Contributions aux méthodologies et outils de conception des System-on-Chip : capture applicative, architecture et sécurité

Jean-Christophe Le Lann

DOCTORAT BRETAGNE LOIRE / MATHSTIC

Université de Bretagne Occidentale

HABILITATION À DIRIGER LES RECHERCHES DE

L'UNIVERSITÉ DE BRETAGNE OCCIDENTALE

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : Informatique

Par

Jean-Christophe Le Lann

Contributions aux méthodologies et outils de conception des *System-on-Chip* : capture applicative, architecture et sécurité

HDR présentée et soutenue à Université de Bretagne Occidentale, le 18 Juillet 2025
Unité de recherche : Lab-STICC, UMR CNRS 6285

Rapporteurs avant soutenance :

Roselyne Chotin	Professeure des Universités à Sorbonne Université / LIP6
Jean-François Nezan	Professeur des Universités à l'INSA de Rennes /IETR
Frédéric Rousseau	Professeur des Universités à l'INP - Université Grenoble Alpes / TIMA

Composition du Jury :

Président :	Frank Singhoff	Professeur des Universités, Université de Bretagne Occidentale, France
Examinateurs :	Bertrand Granado	Professeur des Universités à Sorbonne Université / LIP6
	Steven Derrien	Professeur des Universités à l'Université de Bretagne Occidentale / Lab-STICC
	Philippe Coussy	Professeur des Universités à l'Université de Bretagne Sud / Lab-STICC
	Koen Bertels	Professeur à l'Université de Ghent, Belgique

A mes parents.

A Sabine et Evariste.

Mémoire d'Habilitation à Diriger des Recherches

de l'Université de Bretagne Occidentale

Contributions aux méthodologies et outils de conception des *system-on-chips* : capture applicative, architecture et sécurité

Jean-Christophe Le Lann

Présenté le 18 juillet 2025 devant une commission composée de :

Rapporteurs :

Pr. Roselyne Chotin	Sorbonne Université / LIP6
Pr. Jean-François Nezan	INSA Rennes / IETR
Pr. Frédéric Rousseau	Grenoble INP - UGA / TIMA

Examinateurs :

Pr. Bertrand Granado	Sorbonne Université / LIP6
Pr. Koen Bertels	Université de Ghent, Belgique
Pr. Philippe Coussy	Université de Bretagne Sud / Lab-STICC
Pr. Franck Singhoff	Université de Bretagne Occidentale / Lab-STICC
Pr. Steven Derrien	Université de Bretagne Occidentale / Lab-STICC

Table des matières

Chapitre 1

Curriculum vitae

Even a poor plan is better than no plan at all.

Mikhail Chigorin
Grand Maître International d'Echecs

1.1 Etat civil – informations générales

Nom : Le Lann
Prénom : Jean-Christophe
Date de naissance : 18/02/1971 (54 ans)
Lieu de naissance : Brest, Finistère (29)
Poste actuel : Maître de Conférence de l'ENSTA Bretagne

3, Terrasses de l'Aber, 29270 Landéda

06-32-72-40-96

jean-christophe.le_lann@ensta.fr
jc.lelann@gmail.com

- Vie familiale :
  - Pacsé
  - 1 enfant
- Internet et réseaux sociaux :
  - http ://www.jcll.fr
  - https ://www.linkedin.com/in/jean-christophe-le-lann/
  - https ://github.com/JC-LL
  - https ://gitlab.com/JCLL

1.2 Formation universitaire

1.2.1 Résumé

- 1993-94 **Maîtrise de Physique** – Université de **Brest**.

- 1994-95 **D.E.A de Microélectronique**. Institut national polytechnique de **Grenoble**, Université Joseph Fourier.
- 1995-96 **Service national** - Enseignant en Physique et traitement du Signal. Ecole Navale & U.B.O
- 1996-99 **Thèse de Doctorat d'Informatique**. INRIA/Industrie. IRISA, **Rennes**. Conception conjointe logicielle/matérielle, temps-réel.

1.2.2 DEA de Microélectronique - INP Grenoble

- **1995 : Laboratoire CSI** (Gabrielle Saucier) : laboratoire de conception des systèmes intégrés.
- Sujet : Conception et Synthèse VHDL de microprocesseurs 32 bits tolérant aux fautes transitoires.
- Encadrant : Raphaël Rochet.
- Sujet : il s'agissait d'expérimenter un flot de synthèse VHDL de l'outil ASYL+ développé par le laboratoire, qui permettait notamment l'insertion automatique de dispositifs de détection / évitement de fautes transitoires dans des automates complexes. Ces dispositifs étaient de deux types : la triplication simple de blocs complexes (et leurs voteurs), ainsi que la génération de signatures à l'exécution de machines d'états complexes, le long de chemins d'exécution. En particulier, on a cherché à en vérifier l'applicabilité de bout-en-bout en ce qui concerne des descriptions de microprocesseurs. J'ai décrit un microprocesseur appelé HSURF32 et réalisé un ensemble de synthèse attestant de la robustesse de l'outil.

1.2.3 Doctorat d'Informatique - IRISA Rennes

- **1996-1999**. Equipe EPATR : Environnement de programmation des applications temps-réel.
- Directeurs : Paul Le Guernic et Christophe Wolinski.
- Intitulé : **Simulation et synthèse de circuits s'appuyant sur le modèle synchrone**

De type *INRIA-Industrie*, elle a été initée par la société Motorola, et visait à étudier les capacités des *langages synchrones* –et le langage Signal en particulier– en terme de synthèse automatique de circuits. Pour rappel, les langages synchrones sont nés en France au cours des années 1980 : *Esterel*, inventé et popularisé par Gerard Berry aux Mines de Paris (antenne de Nice Sophia Antipolis), *Lustre* (Halbwachs, Caspi) à Verimag Grenoble et Signal (Le Guernic, Benveniste) à Rennes. Inspirés des circuits synchrones, leur sémantique est non-ambigüe et se prête bien à des spécifications et analyses rigoureuses : je me suis intéressé à la préservation de propriétés lors des transformations opérées sur ces spécifications (preuves de raffinement). J'ai démontré au cours de ma thèse d'autres propriétés intéressantes, comme la capacité des compilateurs synchrones à générer des simulateurs *cycle-based* performants. Le troisième aspect défendu a trait aux représentations intermédiaires de ces langages : à l'instar des travaux d'Apostolos Kountouris et Christophe Wolinski [M72], j'ai exploré la notion de HCDG (hierarchical conditional dependency graphs) : les HCDG sont des graphes de flot de contrôle et de données (CDFG) particuliers, qui exposent de manière très précise les gardes booléennes de définition et utilisation des variables. Utilisées au cours de la synthèse comportementale, ces gardes permettent d'élaborer de nouvelles heuristiques, notamment les phases d'ordonnancement et d'allocation de ressources.

Valorisation des travaux de thèse Ces travaux ont été publiés avec succès en conférence et ont donné lieu à 2 revues :

- Polychrony for system-design avec Jean-Pierre Talpin et Paul Le Guernic. Journal of Circuits, Systems, and Computers 12 (03), 261-303. **environ 350 citations**
- High-level synthesis using hierarchical conditional dependency graphs in the CODESIS system – Journal of Systems Architectures - 2001.

1.2.4 Année sabatique au LESTER

En 2006, j'ai effectué une année sabatique qui marquait ma volonté de retour vers le monde académique. J'ai été embauché par le LESTER (Brest, Bernard Pottier) comme ingénieur participant au projet européen FP6 Morpheus. Le sujet traité portait sur le mouvement de données sur la future puce (fondue par ST Microelectronics). Nous avons défini un procédé de compilation de haut niveau (outil de Ronan

Keryell), permettant de générer des descripteurs DMA de mouvements de données massives, exécutés par des unités reconfigurables et multi-canaux. Ceci a conduit à un prototype écrit en C et SystemC.

1.3 Parcours industriel

1.3.1 Thomson R & D France : conception de SoC pour la compression vidéo.

J'ai été embauché en 1999 comme ingénieur par la *design house* ICDH de Thomson Multimedia. Cette entité –bientôt renommée TSC **Thomson Silicon Components**– était en charge de la conception et de la commercialisation de circuits numériques de type ASIC pour des produits du groupe, dont des décodeurs pour set-top-box grand public ("consumer"), mais également pour les professionnels (studio de télévision, production et postproduction cinématographique). J'ai pu, au cours des 10 dernières années du groupe, bénéficier de l'expérience accumulée de cette entreprise de pointe. J'ai progressivement pris des responsabilités techniques dans la conception des architectures des circuits (SoC) de compression video, notamment pour des standards émergeants, dont le H264 (TNT etc).

L'entité regroupait environ 300 ingénieurs répartis entre l'Allemagne (Villingen), les Etats-Unis (Indianapolis) et la France (Rennes). Par ailleurs, la complexité des circuits et le coût de leur NRE a également forcé l'ouverture de nouveaux centres de design (Bangalore et Pekin), avec lesquels j'ai pu collaborer efficacement.

System-on-chip conçus et responsabilités : 1999-2008

- 1999-2001 : circuit Viper - Encodeur Mpeg2 pour caméras professionnelles pour Thomson Nextream.
  - Reprise d'un front-end (400kgates). Simulations sous Cadence Ncsim et émulateur Ikos.
  - Mise en place d'un environnement de preuve formelle d'équivalence Synopsys Formality.
  - Flot de fonderie : IBM.
  - Gestion de versions.
- 2001-2002 : circuits TCI4/5
  - Décodeur Audio MP3 pour pick-up CD.
  - Conception mixte VHDL et Verilog d'un déboggeur intégré multi-DSP.
- 2002-2008 : SoCs Prometheus, Brubeck, Montgomery, Django.
  - Coeurs de décodeur video programmable et multistandard pour S/HDTV.
  - 2,5 Million de portes logiques CMOS 130nm, 90nm et 45nm. Technologie Texas Instrument et TSMC.
  - SoC multiprocesseur RISC + périphériques d'accélération sur interconnect Amba AHB (SIMD, ASIP, accélérateurs dédiés).
  - Responsable du bloc IP de décodage entropique (VLD,CABAC,CAVLC) et reconstruction vecteur
  - Conception VHDL : simulation Modelsim (Mentor Graphics), et Synopsys VCS.
  - Drivers sur chaine de cross-compilation GNU-GCC.
  - Co-vérification HW/SW full-system sur émulateur Cadence Palladium.
  - Intégré de l'IP dans l'encodeur Mustang MPEG4 HD de Grass Valley.
  - Normes : MPEG2,4, H264/AVC,AVS,VC1,DV.

1.3.2 Montage de projets collaboratifs

Au cours des deux dernières années de l'activité TSC, le groupe s'est ouvert au monde universitaire, à travers le montage de projets collaboratifs nationaux (ANR) et Internationaux (FP6,FP7, Medea+, Catrene) : cela a été l'occasion pour moi de nouer des relations particulièrement intéressantes et solides avec des Universités étrangères (Italie, Pays-Bas notamment). En étroite relation avec le responsable groupe (Henk Heijnen), j'avais les 4 missions principales suivantes :

- Prospection partenariale
- Montage des dossiers techniques

- Animation et vie des projets
- Montage et encadrement de thèses CIFRE

Thèses CIFRE J'ai monté trois thèses CIFRE au cours de cette période, en relation avec différentes universités et écoles d'ingénieurs. Je les ai encadrées en interne durant leur première moitié, jusqu'à la cessation d'activité du groupe Thomson Silicon Components.

- Thèse de Stéphane Lecomte avec Christophe Moy et Pierre Leray à Supelec Rennes, intitulée *Méthodologie de conception basée sur les modèles de haut niveau pour les systèmes de radio logicielle* La thèse s'est déroulée dans le cadre du projet ANR Mopcom sur les méthodologies de conception de SoC à partir de modélisation de type UML-MARTE.
- Thèse de Erwan Raffin, avec François Charot et Christophe Wolinksi à l'IRISA portait initialement sur la modélisation d'architectures multiprocesseurs en SystemC, notamment à l'aide de simulateurs de jeux d'instructions (ISS) décrits dans la bibliothèque SoCLib. La thèse s'est ensuite réorientée vers le *Déploiement d'applications multimédia sur architecture reconfigurable à gros grain (CGRA) : modélisation avec la programmation par contraintes* Erwan travaille aujourd'hui pour Atos.
- Thèse de Muhammad Rachid, avec Bernard Pottier à l'UBO, s'intitule *Holistic Approach to Design Heterogeneous Reconfigurable Systems*. Elle a été soutenue en 2009.

1.3.3 Fondateur et Directeur technique de la startup Modaë Technologies

Démarrage En 2008, l'opportunité nous a été donnée de solliciter un fonds spécifique de montage de startups, lié à la cessation de l'activité TSC. Parmi les 2 projets retenus, figuraient le projet InPixal, ainsi que mon projet Modaë Technologies. Avec mon collègue Pierre-Laurent Lagalaye (aujourd'hui DGA-MI Bruz), nous avons capitalisé nos savoirs-faire en matière de :

- Modélisation applicative.
- Modélisation d'architecture hétérogènes multiprocesseurs.
- Portage d'applications.
- Développement d'outils : simulateurs, synthétiseurs de haut niveau (HLS), compilateurs.

Fondateur et directeur technique Initiateur du projet, j'ai organisé le montage et les idées techniques clés de la société. Ce rôle de CTO comportait notamment les aspects suivants :

- Présentation du projet : concept, marketing, etc. auprès de différents organismes dont notre incubateur Emergys, Bretagne Entreprendre, OSEO/BPI, Ministère de l'Enseignement Supérieur et la Recherche.
- Suivi du montage administratif et financier du projet, en étroite relation avec Pierre-Laurent (CEO).
- Pilotage et développements techniques.
- Dépôt de brevet [P3]
- Recrutement de 2 ingénieurs : Gildas Cocherel et Jean-Charles Roger, localisés à l'INSA Rennes et l'ENSTA Bretagne.
- Entrée de l'actionnariat d'IT-Translation (INRIA Daniel Pillaud, Laurent Kott etc)

Technologie développée

- Nous avons développé un outil de codesign logiciel-matériel polyglote.
- Il assure le passage entre des spécifications exécutables écrites sous forme de processus communicants (synchrones et asynchrones) et des implémentations logicielles-matérielles sur ARM-FPGA.
- L'originalité de l'approche réside dans le recours à des langages interprétés standards et open-source (Ruby et Python) lors de cette capture applicative. Ces langages sont conviviaux et désormais très répandus, du fait de leur caractère interprété et leur orientation objet.
- Cette convivialité rend la tâche de compilation logicielle-matériel complexe, mais va dans le sens d'une élévation du niveau d'abstraction rencontrée dans le domaine de l'électronique. Le logiciel assurait un confort d'utilisation (simulateur transactionnel, debuger, viewer). Il assurait également la synthèse des communications ARM-FPGA à travers la génération automatique de modules noyaux Linux.

Prix et distinctions La start-up Modaë Technologies a été lauréate de différents concours.

- Lauréate 2010 du concours d'entreprises innovantes l'OSEO/BPI et le Ministère de la Recherche : catégorie en émergence.
- Lauréate 2011 du concours d'entreprises innovantes l'OSEO/BPI et le Ministère de la Recherche : catégorie en création.
- Lauréate 2010 du dispositif PHAR : prêt d'honneur d'amorçage de la région Bretagne.

1.3.4 Chercheur détaché à l'IRT BCOM

Après mon intégration à l'ENSTA Bretagne et durant 2 ans environ, j'ai été détaché à temps partiel pour travailler sur la thématique de virtualisation des réseaux, en collaboration avec l'institut de recherche technologique B-COM. Cette thématique s'est orientée vers la définition d'architectures matérielles qui soient à même de supporter de telles virtualisations. Ces travaux se sont organisés autour de la thèse de Théotime Bollengier, et la définition d'*overlays FPGA*.

1.4 Parcours d'enseignant-chercheur

Poste occupé actuellement

Depuis octobre 2008, j'occupe un poste de Maître de Conférences de l'ENSTA Bretagne.

Enseignement J'y enseigne la conception des systèmes numériques embarqués, et leur continuum :

- Electronique numérique élémentaire.
- Langages de description matériel.
- Processeurs et accélérateurs parallèles sur FPGA.
- Architecture des *systems-on-chip* (SoCs).
- Compilation (classique ou sur silicium).
- Introduction au parallélisme.
- Modèles de calculs et communication (MoCC).

Recherche

- Mes travaux de recherche portent sur la conception d'outils (logiciels) pour l'Electronique au niveau Système (ESL : Electronic System Level), qui vient en relai de l'Electronic Design Automation (EDA).
- Cette recherche s'effectue au sein du Labsticc, dans l'équipe ARCAD du pôle MOCS (Méthodes et Outils pour la Conception des Systèmes).
- Mes développements logiciels sont nombreux et variés : simulateurs, interpréteurs, compilateurs, synthétiseurs ainsi que nouveaux langages pour l'ESL.

Encadrement doctoral J'ai encadré et encadre plusieurs thèses :

- Thèse CIFRE Thalès (2011-2014) de M.Nader Khammassi (Directeur : Jean-Philippe Diguet, UBS-CNRS) sur la parallélisation automatique et semi-automatique sur architectures multicoeurs. Nader travaille aujourd'hui pour Intel dans l'Oregon. Encadrement estimé : 90%.
- Thèse BCOM de M.Théotime Bollengier (2015-2018) (Directeur : Loïc Lagadec) sur la conception d'overlays FPGA (surcouches reconfigurables simplifiant l'accès aux ressources) pour le Cloud Computing. Théotime est aujourd'hui ingénieur spécialisé en Cyber-sécurité à l'ENSTA Bretagne. Encadrement estimé : 50%.
- Thèse ARED de Hannah Badier (2018-2020) (Directeur : Guy Gogniat, UBS) sur l'obfuscation de code source pour la synthèse de haut-niveau (HLS) dans le Cloud et la protection contre le vol de propriété intellectuelle dans les circuits numériques. Elle est aujourd'hui directrice de sa startup. Encadrement estimé : 33%.

- Thèse ARED de Maélic Louart (2020-2023) (Directeur : Abdel Boudraa, Irenav). Thèse sur l'identification temps-réel de falsifications et anomalies de signaux AIS et l'implentation sur FPGA. Encadrement estimé : 33%.
- Thèse autofinancée de Pierre Filiol (DGA-MI, 2022-) (Directeur : Luc Jaulin). La thèse commencée en 2022 porte sur l'exploration logicielle-matérielle du calcul par intervales, offrant des garanties de calculs. Encadrement estimé : 33%.
- Thèse Creach Lab de Quentin Tual (2022-) (Directeur : Philippe Coussy). La thèse cherche à proposer des méthodes des protections de circuits contre les attaques réalisées par une fonderie malveillante.Encadrement estimé : 50%.
- etc [1]
- Démarrage de 2 thèses en 2025 (projet ANR Dynnamo sur les réseaux de neurones dynamiques, en collaboration avec l'IETR Rennes et Nantes).

Bilan

	nombre
Thèses soutenues	4/4
Thèses en cours	2
Thèses à démarrer (2025)	2
Thèses CIFRE Thomson	3
Total	11

Responsabilités majeures Pendant 3 ans (et jusqu'à septembre 2021), j'étais initiateur et responsable du groupe thématique (GT) Circuits & Systèmes, composé de 6 permanents. Le GT C&S est un des 7 groupes qui composent le Département STIC de l'ENSTA Bretagne. A l'initiative de la création de ce groupe, j'ai fédéré les compétences de mes collègues dans le domaine de l'Electronique Numérique et radio-fréquence (RF), en relation avec la Radio-Logicielle et la Guerre Electronique. C'est un objectif collectif et ambitieux, qui a porté ses fruits avec une reconnaissance avérée au sein de l'établissement, et au delà :

- sollicitation de montages de nouvelles filières d'enseignement (FISE, FIPA).
- montages de chaires industrielles.
- projets DGA de type Rapid & Astrid.
- projets MRIS et AID (Agence d'Innovation de Défense)
- projets CPER (Contrat plan état-région)

Le schéma suivant synthétise graphiquement les ambitions du groupe thématique.

1. J'ai également monté et co-encadré 3 autres thèses CIFRE dans le cadre de mon activité d'Ingénieur chez Thomson R&D France (de 1999-2008), que je ne mentionne pas dans cette introduction.

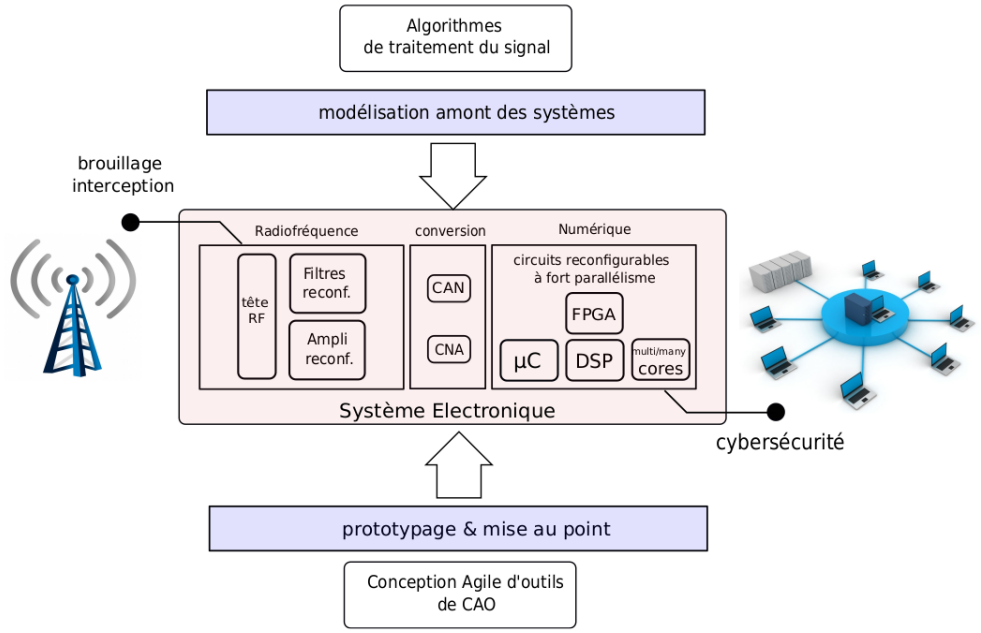


FIGURE 1.1 – Aperçu des activités du GT Circuits & Systèmes de l'ENSTA Bretagne, dont j'ai été un des initiateurs, puis responsable. La thématique est centrée sur les systèmes embarqués : outils de conception (CAO/EDA/ESL), algorithmes de traitement du signal numérique, circuits radio-fréquence, notamment appliqués à la cybersécurité et la guerre électronique.

1.5 Activités d'enseignement

Note : ne sont mentionnés ici que les volumes horaires les plus marquants. Mon volume horaire exigible est de 192 heures ETD. Toutefois, mon temps effectif de présence devant les élèves dépasse les 250 heures ETD. Certains enseignements plus anciens (IFSIC Rennes, Ecole Navale, UBO) ne sont pas listés.

ENSTA Bretagne FISE – Formation d'Ingénieurs sous Statut Etudiant **Brest**

- 1ère année - UV de 30 heures. *2008 → 2013*
  - Participation à l'UV : Introduction à Java et Algorithmique.
  - Participation à l'UV : Electronique numérique.
- 1ère année - UV de 60 heures. *2013 → 2019* (180 élèves)
  - **Responsable de l'UV** : Traitement de l'information (dont Electronique)
- 1ère année - U.E de 30 heures. *2019 → aujourd'hui* (180 élèves)
  - **Responsable de l'UE** : Electronique numérique.
- 2ième année - UV de 60 heures. *2013 → 2019*
  - **Responsable de l'UV** : Architectures numériques et compilation.
  - Participation à l'UV : projet d'ingénierie système (partie 1).
  - Participation à l'UV : projet d'ingénierie système (partie 2).
- 2ième année - UE de 30 heures. *2019 → aujourd'hui*
  - **Responsable de l'UE** : Compilation.
- 3ième année - UV de 30 heures. *2009 → 2011*
  - **Responsable de l'UV** Codesign logiciel-matériel.
  - Montage du cours HLS – 10 heures.
  - Montage du cours MOC Modèles de calculs – 6 heures.
  - Organisation du cours Linux embarqué.

- 3ième année - UV de 30 heures. *2023 → aujourd'hui*
  — Montage du cours Introduction au parallélisme – 10 heures.

ENSTA Bretagne FIPA – Formation d'Ingénieurs Par Alternance **Brest**

- **Acteur majeur la formation FIPA Systèmes embarqués** : *2013 → 2023*
  — Environ 15 étudiants par promotion
  — Responsabilité complète *2013 → 2017*.
  — Initiateur (avec Olivier Reynet) et accompagnement de la réforme.
  — Création du syllabus.
- 2ième année - UV de 30 heures.
  — **Responsable de l'UV** : VHDL 1 *2020 → aujourd'hui*
  — **Responsable de l'UV** : Compilation *2021 → aujourd'hui*
- 3ième année - UV de 30 heures *.2020 → aujourd'hui*
  — **Responsable de l'UV** : VHDL 2.
  — **Responsable de l'UV** : Introduction au System-on-Chip.
  — **Responsable de l'UV** : Projet d'application système (semestre entier).

ENSTA Bretagne Euro Access Graduation program **Brest**

- Encadrement annuel de 8 étudiants chinois sur des projets d'informatique/électronique.
  — *2012 → 2014*

ENSTA Bretagne Formation continue **Brest**

- Introduction à VHDL et aux FPGA
  — *2012 → 2020*
- Initiation à Ruby et à la métaprogrammation.
  — *2012 → 2022*

Université de Bretagne Occidentale **Brest**

- Master 2 Recherche Informatique. 2009
  — Intervention : "Retour d'expérience de 10 ans en conception de SoC" - 3 heures

Telecom Bretagne et IMT Atlantique **Brest**

- Modèles de calcul pour la conception ESL des systèmes embarqués. *2015 → aujourd'hui*

1.6 Animation de la Recherche

Montage de projets collaboratifs à titre industriel Comme expliqué précédemment dans ce dossier, un de mes rôles chez Thomson R&D France consistait à organiser le montage de projets collaboratifs, au niveau national ou européen, puis à les animer. Parmi les projets emblématiques auxquels j'ai pu contribuer autant lors du montage que lors de leur exécution, on peut citer :

- Projet FP6 hARTES : conception d'une chaîne de compilation holistique pour les systèmes embarqués temps-réels embarquant des éléments reconfigurables.
- Projet Catrene CANVAS : co-design des systèmes embarqués hétérogènes (CoWare Leuven).

- Projet CATRENE Mongs : méthodologies de conception orientées objet pour les SoC (Univ. Paderborn, Politechnico di Milano)–avant fermeture d'activité.
- Projet ANR Mopcom : conception de SoC sur FPGA à partir de formalismes de type UML (Thales TAS).
- Projet ANR Lomosa : minimisation de la consommation d'energie dans les systèmes embarqués.

Montage de projets collaboratifs à titre universitaire j'ai participé au montage et à l'animation de différents projets universitaires (liste non exhausive).

- ARTEMIS iFEST (KTH Stockholm, TU Delft, Thales, ENSTA Bretagne,...).
- ANR Gemoc –jusqu'à mon retrait volontaire.
- DGA MRIS Modélisation multi-paradigmes de systèmes Cyber-physiques.
- CPER SSI : Emulation SoC et Plateformes parallèles pour la cyber-sécurité matérielle.
- DGA AID Tectonic (avec l'Ecole Navale) : portage sur cible embarquée d'applications de localisation orientées terrain.

Expertise de projets collaboratifs nationaux et internationaux

- Catrene/MEDEA+ : expertise d'accompagnement DGCIS pour le projet TSAR (Lip6,Bull,...)
- RNTL 2006-2007 (dont SoClib)
- RNTL 2008-2009
- 2013 Expertise internationale Norvegian-Estonian Research Cooperation Programme
- 2015 Expertise internationale Norvegian-Estonian Research Cooperation Programme
- 2017 Membre du Advisory Board du projet H2020 Quanto - processeur quantique.

Jury de thèse – Examinateur

- 2009 – M. Muhammad Rashid (dir : Bernard Pottier - UBO) Holistic Approach to Design Heterogeneous Reconfigurable Systems. Thomson R&D and UBO.
- 2013 –Mme Khawla Hamwi (dir : Ommar Hammami- ENSTA Paritech -Ammar Sharaia -ENIB)– Low Power Design Methodology and Photonics Networks on Chip for Multiprocessor System on Chip.
- 2016 – M.Imran Ashraf (dir : Koen Bertels - TU Delft - Pays-Bas). Communication driven mapping of applications on multicore platforms.

Participation à des comités de lectures de journaux

- **IEEE TCAD** Transactions on Computer-Aided Design of Integrated Circuits. 2019, 2020, 2021, 2024 (Pr.Philippe Coussy).
- **IEEE TCSVT** Transactions on Circuits and Systems for Video Technology. 2012 (Pr.Mattavelli).
- **IEEE Access** 2024 (Pr.Christian Pilato).

Participation à des comités de lectures de conférences

- **DATE** Design Automation & test in Europe : 2017, 2019
- **FPL** Field-Programmable Logic and Applications : 2019
- **VIPES** Virtual Prototyping of Parallel and Embedded Systems : 2013
- **APSEC** Asia-Pacific Software Engineering Conference : 2012 et 2013
- **ReConFig** International Conference on ReConFigurable Computing and FPGAs : 2012,2013,2014
- **Recosoc** International Symposium on Reconfigurable Communication-centric Systems-on-Chip : 2018
- **Sympa** Symposium en Architectures nouvelles de machines : 2011 + PC Member

Participation à des comités de suivi individuel (CSI) de thèses externes

- Thèse C.E.A en Instrumentation pour le nucléaire. 2023-2025 (Julien Mastrangello, dir : Johann Laurent, UBS).

Participation à l'organisation de manifestations scientifiques

- MDD for Distributed Real-time Embedded Systems (MDD4DRES) – Aussois, 2009.
- Animation de la journée B-Ware (embark-day) ENSTA Bretagne – 2012.
- Organisation de Journée Linux Embarqué à l'ENSTA Bretagne (3A) (Armadeus, Alcatel, etc)–2012.

Divers

- Membre de **H.I.P.E.A.C** : High Performance Embedded Architecture and Compilation.
- Point de contact ENSTA-Bretagne : pôle spatial universitaire (2017-2020).

1.7 Publications

Mes publications figurent en fin de document, classées par sections.

Je propose ici un bilan chiffré.

Type	Nombre	Intitulé
Chapitres de livres	3	Springer
Brevets	3	FR et US
Revues internationales	8	JSA(x3), DSP, SCP, Acta Cybernetica,ESA
Conférences internationales	18	DATE(x3),IVLSI, ARC, FPGA4GPC,HPCC,PDCTA, DASIP,ReCoSoC, DSD,...
Workshops internationaux	6	DATE/OSDA(x2),ERTS,...
Conférences nationales	8	Compas, Gretsi,...
Présentations en GDR	5	GDR SoC2, GDR ISIS

Première partie

Introduction et contexte

Chapitre 2

Présentation du manuscrit

Straight roads do not make skillful drivers

Paulo Coelho

Sommaire

2.1 Avant propos

Ce manuscrit présente une synthèse de mes activités de recherche, en grande partie réalisées à l'ENSTA Bretagne et au sein du laboratoire Lab-STICC (UMR 6285) depuis 2008. Cette activité s'est nourrie d'une expérience industrielle préalable d'une dizaine d'années (1999-2008) en tant qu'ingénieur au sein de l'entité Thomson Silicon Components de la société Thomson R&D France, basé à Cesson-Sévigné près de Rennes et acteur mondial de la compression vidéo. J'ai ainsi eu la chance de participer activement à la conception de circuits intégrés de grandes dimensions : les *System-on-chip* (SoC) [1]. A travers les algorithmes de compression et décompression vidéo, j'ai pu ainsi me plonger au coeur de la problématique des SoCs en général : comment, à partir de comportements algorithmiques réputés complexes, faire émerger une structure matérielle performante qui réalise ces calculs ?

Une grande partie de la réponse à cette question repose sur les capacités des outils logiciels utilisés lors de la conception. L'avénement des SoCs n'a été rendu possible que grâce aux efforts soutenus et conjoints de la CAO et de la micro-électronique depuis les années 70. Après mon activité d'architecte et concepteur où j'*utilisais* de tels outils, mon activité de recherche s'est donc assez logiquement recentrée sur l'étude et le prototypage de ces outils. Cette recherche s'est déroulée à la fois dans le domaine académique, mais également par le montage de la startup Modaë Technologies (2009-2014) dans ce même domaine de l'EDA (*Electronic Design Automation*). Cet historique assez long m'amène à couvrir, dans ce mémoire, plusieurs aspects de la CAO : l'élaboration de langages spécialisés (DSL), la synthèse comportementale (HLS), les circuits reconfigurables ou encore la sécurité du flot de conception.

Ces SoCs revêtent une importance considérable à bien des égards : ils permettent à tout un chacun de communiquer, de se divertir et d'interagir à moindre coût dans la vie de tous les jours, dans le confort

1. A titre d'exemple, le circuit Mustang, conçu à Rennes, a longtemps été le plus gros circuit fondu par Texas Instruments, avec une surface de près de 360 mm^2.

et la sécurité, grâce à des économies d'échelle, jusqu'ici jamais atteintes dans l'histoire des sciences et techniques. Ces mêmes technologies permettent aux chercheurs et ingénieurs de concevoir des systèmes embarqués sophistiqués, désormais dotés d'intelligence et destinés à différents secteurs de l'industrie. Ils participent à la stabilité des nations, en s'imiscant dans l'ensemble des dispositifs qui assurent la sécurité nationale, la défense des territoires et le bon fonctionnement des transports et infrastructures critiques, civiles ou militaires.

Pourtant, s'ils constituent la clé de voute de nos sociétés modernes, les *System-on-chip* restent méconnus du grand public : l'avénement rapide de ces SoC, couplé à la technicité et aux ordres de grandeurs vertigineux qu'ils sous-tendent, font que ces enjeux ont de quoi échapper aux non-spécialistes. On peut aisément se méprendre sur les dimensions de ces *puces* bien discrètes, dont le gigantisme effectif ne se mesure qu'en milliards de transistors assemblés et au coût rédhibitoire des nouvelles usines de production.

Il en est de même pour les *outils* qui permettent de les mettre en oeuvre : l'élaboration de ces outils nécessite un effort soutenu de la part de l'Industrie et du monde académique. Ces outils doivent être à même de répondre à la fois à la complexité grandissante des objets manipulés eux-mêmes, mais également à l'alongement des *flots de conception* : la conception des SoCs nécessite en effet la mise en place de chaînes d'outils de plus en plus longues et élaborées, et de méthodologies guidant leur utilisation. Cette conception périlleuse est immanquablement sujette à différentes menaces.

Mes travaux, comme ceux de mes collègues du domaine, visent à parer ces menaces. La plus grande de ces menaces n'est cependant pas, selon moi, celle liée aux cyber-attaques dont les SoCs sont désormais la cible. La plus grande des menaces est celle que j'appelle la *menace de non-conception* : je la définis comme l'incapacité à pouvoir *modéliser* le futur système dans son ensemble, aux stades amont de la conception. L'hétérogénéité des plateformes et la complexité des applications se posent en véritable challenge pour la CAO. Cette modélisation amont est déterminante : elle conditionne le démarrage effectif, ainsi que le succès, de l'ensemble de la chaîne de conception en aval.

Pourtant, si le nombre de ces *design starts* (ASIC ou FPGA) a longtemps été en chute libre [M11], les chiffres du moment suggèrent un *renouveau* de l'activité autour du Silicium [M73] : dopés par différents domaines (automobile, intelligence artificielle, cloud computing, etc), par un mouvement open-source puissant [M62] ainsi qu'une prise de conscience collective, les SoCs connaissent un nouvel essor. Hennessy et Patterson parlent ainsi d'un *Nouvel âge d'or en matière d'architecture des machines*" [M33]. Il s'agit pour la communauté de la CAO (EDA/ESL) d'accompagner ce renouveau de manière appropriée, en proposant de nouvelles manières de concevoir ces produits. Il s'agit là d'un terrain de jeu fantastique !

2.2 Outils d'aide à la conception : de l'EDA à l'ESL

EDA : Electronic Design Automation Le domaine de l'EDA est vaste, allant de la modélisation de dispositifs analogiques aux simulateurs et synthétiseurs de circuits décrits à l'aide de langages de description matériel comme VHDL et Verilog, en passant par le placement-routage ou la vérification formelle. Industriellement, des sociétés américaines comme Synopsys, Cadence ou Mentor Graphics (aujourd'hui Siemens) ont joué le rôle de pionniers et faconné le paysage de l'EDA, aidés par des rachats en avalanche et une croissance organique. Les circuits décrits à l'aide des outils développés par ces sociétés sont parfois quasi-invisibles à l'oeil nu, mais présentent une complexité structurelle et comportementale telle qu'ils ne peuvent être manipulés qu'à travers des *abstractions logicielles* : nous verrons ainsi dans ce manuscrit que les concepts élémentaires de *graphe* et d'*arbre* sont omniprésents. Toutefois ces seules représentations ne suffisent pas et requièrent le recours à des techniques issues du génie logiciel, notamment lorsqu'il s'agit de concevoir de nouveaux langages support de l'activité de conception.

ESL : Electronic System Level L'évolution de l'EDA depuis environ 25 ans a vu émerger l'arrivée d'un autre terme auquel je me réfère plus volontiers pour une grande partie de mes activités : celui de *Electronic System Level* (ESL). L'Electronic System Level peut être considéré comme une simple continuité

des activités précédemment couvertes par l'EDA, mais concerne des activités plus en amont des flots de conception, à savoir :

- ▷ les techniques et outils de capture applicative : modélisation amont, DSLs, transformation de modèles.
- ▷ le prototypage virtuel et l'évaluation de performance
- ▷ la synthèse au niveau algorithmique (composant unitaire ou 'IP') et système (ensemble de composants). Cette synthèse peut être autant matérielle (HLS) que logicielle, s'appuyant sur des modèles de calculs (MoCs). Elle vise autant les FPGAs que les SoCs.
- ▷ la simulation comportementale ou transactionnelle (au bit près, au cycle près, etc)
- ▷ la vérification : par génération de bancs de tests robustes ou par technique plus formelle.

2.3 Structure du document : trois axes, trois menaces

2.3.1 Axes thématiques

Information

Le document est structuré autour de **trois axes thématiques**, que sont :

1. la **capture applicative** des applications portées sur SoC
2. les **aspects architecturaux** des SoC
3. la **sécurité** des SoC

2.3.2 Organisation des chapitres

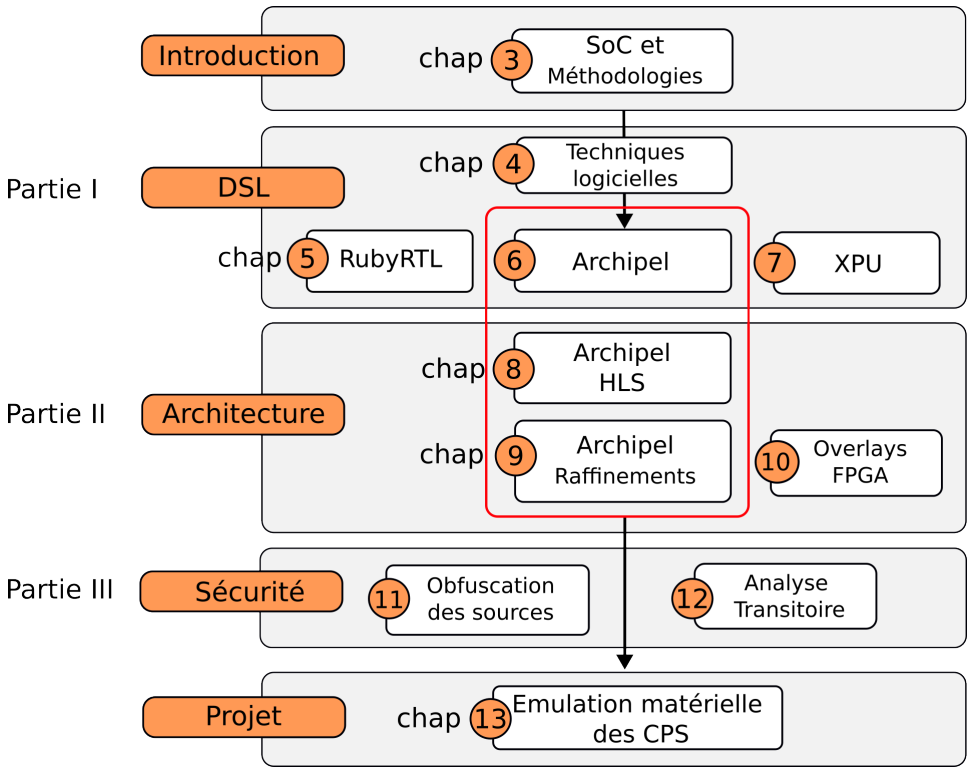


FIGURE 2.1 – Plan de l'exposé

2.3.3 Contributions

1-Contribution à la capture applicative (DSL) Il s'agit de trouver des moyens d'expression *adaptés* aux applications, qui permettent de les traiter de manière plus directe. On parle volontiers de *Domain-Specific Languages (DSL)*. J'ai exploré plusieurs voies concernant l'élaboration de ces DSL, pour 3 niveaux d'abstractions classiques.

- ▷ Le premier niveau est celui de transfert de registres (RTL) : dans le chapitre 5, j'ai exploré le concept de *Hardware Construction Languages (HCL)* à travers l'élaboration d'un tel HCL appelé `RubyRTL`, qui est un *DSL interne* au langage mainstream Ruby. Il possède de grandes similarités avec *LiteX* [2] et son DSL sous-jacent appelé *Migen*, également présenté dans ce même chapitre.
- ▷ Le second niveau d'abstraction étudié est celui associé à la synthèse comportementale de circuits (HLS), qui permet l'obtention automatique de circuits au niveau RTL : dans le chapitre 6, je présente le langage `Archipel` et son outil éponyme. `Archipel` permet la description de systèmes complets sous la forme d'un réseau d'*acteurs flot-de-données* communicants par canaux unidirectionnels et bloquants. `Archipel` permet à la fois la simulation complète des tels systèmes (chapitre 6), mais également la synthèse directe (chapitre 8 et 9) de l'ensemble du système. Le focus est donc plus ici sur l'obtention de ce système complet (*System-level synthesis*), plutôt que l'optimisation forte de chaque élément du système.
- ▷ Enfin, un troisième niveau d'abstraction –celui des tâches– est présenté dans le chapitre 7 : à travers l'outil `XPU`, je décris les travaux autour de la parallélisation explicite et implicite (automatique) sur processeurs multicoeurs sur étagère.

Ces trois niveaux d'abstraction balaient ainsi des paradigmes différents utiles à l'exploration de solutions architecturales lors de la conception de SoC complexes.

2-Contribution aux aspects architecturaux Dans la partie III du document, je propose 3 chapitres.

- ▷ Les deux premiers exposent des techniques de synthèse HLS/SLS que j'ai pu proposer autour d'`Archipel`, ainsi que l'idée générale de raffinements incrémentaux.
- ▷ Dans le troisième chapitre, je présente des dispostifs reconfigurables à grain fin, décrits au niveau RTL : il s'agit des mécanismes de base d'*overlays FPGA*. Ces overlays sont des dispositifs qui possèdent les mêmes propriétés de reconfigurabilité que les FPGA, et synthétisés sur ces derniers, mais qui visent à dépasser certaines limites traditionnelles des FPGA.

3-Contribution à la sécurité des SoC Dans la partie IV du document, j'expose des travaux réalisés autour de la sécurité matérielle.

- ▷ Une idée maîtresse constitue l'ossature de ces travaux : celle de l'*obfuscation des sources logicielles* avant synthèse HLS. Plus précisément, nos résultats les plus intéressants sont basés sur l'idée d'une *obfuscation transitoire* de ces sources : ainsi, il ne s'agit pas de protéger le circuit final, mais de protéger les artefacts logiciels lors des échanges multiples qui ont cours dans les écosystèmes de conception modernes (outsourcing, chaînes de conception longues, etc). Nous avons proposé des procédés de protection de circuits contre deux types de menaces : menace de vol de propriété et menace d'insertion de chevaux de Troie *lors de la conception* (design time).
- ▷ Un second chapitre, esquisse des techniques alternatives à la détection des chevaux de Troie, par une analyse des signaux combinatoires *en cours de stabilisation*. Ce travail, financé par le laboratoire Creach Lab (DGA, Région), reste un travail en cours.

2.3.4 Menaces

J'associe ces trois axes à *trois menaces* distinctes qui planent sur la conception de ces SoC.

La menace de non-conception La première de ces menaces a déjà été évoquée en introduction. C'est celle de la *non-conception*. Je la définis comme l'impossibilité de modéliser un système dans son ensemble aux premiers stades de la conception. L'obstacle majeur est celui de la *complexité applicative* qui, si elle n'est pas traitée de manière sérieuse, a toute les chances de mener à cette impasse de non-conception. A

2. qui est central dans le mouvement open-hardware mondial

titre d'exemple, la complexité du standard de compression vidéo H264 est estimée à 6 à 8 fois celle du standard Mpeg2 précédent : il s'agit là d'une marche abrupte, qui "casse" les démarches de conception précédentes. Là encore, on a vu le nombre d'acteurs industriels se réduire, non-préparés à ce changement d'échelle de la complexité [3]. L'importance de disposer de méthodologies outillées adaptées à l'élévation de la complexité des applications est un enjeu majeur face à cette menace de *non-conception*. Il ne s'agit pas de faire "plus vite" (selon l'argument classique du *time-to-market*) ou "mieux" (selon l'argument classique du *quality-of-service*), mais tout simplement de *pouvoir imaginer* un futur SoC : l'axe 1 de mon activité vise précisémment à dégager des stratégies et des techniques de modélisation amont , qui assurent une capture efficace des applications et une première visibilité sur le futur système. Je reste notamment convaincu que ce problème de non-conception peut-être en grande partie résolu de manière appropriée à l'aide d'une modélisation *dataflow*. Cette modélisation repose sur le concept d' *acteurs*, qui vise à la fois à fournir des abstractions logicielles adaptées à une capture logicielle indépendante de la plateforme, et une génération complète du système. Cette idée n'est pas neuve, mais peine encore à s'incarner dans des outils convaincants.

La menace de l'hétérogénéité des architectures Les techniques de conception actuelles restent pour l'essentiel de type *bottom-up* : dans l'approche dite *platform-based design*, la conception repose sur l'usage de différents blocs de propriété intellectuelle (IP *intellectual property blocks*) : ces IP sont des périphériques d'accélération addressables par un processeur central. Ces IP présentent elles-mêmes différentes granularités, allant de simples interfaces (SPI, contrôleur mémoire,etc) jusqu'à des unités de calcul avancées (coeur de processeur, GPU, CGRA, MPPA, etc). Cette composition reste malheureusement très mal définie. Cette abondance et cette hétérogénéité conduisent à la même menace d'échec que précédemment : les pratiques de composition *ad hoc* de ces éléments disparates se révêlent cauchermardesques ! Deux démarches alternatives sont explorées dans le manuscrit :

- La première repose à nouveau sur une ***modélisation amont*** à l'aide du concept d'acteurs, précédemment évoqué. Le simple fait de délivrer ce concept (sa syntaxe et son outillage) aux concepteurs leur offre la possibilité de modéliser de manière homogène, et les éloigne ainsi de la menace d'hétérogénéité. La démarche favorise par ailleurs la composition progressive de ces acteurs, l'analyse précoce de la composition en résultant, puis la génération automatique d'architectures complètes mixtes-logicielles-matérielles. A l'aide des acteurs, un nouvel écosystème d'*IP comportementales* peut être imaginé. L'outil que j'ai développé autour de cette démarche s'appelle `Archipel` et continue de monopoliser une grande partie de mon temps de recherche. Les plus anciens y reconnaîtront les thèmes récurrents de l'adéquation algorithme-architecture, du codesign hardware-software, et du prototypage rapide.
- Une seconde démarche étudiée au cours de mes recherches, et qui pare les menaces d'hétérogénéité, repose sur l'idée de ***virtualisation***. Il s'agit de chercher à découpler les fonctions à concevoir de celles effectivement mises à disposition par le matériel. Ce découplage fait émerger une couche d'abtraction supplémentaire, qui isole les préoccupations et assure une certaine indépendance des activités d'ingénierie. Cette couche d'abstraction vise ainsi à absorber l'hétérogénéité effective de la plateforme, et permet de manipuler une plateforme virtuelle aux caractéristiques homogènes : les overlays à grain fin (FPGA sur FPGA) ont été étudiés dans la thèse de Théotime Bollengier [Z2].

La menace des cyber-attaques sur silicium La menace cyber est exacerbée par la complexité des circuits et l'allongement des flots de conception : tout à la fois le nombre d'acteurs industriels et humains (*outsourcing*) impliqués, la multiplication des sources d'approvisionnement des IP et l'entremise d'un nombre croissants d'outils de conception tiers augmentent la surface d'attaque. J'ai pu apporter quelques contributions dans le domaine des attaques sur les circuits. Il s'agit toutefois uniquement de menaces liées à la conception matérielle des circuits : dans la thèse d'Hannah Badier[Z1], nous avons ainsi exploré le recours à des obfuscations *transitoires* au cours de la conception (l'obfuscation est alors sensée disparaître dans le circuit fondu) : trois utilisations différentes de ce procédé sont présentées dans le chapitre 11. Dans une seconde thèse, celle de Quentin Tual (en cours), nous cherchons à proposer de nouvelles solutions sur le délicat problème de l'insertion de chevaux de Troie (chapitre 12).

3. Le facteur de 6-8 évoqué est en réalité bien supérieur dans le cas de l'arrivée initiale la TNT : les décideurs institutionnels ont tardé à choisir le standard, et des solutions *multi-standards* ont dû être imaginées...

2.4 Travaux futurs : l'émulation matérielle de CPS

La dernière partie de ce document propose quelques perspectives de recherche autour des systèmes dits *cyber-physiques* (CPS). Il s'agit de systèmes embarqués classiques, plongés dans un environnement physico-analogique et généralement connectés à un système d'information externe : réseaux de capteurs en santé et agriculture, smart grids et villes intelligentes etc. Comme pour les systèmes embarqués classiques, leur comportement, leur fiabilité et leur sécurité (pannes, cyberattaques, etc) doivent être pensés aux stades amont de leur conception. Toutefois leur interconnexion et leur lien avec des phénomènes physiques nécessitent de nouveaux environnements de conception. En particulier, leur simulation à des fins de mise au point, face à des scenario complexes et variables, diffère des approches traditionnelles des SoC précédents : il y a nécessité d'étendre et renouveler la notion de bancs de test tels que nous les connaissons aujourd'hui. Il s'agit de créer des bancs de tests dynamiques, représentatifs de la variabilité de l'environnement du système embarqué. Ces travaux ont d'ores-et-déjà été abordés, notamment dans la thèse de Maélic Louart, soutenue en 2023 [Z4]. C'est cet axe de réflexion qui est exposé dans le chapitre 13.

L'idée d'émuler *électroniquement* ces environnements complexes ouvre selon moi des problématiques de recherche variées, allant de la modélisation du temps physique à des styles de modélisation peu explorés en matière de SoC (modélisation *acausale*, modélisation probabiliste etc). Dans tous les cas, il y a nécessité de repenser les *moyens de capture amont*, sous forme de DSLs [M39], et d'en dériver des architectures spécialisées (DSA : *domain-specific architectures*). Je perçois ce couplage DSL-DSA comme indispensable à une exploration nouvelle des CPS, et peut-être, plus globalement, à l'évolution des SoCs traditionnels.

Chapitre 3

Introduction au System-on-chip :
Du circuit aux méthodologies amont

> People who are really serious about software should make their own hardware.
>
> Alan Kay,
> Turing Award 2003
> Creative Think seminar (1982)

Sommaire

Ce chapitre propose quelques rappels succints sur les SoC : après un historique rapide, quelques exemples concrets de SoC sont présentés, dont certains ont été conçus par mes soins. Ces exemples soulignent dans tous les cas la complexité structurelle évidente qui en émane, et la nécessité de dégager des langages et outils propres à les spécifier dans des flots de conception amont.

3.1 Rappels de Microélectronique

Années 30 et 40 : avénement des semiconducteurs Très tôt, grâce à la théorie quantique et la *théorie des bandes* due à Felix Bloch, il a été possible d'expliquer ce qui distingue un matériau conducteur d'un isolant, et de l'ensemble des matériaux possédant des caractéristiques électriques intermédiaires : les *semi-conducteurs*. L'histoire de la microélectronique se développe alors à une vitesse vertigineuse grâce à plusieurs avancées technologiques clés. L'aventure est évidémment marquée par la découverte du transistor en 1947 par John Bardeen, Walter Brattain et William Shockley aux Bell Labs [1]. Ce composant, capable de fonctionner comme un interrupteur ou un amplificateur, est beaucoup plus petit et plus fiable

1. On peut noter qu'il s'agit d'une des très rares inventions *commandées* de l'histoire de techniques. Bardeen est par ailleurs le seul physicien à ce jour à avoir reçu deux fois le prix Nobel de Physique (1956 et 1972 respectivement pour ses découvertes dans les domaines respectifs des semi-conducteurs et des supra-conducteurs).

que les tubes à vide utilisés précédemment. Le transistor marque le début de l'ère de la miniaturisation en Electronique.

Années 50 et 60 : circuis intégrés Les années 1950 voient la mise au point des premiers *circuits intégrés*. Jack Kilby de Texas Instruments et Robert Noyce de Fairchild Semiconductor sont souvent crédités de cette innovation cruciale en 1958 et 1959. Kilby crée un circuit intégré en utilisant un matériau semiconducteur unique, tandis que Noyce développe une méthode plus pratique pour interconnecter plusieurs transistors sur un même substrat de silicium. Cette découverte révolutionne l'électronique en permettant de réduire encore davantage la taille des composants tout en augmentant leur performance.

Années 70 : microprocesseurs intégrés On assiste non seulement à une élévation de la complexité de la fonction réalisable sur une seule "puce", mais également au *nombre de ces fonctions* : il est désormais possible de réfléchir en terme de "système sur puce" [2]. L'invention du circuit intégré conduit directement ainsi à la naissance des microprocesseurs dans les années 1970. Le premier microprocesseur, l'Intel 4004, est commercialisé en 1971 par Intel. Conçu par Federico Faggin, Ted Hoff et Stanley Mazor, ce microprocesseur contient 2300 transistors et est capable de réaliser des calculs basiques sur 8 et 16 bits. Il est rapidement suivi par des versions au jeu d'instruction plus avancé, comme le 8008 et le 8080 puis, en 1978 le 8086.

Années 80 et 90 : CMOS et loi de Moore Pendant les années 1980 et 1990, la microélectronique continue de progresser avec la loi de Moore, une observation faite par Gordon Moore, co-fondateur d'Intel, qui prédit que le nombre de transistors sur une puce doublerait environ tous les deux ans. Cette tendance favorise une croissance exponentielle de la puissance de calcul tout en réduisant les coûts de fabrication. Les technologies CMOS (Complementary Metal-Oxide-Semiconductor) deviennent dominantes pour la fabrication de circuits intégrés, en raison de leur faible consommation d'énergie et de leur haute densité de transistors.Ces progrès concernant le matériel s'accompagnent bien évidemment de progrès sur l'utilisation de ces microprocesseurs, avec l'avénement de différents langages de programmation, de systèmes d'exploitation et l'émergence de systèmes complets. En retour, les progrès en matière de logiciel participent eux-mêmes à un effet d'entraînement sur le développement du matériel.

Années 00 et suivantes Au tournant du 21e siècle, les avancées en photolithographie et en matériaux permettent de continuer à miniaturiser les composants à une échelle nanométrique. L'intégration se poursuit avec la multiplication du nombre de coeurs de processeurs sur un même substrat de silicium, mais également sur des *chiplets* (plusieurs puces dans un même boîtier). Toutefois, le coût des NRE (*non-recurring engineering costs*) s'élève rapidement, freinant l'essor des SoCs. Les circuits *reconfigurables* gagnent en capacité d'intégration et se présentent, malgré leur coût élevé, comme des alternatives possibles aux ASICs dans de nombreux secteurs.

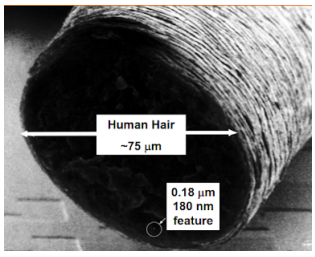


FIGURE 3.1 – Comparaison entre le diamètre d'un cheveu et un transistor (technologie 65 nm). Environ 400 transistors pouvaient occuper ce diamètre dès les années 2000. (credits : MIT Lincoln Lab)

3.2 L'importance des niveaux d'abstractions

Les principes de la miniaturisation obligent à recourir à des *modèles* (appelées ici *représentations* de manière équivalente) qui rendent compte des différentes étapes de fabrication du chips, ainsi que d'al-

2. Le terme ne sera utilisé que plus tard

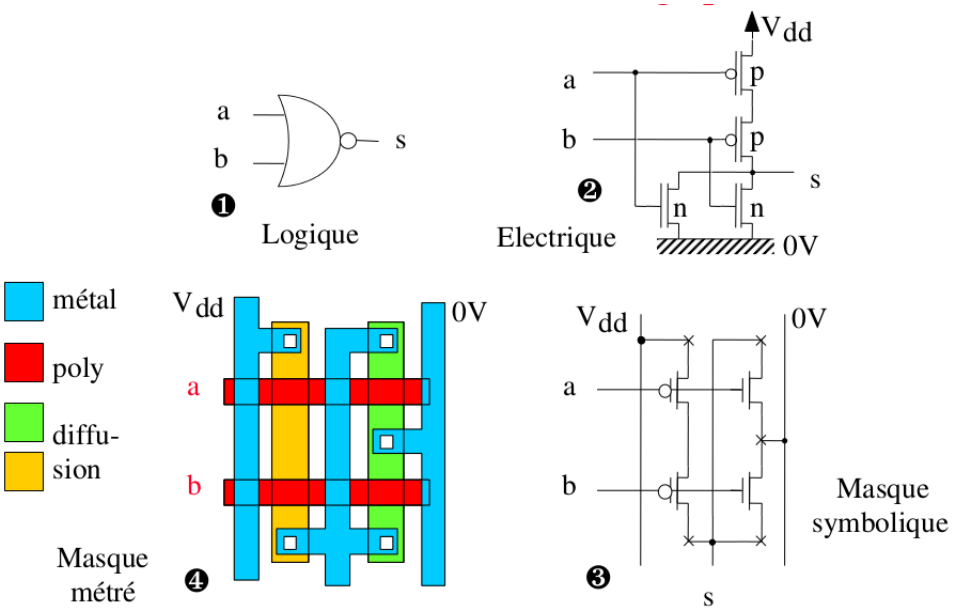


FIGURE 3.2 – Quatre niveaux d'abstraction d'une porte élémentaire `ou` logique. D'après un cours de DEA de Microélectronique de Alain Guyot à l'Université Joseph Fourier 1995.

gorithmes opérant sur ces modèles, de plus en plus volumineux. La figure illustre ainsi quatre niveaux d'abstraction d'une porte logique `nor` : on retouve ❶ le symbole (américain) de la porte, tel qu'il sera utilisé par un concepteur numéricien, la structure interne ❷ qui présente un réseau électrique dual de transistors P et N, un modèle représentant le masque symbolique ❸, qui offre un première incursion dans le domaine topologique, puis un masque métré ❹, qui permet de vérifier le respect de règles de proximité des différents matériaux en présence. Ce masque métré conduit à la génération d'un `gdsII`, format exportable vers des machines de photolitographie. Toutefois, la caractérisation d'un nombre réduit de telles portes logiques, et la constitution de bibliothèques, permet d'éviter une conception *full custom* très coûteuse. La plupart des conceptions numériques se fait selon ce principe de recours à des bibliothèques pré-caractérisées.

Le succès des modèles dans le domaine de l'EDA tient au fait qu'ils permettent d'isoler les préoccupations (*separation of concerns*) de manière remarquable. Probablement très peu de disciplines scientifiques ou techniques peuvent revendiquer un tel succès d'empilement de modèles. L'ensemble de la démarche de conception, y compris sur les phases amont, s'appuie sur l'élicitation de couches d'abstraction : on parle d'approche *constructive* de l'Electronique, mais le terme d'*encapsulation*, désormais plus connu dans le domaine du logiciel, correspond exactement à cette même idée : ainsi, à l'aide de jonctions hétérogènes, on construit des transistors ; à l'aide de transistors on construit des portes, à l'aide de portes on construit des opérateurs arithmétiques, etc. A chaque étape, les propriétés intimes des objets ont été masquées aux interfaces : lors de la réutilisation de ces objets, seules ces interfaces comptent.

Le Y de Gajski-Kuhn Le Y de Gajski-Kuhn est omniprésent dans le cours d'introduction à l'EDA. Ce modèle reste très peu connu hors de la communauté. Pourtant, le Y de Gajski explique de manière remarquable pourquoi, pour concevoir un système performant, il ne suffit pas de se concentrer sur le comportement (ce que fait le programme) : il faut aussi tenir compte de la structure (comment le matériel est organisé pour exécuter le programme) et de la géométrie (où et comment ces éléments sont physiquement réalisés). Ces trois dimensions sont *liées* et *contraignantes* : un programme optimisé peut être ralenti par une architecture inadéquate ou des contraintes physiques, comme la dissipation thermique ou la vitesse de communication. Ce modèle montre qu'ignorer la machine sous-jacente revient à se couper des leviers essentiels pour concevoir des systèmes efficaces. De manière tout aussi importante, il illustre par ailleurs les passerelles qui doivent être réfléchies afin de transiter d'un niveau d'abstraction à un autre (mouvement le long d'un axe) ou d'un artefact à un autre (entre deux axes) : ces suites de passerelles expliquent la constitution des *flots de conception*. Par exemple, passer d'une équation (ou un

programme complet) à une netlist consiste bel et bien à passer du domaine comportemental au domaine structurel, etc.

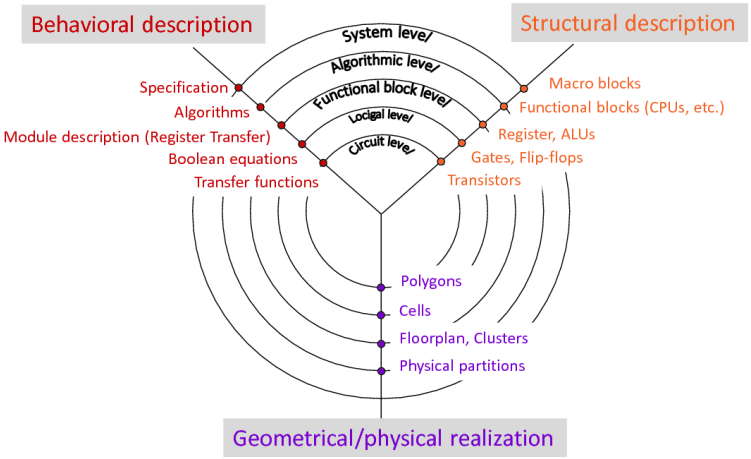


FIGURE 3.3 – Le Y de Gajski-Kuhn représente les 3 axes de représentations des artefacts de l'EDA, et les sauts entre représentations.

Un exemple de transformation : des diagrammes états-transition aux netlists Afin d'illustrer de manière très concrète la notion de représentations et de transformation de modèles, on peut s'intéresser à la transformation qui permet de manière simple de transformer un automate, initialement représenté sous la forme abstraite d'un diagramme états-transitions, en une *netlist*. Cette transformation est selon moi emblématique des "sauts" d'abstraction rencontrés dans le domaine de l'EDA : on passe ici d'une vue "algorithme" (exprimée sous forme d'un diagramme comportemental) à l'axe "structurel" (gates/flip-flops). La figure 3.4 en présente le principe bien connu : la génération automatique et *directe* de la netlist est ici rendue possible par le choix judicieux d'un encodage "one hot" (ou "un bit par état") des états[3]. A l'aide d'un *schéma de traduction* qui porte sur les états initiaux et les états terminaux, la netlist se déduit ici *sans calcul*, de manière parfaitement mécanique.

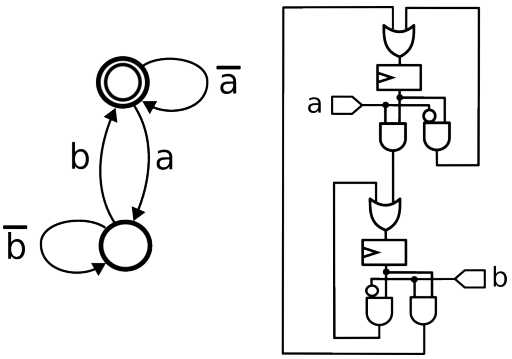


FIGURE 3.4 – Exemple de passage d'un axe comportemental à l'axe structurel : transformation d'un automate d'états finis en netlist, ici à l'aide d'un encodage des états aproprié.

3.3 Complexité des SoC : un aperçu

Comme indiqué précédemment, les SoC modernes embarquent des milliards de transistors, dépassant parfois les 10 milliards pour les processeurs de pointe comme ceux utilisés dans les smartphones (par exemple, les puces Apple M1 ou Qualcomm Snapdragon). Cette complexité des SoC provient directement de leur ambition première : celle d'embarquer sur un même substrat un nombre croissant de

3. Tout autre encodage oblige à établir une table de transition séquentielle et des manipulations booléennes fastidieuses.

fonctions, parfois très variées, et précédemment présentes à l'échelle d'une carte complète, sous forme de composants discrets. Les SoC constituent ainsi un lieu de convergence de technologies et requiert un effort soutenu quant à leur intégration et leur cohabitation. La figure 3.5 donne un aperçu des types de fonctions présentes dans une SoC hétérogène (théorique) :

— Multicoeurs
— Mémoire, contrôleur mémoire et DMA
— Accélérateurs programmables
— Accélérareurs dédiés
— Interfaces de communication et autres

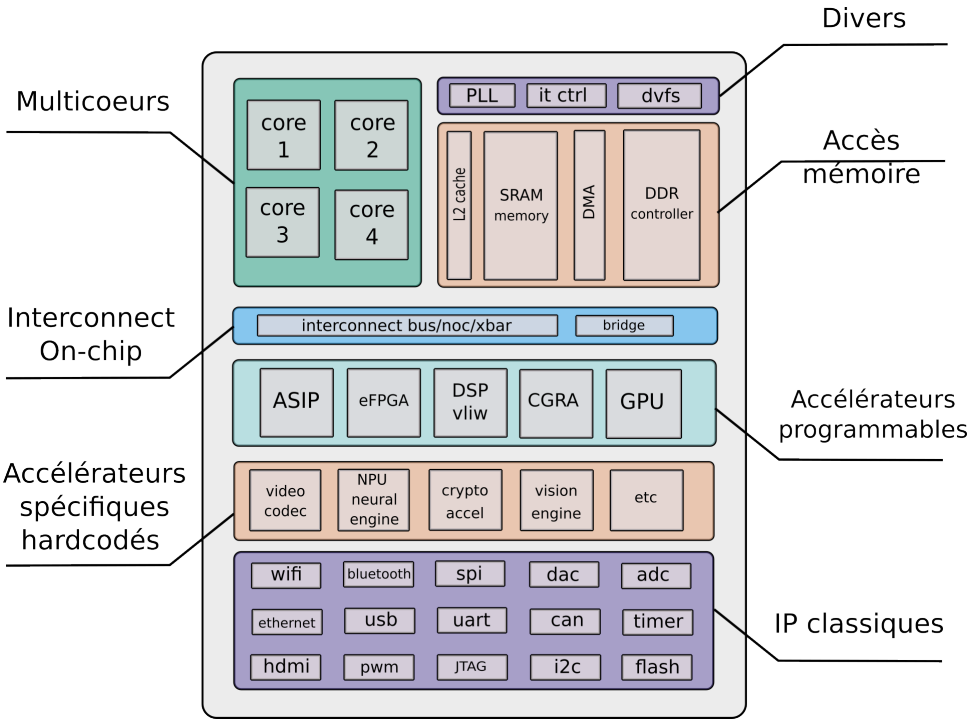


FIGURE 3.5 – Eléments illustratifs de la complexité hétérogène des SoC modernes

IP-based SoC design La constitution du futur SoC nécessite l'adjonction savante d'un très grand nombre de blocs fonctionnels appelé *IPs* (intellectual property blocks), qui sont conçus et délivrés par des sociétés tierces. Sans que cela soit toujours le cas, ces sociétés fonctionnent fréquemment sur le modèle *fabless* : elle vendent ces IPs sous des formes logicielles. Il est parfois nécessaire pour elles de les précaractériser ou les valider en regard de différentes technologies de fonderie. De même, il est possible que ces sociétés délivrent leurs IP sous des formes matérielles spécifiques à ces technologies de fonderie. Des sociétés aussi fameuses que ARM fonctionnent en vendant essentiellement des licences sur leur jeu d'instructions. Deux exemples de SoC réels sont donnés ici, afin d'illuster notre propos.

3.3.1 Exemple du SoC Brubeck de Thomson R&D France

La figure 3.7 illustre l'architecture du SoC Brubeck développé par Thomson R&D France dès 2004, et dont j'étais responsable pour la partie "décodage entropique". Brubeck est le fer de lance d'un ensemble de SoC dédiés à la compression et décompression vidéo. Comme l'illustre la figure, Brubeck est constitué de 4 processeurs Leon (v2) de type SparcV8. Ces processeurs open source (licence LGPL) se présentent comme des *soft-IP* et ont été écrits en VHDL par Jiri Gaisler, de l'ESA (agence spatiale européenne). Un flux de données complexe circule entre la mémoire DDR externe et ces processeurs, via des interconnexions de bus Amba. Trois de ces processeurs sont chacun aidés d'un coprocesseur spécifique. Le coprocesseur lié au décodage entropique est en grande partie *hard-codé*, tandis que les deux autres sont des coprocesseurs SIMD (single instruction, multiple data). A titre d'ordre de grandeur, ces coprocesseurs, essentiels dans le décodage, ont été entièrement développés par une équipe d'une quinzaine de

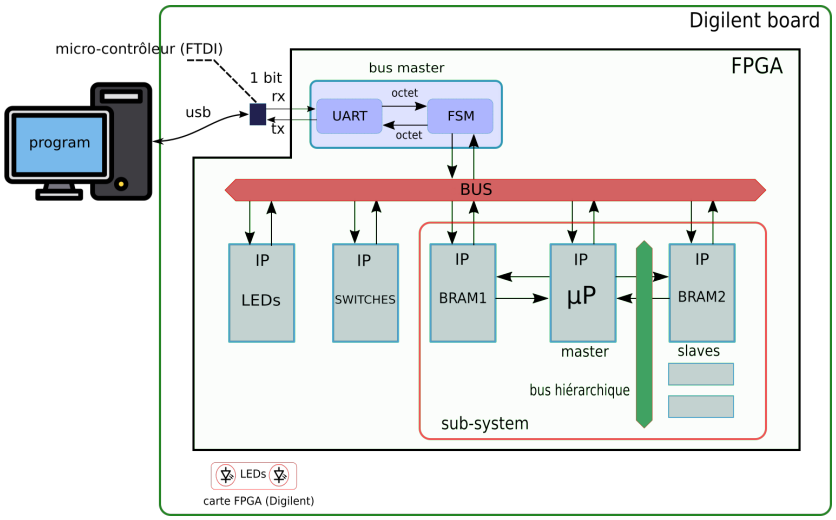


FIGURE 3.6 – SoC conçu par mes soins, comme support des travaux pratiques à l'ENSTA Bretagne. Ce SoC permet l'intégration progressive de plusieurs IP réalisées par les étudiants (FIPA Systèmes Embarqués 3A), dont un microprocesseur 32 bits de type Harvard, appelé Simji. Le bus maison Ottokar est piloté par une UART maître. L'ensemble est totalement indépendant des IP propriétaires Xilinx ou Altera.

personnes à Rennes. Pour information, la fonction du décodage entropique, très irrégulière, agit comme un *parseur* : son but est de récupérer le flux compressé (ici Mpeg2, H264 et DV) présent en DDR et de recomposer un nouveau flux contenant les éléments de syntaxe décompressés pour les traitements en aval (reconstruction image).

3.3.2 Exemple du SoC Precursor

Le second exemple est celui d'un SoC complet appelé Precursor. Je précise que je n'y ai pas participé, mais il me semble intéressant à plusieurs titres. Le projet Precursor est une plateforme open-source dédiée à la conception de System on Chip (SoC) pour la sécurité matérielle. Le projet a été financé par crowd funding où il a recolté un demi-million d'euros. Créée par Bunnie Huang et son équipe, elle vise à fournir un appareil portable sécurisé que les utilisateurs peuvent comprendre, vérifier, et contrôler à différents niveaux, du matériel jusqu'au logiciel. Tous les artefacts de conception, du matériel (schémas, design PCB) aux logiciels, sont disponibles en open source. Cela permet aux utilisateurs et développeurs de vérifier chaque composant du système, réduisant les risques de failles de sécurité cachées. Precursor intègre des fonctionnalités de sécurité avancées, comme l'absence de firmware propriétaire, un bootloader vérifiable et un système basé sur un FPGA (Field Programmable Gate Array), ce qui permet une flexibilité et un contrôle total sur les processus internes. Le cœur de Precursor repose sur un FPGA. Precursor utilise Xous, un micro-système d'exploitation conçu pour offrir des garanties de sécurité fortes et être adapté aux plateformes SoC sécurisées. Xous est minimaliste et écrit avec une attention particulière à la sécurité et à la simplicité. Precursor est conçu pour être utilisé dans un téléphone mobile, avec une interface utilisateur simple mais fonctionnelle. Il offre des performances optimales pour les applications sécurisées. Precursor s'adresse particulièrement aux chercheurs en sécurité, aux développeurs et aux utilisateurs soucieux de leur vie privée qui veulent comprendre et contrôler tous les aspects de leur matériel et de leurs logiciels, contrairement aux solutions commerciales fermées.

3.3.3 Apports des circuits reconfigurables

L'avénement des circuits reconfigurables marque un tournant majeur dans la manière de concevoir les SoC. Le premier FPGA est attribué à Ross Freeman, cofondateur de Xilinx, qui en a eu l'idée pour créer une puce reprogrammable après fabrication. Cette invention permet de reconfigurer la logique d'un circuit sans avoir à produire de nouvelles puces, simplifiant le développement et réduisant les coûts pour des applications variées. Les premiers FPGA, qui n'offraient que très peu de ressources, ont longtemps été cantonnés à des tâches annexes, comme l'adaptation de protocoles. Les FPGA modernes

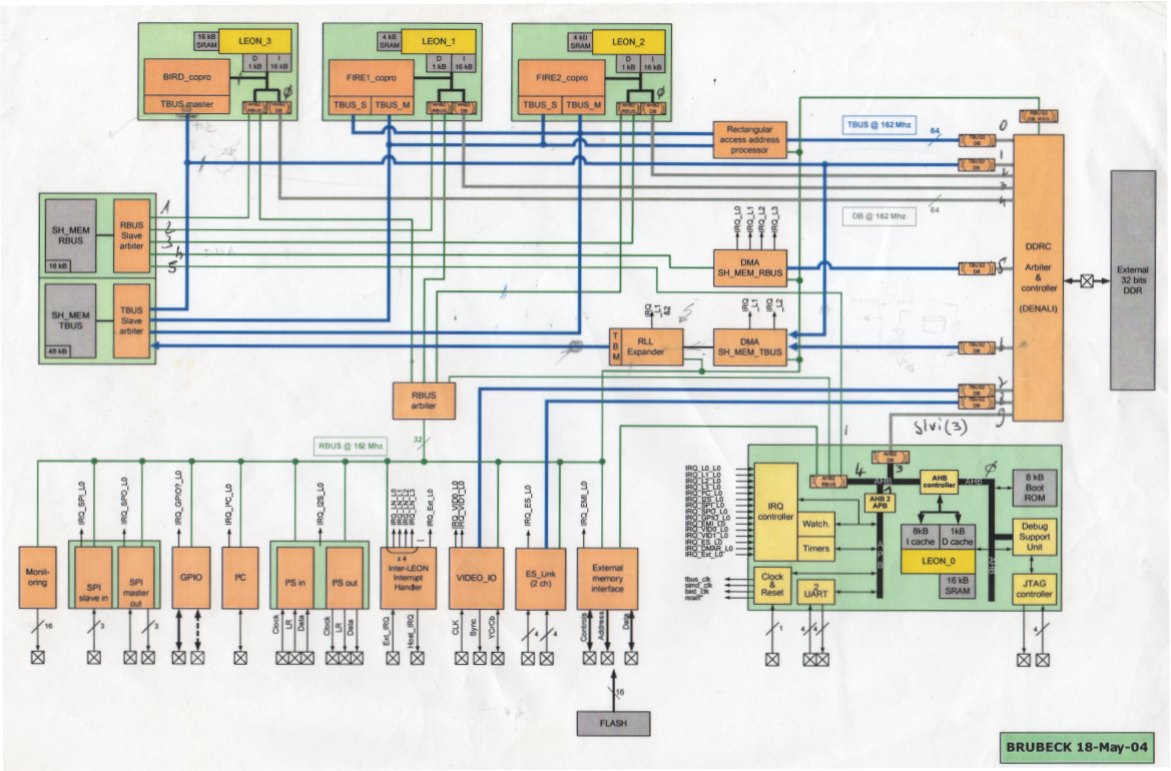


FIGURE 3.7 – Exemple du projet SoC Brubeck multi-standards, développé par Thomson R & D France en 2004. Chaque bloc jaune correspond à un processeur SparcV8 LEON2. J'étais responsable du coprocesseur Bird (en haut à gauche) lié au décodage entropique de standards de compression vidéo couvrant h264, Mpeg2 HD et DV.

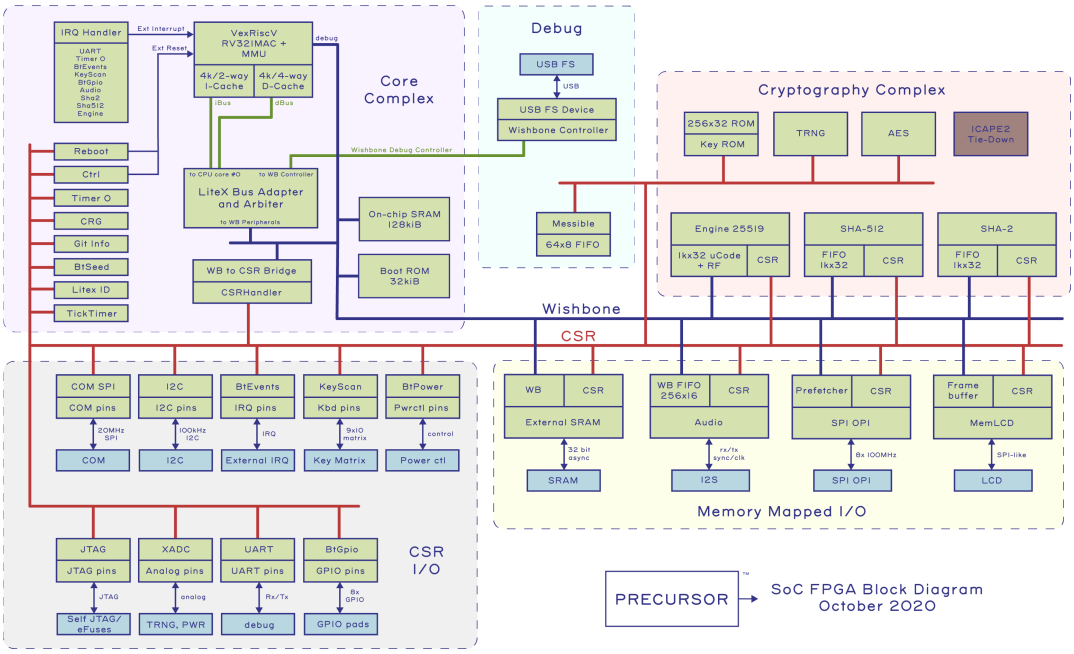


FIGURE 3.8 – Exemple du projet de SoC Precursor, basé sur un processeur VexRiscV, et conçu avec Litex (voir chapitre 5). Il illustre la complexité structurelle de SoC qu'il est désormais possible de développer en open-source.

sont désormais à l'image d'une vaste page blanche sur laquelle un concepteur peut écrire et réécrire

à volonté et y synthétiser des SoCs complexes. De tels dispositifs ouvrent également la porte d'une multitude de cas d'usage variés : reconfiguration lors de l'exécution, exploration d'architecture [M22], prototypage rapide, etc. Cette reconfigurabilité a toutefois un coût non négligeable : en prix et en performance, les FPGA restent pour l'instant rhédibitoires pour bon nombre de secteurs de l'Electronique, qui ne peuvent se permettre de les embarquer dans des produits finaux. Toutefois, une véritable tendance semble amorcée : comme l'indique [M14], le couplage entre FPGA et HLS (High-level synthesis) autorise à imaginer une customisation "démocratisée" et généralisée des accélérateurs de calculs (*DSA : domain-specific accelerators*) et plus largement l'avénement du calcul spatialisé (*spatial computing*, [M17]). Enfin les architectures de type CGRA (*Coarse Grain Reconfigurable Array* [M47]) et leurs flots de conception –qu'il s'agira également de concevoir de manière spécifique– promettent de diminuer drastiquement les temps de compilation sur de telles architectures reconfigurables, et franchir la dernière barrière qui subsiste entre logiciel et matériel.

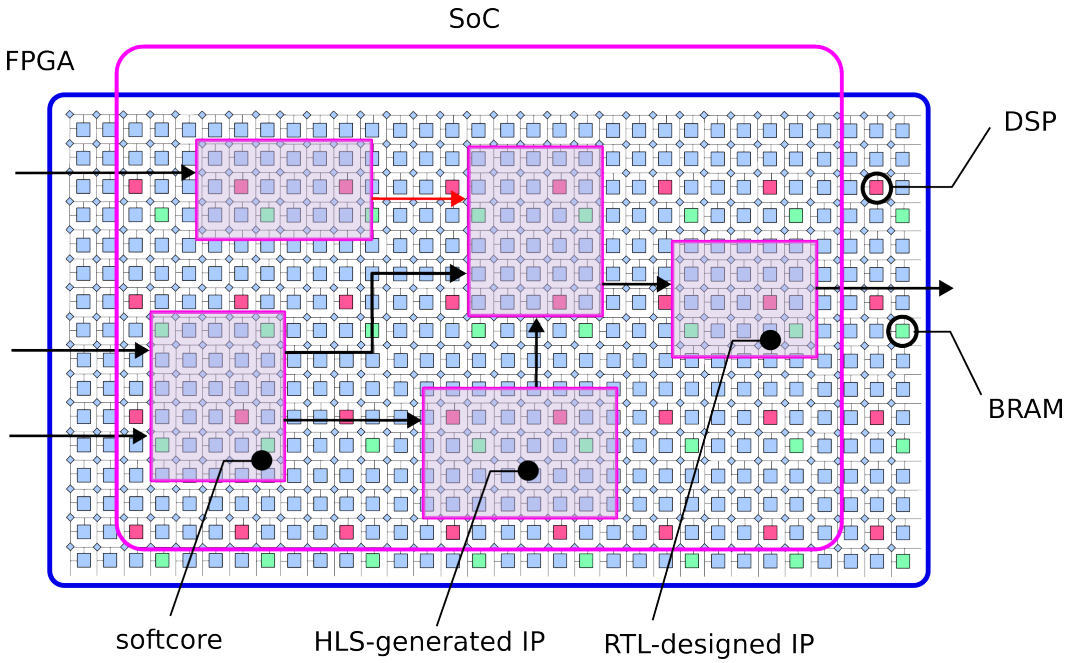


FIGURE 3.9 – Les FPGA modernes permettent désormais le prototypage et le déploiement de SoCs complexes, et participent à l'avénement des *DSA : domain-specific accelerators*.

3.4 Méthodologies et outils de conception *amont*

3.4.1 Sofware-defined SoC

La *définition logicielle* des systèmes sur puce s'inscrit dans une tendance technologique plus large où de nombreux domaines techniques sont progressivement redéfinis par des approches logicielles *domain-specific*.
À l'instar de la radio logicielle (SDR, [M66]) ou des véhicules définis par logiciel (SDV pour *software-defined vehicules* [M43]), qui révolutionnent respectivement les communications sans fil et l'industrie automobile, les SoCs, ainsi définis par logiciel, représentent une évolution technologique majeure dans l'électronique moderne [M71]. Cette transformation, largement entamée dès l'arrivée des HDLs traditionnels, exige désormais une approche collaborative plus étroite entre les concepteurs matériels et logiciels, impliquant des méthodes innovantes comme la conception conjointe logiciel-matériel et le prototypage virtuel (voir figure 3.12). Ces nouvelles pratiques permettent de réduire les cycles de développement, d'optimiser les performances et d'anticiper les contraintes techniques dès les premières étapes de conception. En intégrant des outils de simulation avancés et des méthodologies de co-design, les ingénieurs cherchent à explorer et valider des architectures complexes de manière plus rapide et précise.

Nous avons défendu cette idée, qui s'appuie largement sur l'IDM/MDE, dans le projet ANR Mopcom, piloté par Thalès. Le flot défendu dans Mopcom est illustré sur la figure suivante 3.11.

Approche traditionnelle et *early-binding* Sur la gauche de ce schéma 3.11, l'approche traditionnelle de co-conception est rappelée : l'idée est de scinder les activités de conception logicielle et matérielle *au plus tôt*. Les concepteurs matériels s'engagent à délivrer des fonctionnalités performantes aux développeurs du logiciel embarqué. L'échange d'information entre concepteurs se fait par la rédaction d'une *memory-map* précise, qui indique les adresses des registres d'interface (jusqu'à quelques centaines en général) ou de régions mémoires utilisées comme *buffers*, directement accessibles. Cette ligne de "partage des eaux", qui intervient très tôt entre entre logiciel et matériel n'a pas que des désavantages : elle permet notamment aux équipes respectives de développer ses activités en toute indépendance, à l'aide d'outils bien maîtrisés de part et d'autre. Toutefois, cette méthode traditionnelle nécessite une prise de décision précoce du partitionnement logiciel-matériel. L'histoire montre que, de manière récurrente, ces prises de décisions sont très difficiles sans expérimentations et sans mesures de performances appropriées : un mauvais partitionnement peut ne se découvrir que très tard, lors de la phase d'intégration, ce qui se révèle catastrophique pour le projet lui-même.

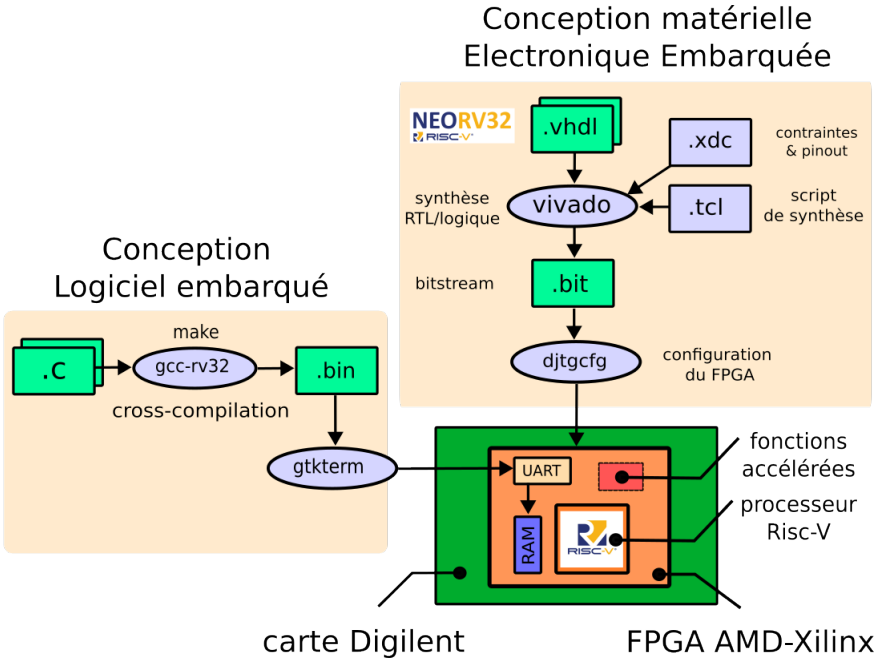


FIGURE 3.10 – Illustration d'un **flot de conception classique**, d'après une serie de travaux pratiques que je réalise avec les étudiants de l'ENSTA Bretagne (FIPA Systèmes Embarqués). Une plateforme matérielle basée RiscV (NeoRV32) est synthétisée sur FPGA, puis un logiciel embarqué utilisant cette platforme est cross-compilé. Certaines fonctions peuvent être accélérées matériellement.

Approche par transformation de modèles et *late-binding* Une seconde approche, qu'on peut qualifier d'*holistique*, prône une approche de transformations *progressives* de modèles, allant de la spécification exécutable jusqu'à l'implémentation mixte logiciel-matériel. Ainsi, c'est intrinsèquement le même modèle, transformé progressivement, qui focalise les activités de l'ensemble de l'équipe de conception. La scission entre métier logiciel et métier matériel ne se fait que tardivement : on parle de *late binding*. Si cette idée de transformations de modèles est effectivement attrayante sur papier, elle n'est toutefois pas sans difficultés. Tout d'abord, il semble difficile de définir ce que sont réellement ces transformations : elles ne sont plus spécifiquement des transformations habituelles, réalisées dans un compilateur standard. Elles sous-tendent l'existence de *profileurs intelligents* capables de piloter ou suggérer les transformations. Ces transformations nécessitent probablement, en outre, la définition de langages spécifiques se prêtant effectivement à ce type de flot de conception. C'est cette approche que je tente de défendre dans les chapitre 6,8 et 9, à propos de `Archipel`.

Un challenge encore plus important s'associe à cette manière de procéder : il s'agit de former une nouvelle gamme d'ingénieurs (ni issus spécifiquement du logiciel, ni du matériel ni d'une discipline déjà bien établie) capables d'appréhender et d'adhérer à cette nouvelle manière de concevoir.

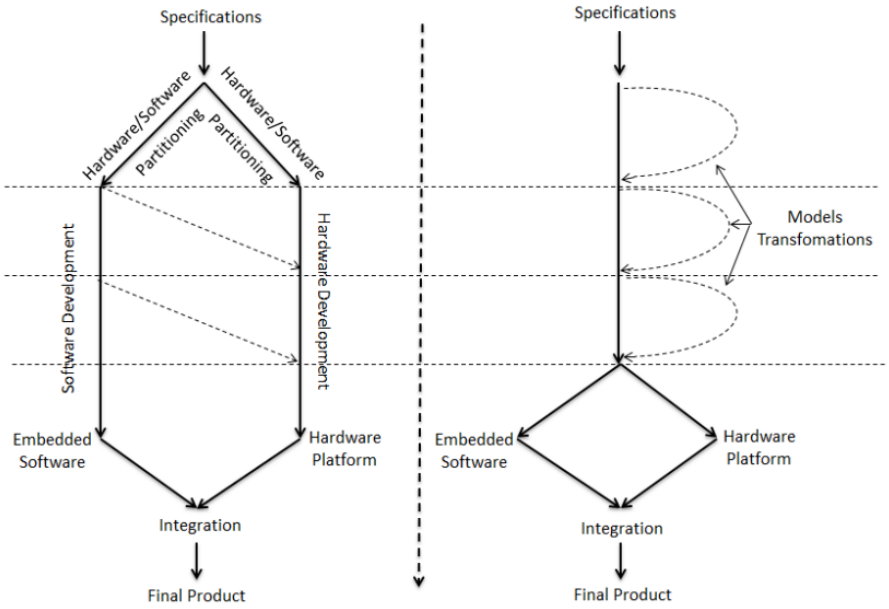


FIGURE 3.11 – Approche de conception HW/SW traditionnelle (à gauche) versus approche par transformation de modèles (à droite), telle que défendue dans le projet ANR Mopcom [B3].

Un flot complet, illustrant ces phases amont *System-level*, ainsi que l'enchaînement avec des aspects plus classiques où logiciel embarqué et matériel sont clairement identifiés, est présenté sur la figure 3.12. Il est inspiré d'une présentation commerciale de l'outil Cierto VCC de Cadence, qui cherchait à s'imposer dans les années 2000.

3.4.2 Modélisation comportementale au niveau système

L'idée de fonder la conception des SoC sur des transformations de modèles parait naturelle. Elle permet d'éviter des couplages précoces entre comportements et structures. Mais comment spécifier ces fameux comportements "amont" ? Un grand nombre de travaux ont tenté de proposer différents langages et paradigmes associés à leur capture.

Modélisation par automates d'états finis Afin de dépasser les barrières culturelles puissantes entre spécialistes du logiciel et spécialistes du matériel, il est nécessaire de réfléchir à des représentations admises par les deux communautés de développeurs. Les *automates* constituent en cela des représentations intermédiaires intéressantes de ce point de vue : ces automates autorisent des descriptions de systèmes, sans tropisme logiciel ou matériel prononcé. La figure 3.14 illustre notre propos. Couramment codés, autant par les spécialistes du logiciel ou du matériel, les formalismes à base d'automates constituent ainsi un terrain *neutre*, où l'ensemble des deux communautés peut s'exprimer de manière naturelle. Des extensions au automates classiques ont en outre été proposés, afin d'en étendre l'expressivité. On peut citer les *Statecharts* de David Harel [M31], qui étendent l'expressivité des automates classiques par l'adjonction d'états hiérarchiques et d'états concurrents (*HCFSM*). Dans le premier cas cela signifie que la description d'un état peut conduire à y décrire un sous-système à base d'un nouvel automate, alors que la concurrence permet de juxtaposer deux automates fonctionnant en parallèle (voir illustration 3.13).

Langages synchrones Les langages synchrones (Esterel, Lustre et Signal) [M4] continuent d'influencer considérablement la modélisation amont des systèmes embarqués. Plus expressifs que des automates, leur compilation vise ainsi pour l'essentiel à exhiber des automates sous-jacents souvent fort complexes, et ce de manière efficace (le schéma 3.15, que nous allons discuter, illustre ce propos). On peut d'ailleurs

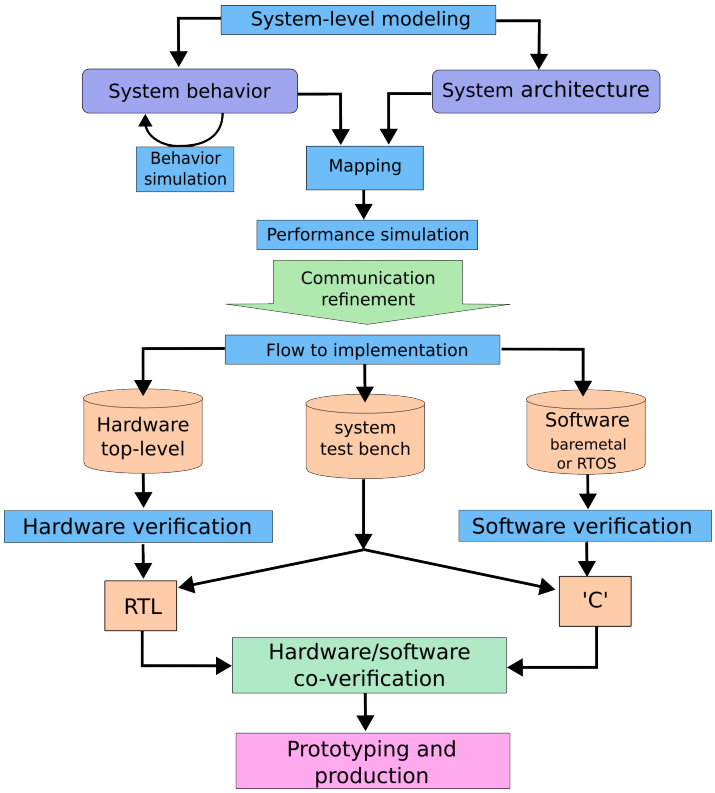


FIGURE 3.12 – Flot de conception : connexion entre les aspects *System-level* amont, et le flot aval mixte logiciel-matériel.

souligner que des travaux proposent très légitimement de profiter des deux mondes évoqués jusqu'ici : l'approche synchrone et les automates semblent hautement compatibles. Les travaux de Florence Maraninchi autour d'`Argos` [M45] sont emblématiques de ce pont construit entre les deux communautés. Le succès des langages synchrones tient à leur focalisation sur la recherche d'une représentation pertinente du *temps* : central dans les préoccupations des ingénieurs réalisant des systèmes dits *temps-réel*, ils en ont paradoxalement trouvé une représentation abstraite ! Il s'agit d'un temps discret et logique, manipulé par des compilateurs, et qui se focalise sur la *chronologie* et la *causalité* des *événements* dans les systèmes, plutôt que sur le temps *chronométrique*, auquel on a pourtant recours lors de la mesure habituelle de leurs performances. Ce changement de représentation, qui reste intuitif, est riche de conséquences : il devient possible de garantir que les événements suivent un ordre causal bien défini, indépendamment des fluctuations du temps physique (latences, délais matériels, etc.) et de garantir un déterminisme fondamental pour ces systèmes. Travailler dans un cadre où le temps est vu comme une séquence d'instants discrets (ticks) permet de se concentrer sur leur logique fonctionnelle, sans se soucier des détails liés à l'exécution physique. Il s'agit d'ailleurs de la même hypothèse que celle sur laquelle s'appuie la conception des circuits numériques, cadencés par une horloge : l'échantillonnage des données par des bascules "absorbe" les fluctuations des chemins combinatoires que suivent les données. Le modèle logique fournit ainsi une base idéale pour appliquer des techniques de vérification formelle (comme la model checking ou les preuves de propriétés) ou raisonner sur des transformations.

Esterel [M5] est probablement le plus connu des langages synchrones. Alors que Lustre et Signal suivent une approche déclarative (orientée flot-de-données), Esterel reste plus proche des langages impératifs, ce qui explique son adoption dans des contextes où une logique de contrôle séquentielle est requise. Son pouvoir de modélisation est très intéressant : à titre d'illustration, on peut observer la figure 3.15 (adaptée de [M57]). Le programme (appelé `ABRO` et souvent cité) vise à attendre les deux signaux `A` et `B`. Lorsque le dernier de ces signaux est reçu, le programme émet un signal `O` et recommence le processus à chaque arrivée d'un troisième signal `R`[4]. La figure illustre l'automate d'états finis correspondant à ce comportement. On constate une complexité importante de cette FSM à 4 états. En général, cette complexité en nombre d'états croît exponentiellement en fonction du nombre de lignes de code

4. L'arrivée de R est prioritaire sur celle de `A` et `B`

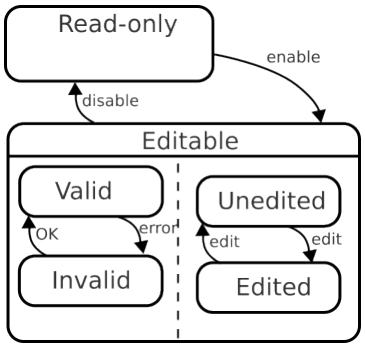


FIGURE 3.13 – Notion de *Statecharts* ou HCFSMs : machines d'états finis présentant des états hiérarchiques et concurrents.

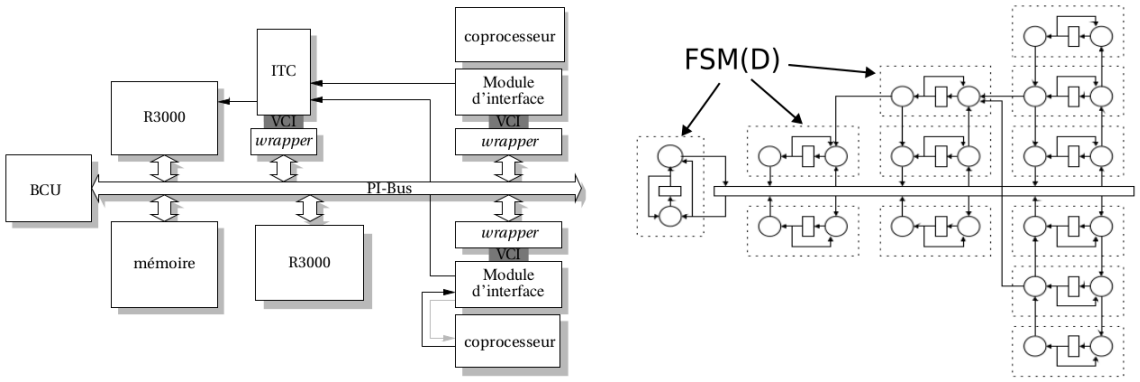


FIGURE 3.14 – Illustration : une architecture SoC peut s'abstraire sous forme d'une composition d'automates (d'après la thèse de Denis Hommais (LIP6) [M35]).

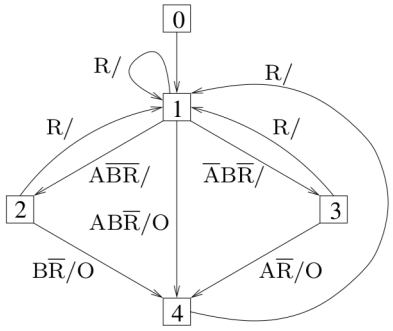
`Esterel`, ce qui dénote une expressivité remarquable. Toujours à titre d'exemple, le code `C` équivalent à `ABRO` est également présenté ; sa complexité contraste à nouveau fortement avec la simplicité du code `Esterel`.

Très différent d'Esterel, et partageant pourtant la même idée de synchronisme parfait, **Signal** [5] et l'environnement **Polychrony** [R5] reposent sur une approche par flots de données et multi-horloges, avec un fort ancrage dans la *programmation par contraintes équationnelles*. Tout comme dans le cas de `Lustre`, les flots représentent des séquences de valeurs associées à des instants de temps logique, chaque flot étant défini sur une horloge, qui lui est propre, et qui spécifie les instants où il est actif (ou "*présent*"). Signal permet d'exprimer des *relations entre ces flots* à travers des équations logiques ou mathématiques, formalisant ainsi des contraintes (dites *contraintes d'horloges*) qui capturent la causalité et les synchronisations entre événements. Les horloges utilisées pouvant ne pas être liées entre elles, les modèles polychrones ne sont pas basés sur un modèle linéaire du temps. Par conséquent, les réactions d'un système polychrone sont seulement partiellement ordonnées. Deux instants ne peuvent être comparés dans l'échelle temporelle que si tous deux contiennent des événements d'un signal partagé. Sa gestion du multi-horloge le rend particulièrement adapté à la modélisation de systèmes intrinsèquement hétérogènes où différentes parties fonctionnent à des rythmes distincts, en permettant de définir des relations explicites entre horloges (comme la restriction ou la fusion). L'environnement de modélisation génère un code réputé fiable et efficace, essentiel pour les systèmes critiques, comme en aéronautique ou en automobile, où les comportements doivent être garantis.

Ce modèle abstrait facilite la *modélisation modulaire* des systèmes complexes [M25], et leur vérification formelle pour détecter des incohérences (instrinsèques ou accidentelles) : cette modélisation modulaire revient strictement à juxtaposer des systèmes d'équations d'horloges inférées par les déclarations du langages (ses "instructions"), et à *résoudre* ce système. Selon le succès ou d'échec de cette résolution, le compilateur peut accepter ou refuser de générer du code pour cette modélisation. Il arrive fréquemment

5. Pour rappel, ma thèse portait sur la simulation et la synthèse de circuits s'appuyant sur `Signal` [T1]

```
module ABRO:
input A,B,R;
output O;
loop
    [await A || await B];
    emit O
each R
end module
```




```
switch(state){
case 0: state=1;
        break;
case 1: if (!R) if (A) if (B) {O=1; state=4;}
                       else state=2;
                else if (B) state=3;
                break;
case 2: if (R) state=1;
        else if (B) {O=1; state=4;}
        break;
case 3: if (R) state=1;
        else if (A) {O=1; state=4;}
        break;
case 4: if(R)state=1;
}
```

FIGURE 3.15 – Exemple du programme `ABRO` écrit en Esterel (à gauche), illustrant sa très grande expressivité. Deux représentations équivalentes sont présentées à droite : un automate d'états finis et un code `C`.

que la modélisation naturelle d'un sous-système, traitée isolément, conduise à un rejet par le compilateur, mais que la composition avec le reste du système permette la résolution du système global.

`CCSL`, initialement introduit comme DSL support du profile `MARTE` d'UML, suit une approche très similaire [M74], et explicite de nouvelles typologies de relations entre événements[M44]. Dans les deux cas, cette approche *déclarative*, *flot de données* et *équationnelle* est à la fois pleine de sens, ambitieuse, mais elle se révèle également déroutante pour l'ingénieur plus accoûtumé à la programmation classique, au style impératif et séquentiel [6].

Modèle Synchronous Data Flow (SDF) de Lee et Messerschmidt Malheureusement, les automates semblent ne pas répondre à tous les besoins en matière de modélisation amont. Le traitement du signal, en particulier, nécessite une focalisation sur les *flot de données*. Le modèle Synchronous Data Flow (SDF) de Lee et Messerschmidt [M42] est à ce titre remarquable à plusieurs titres. Introduit en 1987, ce modèle permet de décrire des applications de traitement du signal s'exécutant *in fine* de manière concurrente sur un matériel parallèle. Un modèle SDF se présente sous la forme d'un graphe orienté dont les noeuds représentent des fonctions connues et les arcs des transferts de données. Ces noeuds sont dénommés *acteurs*. Contrairement à des descriptions similaires invitant à une exécution asynchrone, les extrêmités des arcs du graphe sont annotées par une indication de *rythme de production* et de *rythme de consommation des données*. A l'aide de ces indications, il est possible de réaliser un calcul permettant de conclure si l'application ainsi décrite est exécutable *sans accumulation de données sur les arcs* et donc *sans débordement de la mémoire*. En outre, le calcul permet de déduire aisément un code incarnant l'ordonnancement statique trouvé, soulageant ainsi l'implémentation d'une gestion dynamique du flot de données, souvent coûteuse.

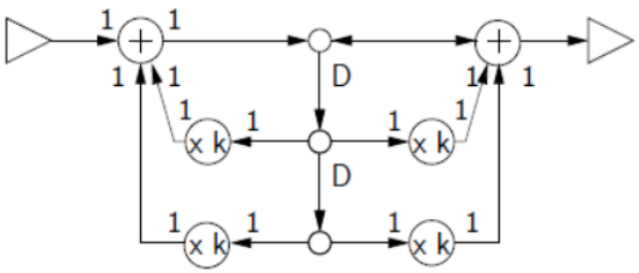


FIGURE 3.16 – Filtre à réponse impulsionnelle infinie, modélisé comme un diagramme SDF : les arcs sont annotés par un rythme de production et de consommation [M42]

Formellement, on note $n_{k,m}$ le nombre de valeurs écrites/lues par un acteur k sur l'arc m. En prenant comme convention que $n_{k,m}$ est positif quand il s'agit d'écrire sur un arc et négatif quand il s'agit de lire, on peut décrire les consommations et productions du système complet comme un ensemble d'équations

6. De l'*ordre total* on est passé à des spécifications plus flexibles fondées sur des *ordres partiels*

diophantiennes (polynomiales, dont on cherche des solutions en nombre entier) dite d'*équilibre* (très similaire aux *équations des noeuds* de Kirchoff) :

$$\forall m, \sum_{k=1}^{K} q_k n_{k,m} = 0$$

Le vecteur $\vec{q} = (q_1, \ldots, q_K)$ est appelé *vecteur de répétition*.

A l'instar des langages synchrones, la compilation d'un modèle SDF s'accompagne ainsi d'un *modèle de calcul* (MoC) rigoureux permettant à la fois de déterminer si la modélisation a un sens, et de générer un code efficace de manière automatique.

Extensions du modèle SDF Le modèle de calcul SDF a clairement ouvert la voie à de nouvelles manières de modéliser et concevoir des systèmes [M54]. Malgré ces qualités, il reste beaucoup à faire dans le domaine. Le modèle SDF souffre de plusieurs limitations qui restreignent son applicabilité. Sa contrainte de taux de consommation et de production fixes, bien qu'utile pour l'analyse statique, limite son expressivité pour modéliser des systèmes dynamiques ou adaptatifs, tels que les codecs vidéo ou les algorithmes dépendant des données. L'absence de prise en charge native des comportements conditionnels ou des variations dynamiques a conduit au développement de modèles tels que le Boolean Dataflow (BDF), qui introduit des dépendances conditionnelles basées sur les données. Cependant, ces extensions augmentent considérablement la complexité de l'analyse statique. Enfin, bien qu'efficace dans un environnement synchrone, il montre des limites pour intégrer des scénarios où la synchronisation stricte n'est pas garantie.

Vers une approche *Dataflow* pure L'idée de se départir de l'exécution séquentielle, imposée par le modèle de Von Neumann, et qui irrigue toute l'Informatique et le coeur des Systèmes Embarqués, n'est pas récente. L'approche *Dataflow*, que nous qualifierons ici de *pure*, continue de se poser en alternative de choix. Dans ce modèle d'exécution intrinsèquement parallèle, seule la disponibilité des données en amont d'un *acteur* déclenche un calcul de sa part, qui produit de nouvelles données en aval : cette exécution est naturelle, intuitive et offre une grande souplesse en terme d'échange de données entre acteurs.

RVC-Cal Parmi les langages et outils remarquables, on peut noter le langage `Cal`, issu de Berkeley, qui a connu une attention soutenue dans le domaine de la compression vidéo : en effet, le consortium MPEG l'a proposé comme un moyen de spécifier de futurs standards de compression vidéo, à travers le langage RVC-CAL. Une norme ISO/IEC 23001-4 :2017 en explique l'utilisation. Le laboratoire IETR de Rennes a été particulièrement actif[M52], en proposant très tôt des outils de génération de code logiciels autour de Cal [M70], [M48].

Approches MDE/UML des systèmes embarqués Les approches de développement dirigé par les modèles (Model-Driven Engineering, MDE) semblent évidemment adaptées à notre propos quant aux besoins en terme de modélisation amont des applications. La capacité du MDE à *éliciter* les modèles, puis à les manipuler de manière consistante, constitue un point de départ idéal. Plusieurs travaux ont proposé d'adapter cette approche, initialement orientée logicielle, à la conception des SoC. On peut notamment citer les travaux de Dekeyser et son équipe à Lille [M20], [M26]. Leur but est d'abstraire la complexité des SoC, en facilitant la conception de systèmes hétérogènes, envisagées à plusieurs niveaux (ou couches) d'abstraction. En MDE, des modèles de haut niveau décrivent les fonctionnalités, l'architecture et les contraintes du système, ce qui permet de simuler, valider et optimiser précocement le design. Ces modèles servent également de base pour la génération automatique de code, réduisant ainsi les erreurs humaines et accélérant le développement. Dans le cas des SoC, le MDE favorise l'intégration entre le matériel (IP cores, interconnexions) et le logiciel (firmware, drivers), en s'appuyant sur des métamodèles standardisés comme UML/MARTE ou SysML pour capturer à la fois les aspects fonctionnels et non fonctionnels (performance, consommation énergétique). Cette approche semble en particulier prometteuse lorsqu'il s'agit d'améliorer la collaboration interdisciplinaire, la traçabilité des exigences et la réutilisation des composants dans des projets complexes et soumis à des contraintes strictes de temps et de coûts.

Plusieurs spécialisations (appelés *profils*) de diagrammes UML ont été proposés. L'un des plus connus est effectivement le profil MARTE (Modeling and Analysis of Real-Time and Embedded Systems), standardisé par l'OMG (Object Management Group). Ce profil enrichit UML avec des stéréotypes, des annotations et des concepts dédiés aux systèmes temps réel et embarqués, incluant les SoC. Le profil MARTE permet de modéliser les aspects fonctionnels (comportements des composants), architecturaux (topologie matérielle et logicielle), ainsi que les propriétés non fonctionnelles (comme la consommation énergétique, les délais ou la latence). Par exemple :

— Il fournit des stéréotypes comme "*Resource*" pour représenter les éléments matériels d'un SoC (processeurs, mémoires, interconnexions).
— Il inclut des concepts pour modéliser la communication entre composants, comme les bus ou les réseaux sur puce (NoC).
— Il permet l'annotation de contraintes temporelles ou énergétiques directement dans les modèles, facilitant ainsi l'analyse précoce des performances via des outils tiers.

Avec MARTE, il est possible d'associer une vue logicielle (modèles de tâches, exécution concurrente) avec une vue matérielle (allocation des tâches sur les ressources matérielles du SoC). Cela favorise une co-conception matériel/logiciel en simulant l'impact de chaque choix sur les performances globales. Un survey récent [M53] permet de se faire une idée des ambitions de MARTE.

Nous avons expérimenté ces approches dans au sein du projet ANR Mopcom [B3] [C11]. Malgré sa standardisation par l'OMG, l'adoption de MARTE dans l'industrie reste relativement faible. Les entreprises peuvent être réticentes à investir dans une technologie perçue comme complexe et peu soutenue par des outils commerciaux. Nous n'avons pas donné suite à ces travaux, pourtant initialement supportés par une division de Thales. Force est de constater que les domaines des SoC, de l'EDA et de l'ESL restent très conservateurs et requièrent des niveaux de maturité importants, avant qu'un outil ou une méthodologie connaisse une éventuelle adoption.

3.4.3 Outils d'estimation de performances

Le succès d'un SoC tient à la justesse des décisions prises en amont et son adéquation algorithme-architecture. Cette prise de décision doit s'appuyer sur des mesures précises de débit, latences, occupation mémoire, puissance, etc. Elle reste difficile pour de multiples raisons :

— Difficulté à appréhender un comportement d'ensemble.
— Présence de comportements dépendants des valeurs de données.
— Difficulté à simuler le système avec des jeux d'entrées réalistes.
— Difficulté de mise en oeuvre de simulations suffisamment rapides pour l'exploration.
— etc.

Exemple d'outils industriels Parmi les outils industriels utilisés dans cette phase d'estimation de performances, on retrouve notamment l'outil `Cofluent` d'Intel et l'outil `VisualSim` de la société Mirabilis Design. Ils couvrent un ensemble d'activité allant de la modélisation, jusqu'à cette estimation de performances et ceci *à des stades précoces* de la conception. Le premier repose sur UML et Sysml lors de la capture applicative, et le second possède son propre environnement graphique. Cette estimation de performances, alors que l'ensemble des éléments (processeurs, mémoires, etc) ne sont pas encore parfaitement connus, permet de déduire des grands comportements d'ensemble du système, où des *propriétés émergentes* peuvent être étudiées. Au niveau purement applicatif cette fois, `Matlab`, `Simulink` et `Scade`[7] jouent également ce rôle. On peut souligner que la technologie `Cofluent` est issue de a methode `MCSE` due à Jean-Paul Calvez [M13] à l'Université de Nantes. La méthode MCSE offre un cadre méthodologique d'exploration remarquable : elle insiste en particulier sur la description *fonctionnelle* des systèmes électroniques, leur annotation temporelle externe, et la notion de *mapping*.

Apport de SystemC et de la modélisation transactionnelle Comme indiqué précédemment, l'approche MDE a un temps focalisé l'attention en matière de conception amont "globale", mais la mise en pratique de la démarche, souvent très laborieuse, semble avoir rebuté la communauté. La communauté s'est repliée sur des techniques plus simples, "orientées langage". `SystemC` notamment reste le langage de choix pour ces mesures d'estimation de performances : fer de lance de la modélisation transactionnelle (TLM, [M27]), `SystemC`, issu d'une communauté électronicienne, permet de rapidement

7. Scade est dérivé de Lustre, langage synchrone déjà évoqué ici.

créer des scenario de partitionnement d'architectures, et leur simulation [M41]. Au niveau TLM, les interactions entre les composants sont modélisées en termes de transactions de haut niveau, telles que des transferts de données ou des requêtes, plutôt que des signaux bas niveau ou des détails spécifiques à l'implémentation matérielle. En particulier, on s'autorise à perdre la notion de cycle d'horloge et le contrôle fin qu'il autorise aux interfaces des composants. Concrètement, ces nouveaux types de transactions sont réalisés à travers des appels de méthodes, plutôt que sur des "fils". L'objectif est de faciliter

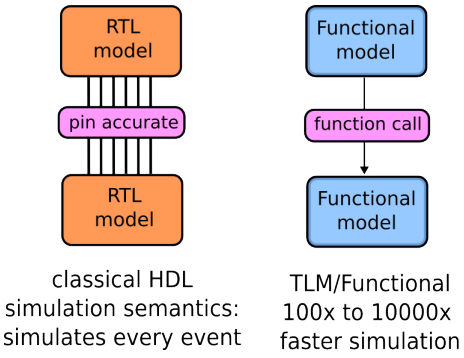


FIGURE 3.17 – La modélisation au niveau TLM ou *fonctionnelle* permet d'accélérer la simulation des SoC.

la simulation rapide et le développement en amont de l'architecture, en se concentrant sur les fonctionnalités et les performances globales du système plutôt que sur les minuties du matériel. L'existence de synthétiseurs comportementaux acceptant des codes `SystemC` permet d'envisager une approche descendante crédible vers le RTL et donc le silicium. Des transacteurs, permettant le passage d'un niveau d'abstraction à l'autre ont été proposés dans la littérature [M29](y compris dans le sens RTL-TLM [M7]).

gem5 Enfin, on ne peut évoquer l'estimation de performances sans parler de `gem5` [M6]. Résultat d'une collaboration entre plusieurs universités et centre de recherche indusriels, `gem5` est un simulateur (open source) d'architectures de processeurs très utilisé dans la recherche en architecture. Ecrit principalement en `C++` et `Python`, il permet de modéliser avec précision le comportement de SoC à l'étude. Il prend en charge plusieurs jeux d'instructions (comme x86, ARM et RISC-V) et permet de simuler différents niveaux d'abstraction, allant de l'exécution d'instructions détaillées (cycle-accurate) à une modélisation plus rapide au niveau fonctionnel. Il offre ainsi un moyen alternatif à la simulation RTL. Il est particulièrement utile pour cette activité d'évaluation des performances et l'exploration de nouvelles architectures. Dans [M59], les auteurs ont démontré sa pertinence dans le cas de simulations incluant des accélérateurs dédiés, modélisés à l'aide de LLVM ("pre-RTL").

Des outils plus formels, issus de la théorie de l'ordonnancement [M63], peuvent être évoquées [M32], [M64] concernant l'évaluation de performances des SoC, mais ils n'offrent (à notre connaissance) pas un tel chemin vers le silicium.

3.5 Conclusion

Ce chapitre introductif rappelle la structure des SoC, et illustre quelques unes des difficultés qui entâchent leur conception : outre leurs dimensions, leur hétérogénéité structurelle apparait comme un frein puissant. Le recours à des modèles comportementaux *amont*, et des transformations adéquates, promettent de simplifier cette conception. L'utilisation d'outils d'analyse permet d'étayer les choix effectués lors de *raffinements successifs*. Cette approche itérative semble fondamentale pour la conduite de projet, mais continue d'être relativement peu répandue. Elle sous-entend une maîtrise d'abstractions diverses qui ne correspondent pas aux différents rôles traditionnellement associés à la conception des SoC : il ne s'agit ni d'algorithmique, ni de logiciel embarqué, ni de matériel, etc. Comme souligné par Alberto Sangiovanni-Vincentelli dans son article de 2007 "Quo Vadis, SLD?" [M60], il y a nécessité d'élaborer une *"nouvelle science"*, *"mariant le physique et l'abstrait"*.

Dans la suite du document, plusieurs aspects de la création de SoCs sont abordés. Leur dénominateur commun reste probablement l'obtention systématique de tels modèles informatiques et d'outils les manipulant, permettant le prototypage et l'expérimentation. Cette *capture* initiale est fondamentale, quel

que soit le problème traité, et doit pouvoir s'inscrire dans une démarche systématique : il s'agit tout autant d'une question d'appropriation du problème que de sa résolution ultérieure effective. Cette démarche parait pleinement justifiée en particulier sur les phases de modélisation amont des applications, mais s'applique également dans un cadre plus architectural.

Deuxième partie

Contributions à la capture applicative

Chapitre 4

Genie logiciel pour le prototypage d'outils EDA/ESL :

Model-driven engineering & métaprogrammation

> Programming is the art of creating your own DSL.
>
> Dave Thomas
> auteur de *The Pragmatic Programmer*

Sommaire

4.1 Introduction

Les travaux présentés dans le présent manuscrit reposent sur la conception de plusieurs d'outils (compilateurs, simulateurs, etc) que j'ai pu développer, seul ou en équipe. Ce chapitre recense et illustre des techniques logicielles auxquelles j'ai fréquemment recours dans l'élaboration de ces outils EDA/ESL. A priori voués à une annexe secondaire, j'ai décidé de les présenter en tête de cette partie liée à la "capture applicative" : le choix de ces techniques a en réalité été déterminant dans le succès des expériences menées par le passé, et présentées ici. Il était donc légitime de les replacer tôt dans ce mémoire. Certaines de ces techniques se révèlent relativement standard, alors que d'autres le sont moins. Elles ont, dans tous les cas, pu faire leur preuve à de nombreuses reprises, dans différentes situations : de l'expérimentation rapide, jusqu'à l'élaboration de logiciels suffisamment aboutis pour être maintenus sur un temps long.

Ces techniques se rapportent au domaine du génie logiciel, et plus précisément à l'Ingénierie dirigée par les modèles (*Model-driven software engineering*). Je me suis notamment efforcé de dégager des techniques MDE s'appuyant sur la *métaprogrammation*, qui permettent de créer des DSLs *internes* de manière *agile*. Après quelques rappels de définitions classiques autour de l'orienté-objet, ces techniques plus exotiques sont illustrées sur quelques cas illustratifs.

4.2 Démocratisation des DSLs pour la conception des SoC

Il est nécessaire de souligner qu'*industriellement*, les sociétés qui concoivent des SoC (ASIC ou FPGA) reposent sur des outils fournis par les sociétés de l'EDA : simulateurs, compilateurs et outils de synthèse (logique, RTL et comportementale). Quelques sociétés peuvent être amenées à développer en interne des outils et méthodologies amont, afin de réaliser de premiers modèles fonctionnels, ainsi que des analyses de performances. Mais bon nombre d'entre elles reposent là-aussi sur des langages développés en externe (Matlab par exemple) et des outils tout aussi fermés.

Cette ligne de partage des activités, nette et souvent stricte, est ici, selon nous, un frein à l'innovation et plus généralement à la conception des SoCs modernes. Grâce aux progrès importants du *Génie logiciel*, il est désormais à la portée de ces entreprises et centres de recherche de créer plus aisément de nouveaux outils. Il devient ainsi possible de composer de nouveaux flot de conception "à façon".

Plusieurs auteurs ont récemment mis l'accent sur ce renouveau en matière de conception. Il s'agit de *démocratiser* la conception en général. Ainsi dans [M14], Jason Cong [1] et son équipe insistent sur cette possibilité désormais offerte pour ces entreprises de "créer un environnement de programmation et un flot de compilation qui permettent aux programmeurs de créer leurs propres architectures spécialisées de manière efficace et économique sur FPGA". Le mouvement open-source participe bien évidemment également à cette démocratisation. L'avénement de l'ISA RISCV est emblématique de ce mouvement de fond important [M62]. Enfin, l'engouement et l'essor fantastique de *Litex*, développé en open-source par Enjoy-Digital [2] autour de Python est un nouvel indicateur de cette ouverture qui est à l'oeuvre dans le domaine des SoC.

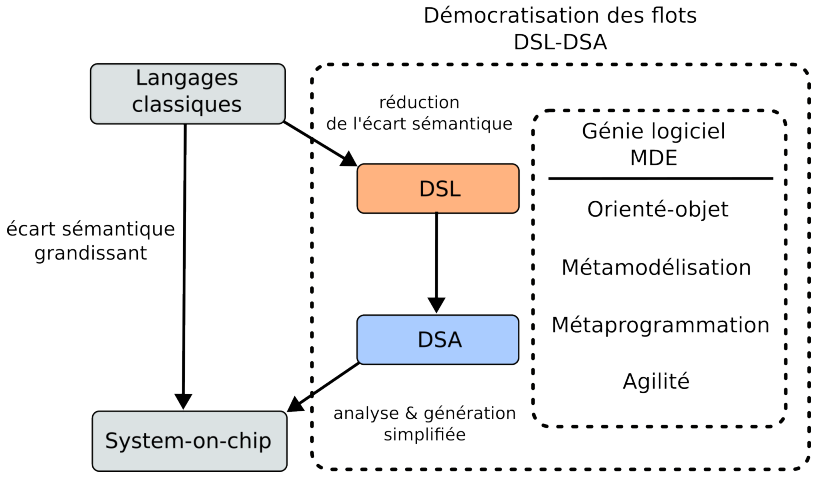


FIGURE 4.1 – L'idée de démocratiser la conception de langages et outils EDA/ESL permet de réduire les écarts sémantiques entre les objets à étudier (SoC, composants) et les langages qui les capturent.

4.3 Approche *Model-driven engineering* (MDE)

Définition L'approche MDE de construction de logiciels place, comme son nom l'indique, la notion de *modèle* au centre de l'activité de développement [M16]. Ces modèles sont une vue épurée d'une réalité

1. Créateur de ce qui est devenu VivadoHLS
2. `htttp://www.enjoy-digital.fr`

plus complexe, et permettent aux développeurs un premier partage commun : celui d'un vocabulaire reserré et non-ambigü, lié au domaine d'étude. Cette démarche, initialement *ontologique*, permet par la même occasion de se dégager des contingences techniques liées à l'implémentation finale dans tel ou tel langage de programmation, qui arrive avec ses propres contraintes et particularismes, et qui a tout lieu d'entraver un développeur dans sa démarche de modélisation. Idéalement, cette démarche MDE permet de découpler les phases amont d'appropriation et d'analyse du modèle de celles d'implementation et d'exécution : il est ainsi possible a priori d'élaborer des modèles logiciels, puis d'en dériver des implémentations dans plusieurs langages cibles, avec un effort minimal. Cette démarche n'est possible que par l'existence préalable de générateurs de code appropriés, ou d'un méta-outillage permettant leur création.

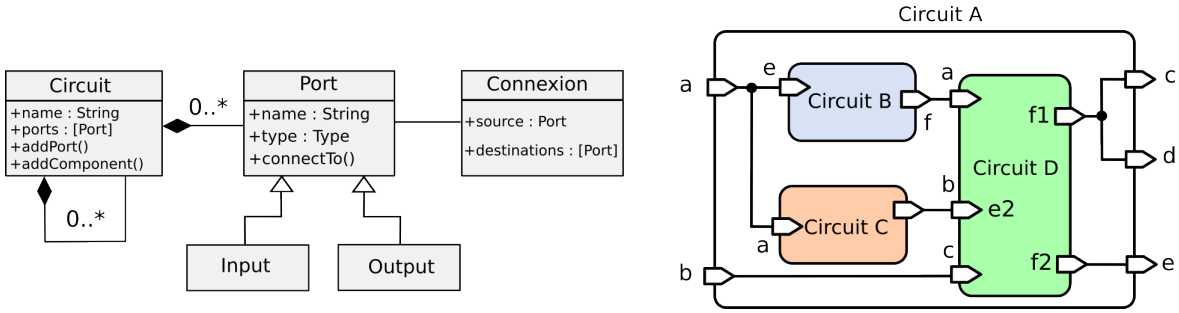


FIGURE 4.2 – Métamodèle d'un circuit hiérarchique (à gauche) et exemple de modèle instancié.

Apport de l'orienté-objet L'apport des langages orientés-objets a été déterminant dans l'avénement de la démarche MDE . L'approche OO (née avec Smalltalk [M40]) introduit le concept de classe comme une abstraction qui peut être instanciée en objets concrets. Cette classe agit comme un moule (c'est d'ailleurs elle-même un objet d'un type spécial !) qui permet aux objets d'hériter de propriétés (attributs) et de comportements (méthodes) hérités de ladite classe. De manière similaire, en MDE, un modèle peut être vu comme une instance d'un métamodèle, respectant à la fois son vocabulaire et ses contraintes éventuelles. L'OO et MDE mettent ainsi toutes deux l'accent sur la modélisation, la réutilisabilité, et la séparation des préoccupations. Enfin, on peut noter que l'*encapsulation*, qui est un pilier de l'OO, consiste à cacher les détails d'implémentation derrière des interfaces publiques. Cela crée une séparation nette entre l'interface d'un objet (ce qu'il fait) et son implémentation (comment il le fait). Enfin, lorsqu'il existe des opportunités de factorisation de comportements, la notion d'héritage (et de mixin) autorise une telle structuration du modèle. Cette manière de penser en termes d'objets a posé les bases d'une méthode descriptive modulaire et abstraite très adaptée à la modélisation d'un domaine spécifique.

MDE selon l'approche UML/OCL/EMF Il existe plusieurs manières de pratiquer une ingénierie MDE. L'un des standards les plus utilisés est UML (Unified Modeling Language), permettant de modéliser des structures, des comportements et des interactions dans un système. Pour garantir la cohérence et la précision des modèles, OCL (Object Constraint Language) est fréquemment utilisé afin spécifier des contraintes et des règles logiques sur les modèles UML (et autres). Outre UML et OCL, un cadre essentiel dans l'écosystème MDE est EMF (Eclipse Modeling Framework), qui a joué un rôle central dans la définition, la gestion et la manipulation de modèles basés sur les méta-modèles. À partir d'un méta-modèle défini (généralement en Ecore), EMF génère automatiquement le code Java pour manipuler les instances du modèle. EMF facilite également l'intégration avec d'autres outils de modélisation, permettant une interopérabilité accrue. EMF peut être utilisé pour créer des DSL à partir de méta-modèles Ecore. Une syntaxe textuelle ou graphique peut accompagner ces modèles, ainsi qu'une édition réflexive. Des outils comme ATL (ATLAS Transformation Language) ou QVT (Query/View/Transformation) permettent de transformer des modèles EMF vers d'autres formats ou langages, facilitant ainsi la génération de code ou la migration de modèles. L'un des objectifs du MDE est de générer automatiquement du code source à partir des modèles, en utilisant des outils comme Acceleo ou Xpand. Enfin, afin s'assurer que les modèles respectent les spécifications et les contraintes définies, des outils de simulation (comme Papyrus ou MagicDraw) et de vérification (comme Alloy) sont utilisés.

4.3.1 DSL : domain-specific languages

MDE et DSL La création de langages informatiques a longtemps été l'apanage de quelques spécialistes des compilateurs. L'avénement relativement récente du MDE a permis de décloisonner et démocratiser effectivement la création de tels outils. Ils n'ont toutefois pas tous vocation à générer du code binaire optimisé, comme le font des compilateurs classiques : leur bénéfice se situe plus en amont, lors de la *capture de code métier* : on parle ainsi de DSL, qui permettent d'exprimer des concepts propres à un domaine d'activité restreint.

Syntaxe abstraite Ces concepts nécessitent avant tout la définition informatique d'une *syntaxe abstraite* : il s'agit de la définition d'une structure de données, qui permet la représentation interne des concepts manipulés. On pourra lui associer une syntaxe textuelle ou graphique, voire une simple édition à travers un éditeur réflexif. L'enjeu de définition de cette syntaxe est qu'elle minimise l'écart entre les concepts métiers effectifs, et leur manipulation en interne, dans les outils : en se fondant sur cette syntaxe abstraite, le développement des outils se révèle naturel, maintenable et évolutif. A l'inverse, la programmation classique repose sur la syntaxe du langage hôte pour "parler" de concepts métiers, puis les manipuler : sans cesse, un écart sémantique, introduit précocement, s'imisce dans le développement. Le *nommage* non ambigü des concepts, à travers un système de types, est une base méthodologique nécessaire : les deux approches de programmation fonctionnelle et orientée-objet se prêtent bien à cette pratique.

DSL Internes et Externes Deux voies s'offrent à nous concernant la construction de ces langages :
- Les DSL *externes* : il s'agit de langages dont la syntaxe est définie "from scratch".
- Les DSL *internes* : ces langages sont *hébergés* par un langage préexistant (et son outillage), appelé langage *hôte*. Les concepts du DSL doivent s'intégrer dans une syntaxe déjà existante.

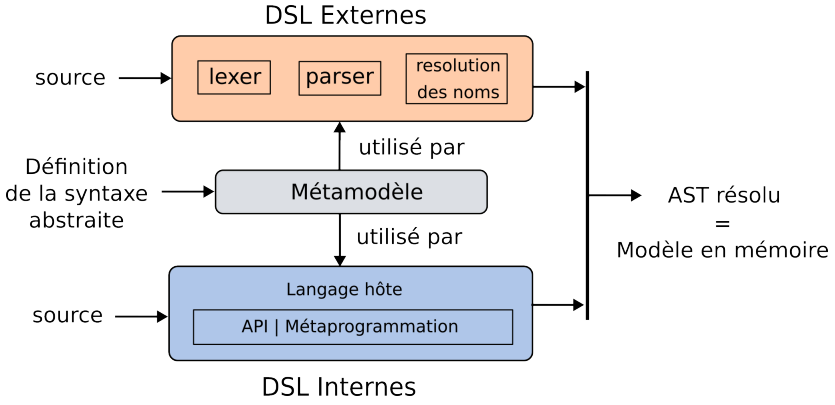


FIGURE 4.3 – Deux voies retenues pour la création de langages : DSL internes et externes partagent une syntaxe abstraite commune, décrite dans un métamodèle.

4.4 Metaprogrammation et DSL internes

4.4.1 Définition

La *Metaprogrammation* consiste à créer des programmes qui manipulent d'autres programmes, voire eux-mêmes en tant que données. Il s'agit ainsi d'une approche MDE au sens strict du terme. Cela inclut :
- la génération de code
- la modification du comportement à l'exécution (réflexion)
- l'ajout de nouvelles fonctionnalités de manière dynamique : à titre d'exemple, en Ruby, on dispose de méthodes comme `define_method` qui, lors de l'exécution du programme, construit une méthode de toute pièce. Des pratiques similaires incluent la création et modification de classes ou modules à la volée.

4.4.2 DSL internes

La création de DSL (Domain-Specific Languages) *internes* fait logiquement partie des pratiques de métaprogrammation. Le DSL est considéré comme *interne* car il est *hébergé par son hôte* (ou encore *embarqué*). Comme pour des DSL *externes*, les DSL *internes* permettent donc de rendre le code plus expressif et adapté à un style de description. Toutefois la grande différence entre les deux approches réside dans la quantité de travail à réaliser pour parvenir aux mêmes effets : dans le cas d'un DSL *externe*, il faudra développer un lexeur, un parseur, un AST, puis réaliser une résolution des noms (lors d'une analyse contextuelle) avant de bénéficier d'un modèle en mémoire enfin exploitable. A l'inverse, dans le cas d'un DSL interne, cet ensemble d'activité de développement est inexistant : le modèle en mémoire est créé par le support du langage hôte. Toutefois tous les langages hôtes ne se valent pas : certains DSL internes ne seront conçus et perçus que comme des APIs pratiques et de haut niveau, mais *in fine* ne reposant uniquement que sur un nommage bien pensé. A l'inverse d'autres langages hôtes comme Lisp, Smalltalk, Groovy, Ruby, C++, etc donneront l'illusion de s'éclipser au profit du langage hébergé : l'utilisateur se retrouve en apparence face à une syntaxe totalement nouvelle, celle du DSL, et parfaitement adaptée à son objectif.

Cas des DSL internes en Ruby Ce genre de pratiques de (méta-)programmation peuvent sembler confiner à de l'acrobatie, ou être suffisamment exotiques pour en rester éloigné. Pourtant, toutes ces pratiques ont été livrées clé en mains par de nombreux langages *mainstream* : dans les paragraphes suivants, nous montrons que ces DSL internes sont remarquablement simples à réaliser en Ruby par exemple. Ma focalisation sur Ruby s'est précisement opérée du fait de ces capacités remarquables en terme de métaprogrammation. Pour rappel, Ruby s'est lui-même fait connaître dans les années 2000 dans le domaine du Web : le framework *Ruby-on-Rails* peut-être présenté comme un ensemble vaste de DSLs internes facilitant la création complète de site web (côte serveur). Un développeur Ruby on Rails utilise ces DSL pour simplifier la configuration et la logique métier. Chaque composant du framework (routes, migrations, modèles, contrôleurs, etc.) est conçu avec des syntaxes expressives qui ressemblent à un langage naturel, permettant de décrire des fonctionnalités complexes avec peu de code. Sans même parler de *Ruby-on-Rails*, c'est toute la pratique de Ruby qui invite à construire et utiliser de tels DSLs internes, fidèle à la citation de David Thomas *"Programming is the art of creating your own DSL."*. L'exemple le plus immédiat pour un débutant est le suivant : la méthode `attr_accessor` crée *dynamiquement* des méthodes d'accesseurs (read et write) aux variables d'instances d'un objet. Il est même fréquent que l'on demande à un objet (par exemple `Struct.new` dans le code ci-dessous) de créer une *classe* (Circuit) de toute pièce, incluant ses accesseurs !

```
class Circuit
  attr_accessor :name,:ports,:components # metaprogramming
end

# even better metaprogramming. Full class with accessors created !
Circuit = Struct.new(:name,:ports,components) # class created
soc = Circuit.new("Encoder",[],[]) # new instance
puts soc.name  # => returns "Encoder". Accessor method name() called without parenthesis.
soc.components << Uart.new
```

Listing 4.1 – Exemples de code Ruby : 1) utilisation d'une méthode de classe qui crée dynamiquement d'autres méthodes : ici l'utilisation de `attr_accessor` évite l'écriture de 6 méthodes distinctes. 2) Second exemple : l'objet `Struct` crée un classe `Circuit` ainsi que ses accesseurs, dont l'utilisation est illustrée.

4.4.3 Un exemple dans le domaine EDA : expressions arithmétiques

Objectifs A titre de première illustration, supposons que nous cherchions à disposer en mémoire d'expressions arithmétiques. Plusieurs cas d'usage sont possibles : création d'un interpréteur, génération de code assembleur, HLS, etc. Il nous faut donc disposer en mémoire de telles expressions arithmétiques, et que nous soyons en mesure de les parcourir à des fins d'analyse, de manipulation ou de génération de code. Une approche standard en compilation serait de développer un parseur complet, aboutissant à un AST en mémoire. Ces expressions sont non-triviales à analyser lexicalement (existence d'identifiants de variables et de litéraux) et syntaxiquement (il faut notamment gérer correctement les priorités des opérations, le parenthésage, etc). Une pratique standard en matière de compilation suggère d'utiliser

un générateur de parseur (comme le couple Flex-Bison en `C/C++` ou ANTRL pour `Java` par exemple) : il serait alors nécessaire de s'approprier cet outil tiers, externe au langage, afin de générer le lexeur et le parseur. A l'aide d'un DSL interne, à l'inverse, ces tâches sont purement et simplement shuntées. La création de l'outil devient triviale, et s'intègre dans le flot de développement du langage hôte, sans détour par des outils tiers.

La vue utilisateur Un moyen simple de donner la possibilité à l'utilisateur de capturer ses expressions est de recourir à la notion de *block closure* : il s'agit d'un bloc de code avec paramètres, qui peut être passé à une méthode, comme un argument banal. Le concepteur d'outil peut donc développer la méthode appelée ici `compile`, qu'il fournit comme porte d'entrée à l'utilisateur. L'utilisateur y aura recours par exemple comme suit :

```
1  compile{|a,b,c,d,e| (a+b)*(c-d)/e}
```

On y retrouve l'appel à la méthode développée spécifiquement, appelée `compile`, à laquelle l'utilisateur est invité à passer un objet de type `Block` : syntaxiquement, le bloc Ruby se présente comme un code encadré par des accolades à l'intérieur desquelles on trouve un passage de paramètres exprimés entre barres verticales, suivies un code Ruby classique. Ici, par exemple, l'appel à `compile` utilise comme paramètre un bloc `{|a,b,c,d,e| (a+b)*(c-d)/e}`. Ce bloc a lui-même quatre arguments `|a,b,c,d,e|` et un corps très simple : en l'occurence une expression, syntaxiquement acceptable en Ruby.

La vue développeur Au moment de l'appel par l'utilisateur, notons que rien n'est indiqué quant aux types de `a,b,c,d,e`. Ainsi, jusqu'ici l'expression `(a+b)*(c-d)/e` n'est pas réellement exécutable : elle ne le deviendrait que si les paramètres du bloc avaient effectivement un type spécifié. Cette solution retenue pour le développeur d'outil est alors astucieuse : il va chercher à intercepter à la fois les symboles passés au bloc, ainsi que l'expression capturée avec la syntaxe Ruby : les premiers pourront être typés comme des variables du DSL, alors que l'expression (initialement Ruby) deviendra un AST du DSL en mémoire, représentant l'expression utilisateur. Seules deux lignes sont nécessaires à l'écriture de cette prouesse !

```
1  def compile &block
2    vars=block.parameters.map{|param| Var.new(param.last)}
3    yield *vars
4  end
```

Nous allons expliquer ces deux lignes. On rappelle bien évidemment que le but n'est ici que d'élaborer un AST en mémoire. On insiste également sur le fait que cet AST n'est pas un AST du langage Ruby, mais celui du DSL interne visé.

Avant de se plonger dans le fonctionnement des deux lignes précédentes, on doit également fournir le code qui définit le métamodèle du DSL métier sous forme d'un ensemble de classes Ruby.

```
1  class Expression
2    def +(other)
3      Binary.new(:+, self, other)
4    end
5    # skipped as similar
6  end
7
8  class Binary < Expression
9    attr_reader :op, :left, :right
10
11   def initialize(op, left, right)
12     @left,@op,@right = left,op,right
13   end
14 end
15
16 class Var < Expression
17   attr_reader :name
18
19   def initialize(name)
20     @name = name
```

```
  end
22 end
```

Explication du fonctionnement Revenons sur le code de la méthode `compile`.

```
def compile &block
2   vars=block.parameters.map{|param| Var.new(param.last)}
  yield *vars
4 end
```

Les deux lignes de la méthode `compile` fonctionnent de la manière suivante.

- La ligne 2 réalise l'interception des paramètres passés au bloc, par *introspection* : il est possible de récupérer le nom des paramètres du bloc spécifié par l'utilisateur, grâce à la méthode d'instance `block.parameters`.
- Toujours sur la ligne 2, on demande ici ensuite à la liste retournée d'effectuer une *application* (methode `block.parameters.map`) sur ses éléments. Ici, cette application consiste à créer autant d'objets de classe `Var` qu'il y a de paramètres au bloc. En une seule ligne, on dispose ainsi d'un tableau `vars` de `variables métier`.
- La ligne 3 est tout aussi impressionnante : la méthode `yield` demande à l'interpréteur Ruby d'exécuter désormais le bloc de code utilisateur (ici l'expression `(a+b)*(c-d)/e`)) ; `yield` reçoit ici un nombre adéquat d'arguments exprimés par `*vars`. Il s'agit donc bien désormais des variables métier de type `Var`. Le résultat de cette seconde et dernière ligne sera également l'objet retourné par la méthode `compile`. Mais quel est ce résultat ? Le `yield` a demandé l'exécution du bloc {`|a,b,c,d,e| (a+b)*(c-d)/e`}, mais désormais chacun des arguments est une variable `Var`.
- L'expression `(a+b)*(c-d)/e` prend alors *seulement* tout son sens. Cherchons à comprendre l'exécution d'une sous-expression (par exemple `a+b`) : l'interpréteur Ruby voit le symbole `+` comme un appel de méthode, qu'on peut reformuler en notation pointée : `a.+(b)`. Or l'objet `a` est de type `Var`. Comme `Var` hérite de `Expression`, c'est la méthode `+` de `Expression` qui est appelée : celle-ci construit effectivement un objet `Binary` en mémoire, dont les deux attributs dont des objets `Var`. De proche en proche, par appels récursifs et mutuels, un AST complet est ainsi construit, et finalement retourné par `compile` : le tour est joué !

Intercession réflexive Ce code utilise la notion d'*intercession* (ou *intercession réflexive*), c'est-à-dire la capacité d'un programme à s'examiner et à se modifier lui-même dynamiquement. C'est un concept clé en métaprogrammation, où le programme peut manipuler sa propre structure ou son comportement à l'exécution. Dans l'exemple précédent, c'est le cas du bloc de code passé en argument, dont les arguments ont été adaptés au DSL visé. Le bloc a été ensuite relancé, avec ces nouveaux arguments. En Ruby, la méthode emblématique de l'intercession réflexive est la possibilité de détecter l'appel à une méthode inexistante : dans ce cas, il est possible de récupérer la *forme* de l'appel à l'aide de l'idiome `def method_missing(m, *args, &block)` afin d'effectuer une *délégation* à un objet approprié (par exemple choisi en fonction du nom de la méthode absente).

4.4.4 Ajouts syntaxiques au langage hôte : ajout de mots clés

Appel de méthode sans parenthésage L'exemple précédent se ramène, peu ou prou, à l'utilisation judicieuse du `Block` Ruby, passé en paramètre d'une méthode. Ce type de programmes est omniprésent en Ruby, mais également dans un grand nombre d'autres langages. Ruby possède toutefois d'autres attraits qui lui sont propres : ainsi, il est possible de donner l'illusion de créer de nouveaux mots clés au langage et donc de l'étendre à volonté. Parmi les facilités syntaxiques qui permettent cet exploit, on trouve l'appel de méthode avec paramètres, qui peut se faire *sans parenthésage*.

Exemples : RubyRTL et RubyESL Le DSL RubyRTL sera présenté en détail dans le chapitre suivant. La méthode simple suivante donne l'illusion de création du mot clé `input` au langage Ruby.

```
def input *arg
  @ast||=Root.new
  process_sig_decl(:input,*arg)
end
```

Cette méthode d'instance permet de passer un nombre variable d'arguments `*arg`. A l'exécution, un AST du DSL est créé, s'il n'existe pas pas déjà. La méthode (non donnée ici) `process_sig_decl` permet de créer et ajouter un nouveau noeud de class `Input` à ce dernier. Des mécansimes similaires permettent à l'utilisateur d'écrire :

```
class And3 < Circuit
  def initialize
    input :a => :bit,  # optional type for single :bit
    input :b,:c        # default is :bit
    output :f
    assign(f <= a & b & c)
  end
end
```

Dans un autre DSL (RubyESL, utilisé comme cible de génération de code de simulation pour l'outil `Archipel`, présenté dans le chapitre 6), le choix a été différent concernant l'adjonction de mots clé : il s'agissait d'ajouter un pseudo-mot clé *à la classe elle-même*. Ainsi, il est possible d'écrire :

```
# random sensor
class Sensor < ESL::Actor
  output :sample
  def behavior
    while true
      send(rand,:sample)
    end
  end
end
```

L'implémentation n'est guère plus difficile :

```
def self.output *symbols
  symbols.each do |name|
    send(:attr_accessor, name)
    #...
  end
end
```

Ici, on a affaire à une déclaration de `méthode de classe` (`self.output`). L'appel de la méthode `send`, native à Ruby, permet de créer (dynamiquement) des méthodes accesseurs pour chacun des noms des entrées décrites lors de l'utilisation du DSL.

4.4.5 Emprunts syntaxiques au langage hôte : domain-specific overloading

En plus de l'ajout de pseudo-mot clé au langage hôte, il est fréquent en métaprogrammation de détourner la syntaxe de ce langage hôte. Cette technique s'appelle *domain-specific overloading*. Dans l'exemple précédent, on a pu remarquer au passage la méthode du DSL `assign` : elle détourne le symbole de comparaison `<=` pour donner à nouveau l'illusion d'une assignation de signal, avec une syntaxe proche de Verilog ou VHDL. Un second exemple est proposé sur la première déclaration d'`input`, qui reçoit en paramètre `:a => :bit` : c'est en réalité un `Hash`, à nouveau utilisé sans sa syntaxe habituelle ({...}), mais qui restitue, du point de vue de l'utilisateur du DSL, l'idée d'association d'une variable et d'un type.

4.5 Limites à l'approche MDE et solutions intermédiaires

4.5.1 Critiques des approches *meta*

L'ensemble de ces techniques MDE de métamodélisation et métaprogrammation accélèrent grandement le *protoypage* de DSL, en évitant des tâches laborieuses de création d'une nouvelle syntaxe *from scratch*. Mais elles ne sont pas exemptes de défauts. D'une part, ces techniques posent la question du haut niveau d'expertise du concepteur du DSL, ce qui peut se révêler problématique dans le cas de la maintenance : si documenter un code "classique" est parfois périlleux, le recours à la métaprogrammation rend la tâche encore plus délicate. Par ailleurs, on peut reprocher à ces techniques un manque évident de *portabilité générale de l'approche* : on retrouve une *adhérence technologique forte*, ici par exemple liée au langage Ruby. Un même reproche revient fréquemment concernant l'approche par métamodélisation classique, avec ces mêmes *adhérences technologiques* notamment au langage Java, utilisé de manière intensive dans le domaine, et au monde `Eclipse` en général.

4.5.2 La solution intermédiaire de scaffolding

Outre l'adhérence technologique, plusieurs reproches sont généralement faits à l'encontre des techniques MDE, les plus fréquents étant le manque de maturité des outils de modélisation (et notamment leur manque de compatibilité) ainsi que leur lourdeur au regard de projets de taille moyenne. Nous ne reviendrons pas sur ces débats, mais suggérons la solution *intermédiaire* qu'est la technique de *scaffolding*. Cette technique (dont le nom signifie "montage d'échaffaudage") fait référence aux techniques de génération automatique de code structurel pour démarrer rapidement un projet ou une fonctionnalité plus locale. Bien que les développeurs qui ont recours au *scaffolding* ne se réfèrent pas forcément à l'IDM/MDE, il s'agit d'une pratique de productivité, belle et bien orientée vers une forme de séparation des préoccupations. Elle permet de s'affranchir de tâches répétitives, bien connues et maîtrisées, *autour* du projet. Le *scaffolding* peut ainsi être vu comme une "IDM à taille humaine", créée de manière adhoc par le développeur lui-même (ou son équipe).

4.5.3 Langage de description de la syntaxe abstraite : ASDL

Parmi les solutions intermédiaires en génie logiciel, on retrouve les ASDL (*Abstract Syntax Description Languages*) : ce sont des langages textuels de description de la syntaxe abstraite [3]. Ils ne permettent pas de définir les grammaires de la syntaxe concrète d'un langage, mais au contraire les structures internes du langage. Ils sont selon moi sous-estimés et sous-utilisés. Bien entendu, les compilateurs de ASDLs génèrent tout ou partie des classes, et plus généralement structures de données permettant la réalisation de l'outil final associé au DSL visé. C'est donc un outil productif et simple, qui possède l'avantage de pouvoir être déconnecté du flot complet : c'est souvent le contraire avec des outils de modélisation et métamodélisation mainstream, plus intrusifs. A titre d'exemple, on peut citer l'ASDL [M67] dû à l'équipe d'Andrew Appel. `Zephyr` permet la génération automatique de structures de données complexes en `C`, `C++`, `Java`, `Standard ML` et `Haskell`. Il est intéressant de souligner que `Zephyr` a été utilisé dans la définition de l'AST de `CPython` (la version officielle de `Python`).

Dans l'exemple de code `Zephyr` ci-dessous, on vient déclarer comment différents objets (const, var, print, etc) sont constitués. Le compilateur peut alors déduire une relation d'héritage entre les classes constitutives et les classes mères `Expr` et `Stmt`.

```
module Lang {
  expr = Const(int value)
       | Var(string name)
       | Add(expr left, expr right)
       | Mul(expr left, expr right)

  stmt = Assign(string name, expr value)
       | Print(expr value)
}
```

Listing 4.2 – Description ASDL en Zephyr

3. Bien que partageant des notions communes, ils ne doivent pas être confondus avec des formalismes comme `ASN.1`, orienté vers la description de la sérialisation des données.

J'ai moi-même développé un tel ASDL, appelé `Astrapi`, qui génère un ensemble de classes à partir d'une description de la syntaxe abstraite du DSL visé. Tout comme `Zephyr`, `Astrapi` fournit une sérialisation et désérialisation de modèles du DSL sous forme de *s-expressions*, ainsi que (par scaffolding) la structure générale d'un compilateur permettant le chargement et déchargement de ces modèles en mémoire.

4.6 Conclusion

Dans ce chapitre, nous avons présenté succintement les ambitions de l'approche du génie logiciel focalisé sur le recours à des modèles : ces modèles sont des instances d'un métamodèle savemment défini en amont, qui fixe le vocabulaire d'un domaine spécialisé. Ces instances sont élaborées en mémoire selon différentes techniques : nous avons insisté et tenté d'illustrer l'approche productive qu'est la Métaprogrammation, utilisée fréquemment dans mes recherches. Le chapitre suivant illustre une telle utilisation à travers le DSL `RubyRTL`.

Chapitre 5

Capture au niveau RTL par HCL : RubyRTL et Litex

Hardware is just software crystallized early

Carver Mead

Publications et projets associés

— [W4] Date/OSDA'19. LiteX : an open-source SoC builder and library based on Migen Python DSL.
— [W3] Date/OSDA'20 : Towards a Hardware DSL Ecosystem : RubyRTL and Friends.

Sommaire

L'acronyme HCL (*Hardware Construction Language*) est apparu il y a quelques années, afin de marquer un changement de paradigme dans la manière de capturer les applications au niveau RTL. Plusieurs HCL ont émergé et se présentent désormais comme des alternatives crédibles à l'usage de VHDL et Verilog. Dans ce chapitre, j'explique l'émergence de ces HCL et les illustre à travers deux publications relatives au HCL `RubyRTL` que j'ai développé [W3], ainsi que la bibliothèque `Litex` que j'ai pu mettre en lumière dans [W4].

5.1 Déficiences des HDL classiques

Les langages Verilog et VHDL ont occupé le devant de la scène de conception numérique depuis plus de 35 ans. Leur adoption initiale n'a pas été sans mal dans la communauté des concepteurs, notamment du fait de l'intuition (incorrecte) que la modélisation passerait par des outils sophistiqués de saisie graphique, et non par une saisie textuelle. Les HDL textuels l'ont emporté. [1] Techniquement, on doit à la notion d'*inférence matérielle* une grande part de ce phénomène d'adoption : en décrivant le comportement de manière adéquate de manière logicielle, les outils de synthèse sont en mesure de construire automatiquement des structures RTL bien plus sophistiquées qu'il ne serait concevable de le faire à l'aide de composants graphiques interconnectés.

Cependant, malgré leur contribution immense à notre capacité à créer des circuits plus complexes, force est de constater que ces HDLs présentent plusieurs défauts majeurs :

Syntaxe verbeuse et vieillotte Le premier de ces défauts tient à leur syntaxe verbeuse. VHDL est souvent pointé du doigt sur cet aspect, mais Verilog présente également des lourdeurs syntaxiques évidentes. A titre d'exemple, il est étonnant qu'aucun mot clé n'ait été introduit dans les langages pour décrire de simples bascules D. Ces bascules D constituent en effet la clé de voute des comportements séquentiels. Le concepteurt HDL doit ainsi *re-décrire* le fonctionnement d'une telle bascule D à chaque fois qu'il cherche à l'inférer. Certains HCL, comme `SpinalHDL`, permettent de s'en affranchir [2].

```
import spinal.core._
case class Counter(width: Int) extends Component {
  // Creation d'un registre pour stocker la valeur du compteur
  val io = new Bundle {
    val enable = in Bool()  // Signal pour activer l'incrementation
    val count  = out UInt(width bits)  // Sortie du compteur
  }

  // Declaration du registre avec une largeur parametree
  val countReg = Reg(UInt(width bits)) init(0)

  // Logique du compteur : Incremente seulement si enable est active
  when(io.enable) {
    countReg := countReg + 1
  }

  // Connecter la sortie au registre
  io.count := countReg
}

object CounterVerilog {
  def main(args: Array[String]): Unit = {
    // Genere le Verilog du compteur avec une largeur de 8 bits
    SpinalVerilog(Counter(8))
  }
}
```

Typage complexe C'est le cas pour Verilog, mais également et surtout pour VHDL, qui hérite de la syntaxe de Ada. Pour rappel, Ada a été développé dans les années 1970-1980 par le Département de la Défense des États-Unis (DoD) pour répondre aux besoins de systèmes embarqués critiques, notamment pour les applications militaires et aéronautiques. Son objectif était de fournir un langage robuste, sécurisé et structuré, permettant de gérer la complexité et la sûreté des systèmes critiques. Ada,toujours utilisé dans ces domaines en 2025, est reconnu pour sa gestion rigoureuse des types et sa capacité à détecter un grand nombre d'erreurs à la compilation. VHDL hérite également de cette rigueur dans le typage. Chez les futurs ingénieurs, désormais adeptes de langages dits *agiles* et typés dynamiquement, comme Python, il s'agit là d'un frein majeur à l'entrée dans le monde de l'Electronique numérique.

1. La modélisation purement informatique (UML, Sysml etc) semble ne pas avoir retenu la leçon.
2. C'est également le cas de langages synchrones comme Signal, où le retard d'un tick d'horloge est dénoté par le symbole dollar $ (car...le temps c'est de l'argent)

Rigidité Plusieurs curiosités de VHDL ont également longtemps dérouté les jeunes concepteurs. A titre d'exemple :

— Jusqu'à l'évolution 2008 du standard [3], il n'était pas possible de relire un signal pilotant une sortie d'entité. Il s'agit là d'une chose étrange pour l'électronicien, car rien n'interdit électroniquement une telle réutilisation (il s'agit d'équipotentielles). Cette déficience force le concepteur à créer des signaux supplémentaires "intermédiaires".

— Cette rigidité se retrouve également dans la description de *structures génériques* décrites à l'aide de construction *generate* du langage. Bien que certaines de ces constructions sont effectivement disponibles depuis longtemps, il a fallu attendre le standard 2008 (et son support tardif dans les outils) pour certaines constructions conditionnelles, comme celle proposée ici :

```
g1: case mode generate
  when 0 =>
    c1 : entity work.comp(rtl1)
         port map (a => a, b => b);
  when 1 =>
    c1 : entity work.comp(rtl2)
         port map (a => a, b => b);
  when 2 =>
    c1 : entity work.comp(rtl3)
         port map (a => a, b => b);
  when others =>
end generate;
```

Sémantique complexe La sémantique des HDL classiques est complexe. Dans un cours que j'enseigne à l'Intitut Mines Télécom depuis plus de 10 ans, j'illustre quelques curiosité du langage VHDL, lié à la sémantique par événements discrets sur laquelle s'appuient les simulateurs. Cette sémantique est *imposée* par la norme IEEE 1076. Parmi ces exemples, le suivant (listing) est particulièrement perturbant. Il s'agit de réaliser la somme d'un signal avec lui-même, après un échantillonnage dans deux bascules : le résultat est forcément pair. Un banc de test stimule le circuit avec une séquence d'entiers incrémentés. Lorsqu'on observe le résultat en simulation et sur carte FPGA, les résultats diffèrent : en simulation le résultat est impair, tandis que sur matériel, le résultat est pair ! Malgré leur excellence, peu d'étudiants sont capables de découvrir le pot aux roses. Pire : les deux résultats sont légitimes (le code ne présente pas de "bugs") ! La mystification de l'exemple tient ici au *renommage* de l'horloge réalisée à l'aide d'une assignation `clk2 <= clk`, en apparence bien innocente : alors que ce renommage n'a aucune conséquence en synthèse, qui utilise une sémantique synchrone, il en est tout autrement dans le cas en simulation. Le simulateur à événements discrets, fondement de VHDL, introduit un *delta delay* entre les deux horloges renommées : ce temps infinitésimal, qui n'est là que pour assurer la causalité dans les assignations, *désynchronise* l'échantillonnage des deux bascules, et induit une séquence de données impaires. Stephen Edwards, de l'Université Columbia, a ainsi qualifié VHDL de *kitchen sink* [M21] ("évier de cuisine" en français) pour qualifier cette tentative d'accumuler les concepts dans un seul langage. Notez que cet exemple, inventé par mes soins, n'est pas propre à VHDL seul, mais est typique des simulateurs à événements discrets en règle générale. Ici, la sémantique de VHDL, bien que respectant la causalité, ne restitue pas le fonctionnement attendu du circuit : en procédant par pas infinitésimaux, le simulateur "rate" la notion de *synchronisation globale* constitutive d'un circuit synchrone.

3. La version 93 est longtemps restée la version par défaut de la plupart des outils de simulation et de synthèse

```vhdl
entity circuit is
  port(
    clk    : in  std_logic;
    sample : in  unsigned(7 downto 0);
    result : out unsigned(7 downto 0)
    );
end entity;

architecture rtl of circuit is
  signal val_1, val_2 : unsigned(7
downto 0);
  signal clk2         : std_logic;
begin

  reg_1 : process(clk)
  begin
    if rising_edge(clk) then
      val_1 <= sample;
    end if;
  end process;

  reg_2 : process(clk2)
  begin
    if rising_edge(clk2) then
      val_2 <= sample;
    end if;
  end process;

  reg_3 : process(clk)
  begin
    if rising_edge(clk) then
      result <= val_1 + val_2;
    end if;
  end process;

  clk2 <= clk;

end rtl;
```

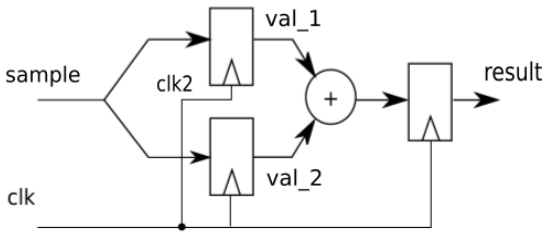


FIGURE 5.1 – Code VHDL et schéma RTL associé, tel que pensé par un concepteur. Un simple renommage d'une des horloges provoque une différence de comportement catastrophique entre le code synthétisé et le code simulé. L'exemple illustre les deux sémantiques de VHDL.

5.2 Principes des HCL

Les HCL présentent des caractéristiques communes, que nous allons passer rapidement en revue.

5.2.1 Abstraction et la modularité

L'aspect le plus notoire tient au fait qu'ils se présentent comme des DSL internes et bénéficient d'emblée d'un support important de leur langage hôte. Par exemple, dans le cas de Chisel ou SpinalHDL, le langage hôte est `Scala` : il s'agit ici d'un langage multi-paradigme (objet et fonctionnel) bénéficiant en outre de l'écosystème `Java`. Les HCL bénéficient ainsi d'un ensembe d'*abstractions* intimement liées à ces langages de programmation modernes : Scala permet par exemple de définir des abstractions riches et réutilisables comme les classes, traits, et fonctions d'ordre supérieur, qui sont mises à profit par le concepteur ayant recours aux HCL dérivés. Ceci facilite la création de modules matériels complexes tout en réduisant la duplication de code. Il est est de même concernant la *modularité* : grâce à des concepts comme les traits et la composition par mixins, Chisel et SpinalHDL peuvent organiser la logique en blocs modulaires et réutilisables, rendant le code plus maintenable et évolutif.

5.2.2 Elaboration interne

Les HCL présentent une seconde caractéristique commune fondamentale : l'exécution de ces DSL internes conduit à l'élaboration en mémoire d'une représentation interne du circuit décrit. Il s'agit ainsi d'une forme de *synthèse RTL*, où une structure de calcul est effectivement élaborée, et qui permet la génération d'un code VHDL ou Verilog en aval. A ce titre, `SystemC` ne peut pas être considéré comme un HCL, bien qu'il se présente lui-aussi comme un DSL interne, hébergé par `C++`. A son exécution, aucune représentation matérielle n'est élaborée.

5.2.3 Vérification

La capacité des HCL à élaborer un circuit directement (sans l'aide d'un outil tiers) offre un avantage important concernant la vérification de ces circuits : le concepteur bénéficie ici de la puissance de langages *mainstream* dans l'élaboration de stratégies de vérification complexes ou plus simplement de génération de vecteurs de tests. Pour rappel, dans le cas de VHDL ou Verilog, la génération de vecteurs de tests est le plus généralement réalisée de manière externe, à l'aide d'un modèle de référence écrit dans un langage support (comme Matlab, C, Python etc) qui fournit des patterns d'entrée-sorties permettant de réaliser cette vérification. On comprend bien ici tout l'intérêt de simplifier ces flots de conception liés à la seule mise au point des circuits.

5.2.4 *HCL is not HLS*

Une des grandes suprises liées à l'avénement de HCL tient au fait qu'ils ne s'appuient pas sur les techniques de HLS (*High-level synthesis*). C'est effectivement une surprise pour plusieurs raisons. La première est que la HLS semblait monopoliser depuis fort longtemps un grand nombre de chercheurs sur le domaine, conduisant à l'avénement d'outils là-aussi robustes et parfaitement utilisables (dont Vivado HLS) : on aurait ainsi pu s'attendre à voir de nouveaux langages s'emparer de la HLS (peu de nouveaux langages ont finalement émergé en ce sens). La second est que les techniques HLS semblent soulager le concepteur d'un certain nombre d'activités complexes comme l'allocation des registres et unités de calcul, ainsi que l'ordonnancement. Il apparaît au final que la volonté du concepteur de *décrire* et *maîtriser* une structure de calcul prédomine sur son *obtention directe*. On peut y voir le souhait de pouvoir concevoir des structures plus élaborées que celles engendrées par la HLS (structures régulières, etc), mieux spécifiées (pipeline précis) et ainsi posséder une *image mentale* du circuit propre à faciliter la phase de vérification. Ces remarques peuvent apparaître à charge sur la HLS au profit des HCL, mais mon propos est plutôt de souligner une certaine effervescence dans le domaine de la conception matérielle : cette période est d'ailleurs comparée par Patterson comme une explosion cambrienne des possiblités en matière de conception matérielle.

5.3 Construction de RubyRTL

J'ai proposé `RubyRTL` dans un papier accepté au workshop *Open Source Design Automation (OSDA)* lors de la conférence *Date'20*. Il fait suit à un premier papier également accepté lors de l'édition de l'année précédente, et dont le but était une présentation de *Litex*, dont nous parlerons également dans la suite de ce chapitre. `RubyRTL` est construit à l'aide de techniques dont certaines ont déjà exposées dans le chapitre précédent, concernant la métaprogrammation. On rappelle que l'objectif premier est de bénéficier non seulement de la syntaxe du langage hôte, mais également de son exécution, le but étant l'obtention finale d'un circuit en VHDL, dans le cas de `RubyRTL`. On rappelle également que ces techniques permettent de s'affranchir des étapes d'analyse syntaxique et de résolution des noms.

5.3.1 Entrées-sorties. Assignations

L'exemple d'un demi-additionneur est présenté ici. Conformément à l'approche orienté-objets, on modélise ce composant à l'aide d'une classe (ligne 1), qui hérite ici de toutes les notions associés à un circuit (en Ruby, l'héritage s'écrit à l'aide du symbole <). La méthode `initialize` (également classique en Ruby) permet de décrire les entrées-sorties du circuit, ainsi que sa structure (ou son comportement) : des méthodes sans parenthèses sont présentes ici afin de préciser ces éléments (méthode `input`, `output`). Le cas de l'assignation de signaux est intéressant : on imite ici le `assign` de Verilog. Afin d'imiter à nouveau le symbole <= présent dans Verilog et VHDL, nous utilisons le symbole de comparaison, en l'ayant préalablement redéfini. Il faut bien comprendre le modèle objet : <= est une *message* envoyé au récepteur. On rappelle que dans un langage orienté-objet pur, ce qui apparait comme une expression `a<=42` est en réalité un appel de méthode (avec notation pointée) `a.<=(42)`. Dans l'exemple donné en RubyRTL, dans la première assignation, cette comparaison est une méthode du récepteur `sum`, qui a été précédemment déclaré comme `output` : l'objet `sum` est ainsi un *signal* du DSL, ce qui nous permet de *redéfinir* la signification de la méthode <= pour cet objet. L'exécution de la méthode renverra en l'occurence un objet de l'AST RubyRTL appelé *Assignment*.

```
1  class HalfAdder < Circuit
     def initialize
3      input  :a,:b
       output :sum
5      output :cout

7      assign(sum  <= a ^ b)
       assign(cout <= a & b)
9    end
   end
11
   ha=HalfAdder.new #instanciation
13 compiler=Compiler.new
   compiler.compile ha # VHDL generated !
```

Listing 5.1 – Circuit `half adder` décrit dans le DSL interne RubyRTL.

5.3.2 Descriptions hiérarchiques

La notion de hiérarchie est illustrée sur le code suivant d'un *full adder*. Pour rappel, on peut construire un tel additionneur 1 bit complet, à partir de deux demi-additionneurs. Le code présente l'instanciation de composants : dans RubyRTL, on précise cette instanciation par le mot clé `component` du DSL, suivi d'un Hash Ruby : ce hash indique le nom du composant (par exemple ici `:ha1` et `:ha2`) suivi du nom du composant (comme dans l'exemple, il peut s'agir de la classe du composant ou d'un objet instance).

```
require_relative 'half_adder' # preceding circuit
2
class FullAdder < Circuit
4  def initialize
     input  :a,:b,:cin
6    output :sum,:cout
```

```
    component :ha1 => HalfAdder     # class...
    component :ha2 => HalfAdder.new # or ...obj

    assign(ha1.a <= a )
    assign(ha1.b <= b )
    assign(ha2.a <= cin)
    assign(ha2.b <= ha1.sum)
    assign(sum   <= ha1.sum)
    assign(cout  <= ha1.cout | ha1.cout)
  end
end
```

Listing 5.2 – Circuit `full adder` en RubyRTL.

5.4 Instanciation programmative

Il est désormais possible de décrire des interconnexions plus complexes, en profitant des capacités conjointes du DSL et du langage Ruby. L'exemple suivant est celui d'un additionneur générique (ici de type *ripple-carry adder*) : on passe au constructeur le nombre de bits et le code *construit* le circuit en mémoire, de manière programmative.

```
class Adder < Circuit

  def initialize nbits
    input  :a    => nbits
    input  :b    => nbits
    output :sum  => nbits
    output :cout

    # create  components
    adders=[]
    for i in 0..nbits-1
      adders << component("fa_#{i}" => FullAdder)
    end

    # connect everything
    for i in 0..nbits-1
      assign(adders[i].a <= a[i])
      assign(adders[i].b <= b[i])
      if i==0
        assign(adders[0].cin <= 0)
      else
        assign(adders[i].cin <= adders[i-1].cout)
      end
      # final sum
      assign(sum[i]        <= adders[i].sum)
    end
  end
end
```

Listing 5.3 – Additionneur RCA générique en RubyRTL, illustrant sa capacité à décrire des dispositifs paramétrés

On notera que nous avons fait le choix de réaliser la connexion à un port nommé de manière pointée (ce qui semble être adopté par la plupart des HCL modernes). A l'aide de cette syntaxe, il est ainsi possible de décrire et générer des composants hiérarchiques interconnectés et paramétrés. La possibilité d'utiliser ainsi l'*intégralité* du langage hôte Ruby au service du DSL hébergé confère à RubyRTL une très grande agilité. Pour l'instant toutefois le comportement de ces composants n'est qu'équationnel. Nous allons désormais présenter du code véritablement RTL.

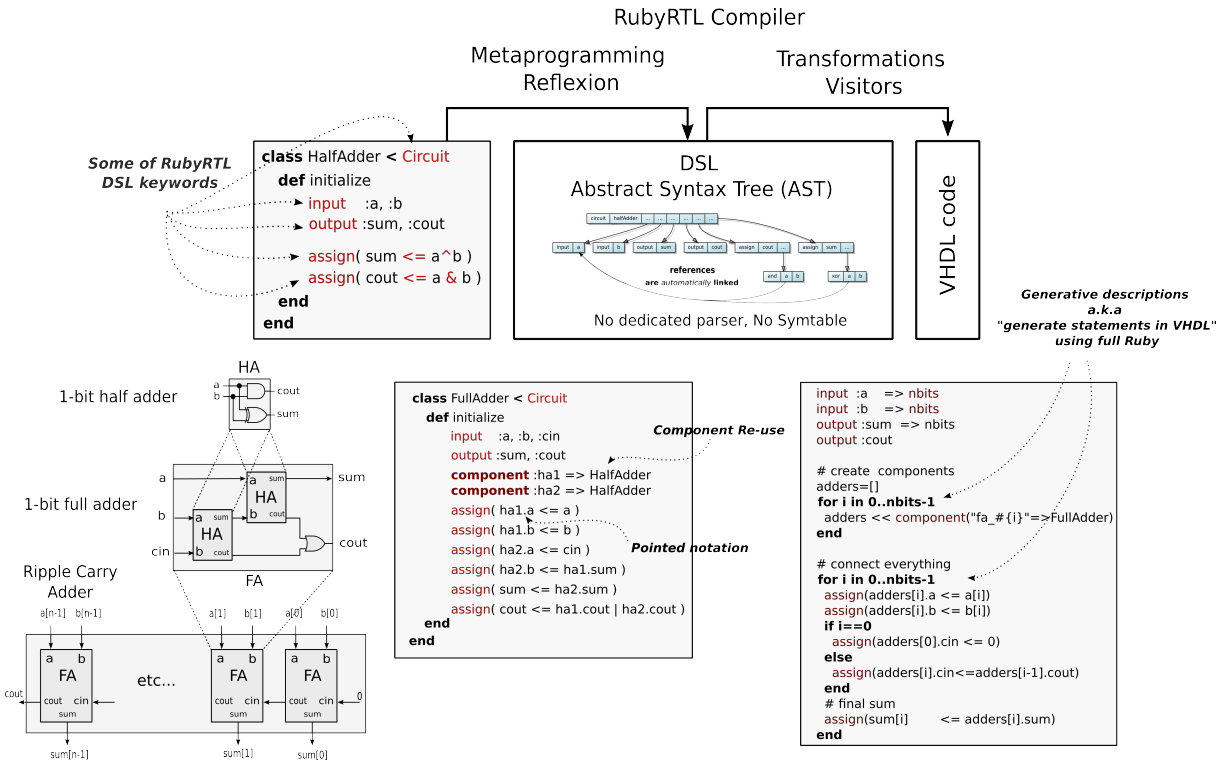


FIGURE 5.2 – Aperçu de RubyRTL [W3]

5.4.1 Inférence automatique des registres

L'exemple suivant d'un simple compteur illustre un comportement typique de l'inférence matérielle. L'appel à une méthode appelée `sequential` permet d'indiquer au compilateur que toutes les assignations devront inférer des registres. On note au passage qu'à la différence de VHDL oude Verilog, on ne doit re-décrire explicitement que ces registres fonctionnent sur le front montant de l'horloge. Cette horloge est par ailleurs implicite : à l'heure de cette rédaction, RubyRTL ne synthétise que des circuits mono-horloges. L'adjonction future d'une référence d'horloge se fera probablement par l'ajout d'un paramètre déclaré en tête du constructeur.

```
class Counter < Circuit
  def initialize
    input  :do_count
    output :count => :byte

    sequential(:strange_counting){
      If(do_count==1){
        If(count==255){
          assign(count <= 0)
        }
        Elsif(count==42){
          assign(count <= count + 42)
        }
        Else{
          assign(count <= count + 1)
        }
      }
    }
  end
end
```

Listing 5.4 – Exemple de descriptions 'comportementales' présentant des structures de contrôles : les `If`,`Elsif`,`Else` du DSL sont en réalité des appels de méthodes dont l'un des arguments est un bloc.

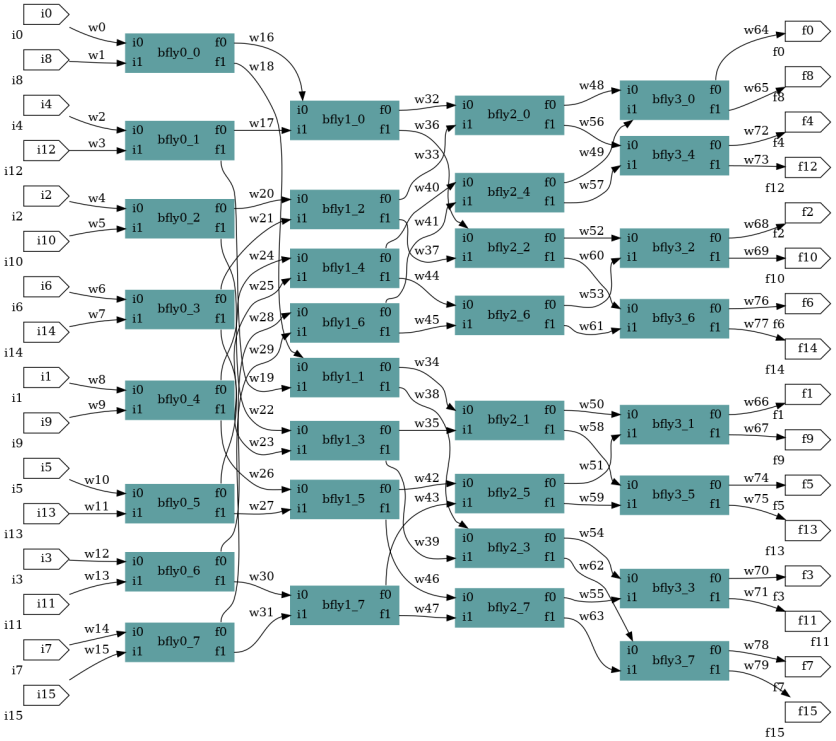


FIGURE 5.3 – Autre exemple de structures aisément décrites en générées en RubyRTL : FFT 16 points comme un réseau de papillons. Le calcul de l'entrelacement de ces papillons profite de la facilité de programmation du langage de script Ruby, hôte du HCL.

Structures de contrôle Comme pour les autres constructions du langage, les mots clés du DSL `If`,`Else` et `Elsif` ont été ajoutés à l'aide d'appels judicieux de méthodes. Il a fallu toutefois distinguer ces mots clés du lexique Ruby natif : pour cela, les majuscules sont requises. A cette exception, on peut noter que les constructions associées sont syntaxiquement très naturelles. Il s'agit pourtant, bel et bien, d'appels de méthodes RubyRTL cachées. Dans l'exemple, en interne, le compilateur doit par exemple reconstruire les trois branches de l'AST à partir des trois appels de méthodes `If`, `Elsif` et `Else`, etc. Mais l'on comprend bien à travers cet exemple qu'aucun frein ne s'oppose à la méthode d'élaboration d'un DSL interne complet et expressif.

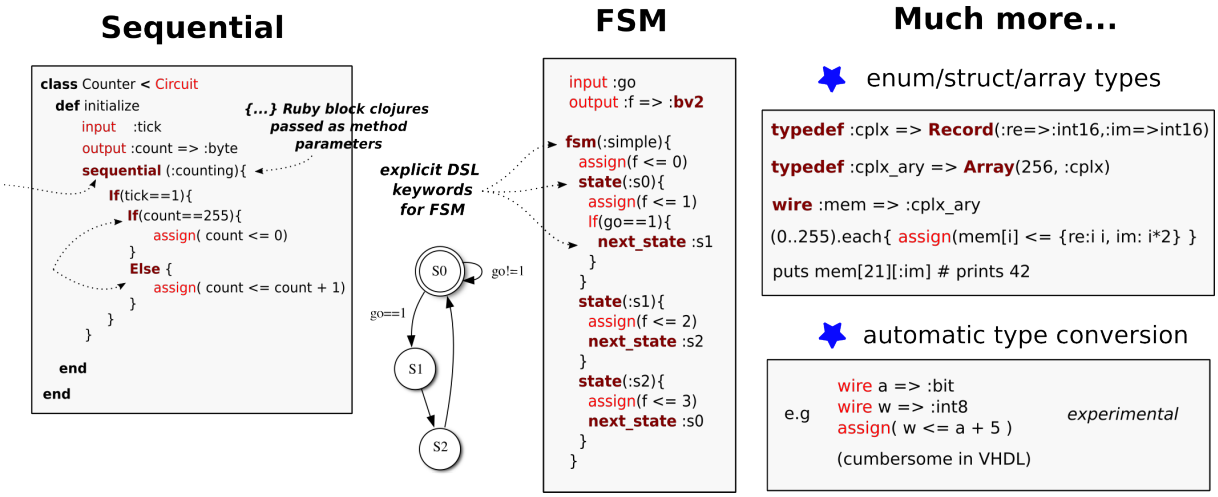


FIGURE 5.4 – Description de circuits séquentiels en RubyRTL [W3]

5.4.2 Machines d'états finis

L'exemple suivant illustre une construction plus avancée, à savoir la description de machines d'états finis. L'exemple est à nouveau suffisamment parlant pour constater la clareté de cette construction. En comparaison de VHDL et Verilog, on évite ici la double déclaration des noms symboliques des états. Il serait ici également possible de recourir à une programmation Ruby traditionnelle pour décrire des machines d'états paramétriques (nombre d'états variables par exemple), ce qui est remarquablement impossible en VHDL.

```
class FSM1 < Circuit
  def initialize
    input :go,:b
    output :f => :bv2

    fsm(:simple){

      assign(f <= 0)

      state(:s0){
        assign(f <= 1)
        If(go==1){
          next_state :s1
        }
      }

      state(:s1){
        assign(f <= 2)
        next_state :s2
      }

      state(:s2){
        assign(f <= 3)
        next_state :s0
      }
    }
  end
end
```

D'autres constructions sont disponibles dans le langage, mais ne seront pas présentées ici. Notamment, il est possible de déclarer des types complexes (type composite comme les `record` de `VHDL` ou des tableaux `array`). Différentes expérimentations concernant le *transtypage* peuvent être facilement expérimentées dans un tel DSL interne. Ceci est également présenté dans notre article [**?**].

5.5 Litex

5.5.1 Aperçu du DSL Migen

Migen [M9] est le DSL sous-jacent à *Litex*. Migen a été élaboré par Sébastien Bourdeauducq dans le cadre initial d'un SoC appelé Milkymist [M8], dédié à la vidéo (Migen signifie d'ailleurs *Milkymist generator*). Il se présente comme un DSL Python conçu pour décrire du RTL sans les lourdeurs des HDL classiques. Les techniques sous-jacentes utilisées dans la construction de Migen ressemblent sur de nombreux aspects aux techniques utilisées par le DSL *RubyRTL*. Là aussi, l'idée était de profiter d'un langage hôte facile à prendre en main et se prêtant suffisamment à la métaprogrammation : il s'agit ici de Python. Une version repensée de Migen s'appelle *Amaranth* [4].

4. Ces outils sont développés par M-Labs autour du projet ARTIQ (Advanced Real-Time Infrastructure for Quantum physics), un système de contrôle open-source spécialement conçu pour la gestion d'expériences de physique quantique.

Exemple 1 : assignation et opérateur L'exemple suivant illustre la construction *directe*, à l'aide des classes de Migen, d'une simple expression `and`.

```
a = Signal()
b = Signal()
x = Signal()
and = _Assign(x, _Operator('&', [a, b]))
```

La syntaxe de cette dernière étant particulièrement lourde, Migen propose de la remplacer par : `and = x.eq(a & b)`. Cette construction syntaxique est un raccourci pour le code précédent.

Exemple 2 : *blinking led* Le code suivant présente un simple compteur qui compte sur 26 bits. Le module matériel modélisé hérite de `Module`, qui représente un circuit. Comme pour `RubyRTL`, le circuit est décrit dans le constructeur. Il est directement possible de préciser quelles instructions sont combinatoires ou séquentielles : de manière interne, on enrichit des branches distinctes de l'AST du DSL.

Par exemple, dans `self.sync += counter.eq(counter + 1)`, on incrémente le `signal` counter, en précisant que cette déclaration est bien séquentielle [5]. Les lignes en dehors de la classe sont également intéressantes et concernent la synthèse sur un FPGA physique, ainsi que le *build*. On peut ainsi créer une plateforme d'accueil, précaractérisée par les soins de Enjoy Digital : ici, par exemple, on entend synthétiser notre compteur sur un FPGA Nexys4DDR de Digilent. Il est alors possible d'instancier notre compteur, en passant au constructeur le fil (`"user_led"`) physiquement connecté à une LED de cette plateforme. Ensuite, le même script Python demande la synthèse RTL. L'ensemble de ces activités est simplifié par rapport aux approches de Xilinx Vivado, où ces contraintes d'association se font dans un fichier TCL externe. Mieux : cette démarche a été homogénéisée dans Litex de manière à banaliser les flots de constructeurs Xilinx, Intel, Lattice, Efinix, Gowin, etc...

```
from migen import *

from litex.boards.platforms import nexys4ddr

# Create a led blinker module
class Blink(Module):
    def __init__(self, led):
        counter = Signal(26)
        # combinatorial assignment
        self.comb += led.eq(counter[25])

        # synchronous assignement
        self.sync += counter.eq(counter + 1)

# Create our platform
platform = nexys4ddr.Platform()

# Get led signal from our platform
led = platform.request("user_led", 0)

# Create our main module
module = Blink(led)

# Build the design
platform.build(module)

```

5.5.2 Litex

Migen se présente comme une bibliothèques de composants, un simulateur et un système de *build*. Toutefois, c'est la bibliothèque *Litex* d'Enjoy Digital, développée par Florent Kermarrec, aidé d'une communauté grandissante, qui est désormais sur le devant de la scène. Litex regroupe un très grand nombre d'IPs permettant de construire la structure de SoCs très variés. Un aperçu de l'ensemble est présenté sur

5. on pourra noter la notation pointée `.eq`, que l'on a évitée dans RubyRTL

la figure. Notez que cette figure, extraite de notre article [], donne un état du système en 2018. Il a fortement évolué depuis, avec notamment l'incorporation d'un grand nombre de processeurs RISCV, OpenSparc, etc. La nombre des projets industriels réalisés à l'aide de ces IPs ne cessent d'augmenter. Plus de 150 cartes FPGA sont désormais supportées.

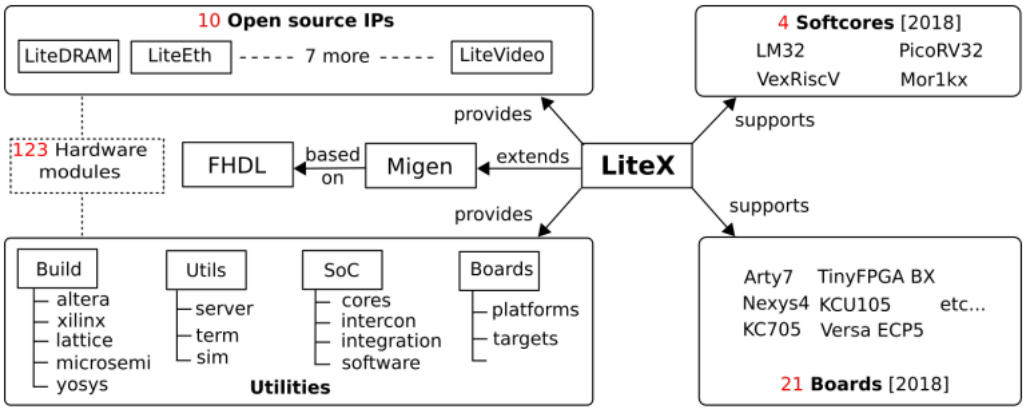


FIGURE 5.5 – Aperçu (2018) des capacités du framework *Litex* [W4].

5.5.3 Vers une interopérabilité de HCL ?

Comme nous avons pu le constater, les deux HCL présentés dans ce chapitre partagent plusieurs points communs. Dans [W3], nous avons proposé une première tentative d'interopérabilité entre ces deux HCL. L'idée centrale est qu'il doit être possible de définir un format commun de représentation des circuits, qui puissent bénéficier aux deux écosystèmes. Le schéma 5.6 explicite cette idée : nous avons défini un format de sérialisation (s-expressions) appelé `Sexpir`, facilement lu et écrit par tout langage de programmation (dont Python et Ruby). Il décrit un circuit RTL réalisé par l'un ou l'autre des langages. Une modification expérimentale a été réalisée dans Migen et RubyRTL : dans l'expérience, nous avons relu dans RubyRTL un design conçu avec Litex. Cette expérience suggère à la communauté d'éviter une certaine dispersion des efforts des concepteurs et des outilleurs, telle qu'elle a pu être constatée autour des HDL classiques.

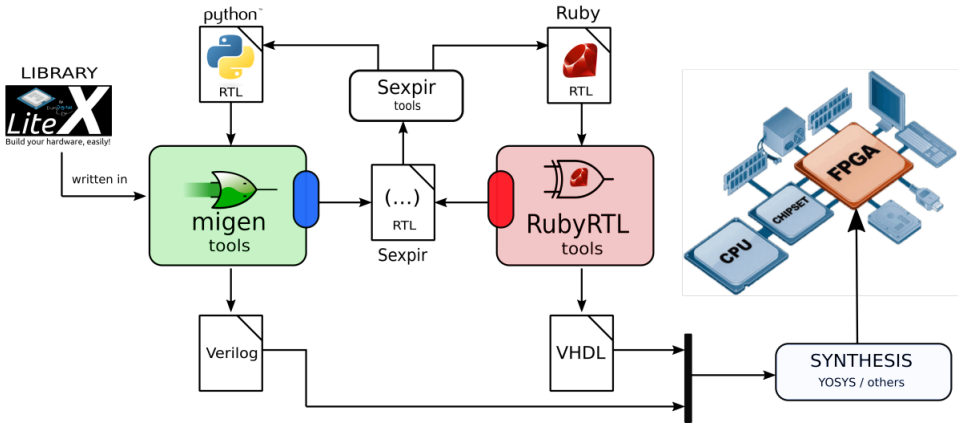


FIGURE 5.6 – Principe de l'interopérabilité entre deux HCL (RubyRTL et Litex)

5.6 Conclusion

Ce chapitre illustre de manière concrète l'utilisation des HCL comme moyen de capture au niveau RTL. Nous avons succintement présenté notre travail autour de RubyRTL et Litex. Ces HCL reposent sur

des langages *mainstream* et incitent ainsi des programmeurs "classiques" à s'intéresser à la conception matérielle et aux SoC en général. Si l'essor des HCL est désormais indubitable, nous avons soulevé le risque potentiel de dispersion des communautés open-source attachées à la conception matérielle, et proposé de réfléchir à l'interopérabilité de ces HCLs. Un nouvel effort conséquent en génie logiciel reste à mener afin de définir des passerelles convaincantes entre de tels DSLs.

Chapitre 6

Capture au niveau comportemental par *Acteurs* : Archipel

> It's very distressing - I'm watching almost with disbelief. The Americans cannot get it out of their heads that if you're trying to build machines with lots of processors, you don't assume that they all share a common memory. The world doesn't have a common database. We pass messages to one another.
>
> David May
> professor of computer science at the University of Bristol
> Doctor Honoris Causa Rennes I, 1997.

Publications et projets associés

- Montage de la startup **Modaë Technologies**.
- [B3] Chapitre d'ouvrage. MoPCoM Methodology : Focus on Models of Computation.
- [C6] An experimental toolchain based on high-level dataflow models of computation for heterogeneous mpsoc.
- [C17] Early exploring design alternatives of smart sensor software with Model of Computation implemented with actors. ESUG'13.

Sommaire

Introduction

Ce chapitre présente `Archipel`, un outil de modélisation et de co-design HW/SW de systèmes numériques qui repose sur le paradigme des acteurs. Nous commencerons par faire quelques rappels sur ce modèle, puis nous étudierons les objectifs et spécificités d'`Archipel` : ce chapitre présente la syntaxe, la sémantique informelle et les techniques de simulation retenues. Deux chapitres complémentaires sont également consacrés à `Archipel` :

— le chapitre 8 se focalise sur la synthèse comportementale de systèmes décrits avec `Archipel`.
— le chapitre 9 sur la notion de raffinement architectural.

6.1 Concept d'acteur

Les acteurs se présentent comme des entités indépendantes, qui maintiennent un état interne qui leur est propre, et qui entrelacent calcul et envoi de messages. L'idée n'est pas neuve, puisque ce concept a été initialement proposé par Carl Hewitt, Peter Bishop et Richard Steiger en 1973, puis développé significativement par Gul Agha dans les années 1980. Cette indépendance était révolutionnaire à l'époque car elle s'écartait des visions procédurales traditionnelles, qui monopolisaient la grande partie des efforts des informaticiens. Les acteurs introduisaient de manière intuitive la notion de concurrence.

Alan Kay lui-même, inventeur de Smalltalk et pionnier de l'approche objet, a souvent évoqué le modèle d'acteurs. Il expliquait à OOPSLA'97 que l'échange de messages est fondamentalement plus important que les objets eux-mêmes, ces derniers étant souvent trop mis en avant. Le modèle initial de programmation orientée objets vise effectivement, selon lui, à décrire le comportement de systèmes distribués complexes à travers des flux de données et de tels envoi de messages entre acteurs du système.

Définition d'un acteur On peut tenter de définir un **acteur** P comme le quintuplet :

$$A = (S, C, \delta, \rho, \pi)$$

où :

— S est l'ensemble des états internes possibles de l'acteur.
— C est l'ensemble des canaux de communication que l'acteur peut utiliser.
— $\delta : S \times M^* \rightarrow S$ est une fonction de transition qui détermine l'évolution de l'état de l'acteur. Elle prend comme entrée l'état courant $s \in S$ et une séquence de messages $m_1, m_2, \ldots, m_k$ reçus à travers ses canaux, et produit un nouvel état $s' \in S$.
— $\rho : S \times C \rightarrow M^*$ est une fonction de réception qui extrait les messages disponibles sur un canal donné $c \in C$, en fonction de l'état courant.
— $\pi : S \times C \rightarrow M^*$ est une fonction d'émission qui détermine les messages que l'acteur envoie sur un canal donné $c \in C$, en fonction de son état courant.

Règles de communication

— Les acteurs communiquent via des **canaux** (C), qui sont des abstractions permettant d'envoyer et de recevoir des messages (M). Les canaux peuvent être modélisés de différentes manières : files d'attente, flux synchrones, etc.
— La communication peut être :
  — **Synchrone** : l'envoi et la réception d'un message bloquent jusqu'à ce que les deux soient effectués.
  — **Asynchrone** : les messages envoyés sont stockés dans une file d'attente associée au canal, et le processus récepteur peut les récupérer ultérieurement.
— Les processus n'ont pas d'accès direct aux états des autres processus. La coordination se fait exclusivement via les canaux.

Acteurs et co-design L'adaptation du concept au co-design HW/SW semble particulièrement intéressante car elle rejoint une des motivations originelles de Hewitt : créer un modèle de calcul qui corresponde mieux à la réalité physique des systèmes que le modèle de von Neumann. Les systèmes matériels sont intrinsèquement parallèles et communicants, ce qui rend le modèle d'acteurs particulièrement adapté. On

doit par ailleurs faire remarquer que les "bonnes propriétés" du modèle d'acteur (encapsulation, communication par message et parallélisme instrinsèque) sont parfaitement *banales* pour un électronicien du numérique, qui les manipule de manière naturelle depuis toujours : tous les circuits fonctionnent *comme des acteurs* ! Cependant, l'électronicien *encode le modèle d'acteurs*, dans des langages adhoc (HDL) et surcontraints (RTL). Il s'agit pour nous ici de proposer une syntaxe explicite pour ces acteurs : ainsi les protocoles de requête et d'acquitement dans un message asynchrone, courant en électronique, sont ici *natifs* à `Archipel`. La partie algorithmique est décrit au niveau *comportemental* à l'aide des constructions habituelles à ce niveau d'abstraction : assignations, conditionnelles, boucles, appels de fonctions, etc.

Idée

Etrangement, la simple idée d'acteurs en tant qu'entités parallèles autonomes est parfaitement banale pour un électronicien numéricien, mais reste difficile à incarner pour les informaticiens.

Acteurs ou processus communicants ? Nous avons légitiment retenu le terme d'acteur ici, mais la notion de processus communicants (CSP, Hoare) semble également très adaptée. Ces derniers font toutefois immanquablement penser aux systèmes d'exploitation eux-mêmes, qu'ils ont fortement inspirés : par glissement, on peut arguer que les processus séquentiels visent à exploiter le matériel préexistant, alors que les acteurs ont une ambition de modélisation abstraite et applicative plus prononcée, pratiquement indépendants du support d'exécution, et plus englobante que ne le sont les processus. Mais dans les faits, la différence est pour nous non significative. Dans notre acception par exemple, les acteurs empruntent aux processus communicants les canaux de communication explicites, par rendez-vous synchrones ou par FIFOs asynchrones. Par contre, nos acteurs ne peuvent pas partager de données, alors que cela est possible entre processus classiques. Nos acteurs sont donc probablement à une forme d'intersection des deux approches.

6.2 Objectifs

Archipel a été conçu pour répondre à plusieurs objectifs :

1. Capturer des applications dans des domaines tels que le traitement du signal et de l'image, où le traffic des données véhiculées est important.
2. Simuler ces applications à différents niveaux de granularité et de performance à l'aide de simulateurs interprétés ou compilés.
3. Évaluer les performances intrinsèques des systèmes abstraits.
4. Explorer les scénarios de mapping pour le partitionnement hardware/software (HW/SW), incluant la génération d'accélérateurs matériels.
5. Synthétiser l'ensemble du système, incluant les composants matériels (génération de code VHDL) et logiciels.

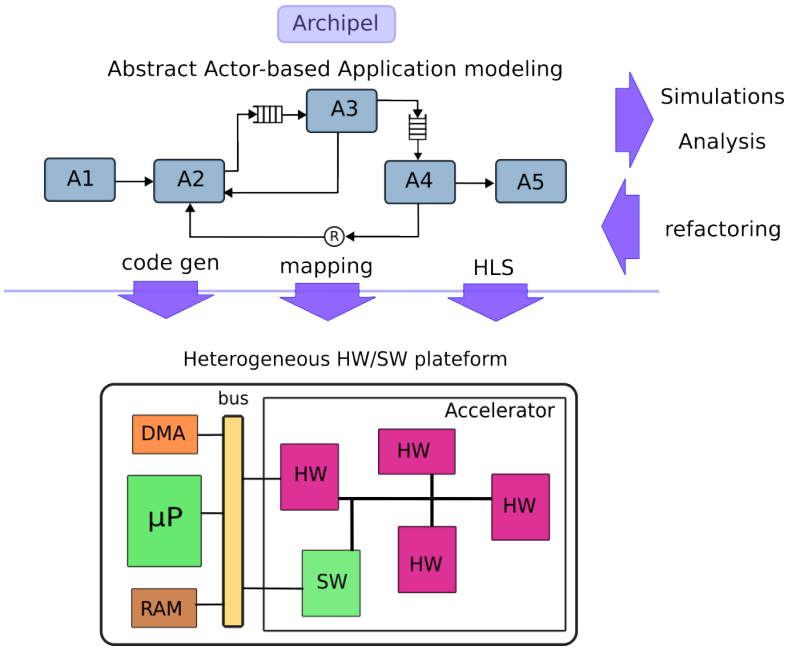


FIGURE 6.1 – Illustration de l'ambition générale de `archipel` : capturer et simuler un système comme un ensemble d'acteurs communicants, et faciliter le portage et la synthèse matérielle sur une cible SoC hétérogène.

6.3 Syntaxe et sémantique

La syntaxe d'un système décrit en `Archipel` est illustrée sur la figure 6.2. On y retrouve la description d'un acteur et celle de l'interconnexion de tels acteurs. Bien que cela n'apparaisse pas dans l'exemple proposé, l'interconnexion peut présenter des boucles, des conditionnelles, etc ce qui permet de décrire des interconnexions complexes et répétitives de manière programmative.

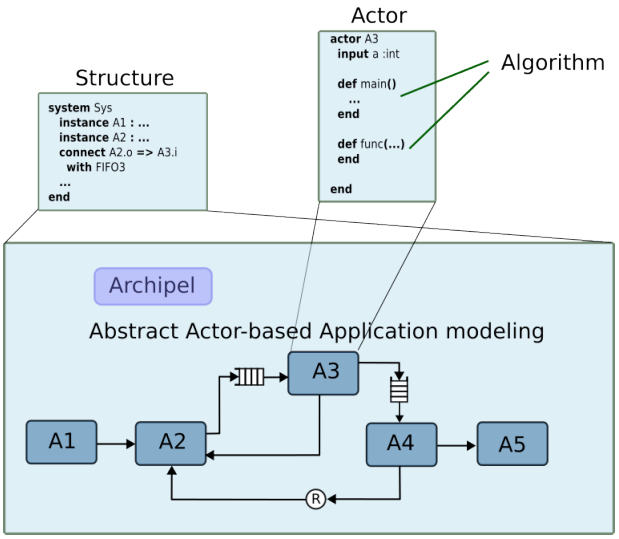


FIGURE 6.2 – Premier aperçu de la syntaxe d'`Archipel`

6.3.1 Syntaxe : acteur et système

Un acteur se présente comme un réceptacle (ou encapsulation) d'un comportement. Il possède ses ports d'entrée et de sortie, ainsi qu'un ensemble de méthodes qui lui sont propres. En ce sens, nos acteurs

peuvent être vus comme des *objets parallèles*, au sens de la programmation orientée-objet. A l'heure de rédaction du manuscrit, l'héritage n'est pas implémenté dans le compilateur.

```
actor Ping
  input  e : int
  output out : int

  def main
    var accu,val : int
    accu=0
    send accu => out
    while true
      receive val <= e
      accu +=1
      send accu => out
    end
  end
end
```

Listing 6.1 – Exemple de l'acteur Ping

```
system PingPong
  instance ping : Ping
  instance pong : Pong
  connect ping.out to pong.e with fifo3
  connect pong.out to ping.e with fifo3
end
```

Listing 6.2 – Exemple du Ping-Pong

Syntaxe du DSL interne La syntaxe présentée ici est celle du DSL externe. Il existe toutefois une description sous forme de DSL interne (utilisé intensivement dans la startup *Modaë technologies*.)

```
class Ping < ESL::Actor
  input :e => :int
  def main
    # etc
  end
end
System.new(:ping_pong) do
  ping=Ping.new
  # etc
end
```

Listing 6.3 – Syntaxe alternative sous forme de DSL interne Ruby

6.3.2 Modules et *mixins*

Les modules permettent de définir des types, constantes et fonctions communes entre plusieurs acteurs. Lorsque ces modules sont inclus dans un acteur, ces définitions lui sont alors connues. C'est un moyen courant de réaliser une forme d'héritage en orienté-objet, en évitant les problèmes classiques d'héritage multiples.

```
module ComplexTypes
  type complex : {re: float ; im: float}
  def add(a: complex ,b:complex) : complex
    return {(a.re+b.re),(a.im+b.im)}
  end
end

actor DSP
  include ComplexTypes # inclusion
  input a,b : complex
  def main
    #.....add(a,b) etc
  end
end
```

Listing 6.4 – Exemple de définition et inclusion de modules

6.3.3 Communications bloquantes et non boquantes

Communications bloquantes : `send/receive` Comme déjà illustré sur le cas de l'acteur *Ping* précédent, on dispose d'une lecture et écriture bloquantes, sous la forme de `send` et `receive`. Lors du calcul de la représentation intermédiaire, le compilateur transforme ces instructions en un protocole séquentiel, comme illustré : une instruction de requête `req(e)` est déposée sur un port `e`, puis, dans le basic-bloc suivant, une scrutation de l'acquittement `ack?(e)` démarre. Dès que l'acquittement est reçu, la requête est dévalidée. Dans un fonctionnement synchrone, le compilateur veille à ce que cette dévalidation se fasse dès réception du `ack`, sous peine qu'une autre requête ne soit immédiatement relancée.

Communications non-bloquantes : `read/write` Avec la même syntaxe, il est également possible de réaliser des lectures-écritures sans validation de protocole : les données sont lues (et écrites) au moment précis de l'exécution. Cette pratique est pour l'instant découragée.

```
actor Receiver
  input f : int
  def main
    var v : int
    v=0
    puts("before. v=,v")
    receive v <= f
    puts("after v=",v)
  end
end
```

Listing 6.5 – Instruction `receive` dans `Archipel`, permettant des réceptions bloquantes sur un canal.

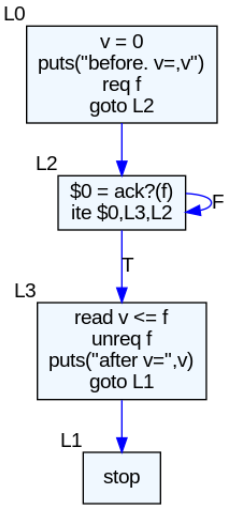


FIGURE 6.3 – IR/CFG correspondant à la méthode `main` précédente, utilisant une instruction bloquante `receive` : de manière sous-jacente, un automate d'état fini apparaît.

6.3.4 Barrière de synchronisation : `synchro`

Les communications bloquantes soulagent le concepteur des mécanismes fins de synchronisation avec les canaux de communication, mais ne sont pas suffisantes dans un environnement de nature parallèle. En particulier, l'ordre d'arrivée des acquittements reste ici problématique : les risques d'interblocages sont omniprésents. Pour rappel, on peut avoir interblocage dans le simple cas suivant :

```
  receive va <= a # l'oeuf ?
  receive vb <= b # ou la poule ?
  #....va+vb
```

Du fait de la nature localement séquentielle du langage, le contrôle peut rester bloquer sur l'attente de réception de la première donnée, alors que c'est la seconde donnée (canal B) qui arrive dans les faits. Pour contourner ce type d'interblocages, nous avons conçu une attente parallèle : il s'agit de l'instruction `synchro` :

```
  synchro
    receive va <= a
    receive vb <= b
  end
  #....va+vb
```

Dans ce cas, le flot de contrôle lance deux requêtes sur les deux canaux `a` et `b`, et se met en scrutation des aquittements, dans un ordre indifférent. L'instruction `synchro` joue le rôle d'une barrière de synchronisation.

```
while true
  synchro
    receive va <= a
    receive vb <= b
  end
  accu+=va+vb
end
```

Listing 6.6 – Instruction `synchro` dans `Archipel`, permettant la réception dans le désordre de données externes à l'acteur

```
L0:
  goto L2
L1:
  stop
L2:
  ite true,L3,L1
L3:
  req a
  req b
  $0 = 0b00
  goto L5
L5:
  $1 = ack?(a)
  ite $1,L6,L7
L6:
  read va <= a
  unreq a
  $0 = ($0|(1<<0))
  goto L7
L7:
  $1 = ack?(b)
  ite $1,L8,L9
L8:
  read vb <= b
  unreq b
  $0 = ($0|(1<<1))
  goto L9
L9:
  $1 = ($0==0b11)
  ite $1,L10,L5
L10:
  accu = (accu+(va+vb))
  goto L2
```

Listing 6.7 – Représentation interne (IR/CFG) du compilateur `Archipel` du modèle précédent relatif à l'instruction `synchro`.

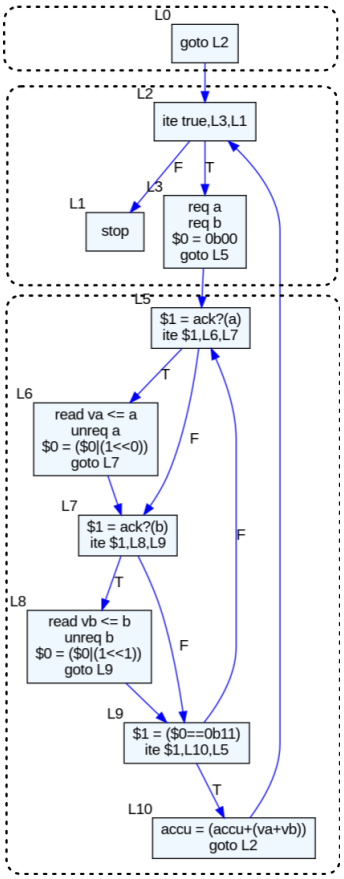


FIGURE 6.4 – Vue graphique de l'IR/CFG exportée par la compilateur. Le chapitre consacré à la HLS montrera l'algorithme de synthèse FSMD (qui conduirait ici à seulement 3 états, représentés en pointillés).

6.3.5 Barrière de séquencement : `step`

Il est également parfois utile de disposer de la possibilité d'attendre un cycle, notamment lorsqu'on cherche à interagir avec un mécanisme matériel pré-existant. L'instruction `step` joue ce rôle.

6.4 Structure du compilateur

La structure générale du compilateur, écrit en Ruby, est illustrée sur la figure 6.6.

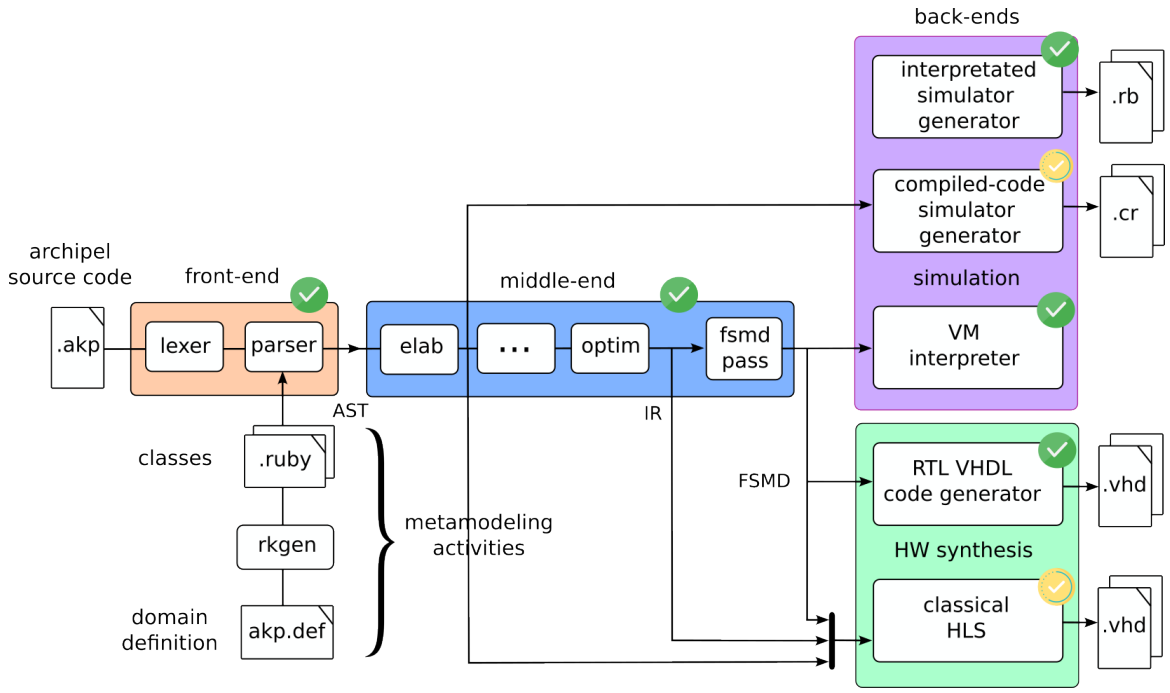


FIGURE 6.5 – Architecture générale du compilateur `Archipel`

— Le **front-end** du compilateur est classique. Nous avons toutefois cherché à décrire de manière externe la constitution du métamodèle, à l'aide d'une description simple et d'un utilitaire permettant de générer le code Ruby des classes correspondantes. Pour rappel, ces classes sont instanciées lors de la création de l'arbre de syntaxe abstraite par le parseur. L'intérêt de cette externalisation et de cette génération systématique est double. D'une part, toutes les classes sont construites sur le même raisonnement. D'autre part, cela permet d'imaginer un portage futur du compilateur vers un autre langage de programmation (à des fins de performances, etc).
— Le **middle-end** vise ici à réaliser la résolution des noms sur l'AST, mais surtout à élaborer la représentation interne (IR), et déclencher diverses optimisations.
— Les **back-ends** sont de différents types, selon les activités envisagées avec `Archipel` : simulation ou synthèse matérielle. Une synthèse HLS particulière, orientée vers le contrôle, sera présentée dans un chapitre ultérieur. On peut noter que certaines de ces activités restent des travaux en cours, comme la génération de code compilé (vers le langage Crystal, réputé très rapide), ainsi que la synthèse HLS dite "classique".

6.5 Représentation intermédiaire (IR)

Le compilateur établit une représentation intermédiaire (IR) spécifique, qui sous-tend un graphe de flot de contrôle (CFG). Sa forme textuelle est systématiquement remontée à l'utilisateur. Contrairement à d'autres IR qui réalisent des optimisations proches de la machine, l'IR d'`archipel` reste de haut niveau et vise à faciliter des transformations *source-à-source*.

Syntaxe abstraite La syntaxe abstraite de cette IR peut se résumer à :
— $v \in \text{Var} = \{v_0, v_1, v_2, \ldots\}$ (Variables)
— $l \in \text{Label}$ (Étiquettes)
— $c \in \text{Const}$ (Constantes)
— $e \in \text{Expr} ::= v \mid c \mid \text{op}_{una}(v) \mid v_1\ \text{op}_{bin} v_2$ (Expressions)
  — $\text{op}_{una} ::= \text{neg} \mid \ldots$ (Opérateurs unaires)
  — $\text{op}_{bin} ::= \text{add} \mid \text{sub} \mid \text{mul} \mid \ldots$ (Opérateurs binaires)
— $i \in \text{Instr} ::= r \leftarrow e \mid \text{stop} \mid \text{ite}(r, l_1, l_2) \mid \text{goto}\ l$ (Instructions)
— $bb \in \text{BasicBlock} ::= l : i_1; i_2; \ldots; i_n$ (Blocs de base)

— $p \in \text{Prog} ::= bb_1; bb_2; \ldots; bb_m$ (Programme)

On remarque les instructions `ite` et `goto` qui permettent des branchements conditionnels et inconditionnels.

Exemple Un exemple d'une telle représentation intermédiaire est donnée ici à la fois sous la forme textuelle et sous forme graphique.

```
actor Computing
  include MyTypes
  input i1 : int
  output f : int

  def main() : nil
    var a,b,res : Matrix
    var i,j,k : uint
    var accu : int
    var tmp :int

    for i in 0..2
      for j in 0..2
        receive a[i][j] <= i1
        receive b[i][j] <= i1
      end
    end

    for i in 0..2
      for j in 0..2
        accu=0
        for k in 0..2
          accu+=a[i][k]*b[k][j]
        end
        res[i][j]=accu
      end
    end

    for i in 0..2
      for j in 0..2
        send res[i][j] => f
      end
    end
  end
end
```

Listing 6.8 – Calcul matriciel

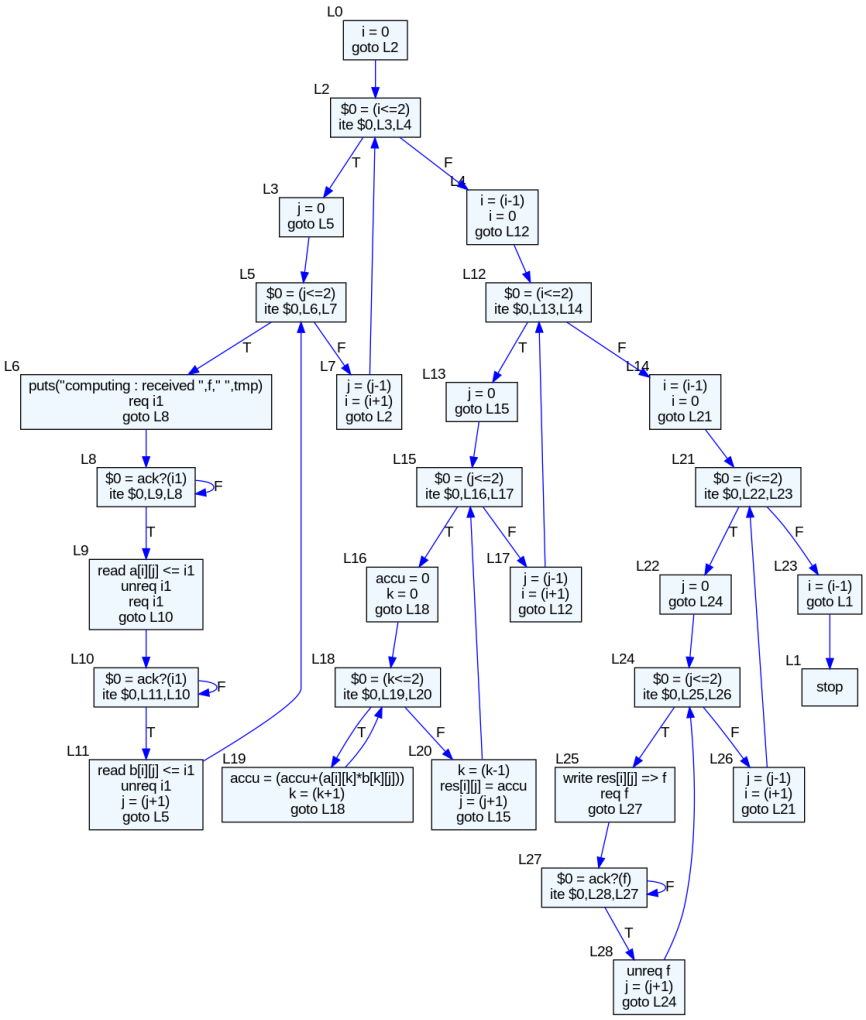


FIGURE 6.6 – CFG de la méthode `main` de l'acteur `Computing` précédent.

6.6 Machine d'états étendue : FSMD implicite

Le compilateur est capable de transformer cette IR en une nouvelle représentation, à savoir une machine d'états finis étendue ou FSMD (finite state machine and datapath) *implicite*. Il existe classiquement deux sortes de FSMD :

— FSMD implicite : le *datapath* reste implicite. Il n'est pas séparé de la FSM. Les actions dans chaque état se présentent comme du code classique : assignations et conditionnelles éventuellement imbriquées apparaissent dans chaque état.

— FSMD explicite : le *datapath* est explicite. Il est vu comme un composant spécifique, piloté par des signaux de contrôle émis par la FSM, qui réagit en retour à des signaux de status.

Une telle FSMD implicite, retenue dans `Archipel`, permet de fournir, avec relativement peu d'efforts, une exécution matérielle ou logicielle qui reste compréhensible par les deux communautés afférentes. Le procédé de passage de l'IR à la FSMD est exposé dans le chapitre 8.

6.7 Simulation fonctionnelle

La simulation fonctionnelle permet de simuler le réseau d'acteurs et d'obtenir une première analyse du comportement d'ensemble du système ainsi décrit. Plusieurs styles de simulateurs peuvent réaliser une telle simulation.

— La première option retenue est celle d'un interpréteur (que nous appelons également VM).

— Une seconde option est celle d'un code (Ruby) généré, basé sur les *coroutines*.

— Enfin, un troisième simulateur se présente sous la forme d'un code généré puis compilé dans le langage Crystal (qui possède également des *coroutines* efficaces).

6.7.1 Interpréteur/VM IR et FSMD

L'interpréteur est conçu, comme l'ensemble des passes du compilateur, selon le design pattern Visiteur. Il a la particularité de pouvoir à la fois simuler l'IR, mais également la forme FSMD implicite obtenue, proche du matériel (voir chapitre suivant). L'interpréteur opère en *lockstep*, de manière synchrone, par cycles abstraits et maintient l'état de toutes les variables relatives à chacun des acteurs. La terminaison d'un acteur le retire de la liste des acteurs en cours d'évaluation.

```
while @running_actors.any?
  @running_actors.each
    step actor
  end
  update(sys.connexions)
  @time+=1
end
```

Listing 6.9 – principe de l'interpréteur synchrone

6.7.2 Simulateur de code exécutable interprété

Le compilateur `Archipel` peut également générer un code de simulation plus rapide, basé sur les *coroutines*. Ces coroutines (appelées Fiber en Ruby) sont particulièrement utiles lorsqu'il s'agit d'implémenter des générateurs ou des itérateurs personnalisés et de créer des machines à états finis (FSM) où l'exécution peut être suspendue et reprise à des points spécifiques : ces coroutines permettent d'exprimer de manière explicite le flot de contrôle du simulateur. Dans le cas d'Archipel, ce flot de contrôle s'interrompt lors des communications. A titre d'exemple d'utilisation des `Fiber` Ruby, on donne l'exemple suivant, où un petit système multitâche est exposé : chaque tâche créée est caractérisée par un nombre de `tick` où elle devra s'exécuter. A chaque `tick`, une tâche s'exécute et rend la main (`Fiber.yield`).

```
class TaskScheduler
  def initialize
    @tasks = []
    @current_time = 0
  end

  def create_task(name, duration, &block)
    fiber = Fiber.new do
      start_time = @current_time
      while @current_time < start_time + duration
        Fiber.yield
      end
      block.call if block_given?
      puts "#{name} termine au temps #{@current_time}"
    end

    @tasks << fiber
  end

  def run
    until @tasks.empty?
      @current_time += 1
      puts "\nTemps actuel: #{@current_time}"
      @tasks.each_with_index do |task, index|
        if task.alive?
          print "  Execution de la tache #{index + 1}..."
          task.resume
        end
      end
      # Supprime les taches terminees
      @tasks.reject! { |task| !task.alive? }
      # Petite pause pour mieux visualiser
      sleep 0.5
    end
```

```
35    end
  end
37
  # Exemple d'utilisation
39  scheduler = TaskScheduler.new

41  # Creation de plusieurs taches avec differentes durees
  scheduler.create_task("Tache rapide", 2) do
43    puts "-Action de la teche rapide executee !"
  end
45
  scheduler.create_task("Tache moyenne", 4) do
47    puts "-Action de la teche moyenne executee !"
  end
49
  scheduler.create_task("Tache longue", 6) do
51    puts "-Action de la teche longue exexcutee !"
  end
53
  # Lancement du scheduler
55  scheduler.run
```

Listing 6.10 – Illustration de l'utilisation des co-routines (fibers) : elles offrent ici l'avantage d'un contrôle fin de l'ordonnancement de tâches.

6.7.3 Simulateur de code exécutable compilé

Un code similaire *compilé* est également généré par `Archipel`. Ce code est généré dans le langage `Crystal`, qui est souvent présenté comme un Ruby compilé (bien qu'il soit un langage à part entière, qui s'est accaparé la syntaxe de Ruby). Les performances de Crystal sont très élevées, proches du langage `C`.

6.8 Perspectives et Conclusion

Dans ce chapitre, nous avons succintement présenté le langage `Archipel`. C'est un langage à acteurs communicants, qui permet de prototyper des systèmes parallèles définis statiquement qui communiquent par port, connectés à des canaux de type FIFO ou Rendez-vous synchrones. Ce langage a été conçu à titre exploratoire : il a permis de prototyper à la fois la syntaxe, d'introduire des facilités de synchronisation et flot de contrôle (comme le `synchro`), mais également des simulateurs associés. D'autres aspects d'`Archipel` seront également présentés aux chapitres 8 et 9.

— Dans le chapitre 8, nous allons présenter les capacités d'`archipel` en matière de synthèse matérielle basée sur la HLS.
— Comme nous allons le voir au chapitre 9, `Archipel` est également le support de ma recherche en matière de co-design logiciel-matériel : notamment, il permet d'explorer, en simulation, les effets des choix d'allocation et de mapping sur processeurs abstraits.

Chapitre 7

Capture au niveau tâche pour architectures multi-coeurs : XPU

Free Lunch is over

Erb Sutter

Publications et projets associés

— [C1] Thèse de Nader Khammassi : Modèle de programmation de haut niveau pour la parallélisation explicite et automatique : application aux architectures multicoeurs.
— [C1] IVLSI'15 Communication-Aware Parallelization Strategies for High Performance Applications.
— [C10] HPCC12 MHPM Multi-Scale Hybrid Programming Model : A Flexible Parallelization Methodology.
— [W5] ERTS'14 Tackling Real-Time Signal Processing Applications on Shared Memory Multicore Architectures Using XPU

Sommaire

7.1 Introduction

L'année 2005 marque un tournant décisif dans l'histoire de l'informatique avec l'adoption généralisée des processeurs multicoeurs. Confrontés aux limites physiques de l'augmentation des fréquences d'horloge, les fabricants introduisent des processeurs équipés de plusieurs cœurs sur une seule puce, comme l'Intel Pentium D, l'AMD Athlon 64 X2 et le processeur Cell, développé conjointement par IBM, Sony

et Toshiba. Ce dernier a marqué les esprits, notamment parce qu'il est au coeur de la réussite industrielle de la Playstation 3 de Sony. Il est doté d'un cœur principal PowerPC (le PPE, ou Power Processing Element) et de huit unités de traitement symétriques (SPE, Synergistic Processing Elements), dédiées à des tâches de calcul hautement parallélisées. Cette structure offrait un modèle de programmation radicalement différent des processeurs traditionnels et a mis en évidence –si besoin était– l'importance du parallélisme pour obtenir des performances élevées. Bien que sa programmation soit complexe, le Cell a ouvert la voie à une plus grande adoption du parallélisme, y compris dans des produits destinés au grand public. Ce changement oblige les développeurs à abandonner leur dépendance à la simple augmentation des fréquences pour améliorer les performances, donnant lieu à la célèbre observation de Herb Sutter : "Free lunch is over." Ce constat souligne que l'optimisation logicielle doit désormais reposer sur l'exploitation explicite du parallélisme, marquant la fin d'une époque où les gains de puissance étaient obtenus sans effort supplémentaire de programmation. Cette révolution multicoeur redéfinit ainsi les bases du développement logiciel et ouvre la voie à de nouvelles avancées dans les systèmes embarqués, les consoles de jeu, les serveurs et le calcul haute performance.

7.2 Objectifs de XPU

L'histoire de XPU est celle de Nader Khammassi, qui a réalisé sa thèse CIFRE Thalès à l'ENSTA Bretagne. Je remercie Jean-Philippe Diguet (directeur de thèse) de m'avoir fait confiance quant au pilotage technique de cette thèse. L'idée clairement énoncée par Nader et Thalès (Alexandre Shrizniaz) était de développer un environnement de programmation des architectures multicoeurs du moment, en un simplifiant à la fois l'accès et plus encore la démarche-même de *parallélisation explicite*.

Parallélisation explicite Cette notion mérite qu'on y prête attention, car elle peut avoir deux acceptions différentes. Dans le premier cas, le plus courant, il s'agit simplement de coder *d'emblée* une application en profitant d'une API [1] pré-existante, qui permet de bénéficier du parallélisme présent dans les machines. C'est ce qui est fait lorsqu'on a recours à la programmation multithread ou multi-processus par exemple. Une seconde acception est toutefois possible : il s'agit de *transformer* une application *séquentielle existante* en une application parallélisée. Il s'agit d'une voie intermédiaire entre la première acception et une troisème forme de parallélisation : la parallélisation automatique. C'est bien cette seconde acception (le fait de transformer de manière explicite une application initialement séquentielle) qui a d'abord été étudiée dans la thèse.

7.2.1 Simplifier la parallélisation explicite

Pour bien comprendre le contexte, on peut se référer à deux bibliothèques largement utilisées pour la parallélisation : les classiques pthreads, et Intel Threading Building Block (TBB).

Pthreads Le manque de flexibilité du modèle de programmation PThreads entraîne de nombreuses modifications du code existant lors de la parallélisation d'un code séquentiel. En outre, un grand nombre de codes supplémentaires liés au paradigme de programmation sont généralement introduits, ce qui rend le code verbeux, moins lisible, sujet aux erreurs et difficile à maintenir. Cette mauvaise programmabilité accroît considérablement la charge du programmeur et rend la réutilisation du code séquentiel difficile. Un tel exemple est donné à titre illustratif : on essaie ici de réutiliser une fonction d'origine en tant que callback PThreads, le code séquentiel d'origine est considérablement modifié, car le code ciblé doit répondre au prototype de callback natif `"void * function(void *)"`, ce qui impose de nombreuses restrictions au programmeur : par exemple, seules les fonctions statiques peuvent être utilisées en tant que callback ; de même une méthode objet `C++` ne peut pas être utilisée directement. En outre, l'ensemble des données consommées ou produites par le callback doit être stocké dans une structure opaque intermédiaire commune (`void *`), puis extrait et restauré dans son type d'origine.

Intel TBB La bibliothèque d'Intel propose des abstractions de plus haut niveau sous forme de classes et d'objets pour représenter les tâches parallèles, les structures de données thread-safe (comme les queues concurrentes ou les hashmaps), ainsi que les outils de gestion des threads. Par exemple, les

1. API : application programming interface

```
1
2  void * callback(void * args)
3  {
4    // unpack arguments then
5    // call the original code
6  }
7
8  void main()
9  {
10   int   arg1, arg2;
11   float arg3;
12   pthread_t      id;
13   pthread_attr_t attr;
14   struct custom_struct args; // custom structure to pack all arguments
15   // pack arguments
16   args.first = arg1;
17   args.second = arg2;
18   args.third = arg3;
19
20
21   pthread_create(&thread, &attr, callback, (void *)&args);
22
23   // ...
24 }
```

FIGURE 7.1 – Principe de la programmation à l'aide des Pthreads Posix et de function "callback".

classes comme `tbb::parallel_for`, `tbb::task_group`, ou tbb : :flow : :graph encapsulent des concepts parallèles. Cette approche est assez commune (on la retrouve par exemple dans Cilk).

```
1  class task_1 : public tbb::task
2  {
3   public:
4     task_1(double * data_1, int data_2) : m_data_1(data_1), m_data_2(data_2)
5     {
6       // initialization
7     }
8
9     tbb::task * execute()
10    {
11       // ... function code ...
12       return 0;
13    }
14
15  private:
16
17    double * m_data_1;
18    int      m_data_2;
19 };
```

FIGURE 7.2 – Exemple de code Intel Threading Building Blocks (TBB)

Si cette programmation à l'aide TBB se veut efficace, modulaire et réutilisable, elle ne résout en rien l'épineux problème de *migration* d'un code séquentiel vers une machine multicoeurs. XPU cherche réellement à simplifier cette migration. L'idée est de recourir tout d'abord à la métaprogrammation afin d'encapsuler les fonctions `c` (ou méthodes `c++`), initialement pensées en terme séquentiel dans des *tâches* parallélisables. Dans un second temps, le programmeur peut *explicitement* organiser ces nouvelles tâches comme il l'entend.

```
// simple functions with different arguments count and types
int read(const char * file, int * stream)
{
  // code...
}
int sort(int * data, int size)
{
  // code...
}

int main()
{
  int  size
  int * data;

  // tasks definition
  task read_t(read, "file.dat", data);        // unnamed task
  task sort_t(sort, data, size);              // task with different argument types
  task named_sort_t("sort", sort, data, size); // named task

  // running the task
  read_t.run();
}
```

FIGURE 7.3 – Illustration de la création de tâches XPU à partir de procédures pré-existantes, et du passage d'arguments. On note qu'il est possible de passer le *nom* de la fonction initiale à la tâche créée, ce qui se révèle très pratique dans les faits (debug).

```
class image
{
    public:

      int sharpen(int val)
      {
      }

      int blur(int x, int y)
      {
      }
};

int main( )
{
   image img("img.jpg"); // object instantiation
   task sharpen_t(&img, &image::sharpen, 10);
   task blur_t("blur", &img, &image::blur, 4, 16);
}
```

FIGURE 7.4 – Illustration de la création de tâches XPU à partir d'instances d'objets c++ définis par des classes pré-existantes.

7.2.2 Compositions de tâches

Notre idée a été de faciliter la transition entre un code procédural traditionnel et la composition explicite de tâches. Cette composition se fait à l'aide d'appels de fonctions judicieusement nommées, et dont les prototypes sont suffisamment flexibles pour accepter les passages de paramètres des fonctions initiales à réorganiser.

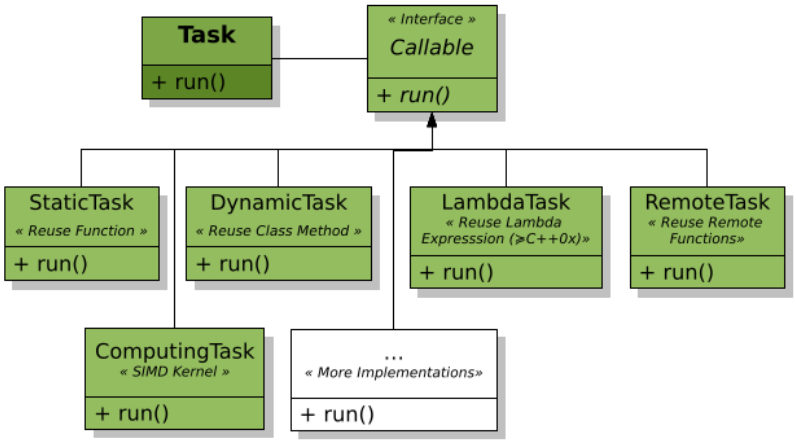
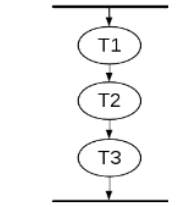


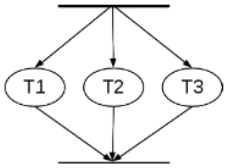
FIGURE 7.5 – XPU présente une hiérarchie variée de tâches internes, qui permet l'adaptation de codes séquentiels initiaux écrits avec différents *styles* : procédural, orienté-objet, fonctionnel (lambda expression), etc

```
1: void main() {
2:  task t1(f1, data_1), t2(f2) …;        // task definition
3:  task_group * program;
4:  program = sequential(&t1, &t2, &t3); // construct task graph
5:  program->run();                       // execute it
6: }
```



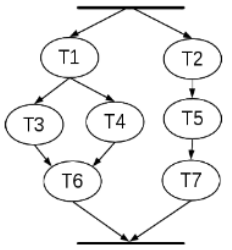
(a)

```
1: void main() {
2:  task t1(f1, data_1), t2(f2) …;        // task definition
3:  task_group * program;
4:  program = parallel(&t1, &t2, &t3);   // construct task graph
5:  program->run();                       // execute it
6: }
```



(b)

```
1: void main() {
2:  task t1(f1, data_1), t2(f2) …;        // task definition
3:  task_group * program;
4:  program = parallel(sequential(&t1,   // construct task graph
5:                                parallel(&t3, &t4),
6:                                &t6),
7:                     sequential(&t2,&t5,&t7));
5:  program->run();                       // execute it
6: }
```



(c)

FIGURE 7.6 – Un ensemble de tâches xpu est organisé en *groupes de tâches*, éventuellement appelés de manière hiérarchique, selon des motifs séquentiels-parallèles. Les tâches sont passées aux groupes de tâches par référence.

7.2.3 Fonctionnement interne et exécution

Runtime Le principe du runtime est présenté sur la figure 7.7 pour différents types d'organisation de tâches [2]. Les groupes de tâches capturés grâce au DSL (API de haut niveau) sont dispatchés en fonction de leur type.

Protection des données partagées Le runtime xpu est capable de détecter les types de dépendances de données à partir de la déclaration des prototypes de fonctions appelées (figure 7.8). Ceci est fondamental

2. On parle également de *parallel skeletons* dans la littérature.

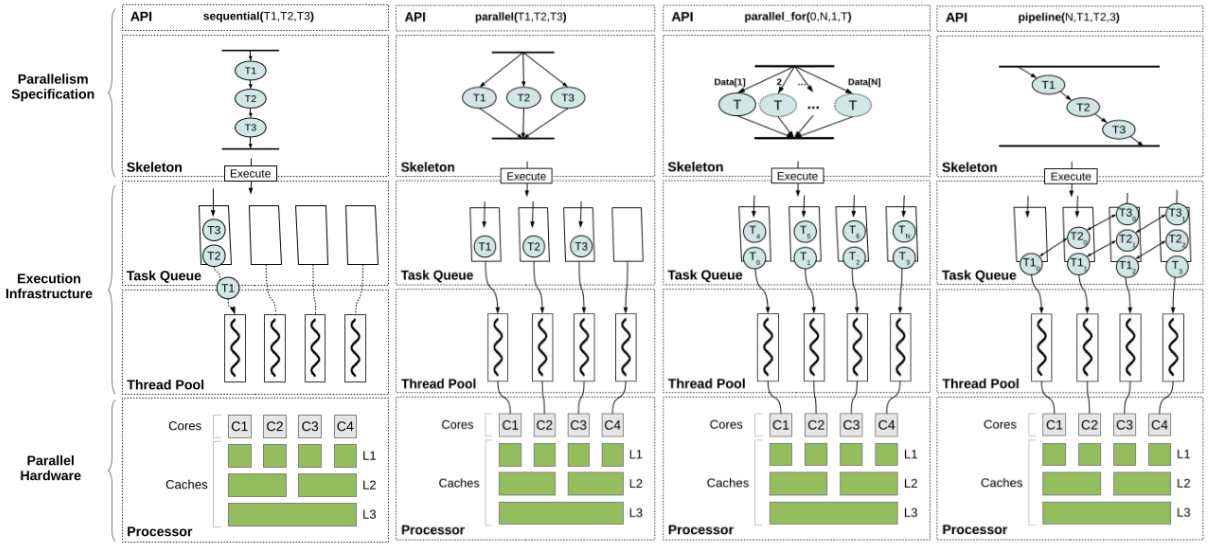


FIGURE 7.7 – Principe du runtime (IRTS) de `xpu` : cas des tâches séquentielles, parallèles, boucle `for` parallèle (parallélisme de données) et exécution pipelinée.

afin d'assurer la cohérence des données, potentiellement manipulées par des threads différents. `xpu` va donc protéger automatiquement les données concernées, à l'aide de sections critiques, comme illustré sur la figure 7.8.

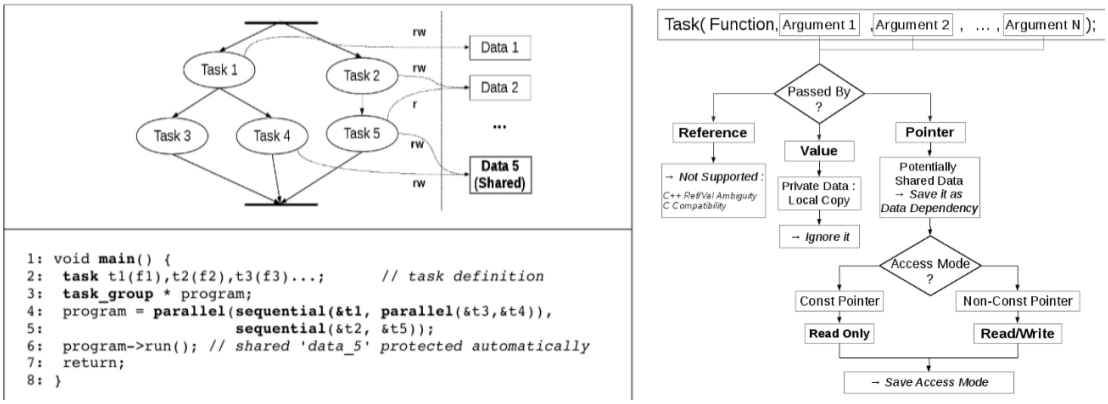


FIGURE 7.8 – Mécanisme de détection et protection des données partagées.

Introspection de la machine hôte `xpu` récupère également des informations précises quant à l'architecture sur laquelle il s'exécute. Dans [C9], nous avons ainsi expérimenté la prise en compte de la topologie des niveaux caches.

7.3 Exemple en traitement du signal Radar

7.3.1 Présentation de l'application

Dans [W5], nous avons proposé une application cible afin d'illustrer les capacités de `XPU` : un algorithme de traitement des signaux radar qui traite un signal numérisé d'un système radar à réseau phasé. L'application a l'intérêt de nous permettre des mesures de scalabilité sur une architecture multicoeurs. Le volume de données augmente de manière significative avec le nombre de canaux activés. L'algorithme effectue sa tâche en huit étapes. Ces étapes peuvent être résumées en trois étapes principales :

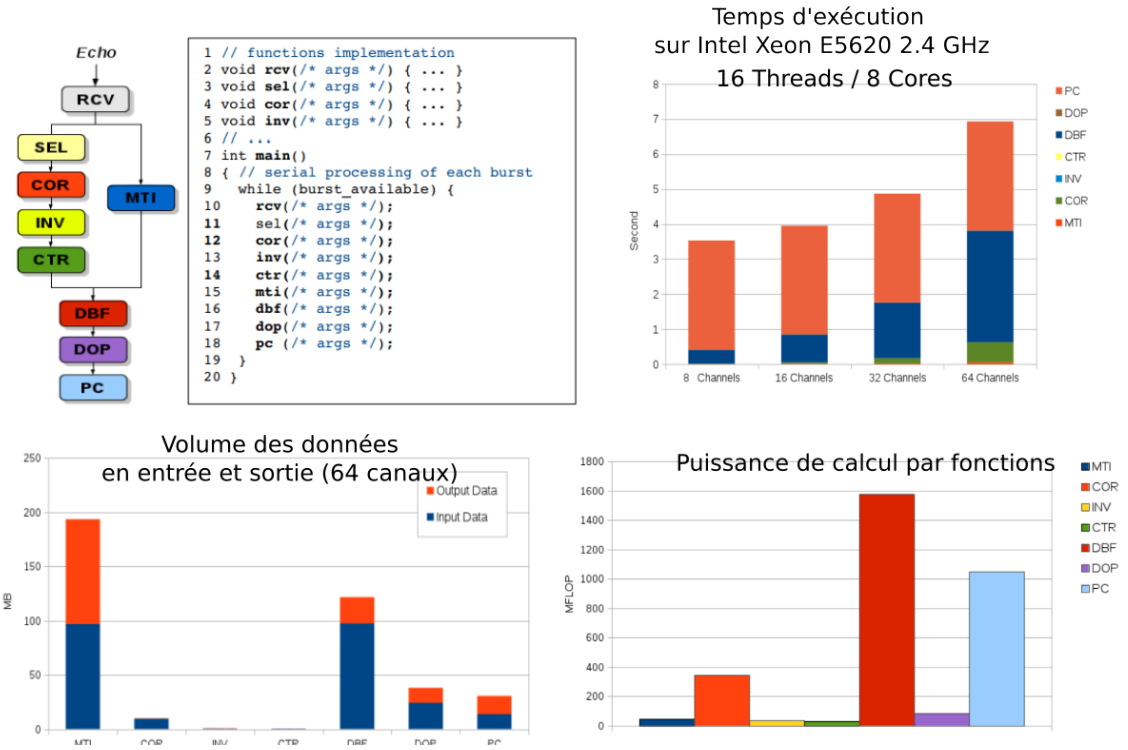


FIGURE 7.9 – Aperçu de la structure initiale de l'application Radar [W5], et des performances séquentielle sur architecture Xeon.

— Formation du faisceau numérique (vert)
— Filtrage Doppler (bleu)
— Compression des impulsions (rouge)

Les échos sont reçus sous forme de *bursts* périodiques. Pour chaque salve, les échantillons du signal reçu alimentent la chaîne de traitement du signal qui doit effectuer toutes les opérations nécessaires avant la réception de la salve suivante afin de respecter le traitement en temps réel. Les bursts sont répétés toutes les 20 ms. La figure 7.9 illustre la structure générale de l'application. La première fonction "RCV" est responsable de la réception des données du simulateur de radar et introduit donc une charge de travail négligeable. Les quatre blocs suivants : SEL (Sélection), COR (Corrélation), INV (Inversion) et CTR (Contrôle) effectuent une corrélation des canaux de réception et guident la formation de faisceaux numériques. Le bloc MTI (Moving Target Indication) permet la discrimination des cibles mobiles par rapport au fouillis stationnaire. Le DBF (Digital Beam Forming) forme des faisceaux en utilisant les canaux d'entrée traités. Le traitement Doppler est ensuite effectué à l'aide de ces faisceaux par le bloc DOP (Doppler) et enfin la compression des impulsions est achevée par le dernier bloc PC (Pulse Compression). Cette implémentation séquentielle de base prend environ 7 secondes pour traiter une salve de 64 canaux avec un nombre de faisceaux fixe sur une plateforme SMP avec deux Intel Xeon E5620 à 2,4 GHz. Elle est donc 350 fois plus lente que le temps de traitement en temps réel requis (20 ms).

7.3.2 Première itération : modélisation des tâches

La première itération consiste à la transformation directe de fonctions en tâches parallèles, ce qui est la vocation première de xpu. Le principe est illustré sur la figure 7.10, ainsi que les performances qui en découle. Sans surprise, cette seule transformation aboutit à des gains en performances modestes, qui ne suffisent pas à atteindre nos objectifs temps réel. Une seconde itération est nécessaire.

7.3.3 Seconde itération : vectorisation, boucles parallèle et pipeline

L'application Radar exhibe un parallèlisme de donnée important, que nous avons exploité grâce à la vectorisation SSE. Les résultats intermédiaires ne sont pas présentés ici (mais figurent dans [W5]), mais

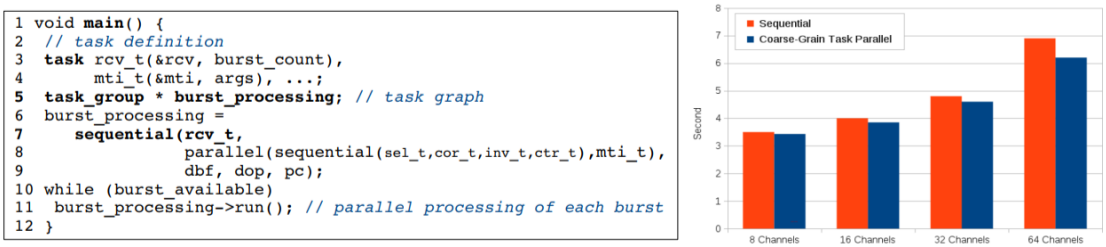


FIGURE 7.10 – Principe de la première itération : passage des fonctions aux tâches.

cette vectorisation fournit un gain substantiel, qui reste toutefois insuffisant. Il est nécessaire de profiter des boucles parallèles, mais également d'un pipeline savamment élaboré pour atteindre le temps-réel exigé. Les résultats sont présentés sur la figure finale 7.12.

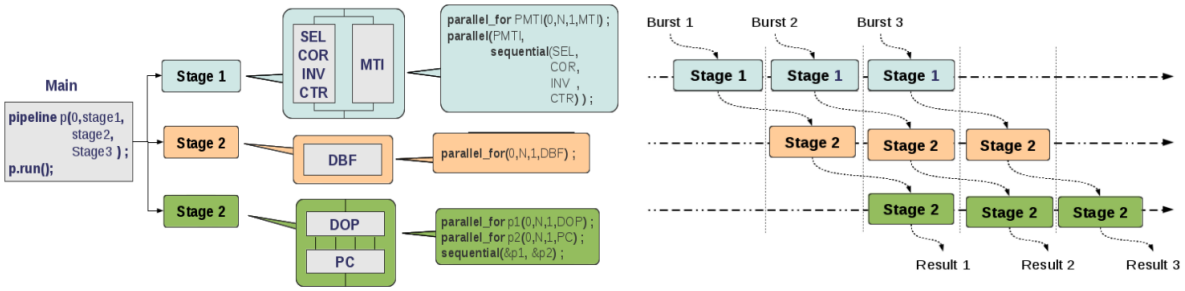


FIGURE 7.11 – Second itération : pipeline et vectorisation

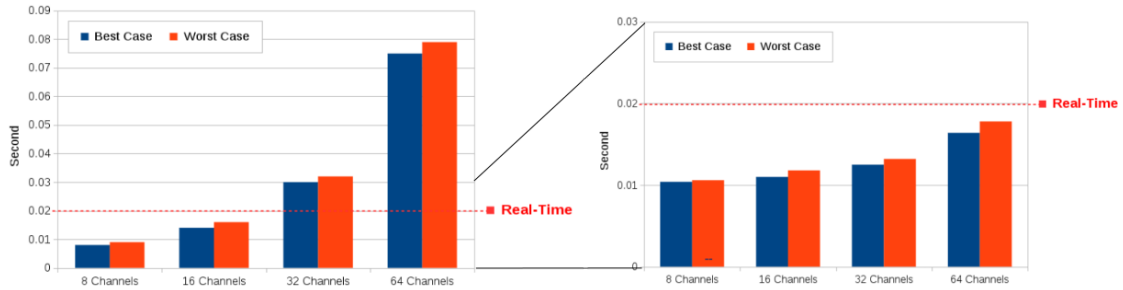


FIGURE 7.12 – Résultats finaux sur plateforme : pire et meilleur temps d'exécution de la version parallèle avec pipeline pour 8,16,32 et 64 canaux en burst sur l'Intel Core i7 Q720 1.6 GHz (8 Threads)

7.4 Parallélisation automatique

7.4.1 Contextes et objectifs

A l'inverse des travaux précédents, où l'on cherchait à faciliter l'expression *explicite* du parallélisme supposé connu du programmeur, la parallélisation automatique du code vise à répartir les tâches d'un programme entre plusieurs unités de calcul pour en accélérer l'exécution, sans intervention manuelle explicite du programmeur. Elle peut opérer à différentes granularités : au niveau des instructions, en exploitant des dépendances fines grâce à des techniques comme le parallélisme d'instructions (ILP), ou à un niveau plus grossier, en divisant des boucles ou des blocs de code indépendants sur plusieurs threads ou processus. Si l'idée semble prometteuse depuis les débuts de l'Informatique, elle se heurte à des défis persistants. L'analyse des dépendances est complexe, et il est difficile de garantir une répartition optimale des charges ou d'éviter des blocages liés à la synchronisation. Depuis les années 1960, de nombreux outils et langages ont tenté d'automatiser la parallélisation, mais avec des résultats souvent décevants

en pratique, en raison des spécificités du matériel et des caractéristiques imprévisibles des programmes. Cependant, cela reste un sujet hautement critique et d'un réel intérêt. Nous avons tenté de nous immiscer dans cette quête, avec beaucoup d'hésitations. Contre toute attente, nous avons obtenu des résultats de qualité, au prix d'un effort relativement modeste.

Principe général Le principe retenu est illustré sur la figure 7.13 : on cherche à transformer les fonctions en tâches, de manière systématique, puis à élaborer en interne un graphe de tâches sans cycle (DAG). Ce graphe de tâches est exécuté sur différents threads disponibles sur le système hôte.

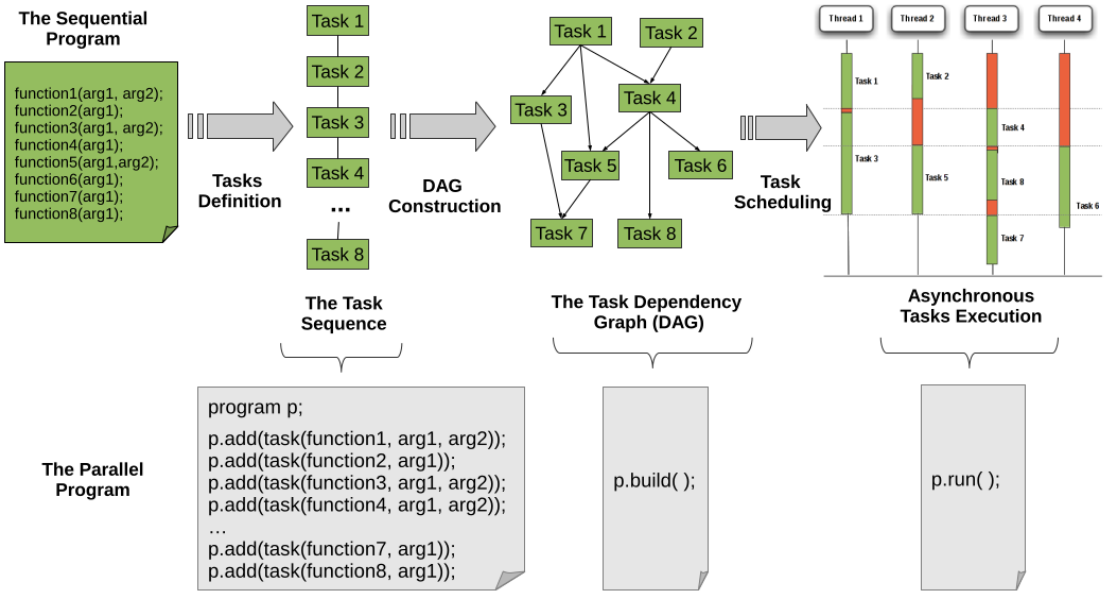


FIGURE 7.13 – Illustration du principe général de la parallélisation implicite : passage des fonctions aux tâche, élaboration d'un DAG et exécution sur cible multithread.

Procédure de construction du DAG La procédure de construction du DAG est illustrée sur la figure 7.14. Elle procède en deux étapes. La première consiste à déterminer les types d'accès aux variables (RAW, WAW, etc), grâce à une simulation parfaitement séquentielle. La seconde déduit les précédences effectives entre tâches.

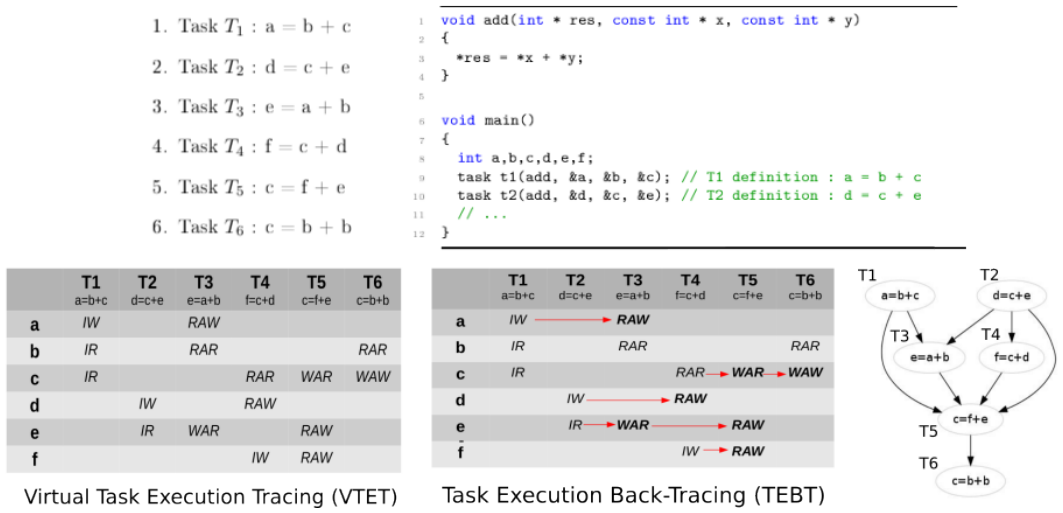


FIGURE 7.14 – Principes de construction du DAG

7.4.2 Expérimentations

Cholesky Nous avons expérimenté cette nouvelle forme de parallélisation sur la décomposition de Cholesky. Cette méthode d'algèbre linéaire est très utilisée en calcul haute performance (HPC) : sa méthode numérique est importante pour résoudre des systèmes d'équations linéaires, optimiser des calculs matriciels et traiter des problèmes de grande taille de manière efficace. La figure suivante illustre notre intention : notre but est une nouvelle fois de *minimiser l'écart syntaxique* entre le code source initial et séquentiel, et le code parallélisé. Comme on peut le constater, la structure générale des deux codes est similaire.

```
1  for (int j=0; j<nb; j++) {
2      for (int k=0; k<j; k++)
3          for (int i=j+1; i<nb; i++)
4              sgemm(a(i,k), a(j,k), a(i,j));
5      for (int i=0; i<j; i++)
6          ssyrk(a(j,i),a(j,j));
7      spotrf(a(j,j));
8      for (int i=j+1; i<nb; i++)
9          strsm(a(j,j), a(i,j));
10 }
```

Sequential Execution

```
1  program p;
2  for (int j=0; j<nb; j++) {
3      for (int k=0; k<j; k++)
4          for (int i=j+1; i<nb; i++)
5              p.add(task(sgemm, a(i,k), a(j,k), a(i,j));
6      for (int i=0; i<j; i++)
7          p.add(task(ssyrk,a(j,i),a(j,j));
8      p.add(task(spotrf,a(j,j));
9      for (int i=j+1; i<nb; i++)
10         p.add(task(strsm,a(j,j), a(i,j));
11 }
12 p.build();
13 p.run();
```

FATMA Parallel Code

FIGURE 7.15 – Code séquentiel (gauche) et code réécrit avec `xpu-fatma` (droite) pour la parallélisation : cette parallélisation est facilitée par le DSL de haut niveau, qui entraîne très peu de changement dans la structure du code

Comparaisons Nous avons utilisé `XPU-FATMA`, `Quark` et `SMPS` pour paralléliser la factorisation de Cholesky par tuile. Nous utilisons une implémentation séquentielle de base comme code de base commun. Cette implémentation commune est tirée des exemples de programmes fournis dans le package SMPSs v2.4. La taille des tuiles utilisée est de 200 pour toutes les configurations. Le programme Cholesky nécessite les bibliothèques LAPACK et BLAS. Nous avons eu recours à la bibliothèque Intel MKL version 10.3 qui implémente les routines BLAS et LAPACK. Nous utilisons le compilateur Intel `icc` (version 12.0.5) pour compiler chacune des trois implémentations parallèles. Les applications sont toutes compilées sur processeur Intel Core i7 Q720 (8 threads matériels et 4 cœurs physiques), sans les technologies spécifiques de Turbo Boost et Hyperthreading. Les résultats démontrent les excellentes capacités de `xpu`, qui s'avère compétitif par rapport à des frameworks très étudiés.

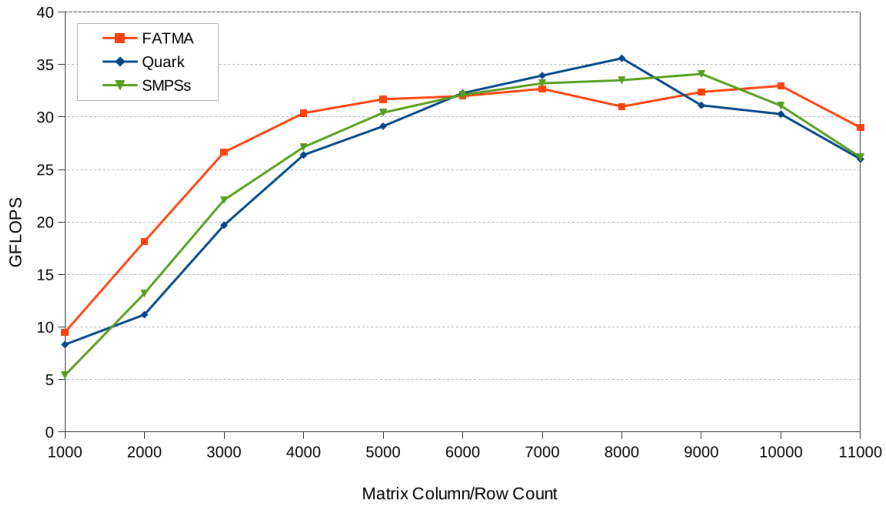


FIGURE 7.16 – Comparaison des exécutions de Cholesky sur machines parallèles entre `xpu-fatma`, Quark et SMPS

7.5 Parallélisation guidée par le profiling

Si notre démarche s'est révélée concluante, elle reste encore expérimentale et a ouvert de nouvelles pistes de réflexion. Dans une collaboration avec l'Université de Delft, nous avons ainsi exploré une parallélisation pilotée par un *profiling applicatif*. Dans [C1], nous avons ainsi proposé un couplage entre l'outil `MCPROF`, développé par Imran Ashraf et `xpu`, développé par Nader Khammassi. L'idée est de mesurer les échanges flot-de-données entre fonctions (langage C) : à la fois leurs fréquences et leur volumes sont enregistrés par l'outil de manière très précise. Ces informations sont cruciales pour déterminer la pertinence d'un choix de parallélisation. En particulier, j'ai développé une passerelle appelée `MXIF` (MCFROF-XPU inferface), qui assure la conversion entre les sources `C` initiales et `xpu`, après profiling. La figure 7.17 illustre cette chaîne d'outils. La figure 7.18 illustre le profiling réalisé par `MCPROF-XPU` et les résultats obtenus lors de l'exécution parallèle.

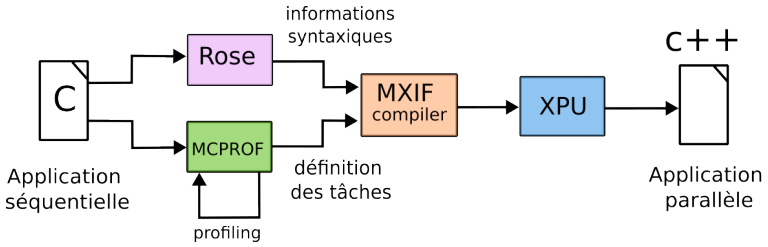


FIGURE 7.17 – Illustration de la chaîne d'outils développée entre TU Delft et L'ENSTA Bretagne, sur l'expérience de parallélisation guidée par le profiling

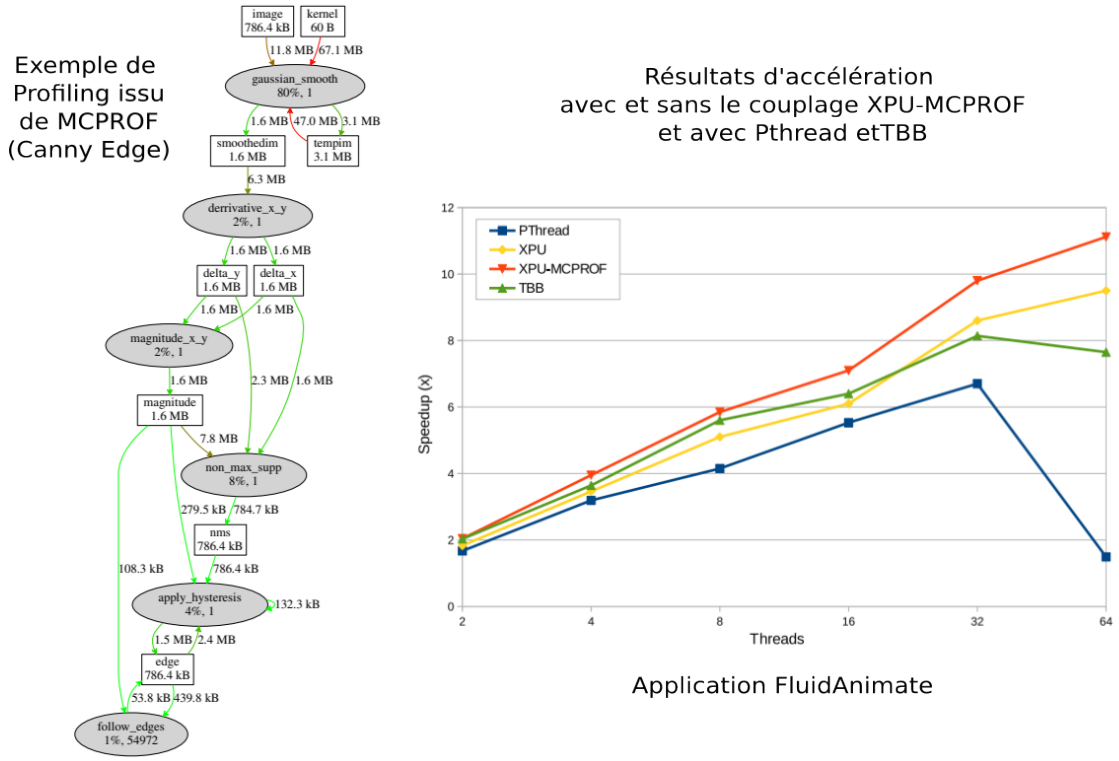


FIGURE 7.18 – Profiling réalisé par MCPROF (graphe de droite) avec les annotations de volumes des données véhiculées à chaque appel de fonction. Sur la droite, résultats comparés du couple `MCPROF-XPU` dans le cas de l'application `FluidAnimate` fondée sur les équations de Navier-Stokes

7.6 Conclusion

Ce chapitre a présenté un aperçu de mon activité passée dans le domaine de la parallélisation explicite et implicite sur architecture multicoeurs. L'encadrement de la thèse CIFRE de Nader Khammassi m'a permis, à partir de 2012, de me familiariser à ces technologies que je connaissais peu. La thématique de cette activité n'est pas à strictement parler associée aux SoC, elle partage un aspect commun avec les autres chapitres du manuscrit : cette activité est centrée sur la définition du DSL (ici `c++ XPU`) , qui permet de modéliser une intention précise du concepteur, et de l'accompagner dans des raffinements lors du portage sur la cible finale.

Troisième partie

Contributions aux aspects architecturaux

Chapitre 8

Synthèse comportementale dans `Archipel`

> I've asked many chip designers, "How did you get from the English to the logic design?" To some, I seemed to be asking how thinking works.
>
> Nick Tredennick
> IBM T. J. Watson Research Center (1981)

Publications et projets associés

— [C18] An HLS Algorithm for the Direct Synthesis of Complex Control Flow Graphs Into Finite State Machines with Implicit Datapath. DSD'24.
— [P3] Procédé de synthèse de haut niveau d'une application. Brevet 2011.
— [N4] Modélisation algorithmique et synthèse d'architectures assistees par model-checking.

Sommaire

8.1 Introduction

Ce chapitre présente quelques travaux autour de la synthèse HLS au niveau système, telle qu'envisagée dans `Archipel`. Deux aspects sont abordés :

— HLS orientée vers le contrôle à travers des FSMD *implicites*.
— Elaboration de nouvelles représentations de CDFG.

D'autres aspects liés seront présentés dans le chapitre suivant : synthèse des mécanismes de communication dans `Archipel`. L'ensemble vise à une synthèse comportementale au niveau système d'acteurs.

8.2 Synthèse HLS de FSMD *implicites*

La synthèse HLS orientée par le contrôle est un domaine probablement moins étudié que la HLS dominée par les données, mais qui intervient pourtant dans de nombreux systèmes numériques : applications de contrôle dans des systèmes de navigation etc, mais également dans la décompression vidéo, où le *parsing* du bitstream compressé est source de nombreuses difficultés calculatoires (CABAC peut notamment être cité). Il s'agit d'algorithmes fortement séquentiels, où le concepteur ne peut s'appuyer que sur très peu de parallélisme. Les prises de décisions interviennent à chaque cycle d'horloge et rendent les techniques de scheduling de graphes de flots de données inopérantes.

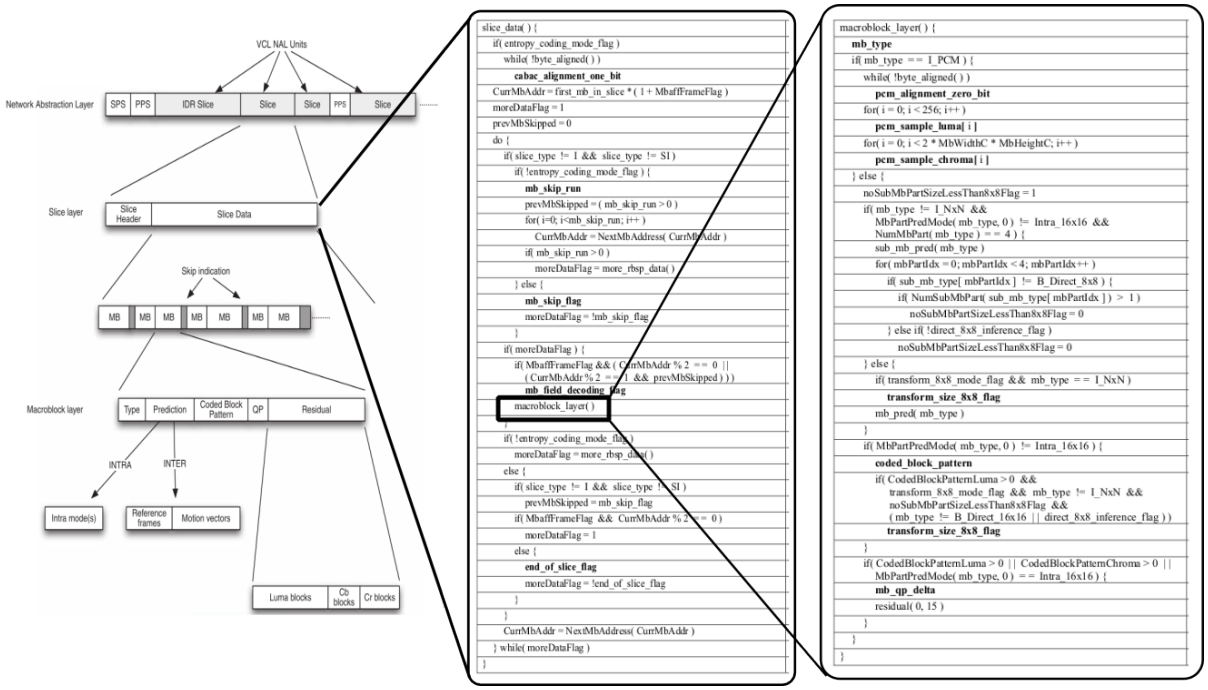


FIGURE 8.1 – Aperçu de la syntaxe du bitstream h264 et des besoins en décodage séquentiel : nous avons affaire ici à des machines d'états imbriquées (dont on peut chercher une mise à plat). Comme pour un parseur de langage, la rapidité de ce décodage a un impact important sur la performance globale du décodeur vidéo complet. Le standard h264 présente 17 pages de texte concernant la seule syntaxe (hors éléments lexicaux CABAC ou CAVLC).

Ce domaine de la HLS m'a particulièrement intéressé, car il est directement connecté aux représentations du contrôle que j'ai également pu aborder lors de ma thèse sur les langages synchrones. Une FSMD (Finite State Machine with Datapath) implicite est une architecture où la machine à états finis (FSM) et le chemin de données (datapath) sont intimement entrelacés, contrairement à une approche explicite où ils seraient clairement séparés. Dans ce type d'implémentation :

— Les registres du chemin de données font partie intégrante du codage des états.
— Les transitions d'états sont directement liées aux opérations sur les données.
— Le contrôle et le traitement des données sont fusionnés dans une même structure.

Définition Une FSMD est définie comme un 7-tuple $< Q, q0, I, O, V, f, h >$, où,

— Q est un ensemble fini d'états.
— $q_0 \in Q$ est l'état initial au reset.
— I est un ensemble fini de ports d'entrée.
— O est un ensemble fini de ports de sortie.
— V est un ensemble fini de variables internes.
— $f : Q \times 2^S \rightarrow Q$ est la fonction de changement d'état,
— $h : Q \times 2^S \rightarrow U$ est la fonction de mise à jour. Ici, S représente un ensemble de relations sur des expressions arithmétiques et logiques, et U reprsente un ensemble d'assignations de variables internes et de sorties.

Ces FSMD sont parfois présentées dans la littérature comme des *machines d'états finis étendues*, dans le sens où ces machines d'états finis peuvent opérer sur des variables internes. Les conditions de transitions peuvent alors s'exprimer non pas seulement en fonction des entrées, mais également en fonction de ces variables internes. On retrouve également, dans un grand nombre de livres d'introduction à l'électronique numérique, la notion de *flowcharts*, qui ne sont qu'une version graphique explicite du graphe de flot de contrôle.

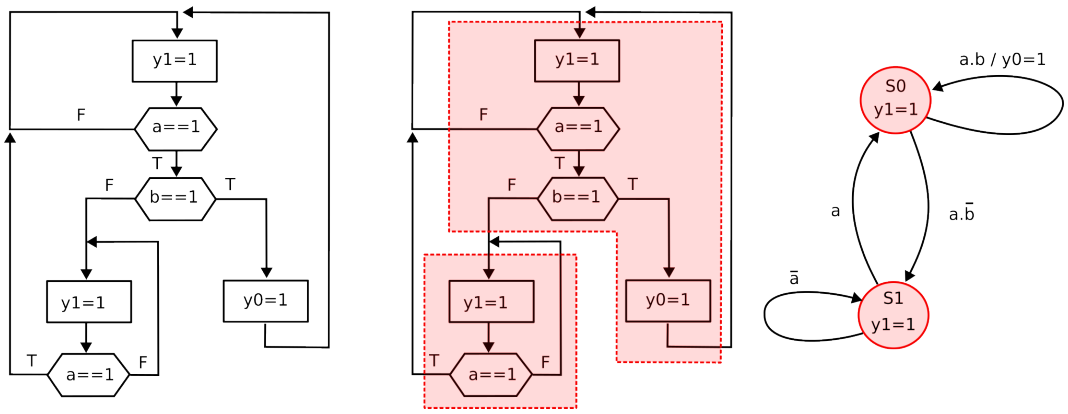


FIGURE 8.2 – Passage d'un *flowchart* à une FSMD (ici à 2 états) tel que souvent présenté dans des ouvrages d'électronique de base, mais jamais formalisé. Mon article DSD'24 [C18] explicite l'algorithme, dans le cadre de `Archipel`.

8.2.1 Syntaxe dans `Archipel`

Ces FSMD peuvent être écrites directement par l'utilisateur, ou inférés par `Archipel` à partir de l'IR (présenté chapitre 6).

8.2.2 Position du problème et cas illustratif

Afin de comprendre l'objectif de notre synthèse, observons le cas de la figure 8.3. Il s'agit d'un simple PGCD (Algorithme d'Euclide) entre deux variables a et b, dont les valeurs ont été lues sur les ports d'entrée d'un acteur (non représenté ici). Le compilateur établit une IR équivalente à ce comportement algorithmique séquentiel. Une fois cette IR obtenue, nous cherchons une exécution sous forme d'une FSMD, exprimée à l'aide d'un *nombre minimal* d'états. Cette FSMD est également représentée sur la même figure : deux états notés A et B suffisent. La version textuelle générée par `Archipel` est enfin écrite sur la droite du schéma. Quels principes ont régi l'obtentio de ces 2 seuls états ? Il s'avère que comme nous cherchons un automate matériel, ce sont les principes de conception numérique qui s'imposent : notamment, on cherche à isoler tous les comportements "locaux" ne présentant *pas de boucle combinatoire interne*. Bien que suggéré comme trivial dans la littérature, cet algorithme présente quelques difficultés.

8.2.3 Algorithme détaillé

L'algorithme (que nous avons publié à DSD 2024 [C18]) est présenté sur la figure 8.4.
Les grandes étapes de l'algorithmes sont les suivantes :

1. A partir du CFG initial, l'algorithme cherche les basic-blocks qui marquent le début d'une boucle. Cette procédure implique uniquement un parcours en profondeur d'abord (DFS). Les basic-blocks trouvés sont marqués par un point rouge sur la figure.
2. Une deuxième étape consiste à constituer le corps des états de la FSMD (state population) : à partir des états précédemment marqués, l'algorithme regroupe les états ayant un même point de départ, jusqu'à tomber sur un nouvel état marqué. Sur l'exemple, on obtient ainsi 3 états A, B et C regroupant respectivement 1,2 et 4 basic-blocks.
3. Toutefois, ce regroupement se révèle incorrect du point de vue des transitions entre basic-blocks associés à ces états. Sur l'exemple, il est ici impossible de transiter de l'état B à C précédemment

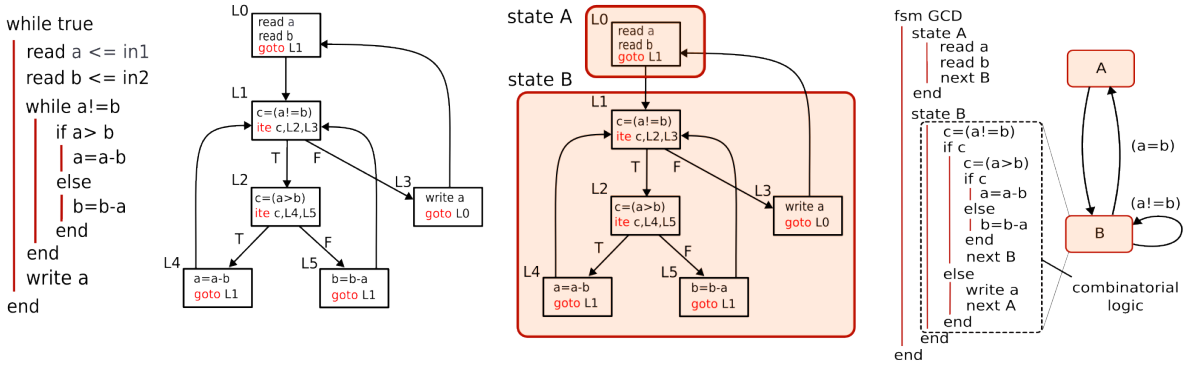


FIGURE 8.3 – Passage de l'IR à la FSMD : 2 états sont trouvés par notre algorithme.

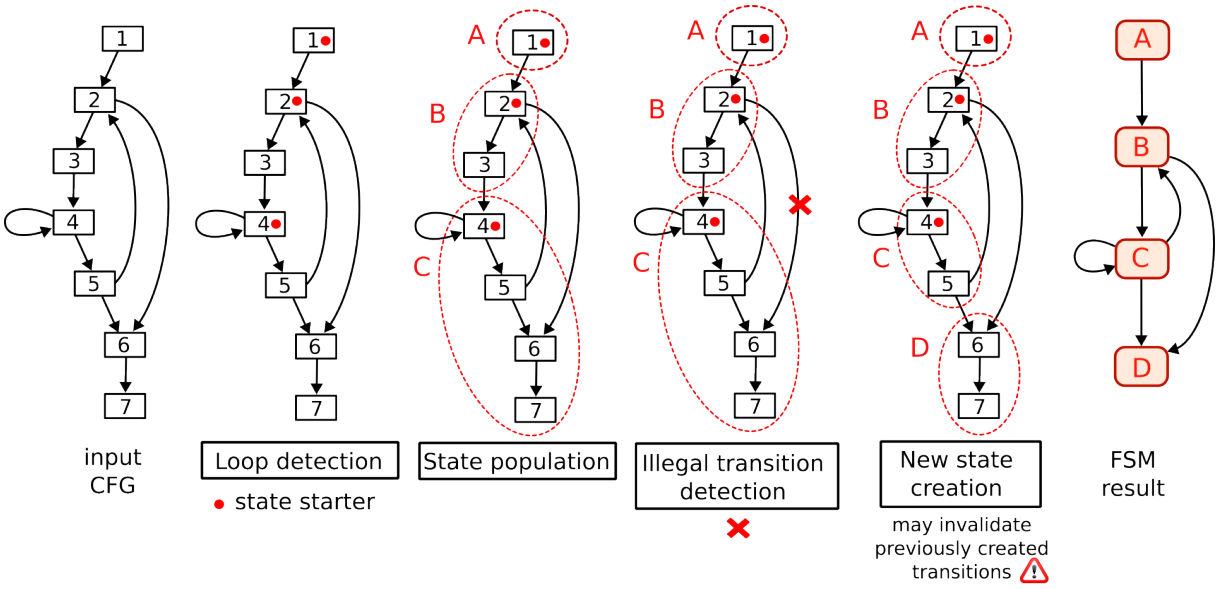


FIGURE 8.4 – Principe de l'algorithme de HLS présenté à [C18].

construits car une transition (marquée par une croix rouge), fait transiter du bloc de base 2 à 6, c'est-à-dire "au milieu" de l'état *C*. Cette transition apparaît impossible à formuler à l'aide du paradigme FSMD. Cette détection de transitions illégales conduit à la création d'un nouvel état *D* , qui se subtitue en partie à *C*.

4. Le procédé est alors itéré jusqu'à l'obtention de la FSMD finale (dessinée à droite).

Il est à noter qu'il est alors possible de regénérer le code "comportemental" de chacun des états, qui apparaissent comme des tests conditionnels (éventuellement profonds) et des assignations. `Archipel` réalise cette regénération textuelle, assez spectaculaire. Un exemple d'un tel code est donné section suivante.

8.3 Exemple

Le code suivant (gauche, appelé `algo.akp`) présente un programme `archipel` factice permettant d'illustrer l'effet technique remarquable de notre algorithme. Il présente des réceptions de données, des assignations, une conditionnelle ainsi que des boucles `while`. Le listing 8.2 (droite) présente le résultat de la FSMD implicite élaborée par notre algorithme (on peut rappeler qu'il s'agit également d'un pro-

gramme `archipel`). Chaque état de la FSMD présente typiquement un ensemble d'assignations conditionnées par des conditionnelles imbriquées. Les boucles initialement présentes de manière explicite ou implicite (`receive`) ont été transformées et mises à plat.

```
actor Algo
  input a : int
  def main
    var v1,v2,accu,i : int
    receive v1 <=a
    v2+=v1*i
    if v2==42
      v1=0
      while v1!=1
        receive v1 <= a
        accu+=v1
      end
    else
      while accu < 100
        case accu
        when 13
          accu*=3
        when 14
          accu+=4
        else
          puts("test")
        end
        puts("accu=",accu)
      end
    end
  end
end
```

CODES/algo.akp

Listing 8.1 – Exemple de structure de code orienté par le contrôle.

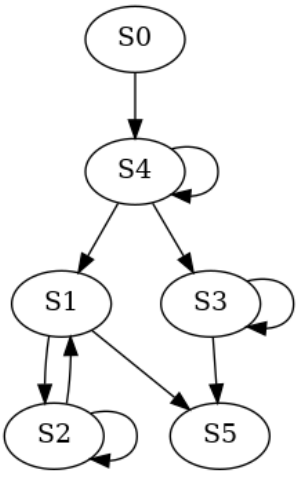


FIGURE 8.5 – Automate sous-jacent qu'il s'agit d'exhiber.

```
actor Algo
  input a : int

  def main() : nil
    var v1 : int
    var v2 : int
    var accu : int
    var i : int
    var $0 : bool

    state S0
      req a
      next S4
    end

    state S1
      $0 = (v1!=1)
      if $0
        req a
        next S2
      else
        next S5
      end
    end

    state S2
      $0 = ack?(a)
      if $0
        read v1 <= a
        unreq a
        accu = (accu+v1)
        next S1
      else
        next S2
      end
    end

    state S3
      $0 = (accu<100)
      if $0
        $0 = (accu==13)
        if $0
          accu = (accu*3)
        else
          $0 = (accu==14)
          if $0
            accu = (accu+4)
          else
            puts("test")
          end
          puts("accu=",accu)
          next S3
        end
      else
        next S5
      end
    end

    state S4
      $0 = ack?(a)
      if $0
        read v1 <= a
        unreq a
        v2 = (v2+(v1*i))
        $0 = (v2==42)
        if $0
          v1 = 0
          next S1
        else
          next S3
        end
      else
        next S4
      end
    end

    state S5
      stop
    end

  end
end
```

CODES/algo_fsm.akp

Listing 8.2 – Code `Archipel` déduit par notre algorithme, présentant de manière explicite l'automate.

8.3.1 Résultats sur benchmarks synthétiques

Nous avons mené une campagne de tests sur des benchmarks synthétiques : il s'agit de CFG générés aléatoirement et soumis à `Archipel`. Le compilateur calcule les FSMD et regénère un code approprié.

Les figures 8.6 et 8.7 rendent compte de ces expériences. La figure 8.6 représente l'évolution du nombre d'états trouvés par notre algorithme en fonction du nombre de basic blocks présents dans le

CFG : un peu plus de 3 basic blocs sont ainsi regroupés par états de la FSMD finale, sur les benchmarks synthétiques. La figure 8.7 montre quant à elle les temps de calculs mesurés lors de l'exécution de notre algorithme. Ces tests permettent de déterminer la complexité algorithmique de notre procédé itératif. La complexité mesurée est polynomiale en $O(n^{1.58})$, ce qui rend le procédé utilisable en pratique.

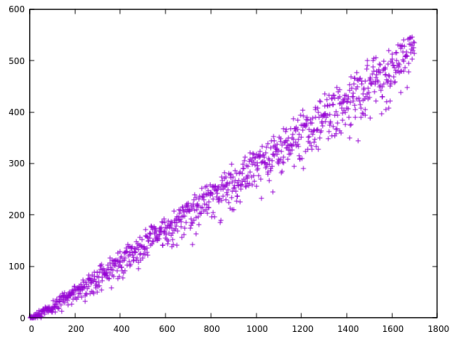


FIGURE 8.6 – Nombre d'états en fonction du nombre de basic-blocks

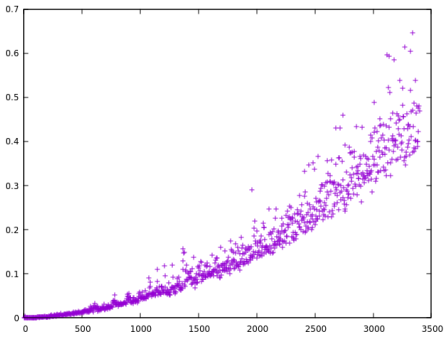


FIGURE 8.7 – Temps de calcul de la FSMD en fonction de la complexité cyclomatique du CFG

8.3.2 Discussion

Notre algorithme, qui compile un code comportemental en FSMD, correspond à une HLS sans contrainte matérielle, sans ordonnancement et sans allocation/binding. Cette HLS ne peut clairement se substituer à une HLS classique dans un cas général. Par contre, elle nous parait intéressante à plusieurs titres.

— **pédagogie** : tout d'abord très peu d'ouvrages, ni d'outils, ne montrent le résultat d'une transformation de code séquentiel riche en structures de contrôle.
— **code intermédiaire pour le raffinement et abstraction** : notre transformation permet de se rapprocher d'un code matériel en introduisant une première étape de *séquentialisation explicite*. Ceci semble se prêter à des activités de vérification formelle, autant dans le sens "descendant" (synthèse matérielle) que dans le sens "montant" (preuve d'équivalence, preuves de propriétés).
— **automates comme "objet de première classe"** : l'algorithme permet d'envisager de revisiter la notion compilation en réintroduisant les automates d'états finis comme "objet de première classe" : en effet, à notre connaissance, aucun compilateur ne fait le choix de représenter des programmes sont la forme retenue ici dans `Archipel` : ils sont basés sur les IR, comme LLVM ou Gimple et centrés sur l'analyse dataflow au niveau instructions ou basic-blocks. Il semblerait logique de réintroduire ce formalisme dans des chaînes de compilation traditionnelles.

8.3.3 Synthèse HLS orientée SSA de FSMD *explicites*

En tant que prototype expérimental, Archipel se révèle stable et peu sujet aux régressions. Il invite à de nouvelles explorations autour des transformations de modèles, orientées vers la synthèse des systèmes. Parmi elles, nous réfléchissons à une HLS complète : elle doit permettre de synthétiser des FSMD *explicites* où contrôleur et datapath sont clairement séparés comme il se doit [M24],[M18]. Plusieurs alternatives sont à l'étude :

— Recours à une HLS externe, comme Vivado HLS (aka Vitis HLS). L'attention particulière tient au respect des modèles de communications associés à Archipel.
— Conception d'une HLS traditionnelle, basée sur des CDFG d'ores-et-déjà à disposition dans Archipel, y compris en bénéficiant de leur forme SSA. Pour rappel, la forme SSA (Static Single Assignment) est une représentation intermédiaire utilisée en compilation, où chaque variable est assignée exactement une fois [M19]. Cela simplifie l'analyse et les optimisations du code. Plusieurs algorithmes permettant l'obtention de la forme SSA sont à disposition dans la littérature [M58]. `Archipel` embarque une telle passe.
— Conception d'une HLS traditionnelle, mais fondée sur l'élaboration d'un CDFG particulier, qui consiste en une mise à plat complète des opérations mêlant controle et données.

Nous donnons ici un aperçu des idées liées à cette dernière alternative. Il s'agit d'une représentation que nous avons appelé SCFG : Synchronous Control-Flow Graphs.

SCFG : Synchronous control-flow graphs Les SCFG visent à la *mise à plat* complète du graphe de flot de données et de contrôle (CDFG). Cette représentation est inspirée des travaux autour des langages synchrones[M38], et rejoint la représentation dite "gated-ssa" [M58]. Traditionnellement, on peut voir un basic-block comme un ensemble de calculs "liés" : un basic-block ne sera considéré comme terminé que lorsque toutes les instructions qu'il possède seront terminées. Il existe toutefois une manière orthogonale de considérer ces basic-blocks : au lieu de s'intéresser à la terminaison d'un basic-block on peut s'intéresser à son déclenchement à l'aide d'une *garde* (appelé "horloge" dans la littérature autour des langages synchrones). Chaque instruction des blocs de base est alors conditionnée par la *garde relative à chaque bloc de base*.

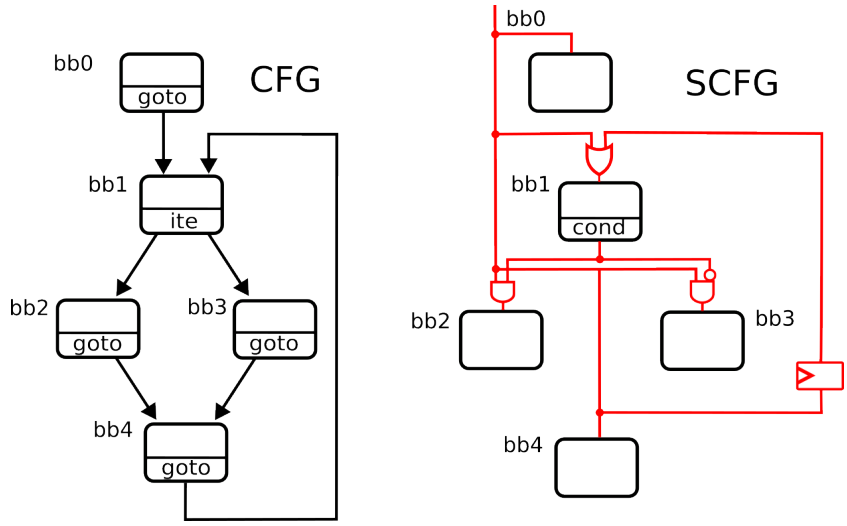


FIGURE 8.8 – Représentation alternative des CFG à l'aide de gardes synchrones d'activation de chaque basic bloc. Couplée à une forme SSA, les SCFG doivent permettre de "libérer" certaines instructions d'une séquentialité arbitrairement imposée lors de l'élaboration mécanique des CDFG traditionnels.

Pour en comprendre l'intérêt profond, considérons la figure 8.8, où l'on cherche à exhiber les gardes de chacun des blocs. On s'aperçoit que la présence d'une instruction `goto` entre deux blocks conduit à une *garde commune entre ces deux blocs* ; cela signifie qu'on peut déclencher simultanément l'*activité dataflow interne* de chacun des blocs de base. A l'inverse, la présence d'un branchement conditionnel entraîne logiquement l'entremise de deux conditions booléennes mutuellement exclusives. Jusqu'ici, ce changement de représentation semble annodin. Pourtant, on se rend compte que le changement de représentation implique la *suppression du* `goto` : en étant gardées par les conditions de déclenchement des blocs de base, chaque instruction se voit ainsi "libérée" des contraintes de séquencement arbitrairement insérées lors de la construction de l'IR. Cette "libération" des instructions individuelles est très intéressante, car elle permet de gagner en parallèlisme, au prix d'un changement de représentation aisé [1].

Pour nous en convaincre, considérons l'exemple suivant 8.9 : il s'agit d'un calcul matriciel présent dans le benchmark *3mm* de l'ensemble *Polybench*. Nous empruntons le schéma de [M30] : on trouve une première partie du CFG liée au calcul de $E = A * B$, puis de $F = C * D$, puis de $G = E * F$. On se rend compte que la capture séquentielle de ces calculs conduit à une sur-contrainte de séquentialité enre E et F. En réalité, ces calculs peuvent être réalisés en parallèle. Grâce au changement de représentation, ces dépendances peuvent être purement et simplement supprimées.

1. Notons toutefois qu'elle demande des aménagements concernant les accès mémoire, afin d'en préserver la cohérence lors des successions d'écritures-lectures

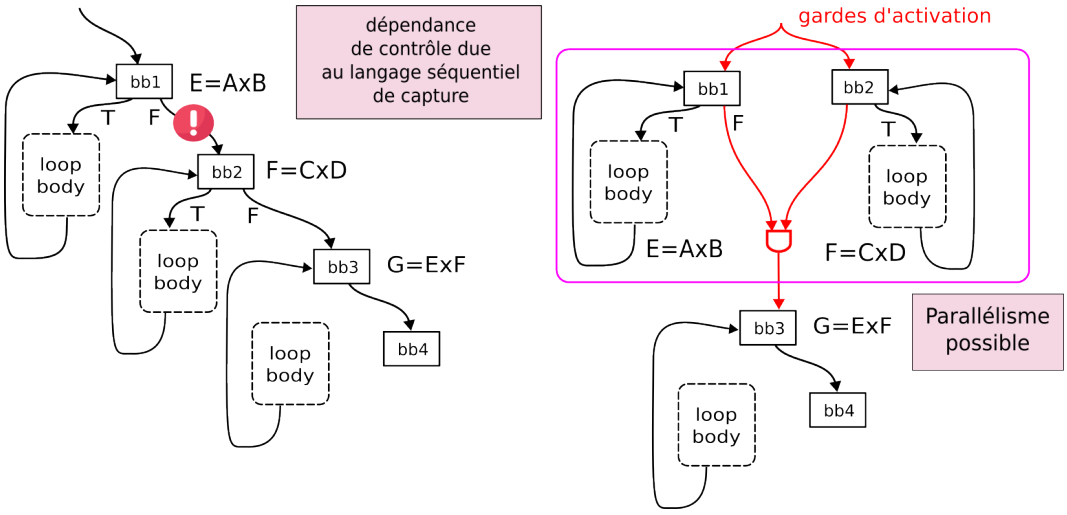


FIGURE 8.9 – Cas d'une multiplication de matrices. A gauche, CFG tel que construit à partir d'un langage séquentiel. Une dépendance de contrôle "bloque" l'expression du parallélisme potentiel de l'application. A droite : recours à des gardes d'activation des blocs de base, qui permettent l'exploitation du parallélisme (calcul des matrices `E` et `F`)

.

8.4 Conclusion

Ce chapitre présente un procédé de HLS particulier, expérimenté dans `Archipel` : il s'agit de générer un code RTL à partir de la seule prise en compte du graphe de flot de contrôle. Cette HLS est particulière dans le sens où l'analyse flot-de-données est ainsi totalement bypassée. Bien entendu, cette HLS orientée contrôle peut être amenée à échouer dans le cas où des calculs importants cohabitent avec le flot de contrôle : dans ce cas, il est nécessaire de basculer sur une HLS traditionnelle. Dans `Archipel`, deux voies sont alors à l'étude : soit le basculement vers un outil de synthèse tiers, comme Vivado HLS, soit la constitution complète d'une véritable HLS traditionnelle.

Chapitre 9

Mécanismes de raffinement des communications et des calculs

A model is a lie that helps you see the truth.

George E. P. Box

Publications et projets associés

— Projet ANR Mopcom

Sommaire

9.1 Raffinement pour le co-design HW/SW : principes et enjeux

Dans ce chapitre, je présente quelques aspects explorés en matière de *raffinement* pour la co-design logiciel-matériel. Pour bien comprendre l'intérêt de cette démarche de conception, on doit tout d'abord faire quelques rappels concernant le co-design.

L'approche traditionnelle est rappelée sur la gauche de la figure 9.1 : à partir d'une spécification de haut-niveau, une prise de décision amont a lieu, qui permet de séparer précocement les tâches et activités de conception logicielles et matérielles. Cette démarche est conduite par le sens pratique : dès lors qu'une interface technique est décidée en commun, la démarche (dite de *late binding*) permet à des équipes orientées logiciel et orientées matériel de s'organiser et réaliser leur travail de manière indépendante. L'ensemble du flot de conception est conduit par l'intégration finale, qui constitue une mise en commun des activités de part et d'autre. Cette démarche organisationnelle reste la plus communément rencontrée dans les équipes autour des SoC. Toutefois, cette démarche est risquée : une mauvaise prise de décision en amont ne se constate que tardivement, lors de cette intégration. Par exemple, il peut s'agir d'un mauvais partitionnement logiciel-matériel ou une conception maladroite de transfert de données entre les deux domaines. Ceci peut aboutir à des systèmes ne respectant pas la performance attendue. De plus, cette méthode perpétue des cultures isolées, au détriment de l'émergence d'une *véritable culture du système*. Une démarche alternative, que l'on peut qualifier d'"*holistique*", consiste à maintenir une représentation commune le plus tard possible dans le flot de conception : cette représentation n'est ni orientée logicielle, ni orientée matérielle, mais permet l'activation de transformations incrémentales. Cette démarche de raffinement progressif doit faciliter des retours arrière (round trips), et l'évaluation continue des bonnes propriétés du système : en ce sens, cette méthodologie alternative constitue une démarche *agile* de conception des SoC. Toutefois, cette nouvelle démarche de conception nécessite la

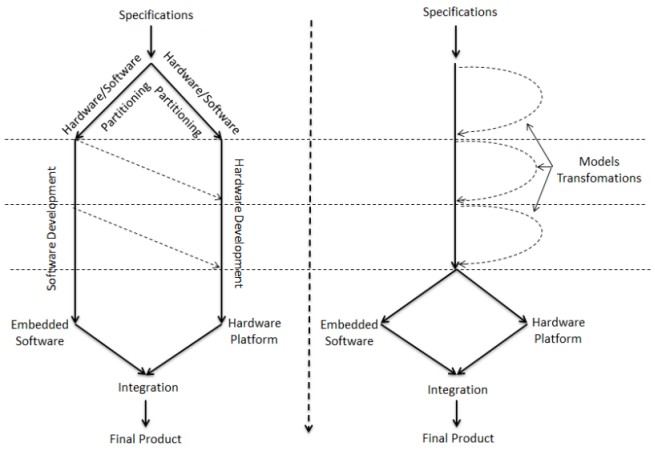


FIGURE 9.1 – Deux approches du co-design HW/SW : sur la gauche, approche classique avec séparation précoce des activités logicielle et matérielle. Sur la droite, approche défendue ici et expérimentée dans `Archipel` : raffinement progressif.

définition effective de cette représentation commune, ainsi que l'émergence de transformations significatives et génériques opérant sur cette représentation, si possible vérifiées formellement (ce que nous ne faisons pas ici, voir [M34]). Bien évidemment, cette démarche se doit d'être outillée et intelligible par des équipes formées à la démarche [M2]. La suite du chapitre illustre quelques idées autour de cette idée de *true* co-design HW/SW.

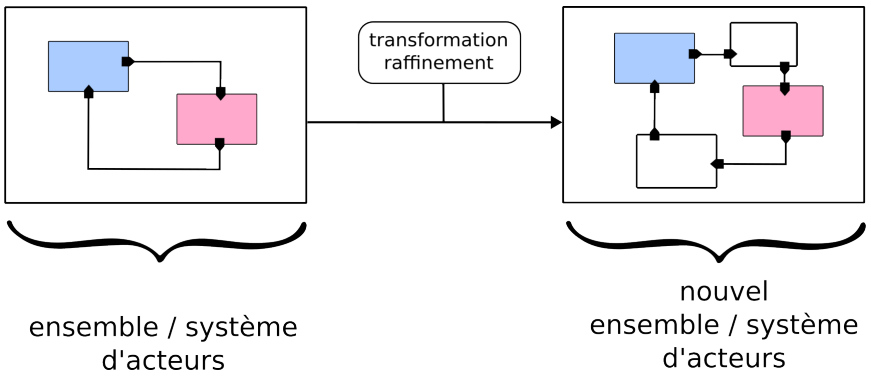


FIGURE 9.2 – Principe des transformations envisagées dans `Archipel` : un raffinement conduit à l'adjonction de nouveaux acteurs, exprimables dans la syntaxe et sémantique du langage.

9.2 Raffinement des communications dans `Archipel`

Raffinement des canaux FIFO et Rendez-vous synchrone La figure 9.3 présente un raffinement des communications, tel qu'implémenté dans `Archipel`. L'idée est simplement de passer d'une annotation comportementale, indiquant le MoC (et son éventuel paramétrage), retenu sur les canaux de communication, à une explicitation des artefacts techniques permettant l'exécution de ce comportement. Dans le premier cas décrit, il s'agit d'expliciter les mécanismes des FIFO et la connexion des ports associés aux acteur et ceux de ces FIFO. On voit ici clairement l'émergence d'une *nouvelle topologie* de connexion. Il en est de même concernant le schéma de droite, où le mécanisme de rendez-vous fait émerger un troisième composant, responsable de l'attente mutuelle de requêtes de la part d'un émetteur et d'un recepteur (il s'agit d'un automate à trois états). Dans ces deux cas, ainsi que pour une démarche générale, `Archipel` invite à une reformulation incrémentale de l'ensemble du système : dans le cas de l'exemple `Ping-Pong`, on passe de deux acteurs à quatre acteurs.

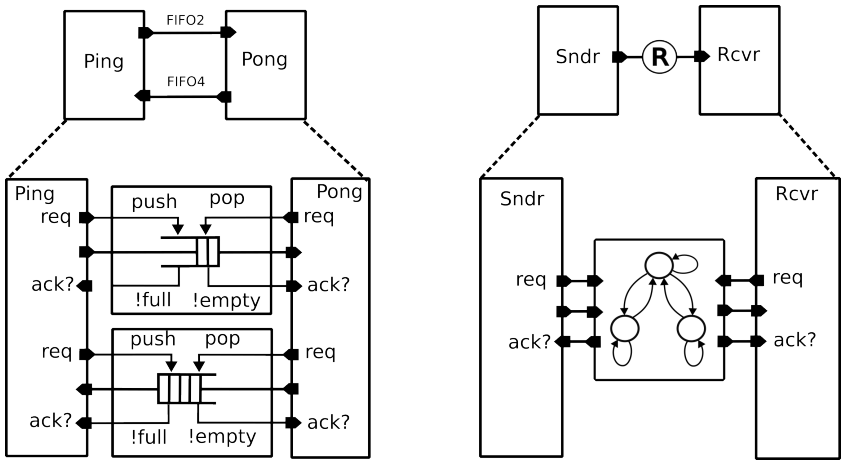


FIGURE 9.3 – Raffinement des communications dans `Archipel` : d'une simple annotation (”fifo2”, ”rendez-vous”, etc), on passe à un nouveau réseau d'acteurs.

Raffinement par acteurs DMA Une seconde transformation est présentée sur la figure 9.4. Elle vise à raffiner les principes d'accès à une mémoire externe, pour des volumes de données nécessitant un tel recours : là encore, le système d'acteurs initiaux se voit transformé en un nouveau système. Les nouveaux acteurs introduits assument ici les transferts effectifs sur une mémoire de type DDR ou SDRAM. Cette transformation n'a à ce jour pas été implémentée dans `Archipel`, mais l'outil est architecturé pour explorer de telles transformations de modèles.

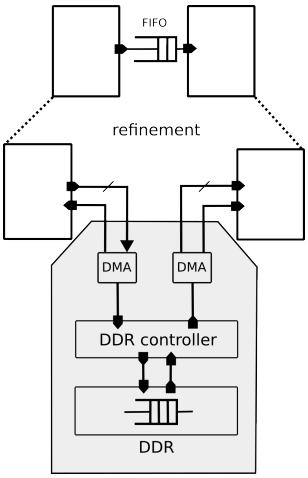


FIGURE 9.4 – Schéma de raffinement de stockage par FIFO : la transformation vise à réaliser la substitution d'une FIFO conceptuelle, au niveau application, par un nouvel ensemble d'acteurs assurant l'accès à une DDR externe par canaux DMA.

9.3 Raffinement des structures de calcul dans `Archipel`

L'idée du raffinement paraît relativement claire et maîtrisable dans le cas des communications, présentées dans les sections précédentes. Qu'en est-il des calculs ? Bien entendu, on peut inclure la HLS elle-même dans l'idée du raffinement. Toutefois, ici l'idée défendue n'a pas trait à la HLS, mais à de nouvelles transformations de modèles, qui permettent de passer des acteurs abstraits et orientés-application, tels que capturés dans `Archipel`, vers des acteurs *orientés-plateforme* : le réseau initial d'acteurs comportementaux doit pouvoir être *organisé* au regard d'éléments de plateforme concrète. Par exemple, si on s'alloue deux processeurs RISC, quel sera l'impact des choix de mapping des acteurs initiaux sur ces deux processeurs, ainsi que sur leur interconnexion, caractérisée par une bande passante, des proto-

coles, etc. Il s'agit donc d'une exploration architecturale incrémentale, qui vise à couvrir un espace de décision qui va bien au dela de la seule HLS.

Syntaxe du mapping des acteurs Dès lors que le réseau d'acteur est supposé fonctionnel, il est possible d'annoter ces acteurs à l'aide d'indications de mapping sur des processeurs : dans la méthodologie, ces processeurs restent *abstraits*, mais on fait l'hypothèse qu'ils ne présentent pas de parallélisme à gros grain : il s'agit de processeurs séquentiels, conventionnels (RISC). Le mapping d'un certain nombre d'acteurs sur de tels processeurs signifie la mise en place d'ordonnanceurs. Là encore, il s'agit de *mécanismes simples* plutôt que de solutions technologiques précises. Cette démarche de rafinement permet de se rapprocher d'une solution technologique, sans engagement définitif.

```
system Sys
  instance A1 : Actor1  mapped on PA
  instance A2 : Actor2  mapped on PA
  instance A3 : Actor3  mapped on PB
end
```

Dans cet exemple, on s'alloue 2 processeurs 1 et 2. Ce simple mapping des calculs entraîne des modifications des performances du système, et de son comportement temporel : nous cherchons à travers *archipel* à évaluer, par simulation, les impacts de ces choix incrémentaux.

Exemple de simulation A titre d'illustration, la figure 9.5 présente l'impact d'un tel mapping au regard du comportement initial d'acteurs communiquants.

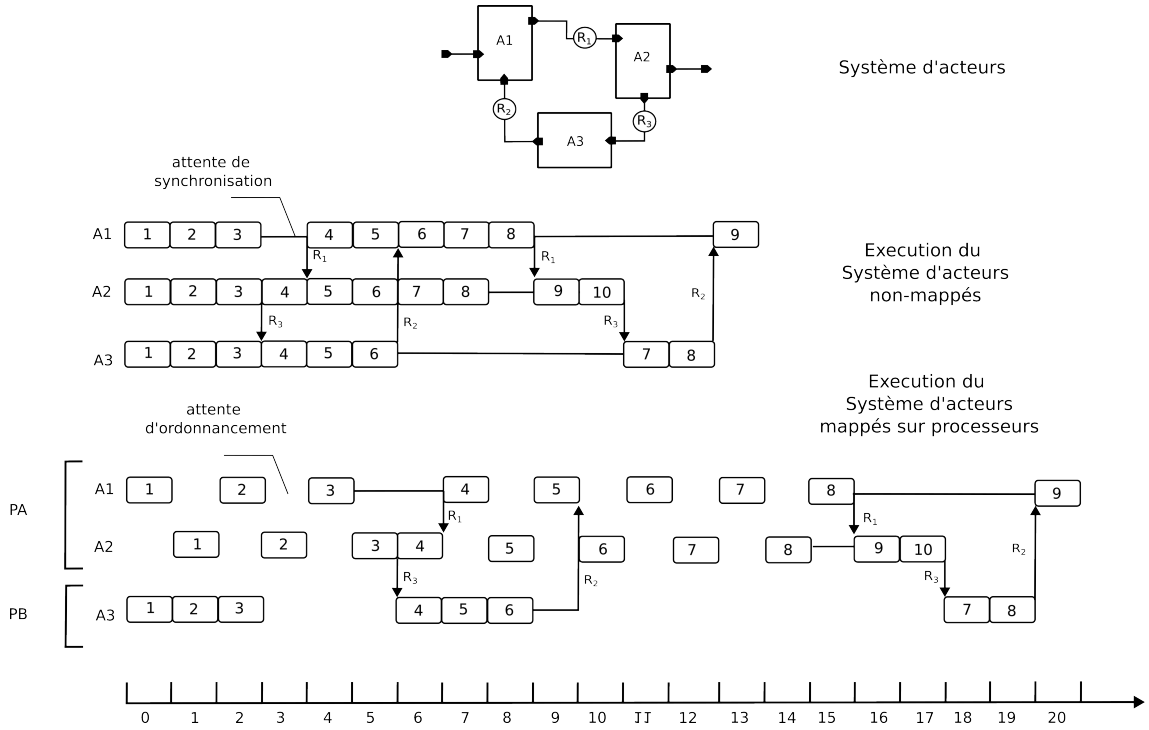


FIGURE 9.5 – Impact du mapping sur processeur sur l'exécution d'un réseau d'acteurs.

Proposition d'algorithme de simulation du mapping L'implémentation de la simulation d'un tel mapping est actuellement incomplète dans `Archipel`. Le projet ANR Dynnamo permettra de le développer et de l'expérimenter sur le cas d'applications de réseaux de neurones dynamiques (capables de s'adapter en fonction d'un contexte). Toutefois, afin d'évaluer la complexité d'une telle mise en oeuvre, on peut dès à présenter illustrer l'idée. On rappelle qu'à chaque cycle de simulation, l'ensemble des acteurs avance d'une "étape" : selon la granularité retenue, cette étape est soit un basic bloc complet, soit un

ensemble de basic blocks constituant un état de la FSMD. L'algorithme de simulation du mapping sur processeur consiste à faire avancer chacun des acteurs mappés sur un même processeur, chacun à tour de rôle. Il suffit de conserver l'historique de l'exécution sur ce processeur : un buffer circulaire est par exemple possible.

```
def visitInstance actor
  processor = actor.get_processor
  if processor.get_runnable_actor==actor
    actor.resume # makes the actor co-routine progress in its execution
    processor.shift_runnable_actors # schedules next actor
  end
end
```

9.4 Raffinement des fréquences des acteurs

Un dernier raffinement est possible dans `Archipel` : il s'agit de préciser les *fréquences propres relatives entre acteurs*. Il est en effet aisé d'attribuer de telles fréquences propres à chacun des acteurs, que cela soit en simulation fonctionnelle ou au niveau matériel, où cela est encore plus naturel. On rejoint ici l'idées des GALS, architecure "globally asynchronous, locally synchonous". Les GALS sont bien plus qu'un simple compromis entre systèmes synchrones et asynchrones. Dans une démarche de System-Level Design, c'est un levier méthodologique qui :

— Structure l'architecture autour de modules indépendants, ce qui est crucial pour le design de systèmes hétérogènes.
— Facilite le raffinement progressif en permettant d'introduire progressivement des contraintes d'implémentation.
— Optimise la consommation et la flexibilité, ce qui le rend essentiel pour les architectures MPSoC, NoC et CPS.

Il s'intègre donc parfaitement dans une approche ESL to RTL, en servant de modèle de transition entre un modèle transactionnel abstrait et une implémentation matérielle détaillée.

9.5 Conclusion

Ce chapitre illustre l'idée de raffinement d'un système décrit par un réseau d'acteurs. Ce raffinement vise à introduire incrémentalement des précisions sur les choix d'architecture. Ce caractère incrémental présente plusieurs avantages :

— Définition mécanique du processus de conception : il s'agit d'éliciter les transformations nécessaires au passage progressif d'un système abstrait vers un système final.
— Constitution d'un corpus de mécanismes comportementaux : il s'agit de capitaliser des IP comportementales représentatives des fonctions classiques essentielles aux SoC (FIFO, DMA, etc). Le cas des mécanismes d'interruption pourrait ainsi être étudiés dans ce cadre. L'enjeu est de pouvoir élever le niveau d'abstration du niveau RTL au niveau comportemental. Une même approche pourrait être défendue concernant les processeurs eux-mêmes : la constitution d'acteurs simulant un processeur pourrait être ainsi envisagée. Plusieurs travaux vont actuellement en ce sens dans le domaine de la HLS.
— Possibilité de retour arrière : bien que non discuté ici, il s'agit de pouvoir inverser le mécanisme de raffinement, et s'autoriser à un mécanisme d'abstraction.

Idée

La modélisation comportementale, associée à l'idée de raffinement, permet d'envisager un écosystème où toutes les fonctions utiles à la constitution d'un SoC sont capturées de manière uniforme, sous la forme d'acteurs fonctionnels organisés en flot de données : ces fonctions peuvent être calculatoire et très spécifiques, mais également plus traditionnelles (DMA, processeurs, etc).

Chapitre 10

Mécanismes reconfigurables RTL pour overlays et eFPGA

The Network is the Computer.

slogan de Sun Microsystems, Inc

Publications et projets associés

— [Z2] Thèse de Théotime Bollengier : Du prototypage à l'exploitation d'overlays FPGA.
— [C15] FPGA4GPC'17 : A cost-effective approach for efficient time-sharing of reconfigurable architectures.
— [C4] ARC 2017 : Soft timing closure for soft programmable logic cores : The ARGen approach
— [C7] ReCoSoC'18 : An Integrated toolchain for Overlay-centric System-on-chip

Sommaire

10.1 Introduction

Je présente ici une modélisation RTL de mécanismes électroniques reconfigurables de base, que nous avons pu étudier et concevoir dans le cadre de la thèse de Théotime Bollengier, menée à l'IRT B-COM. Ces mécanismes constituent le socle de deux types de circuits que nous avons réalisés en laboratoire : des `overlays FPGA` ainsi que des `eFPGA`. Mon apport a été une impulsion initiale dans la conception des LUT,

CLB, switches et routeurs, résumés dans ce chapitre. Ils ont l'originalité d'avoir été conçus au niveau RTL et non au niveau transistor comme c'est le cas le plus souvent. Il est ainsi possible d'en banaliser totalement la conception. Le résultat (overlay) exhibe pourtant toutes les bonnes caractéristiques d'un FPGA...qui s'exécute sur FPGA. Notre but initial était de gagner en portabilité : notre FPGA virtuel (vFPGA) préserve une compatibilité binaire quel que soit son hôte.

La suite de ces travaux a été fructueuse, car nous avons également pu aller jusqu'à la conception d' une puce dédiée, appelée *Elnath*, fondue chez TSMC par l'intermédiaire de la société grenobloise *IC'Alps*.

L'avénement de ces dispostifs reconfigurables modifie totalement le paysage de l'Electronique. Mon expérience industrielle en tant que concepteur "de terrain" m'a maintes fois confronté à ce besoin en matière de reconfiguration. Plusieurs cas se sont présentés :

— Incertitude ou doutes sur l'algorithme à implanter matériellement (e.g standards vidéo en cours de négociation par des consortium,...)
— Absence pure et simple de l'algorithme.
— Complexité importe de l'algorithme, qui laisse planer la menace d'un bug non détecté en phase de vérification.
— Nécessité de remplacer dynamiquement un algorithme par un autre (cas des décodeurs multi-standard).
— etc

10.2 Mécanismes de base des systèmes reconfigurables

10.2.1 Notion d'overlay à grain fin

Notion d'overlay FPGA Un overlay FPGA est une couche d'abstraction placée au-dessus du matériel FPGA brut. Il s'agit en fait d'une architecture préconfigurée qui facilite la programmation et l'utilisation de l'FPGA en proposant une interface plus haut niveau. Les CGRA synthétisés sur FPGA constituent par exemple un tel overlay. Cependant, nous nous sommes plutôt intéressés aux *overlays à grain fin*.

Overlay à grain fin Il s'agit ici pour nous de répliquer les mécanismes logiques présents dans les FPGA, mais selon les techniques de modélisation RTL standards. On conçoit ainsi une architecture numérique "comme une autre", sauf que sa fonction est précisément celle d'un FPGA. On peut d'ailleurs, ainsi, parler de *FPGA virtuel*, qui traduit probablement mieux l'idée dévéloppée. Cependant, cette reconfigurabilité présente un coût intrinsèque, que nous allons sommairement évaluer ici.

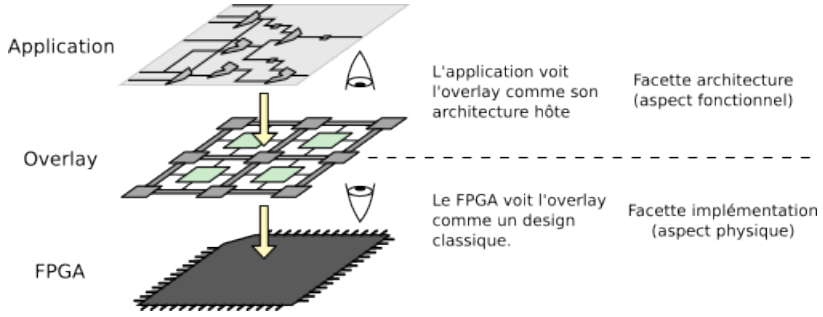


FIGURE 10.1 – Illustration du concept d'overlay [Z2] : une couche d'abstration qui s'interpose entre l'application et l'hôte physique.

10.2.2 Modélisation RTL des *look-up tables* (LUT)

Principes

Les *look-up tables* (LUT) de taille n permettent de réaliser n'importe quelle fonction booléenne $f(x_0, \ldots, x_{n-1})$ de n variables booléennes. Ils peuvent être vus comme une "table de vérité matérielle". La modélisation d'un tel dispositif, en technologie RTL classique, consiste à connecter un registre à décallage constitué de

2^n bascules D avec un *enable* commun, et un multiplexeur à 2^n entrées. Ce multiplexeur possède comme signal de commande les n variables booléennes $x_0, \ldots, x_{n-1}$. Le listing 10.1 présente l'intégralité d'un tel modèle générique. La simplicité de ce dispositif permet aux étudiants de première année de comprendre le mécanisme de reconfiguration : le bitstream est inséré par le registre à décallage et on imagine facilement le *serpentin* qui chemine le long de tous les registres de configuration. Selon la configuration bloquée dans les registres, le dispositif réalise la fonction booléenne escomptée.

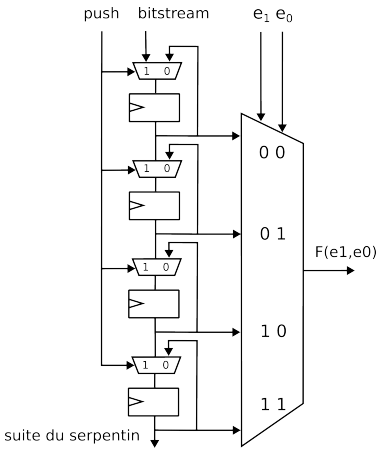


FIGURE 10.2 – Modélisation RTL d'une LUT à deux entrées (LUT2)

Coût matériel

Le coût matériel de cette modélisation reste toutefois important. Prenons le cas d'une LUT_2 (c'est-à-dire à à 2 entrées). Si on considère qu'une bascule D équivaut à 10 portes logiques et qu'un multiplexeur à 2 entrées équivaut à 4 portes, un calcul *cunutesque*[1] conduit à 68 portes logiques. Cela signifie qu'il faut 68 portes logiques pour éventuellement ne configurer qu'une expression booléenne à 2 entrées (et notamment une seule porte logique `and`,`or`, `xor`, `nand`,etc). Ce ratio de 68 explique les réticences initiales de la Silicon Valley à financer le développement des premiers FPGA.

Code VHDL générique

```
1  library ieee;
   use ieee.std_logic_1164.all;
3
   entity lut is
5    generic(n : natural);
     port(
7      reset_n,clk : in std_logic;
       bitstream   : in std_logic;
9      push        : in std_logic;
       inputs      : in std_logic_vector(n-1 downto 0);
11     f           : out std_logic
     );
13 end entity;

15 architecture rtl of lut is
     signal reg : std_logic_vector(2**n-1 downto 0);
17 begin

19   shift_reg : process(reset_n,clk)
     begin
21     if reset_n='0' then
         reg <= (others =>'0');
23     elsif rising_edge(clk) then
         if push='1' then
25         reg(0) <= bitstream;
```

1. Néologisme dû à Alain Guyot, INPG 1994!

```
          for i in 1 to 2**n-1 loop
27          reg(i) <= reg(i-1);
          end loop;
29      end if;
      end if;
31  end process;

33  f <= reg(to_integer(unsigned(inputs)));

35 end architecture;
```

Listing 10.1 – modèle VHDL d'une LUT à n entrées, modélisée au niveau RTL.

10.2.3 Modélisation des *Basic logic element* (BLE) et *Configurable logic block* (CLB)

Un BLE (Basic Logic Element) se compose d'une LUT, d'une bascule D et d'un nouveau multiplexeur permettant de configurer la sortie de la LUT comme le résultat d'une fonction combinatoire ou séquentielle. Ces BLE ont été assemblés en CLB figure 10.3.

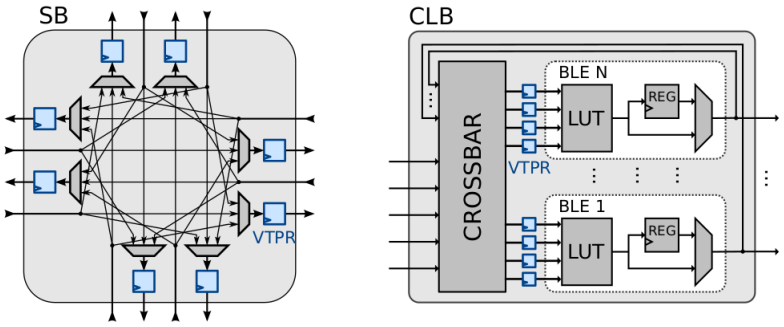


FIGURE 10.3 – Switch Box et CLB synthétisés au niveau RTL : on remarque la présence de registres appelés VTRP (en bleu), qui assure la synthétisabilité RTL sur l'hôte FPGA physique. Schéma issu de la thèse de Théotime Bollengier [Z2].

10.2.4 Modélisation des *Connection Box*

La "Connection Box" est l'interface qui permet de connecter les entrées/sorties d'un CLB aux canaux de routage adjacents. Elle contient des points programmables qui permettent de choisir quelles lignes de routage seront connectées aux pins du CLB.

10.2.5 Architecture générale d'interconnexion

L'architecture générale est illustrée sur la figure 10.4. Les *connection Box* horizontales et verticales permettent d'acheminer les opérandes des fonctions logiques ves les LUTs. Notez que le schéma fait apparaître (en rouge et vert) les mécanismes de VTPR ainsi que ceux associés à la possibilité de réaliser des *snapshots* c'est-à-dire des copies des états des bascules applicatives (et donc de l'ensemble du circuit synthétisé sur l'overlay) à un moment donné : ceci permet la sauvegarde logicielle de l'état du circuit, qui peut être stocké, puis redémarré à volonté (voir démonstration DASIP en fin de chapitre).

10.2.6 Modélisation RTL des *Switch Box*

La "Switch Box" est placée à l'intersection des canaux de routage horizontaux et verticaux. Elle permet d'établir des connexions entre ces différents canaux pour créer des chemins de routage complexes. Au sein des vrais FPGA, ces switchs contiennent des transistors programmables qui déterminent comment les segments de routage sont interconnectés. Ici, du fait de la modélisation RTL retenue, nous retrouvons à nouveau des multiplexeurs, et leur registres de configuration.

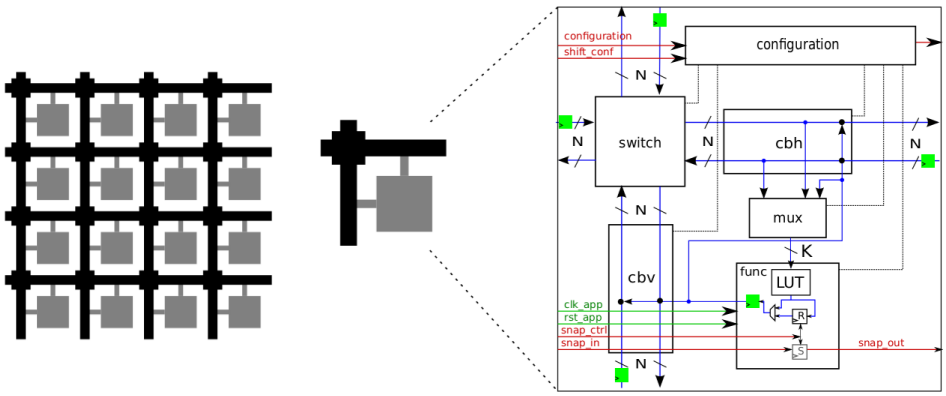


FIGURE 10.4 – Modélisation d'un BLE. On notera l'insertion de registres intermédiaires, que nous avons appelés VTPR, permettant de maîtriser les chemins combinatoires émergeants lors de l'interconnexion des blocks élémentaires. (schéma issu de la thèse de Théotime Bollengier [Z2])

10.3 Propagation d'un temps virtuel : VTPR

10.3.1 Principe

En tant que circuits RTL, ces overlays reconfigurables profitent naturellement des capacités de synthèse des outils standards (ISE, puis Vivado et Quartus dans notre cas). En ce sens, la modélisation au niveau RTL nous impose de respecter les règles courantes de synthétisabilité : en particulier, les boucles combinatoires doivent être proscrites. Or, la topologie régulière de nos overlays – et des FPGA en général– conduit bien évidemment de tels rebouclages combinatoires. Dans le cas des FPGA réels, les temps de propagation ne posent pas de problèmes de fond : ces *timings* sont parfaitement connus et gérés par l'ensemble de la chaîne de synthèse (logique et physique). Pour les overlays, il en est tout autrement : comment notre propre chaîne de synthèse pourrait-elle prendre en compte le résultat de la synthèse sur architecture FPGA réelle ? Le niveau d'indirection intrinsèque à l'overlay rend cette prise en compte très complexe (notamment car les outils propriétaires n'exposent pas leur architecture de manière explicite). Nous avons donc opté pour un choix radical, qui reste aligné sur la nature RTL de nos overlays : notre modélisation va systématiquement *couper* ces chemins combinatoires par une adjonction de bascules spécialement insérées à cette fin : nous avons appelé ces bascules des *Virtual Time Propagation Register* ou *VTPR*. Cela signifie concrètement que les chemins (initialement combinatoires) le long des

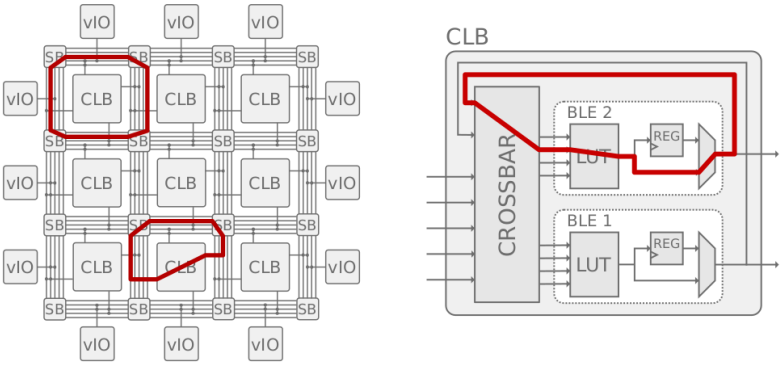


FIGURE 10.5 – Présence naturelle de boucles dans les architectures FPGA, physiques ou virtuels

boucles potentiellement combinatoires sont systématiquement entrecoupées par ces VTPR : les signaux de l'overlay se propagent séquentiellement le long des chaînes de VTRP. Ils *émulent* la propagation combinatoire opérant dans un circuit classique. La fréquence caractéristique des applications synthétisées sur overlay se mesure ainsi comme l'inverse du nombre de VTPR franchis : $f_{virt} = \frac{1}{N_{VTPR}}$. Bien qu'ils se présentent effectivement comme des registres à décalage, on doit cependant noter que la série de VTPR

ne peut pas pipeliner les signaux : les registres applicatifs, de fréquence basse, détiennent des données *stables* pendant toute la durée de propagation de ces signaux le long de la chaîne des VTPR.

10.3.2 Mesure du temps virtuel

Ces VTPR peuvent être vus comme une solution pessimiste qui marquerait une certaine impuissance technologique (impossibilité de prise en compte des timings effectifs finaux, sur la cible physique). Les VTPR possèdent, pourtant, dans les faits plusieurs avantages importants : la logique combinatoire entre deux VTRP est très courte, ce qui permet d'atteindre des fréquences de fonctionnement relativement élevées pour les gammes de FPGA physiques sur lesquels nous avons synthétisé nos FPGA virtuels. Plus intéressant encore, la période de l'horloge de ces VTPR devient un *quantum de temps* ou l'*unité de temps* de mesure de nos chemins applicatifs (voir figure 10.6). Ils nous permettent de gagner fortement en *indépendance technologique*.

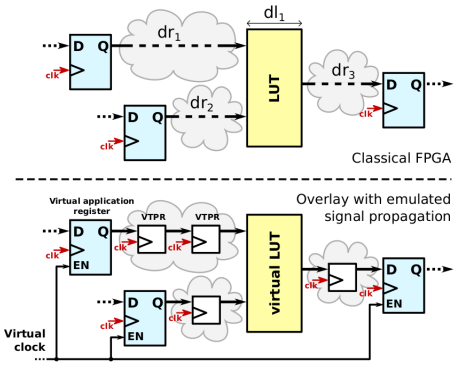


FIGURE 10.6 – Présence des VTPR lors de la synthèse d'applications sur overlay grain fin et conçus au niveau RTL : les temps des chemins combinatoires sont désormais discrétisés par un quantum de temps : celui de la période d'horloge cadençant les VTPR

Synthèse d'applications sur overlay Désormais munis de VTPR, nous sommes en mesure de synthétiser, puis placer et router nos applications sur l'overlay. On pourra se référer à la thèse de Théotime Bollengier [Z2] pour les détails du flot de synthèse. Après placement et routage, nous sommes en mesure de compter le nombre de VTRP franchis sur chaque chemin de l'overlay : cela nous permet de programmer un compteur qui déclenche un signal d'*enable* sur les registres applicatifs. La figure 10.7 illustre ce mécanisme.

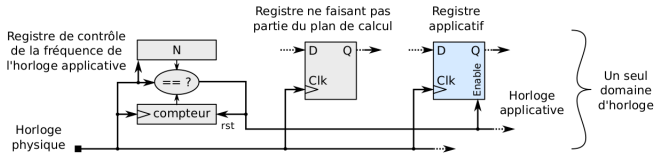


FIGURE 10.7

Circuit applicatif	Synthèse sur Overlay			$\frac{f_{reelle}}{f_{pessimiste}}$	Synthèse sur FPGA (netlist)				Synthèse sur FPGA (RTL)			
	BLEs	Délai N_{VTPR}	f_{reelle} (MHz)		LUTs FFs	f_{max} (MHz)	Freq ratio	Area ratio	LUTs FFs	f_{max} (MHz)	Freq ratio	Area ratio
cordic	611	59	2.89	22.6	598	167.8	58.04	77.65	264	207.4	71.75	175.90
cmult	635	132	1.29	10.1	853	35.9	27.78	56.58	21	135.1	104.59	2298.21
Pmult	412	21	8.12	63.4	259	321.8	39.62	120.90	145	298.4	36.74	215.96
cdivmod	579	166	1.02	8.1	913	74.5	72.53	48.20	730	22.2	21.65	60.28
filtre RII	443	40	4.26	33.3	281	225.9	52.98	119.80	137	174.4	40.90	245.76
Moyenne	536	83.6	3.52	27.5	581	165.2	50.19	84.63	260	167.5	55.13	599.22

FIGURE 10.8 – Résultat de quelques synthèses réalisées sur notre overlay grain fin, et comparaison avec une synthèse native à partir des netlists et du code source RTL.

10.4 Applications des overlays

10.4.1 Mise en oeuvre dans un cadre cloud-computing

Le travail sur les Overlay grain-fins a été réalisé dans le cadre de l'IRT B-COM, dans un cadre cloud-computing. Nous avons cherché à illustrer l'intégration de nos overlays dans ce contexte particulier.

Noeud de calcul et overlays Un nœud de calcul pour overlay est une plateforme matérielle intégrant une mémoire locale, une interface réseau, un overlay et un contrôleur local, appelé hyperviseur de l'overlay. Cet hyperviseur, exécuté sur le processeur du nœud, gère le partage des ressources (overlay, mémoire, communication), ce qui autorise à exécuter plusieurs applications indépendantes. Par analogie avec un hyperviseur de machine virtuelle classique, il ordonne l'exécution des applications sur l'overlay en fonction des priorités et des ressources disponibles. Il reçoit ses instructions d'un contrôleur global, qui peut lui demander d'ajouter ou retirer des applications, ou de fournir des rapports sur l'utilisation des ressources. Les mouvements de données opérés par l'hyperviseur sont représentés sur la figure 10.9. Ces principes ont été publiés dans [R8].

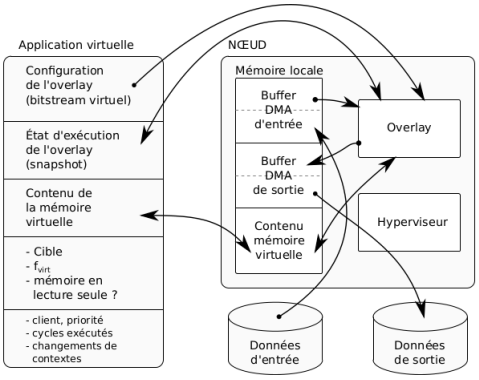


FIGURE 10.9 – Mouvements des données opérés par l'hyperviseur de l'overlay [Z2]

10.4.2 Expérience de migration de tâches

Une expérience démontrée en session "démo" à DASIP en 2016 consistait à utiliser deux overlays FPGA identiques, mais synthétisés sur deux FPGA physiques différents. Il s'agissait en l'occurence de deux FPGA des deux marques Xilinx et Altera, ce qui rend l'expérience plus spectaculaire.Le système se compose de deux nœuds de calcul distincts, reliés à un ordinateur central via Ethernet. Le premier nœud utilise une plateforme Armadeus APF6SP équipée d'un FPGA Cyclone V d'Altera. Le second nœud est basé sur une carte Nexys4 dotée d'un FPGA Artix 7 de Xilinx. Cette dernière est connectée à une Raspberry Pi par une liaison USB/UART (4 Mb/s), la Raspberry Pi faisant office d'hyperviseur. L'ordinateur connecté aux deux nœuds via le réseau Ethernet agit comme contrôleur global du système. Notre overlay (vFPGA) est constitué de 14x13 CLBs de 4 BLEs comportant des LUTs à 4 entrées, avec 16 pistes par canal de routage. L'expérience a permis de mettre en évidence plusieurs attraits de la solution overlay :

— donner une vue homogène d'un ensemble hétérogène de FPGAs.
— ordonnancer différentes applications sur un même overlay.
— offrir un contrôle dynamique via la migration à chaud d'application entre différents overlays.

10.4.3 Dérivation d'un eFPGA

En 2020, nous avons eu l'opportunité, dans le cadre du CPER Cyber-SSI, de pouvoir transformer notre overlay en véritable circuit FPGA "ASIC". Le circuit, appelé Elnath, a été fondu par TSMC en technologie 55 nm ULP. Le flot de conception final a été opéré par la société IC'Alps (Grenoble) à partir de notre code RTL (précédemment émulé sur FPGA puis sur émulateur Synopsys). Elnath comporte 3200 LUTs à 4 entrées et s'est révélé parfaitement fonctionnel lors de sa mise sous tension. Il a par ailleurs permis d'étudier des mécanismes de protection du bitstream in situ.

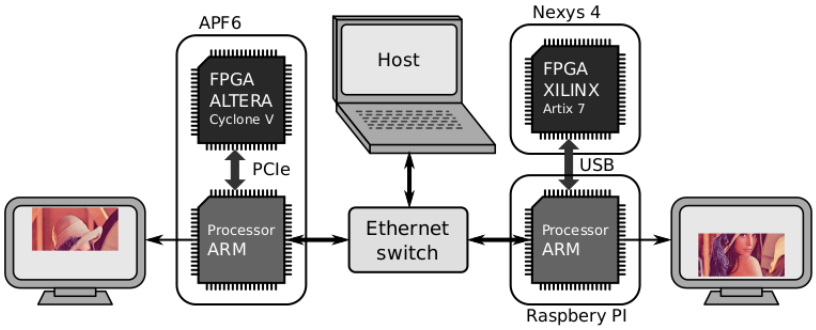


FIGURE 10.10 – Expérience exposée à DASIP 2016 : un *même* overlay synthétisé sur deux FPGA différents assurent une capatibilité binaire de l'application. Grâce à la capacité de *snaphots* (copie d'états applicatifs), une migration de tâches peut notamment être démontrée entre les deux overlays.

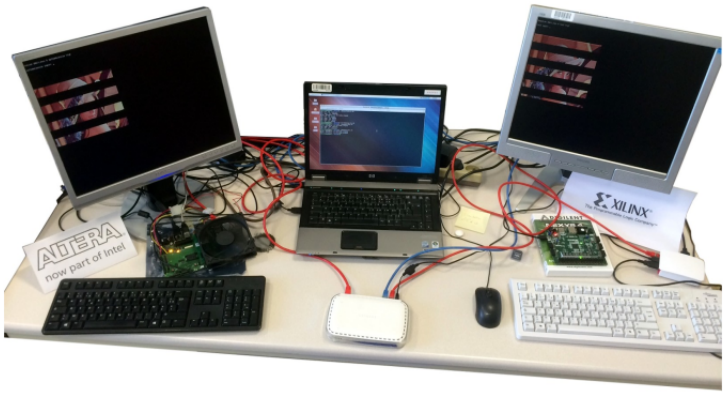


FIGURE 10.11 – Expérience exposée à DASIP 2016

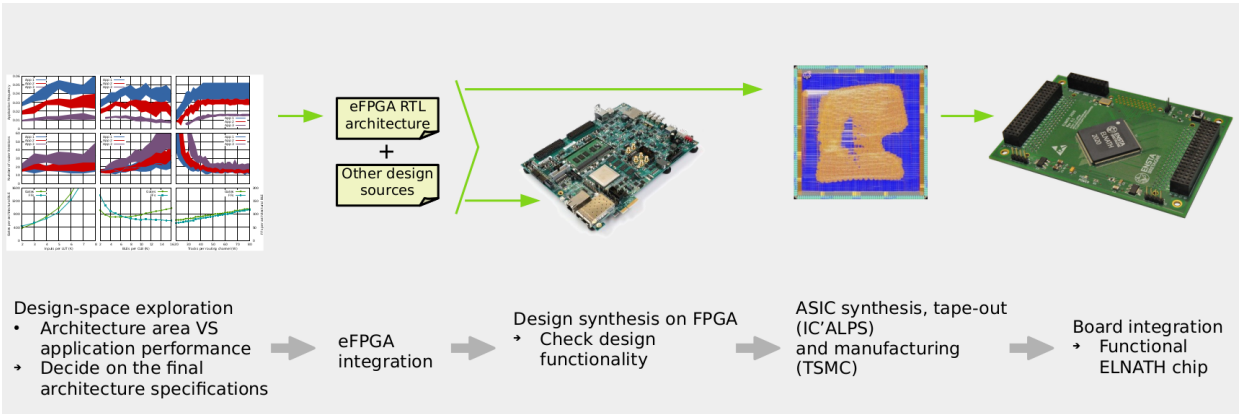


FIGURE 10.12 – Flot de conception du eFPGA ENSTA Bretagne, appelé *Elnath*, comportant 3200 LUTs à 4 entrées. Le circuit a été mis au point au laboratoire, synthétisé par IC'Alps sur technologies TSMC, puis fondu par TSMC.

10.5 Conclusion

La notion d'overlay matériel a été brièvement exposée dans ce chapitre. A l'aide de technologies classiques, de niveau RTL, il est possible de modéliser, simuler puis synthétiser sur FPGA des dispositifs présentant toutes les caractérisitiques d'un nouveau circuit reconfigurable. Cette couche de virtualisation peut se comparer, trait pour trait, aux machines virtuelles (langage ou OS), qui permettent de la même manière de gagner en flexibilité logicielle. Notamment, le portage d'applications, réputé com-

plexe dans le domaine de l'Embarqué, est ici parfaitement simplifié. Toutefois, nous avons illustré le coût matériel important de cette couche de virtualisation, probablement prohibitive pour le cas d'overlays grain fin. Cette expérience reste selon nous une expérience réussie : elle démontre selon nous la possibilité d'une maîtrise "end-to-end" dans ce domaine de l'Embarqué, et invite effectivement à réfléchir à des granularités supérieures où le surcoût matériel peut se révéler acceptable. On pense notamment au CGRA, par ailleurs étudiés au Labsticc (voir les travaux de Kevin Martin [M47]).

Quatrième partie

Contributions à la sécurité des composants

Chapitre 11

Obfuscation de code : trois utilisations

> The best way to hide something is in plain sight.
>
> Edgar Allan Poe

Publications et projets associés

— [Z1] Thèse de Hannah Badier.
— [C3] **DATE'19** Transient Key-based Obfuscation for HLS in an Untrusted Cloud Environment.
— [C2] **DATE'21** Opportunistic IP Birthmarking using Side Effects of Code Transformations on High-Level Synthesis.
— [B1] **Chapitre Springer** Protecting Behavioral IPs During Design Time : Key-Based Obfuscation Techniques for HLS in the Cloud.
— Collaboration avec Christian Pilato. Polytechnico di Milano.

Sommaire

11.1 Introduction

Dans le paysage complexe de la cybersécurité moderne, la menace numérique dépasse largement les traditionnelles attaques réseau et vulnérabilités serveurs. Les systèmes embarqués et les circuits intégrés

constituent désormais des cibles critiques, en raison de leur interconnexion croissante et de leurs architectures de conception spécifiques. Les SoC recèlent des vulnérabilités potentielles liées à leurs flux de conception intrinsèques, offrant ainsi de nouvelles perspectives d'intrusion pour les acteurs malveillants. Ce chapitre résume trois contributions concernant cette menace autour de la conception des circuits. Ces contributions ont été concrétisées dans la thèse de Hannah Badier, ainsi que la collaboration avec Christian Pilato du Polytechnico de Milano, où Hannah a pu séjourner quelques semaines. Ces contributions portent sur 3 aspects :

— La protection des codes sources dans un procédé de HLS déporté dans le Cloud.
— La protection et suivi de la propriété intellectuelle par la notion de *birthmarking* après HLS.
— La détection et suppression de *chevaux de Troie* lors de la HLS.

11.2 Protection des sources pour une HLS dans le Cloud

Ces contributions sont centrées sur l'idée d'obfusquer [1] les codes sources avant HLS : il s'agit d'un cas particulier de techniques de *sécurité par l'obscurité*. L'obfuscation vise à rendre un code particulièrement difficile à exploiter, pour un lecteur malveillant à la recherche d'informations, mais également pour des compilateurs. L'idée d'appliquer l'obfuscation à la HLS a germé lors de mes premiers contacts avec des *prospects* de la startup Modaë technologies : ces sociétés ont souhaité tester notre technologie de modélisation et de synthèse que nous métions à disposition dans le cloud (outil développé par la société rennaise Net-NG) : toutefois, la protection de leurs codes sources (y compris non critiques) apparaissait comme un enjeu majeur. On doit garder en tête que, malgré le chiffrement du code lors de la transmission du code, ce code est, à un moment ou un autre, manipulé en clair par l'outil de HLS déporté dans le cloud. Dans la thèse de Hannah, notre idée à été de permettre à une telle société de "polluer" ses codes utiles à l'aide de constructions inutiles *conditionnées par des clés* ("key-based predicates") : l'ensemble subit la HLS, mais il devient possible d'*inverser* le processus au niveau RTL, en insérant les clés adéquates, qui annihilent l'obfuscation. Cette utilisation de clés peut servir à différents types d'obfuscations. Nous allons expliquer ici les idées maîtresses de ce procédé.

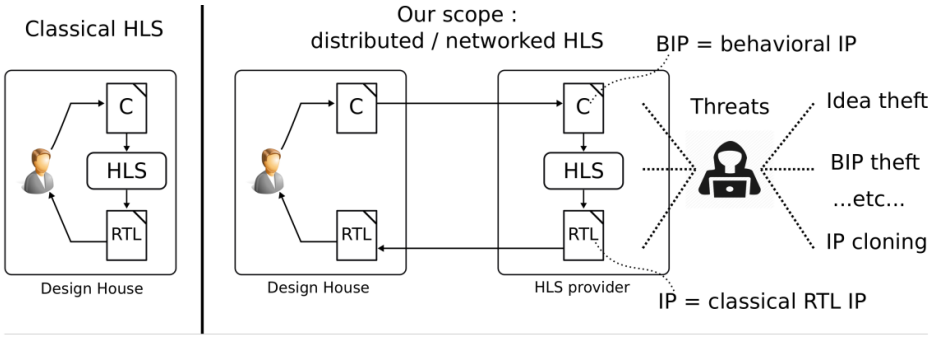


FIGURE 11.1 – Principe de la HLS déportée dans le cloud et ses menaces.

11.2.1 Première technique : expressions obfusquées

Un exemple simplifié d'une telle transformation de code est présentée sur la figure 11.2 : la valeur d'une clé appelée "key" est comparée à la valeur 42. Si la valeur de clé est correcte (ici effectivement égale à 42), l'expression correcte est exécutée. Dans le cas contraire une expression invalide ("*bogus expression*") est exécutée. Nos outils (KaOTHIC et Crokus) effectuent ces transformations d'expressions et remplacent l'expression cible par un flot de contrôle adéquat : une mauvaise valeur de clé conduit à un enchaînement incorrect de calculs. On peut préciser que selon les choix d'obfuscation, l'une ou l'autre des branches contient le code correct à exécuter mais pas forcément celle pour laquelle la comparaison vaut `vrai` : il serait facile pour un attaquant d'observer par exemple la valeur 42 et d'en déduire qu'il s'agit de la valeur de la clé. Par ailleurs, la présence de n variables clés, même identifiables en tant que telles, conduit) 2^n "versions plausibles" du programme, dont l'intérêt est a priori indiscernable.

1. On parle également d'*ofusquer* les sources

On précise que le nouveau code engendré est toujours compilable. Techniquement, nous avons fait le choix de créer des expressions factices similaires aux expressions initiales : une opération arithmétique est remplacée par une opération arithmétique, une opération bit-à-bit par une autre opération bit-à-bit etc, un accès tableau, par un autre accès tableau, etc. De même, les opérandes de ces expressions sont simplement mélangées. Ceci conduit par exemple à :

$$a = (b + c) * 2 \rightarrow a = (2 - c) + b$$

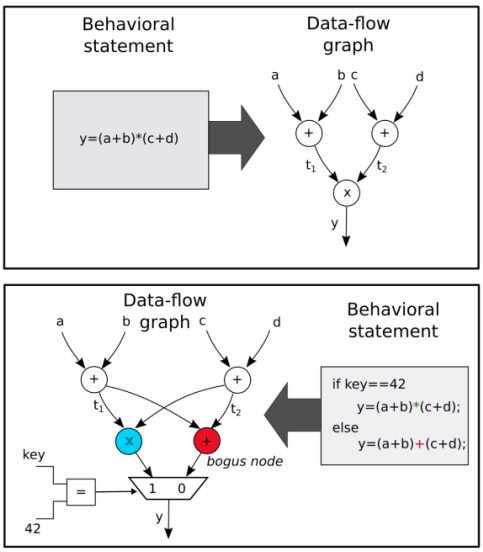


FIGURE 11.2 – Obfuscation d'expressions flot-de-données, pilotée par une clé.

L'ensemble du procédé est applicable au niveau du seul AST du programme.

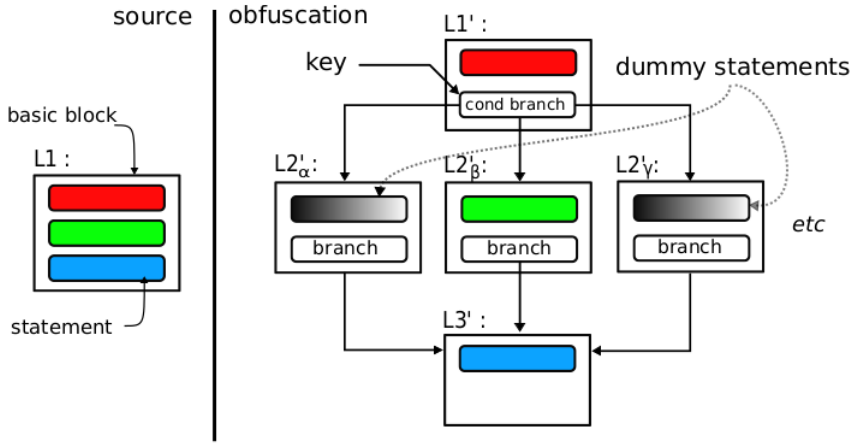


FIGURE 11.3 – Principe d'obfuscation d'expression par prédicat opaque : ici on cherche à cacher l'exécution de l'expression verte, difficile à privilégier par rapport aux deux expressions factices (en gris dégradé).

11.2.2 Autres techniques : flot de contrôle obfusqué

Insertion de blocs de base factices Nous avons également cherché à obfusquer le flot de contrôle présent dans les sources initiales, en créant de toute pièce non plus des expressions factices, mais des blocs de base complets. La technique est cependant similaire à la précédente. Un exemple de création

de ces blocs de base factices est présenté sur la figure 11.3. Nos outils permettent l'obtention, la transformation du graphe de flot de contrôle de ces codes, puis la reconstruction de l'AST correspondant. La génération du nouveau code source est alors possible. Techniquement, certaines difficultés subtiles apparaissent : par exemple, la duplication et modification d'un bloc de base initialement responsable de l'incrémentation d'un index de boucle, peut conduire à une impossibilité de reconstruction de l'AST de la boucle.

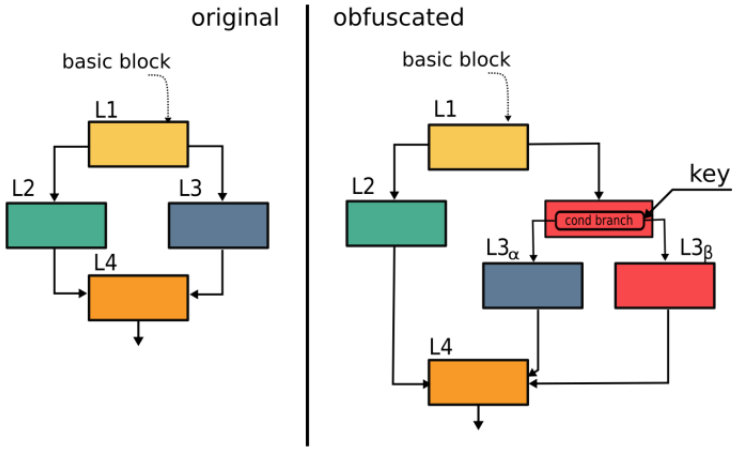


FIGURE 11.4 – Insertion de blocs de base factices.

Control-flow flattening Nous avons également étudié d'autres techniques portant sur l'obfuscation du flot de contrôle. En particulier, la technique de *control-flow flattening* [M15] paraissait intéressante. Elle consiste à remplacer des structures de contrôle telles que des boucles ou des conditionnelles, par une instruction de `switch` global. Le but de cette obfuscation logicielle est de rendre difficile le suivi du parcours de *chemins* entre blocs de bases : le `switch` s'interpose systématiquement entre deux blocs de base qui se suivaient initialement. Il devient difficile de comprendre quel bloc suit l'exécution de quel autre bloc. Cette transformation est illustrée sur la figure 11.5.

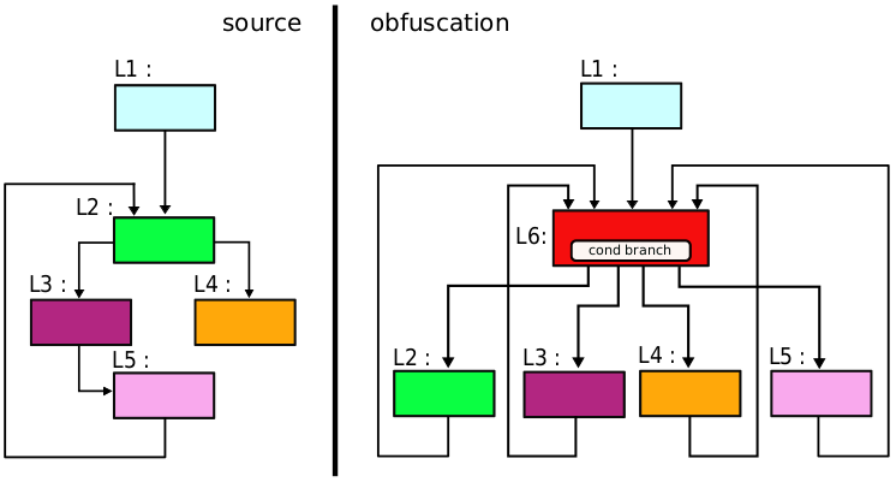


FIGURE 11.5 – Exemple de *control-flow flattening* : l'interposition d'un `switch` global rend l'analyse statique des chemins d'exécution difficile.

Ajout de transitions factices Une autre technique, parmi celles étudiées, consiste à ajouter dans le CFG non plus de blocs factices, mais uniquement des transitions factices 11.6. L'intérêt du procédé tient au fait qu'aucun bloc de base artificiel n'est créé ici : dans les procédés précédent, du fait du caractère

automatique de la création de blocs, certaines expressions pourraient être détectables par un spécialiste de domaine. Ici, seul un saut inconditionnel est remplacé par un saut conditionnel, ce qui reste très discret et plausible.

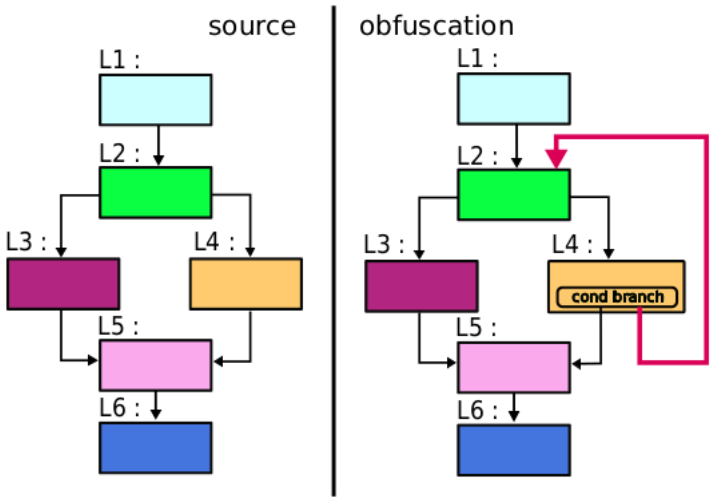


FIGURE 11.6 – Exemple d'ajout de *transition factice* : un saut inconditionnel est remplacé par un saut conditionnel

11.2.3 Obfuscation transitoire : vers la réversibilité

Comme indiqué en introduction, nous avons cherché à appliquer ces obfuscations dans le cadre particulier de la synthèse HLS. Notre but était par ailleurs double : d'une part il s'agissait de rendre l'obfuscation suffisamment convaincante pour faire circuler des codes sources *en clair*, en les rendant inexploitable sans la connaissance des clés. D'autre part, cette obfuscation doit pouvoir *s'inverser* : au niveau RTL, après compilation et insertion des clés, il doit être possible de retrouver le comportement initial du programme avec un overhead si possible minimal. Quel est cet overhead ? Est-il rédhibitoire ou non ?

Désobfuscation par écroulement de la logique Le principe que nous avions en tête pour assurer une certaine réversibilité repose sur la logique booléenne. Observons la figure . Elle présente le multiplexeur qui assure le routage correct entre deux signaux `a` et `b` (ici sur 32 bits), après comparaison d'une clé `k` et d'une valeur `v`. Ce mécanisme représente ici environ 300 portes logiques. Il doit être logiquement présent dans les résultats issus de la HLS, au niveau RTL. Cependant, notre connaissance des clés secrètes nous permet de forcer la bonne valeur de clé à ce niveau. Les synthétiseurs logiques obtiennent alors un ensemble de constantes propagées, qui viennent remplacer le multiplexeur complet et le remplacent par un seul ensemble de fils : aucune porte logique n'est nécessaire. Par exemple, sur la figure, toute la partie logique en amont du signal `b` disparaît purement et simplement. Pour rappel, dans notre cas, cela correspond à l'ensemble des calculs inutiles issus du procédé d'obfuscation. Nous parlons ainsi d'*écroulement de la logique*, qui autorise à penser qu'une *désobfuscation* est possible après HLS. Nous verrons dans un instant que certains obstacles demeurent, mais peuvent être *contournés*, mais également *exploités* de manière astucieuse.

Exemple Afin d'illustrer de manière plus convaincante le procédé d'obfuscation et désobfuscation appliqué à la HLS, on peut observer la figure 11.8. Un code source initial (en haut à gauche) subit cette obfuscation. Ici on a cherché à rendre l'exécution de l'expression initiale $t_1 * c$ dépendante de la clé k. Une mauvaise valeur de clé [2] conduit à l'exécution de l'expression factice $t_1 + c$. La partir (b) du schéma expose une vue RTL (partielle) obtenue après HLS (Vivado). On retrouve l'opérateur initiale d'addition et l'opérateur factice de multiplication, ainsi que l'ensemble du routage des données, mais également

2. Ici la bonne valeur de clé est *toute* valeur *différente* de 42

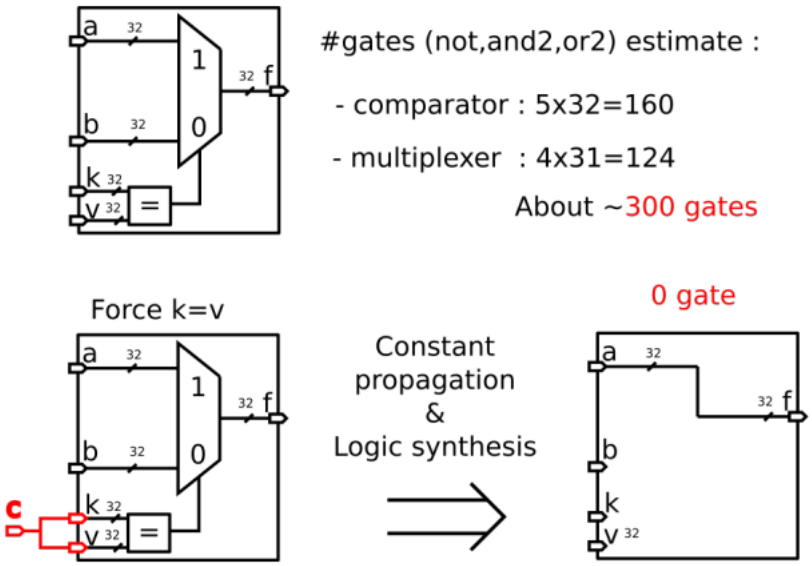


FIGURE 11.7 – Principe de la désobfuscation par écroulement de la logique combinatoire.

leur stockage sur 3 pas de contrôle. L'influence de la bonne valeur de clé est représentée en bleu. Ce qu'il est intéressant de remarquer sur la partie (c), c'est que le mécanisme d'écroulement de la logique ne fonctionne que dans un ensemble combinatoire, mais ne franchit pas les barrières temporelles que constituent les registres *entre c-steps*. Le registre entre le c-step 1 et 2 reste présent et rend impossible la propagation de la constante 0 issu de la comparaison des clés. Nous avons appelé cette première application de clés une *injection de clé naïve* : le circuit se révèle uniquement partiellement désobfusqué.

Désobfuscation par analyse du code RTL Les résultats mitigés issus de l'injection de clé naïve nous a forcé à réfléchir à une méthode visant à injecter le résultat de la comparaison de la clé, de manière directe, à l'entrée du multiplexeur entre les expressions (partie (d) du schéma précédent). Cette désobfuscation a été automatisée par nos outils, de manière chirurgicale, pour la sortie VHDL RTL de VivadoHLS. Elle est cependant très dépendante du style RTL retenu par l'outil de HLS. Une seconde désobfuscation, plus robuste aux variations syntaxiques, a été réalisée à l'aide du parseur `PyVerilog` [] : une première passe *forward* consiste à réaliser la propagation des constantes issue de la comparaison des clés et à substituer la sortie des multiplexeurs commandés par cette constante par leur entrée afférente. Une analyse *backward* supprime également la logique amont non-utilisée pilotant les entrées non-utilisées des multiplexeurs précédents.

11.2.4 Outils et paramètres d'obfuscation

Outil Kaothic Un schéma d'architecture générale du compilateur Kaothic est présenté sur la figure 11.9.

Paramètres d'obfuscation Nos outils prennent en compte deux paramètres importants :

1. Le niveau d'obfuscation : il s'agit d'un paramètre indiquant le nombre d'items obfusqués dans le code initial. La nature de ces items dépend de la technique d'obfuscation utilisée : expression ou CFG. Un niveau d'obfuscation de 100% sur les expressions signifie que toutes les expressions du code ont subi la transformation.
2. Le degré de branchement (*branching degree*) : il s'agit du nombre d'éléments factices ajoutés pour chaque item obfusqué (expression ou bloc de base).

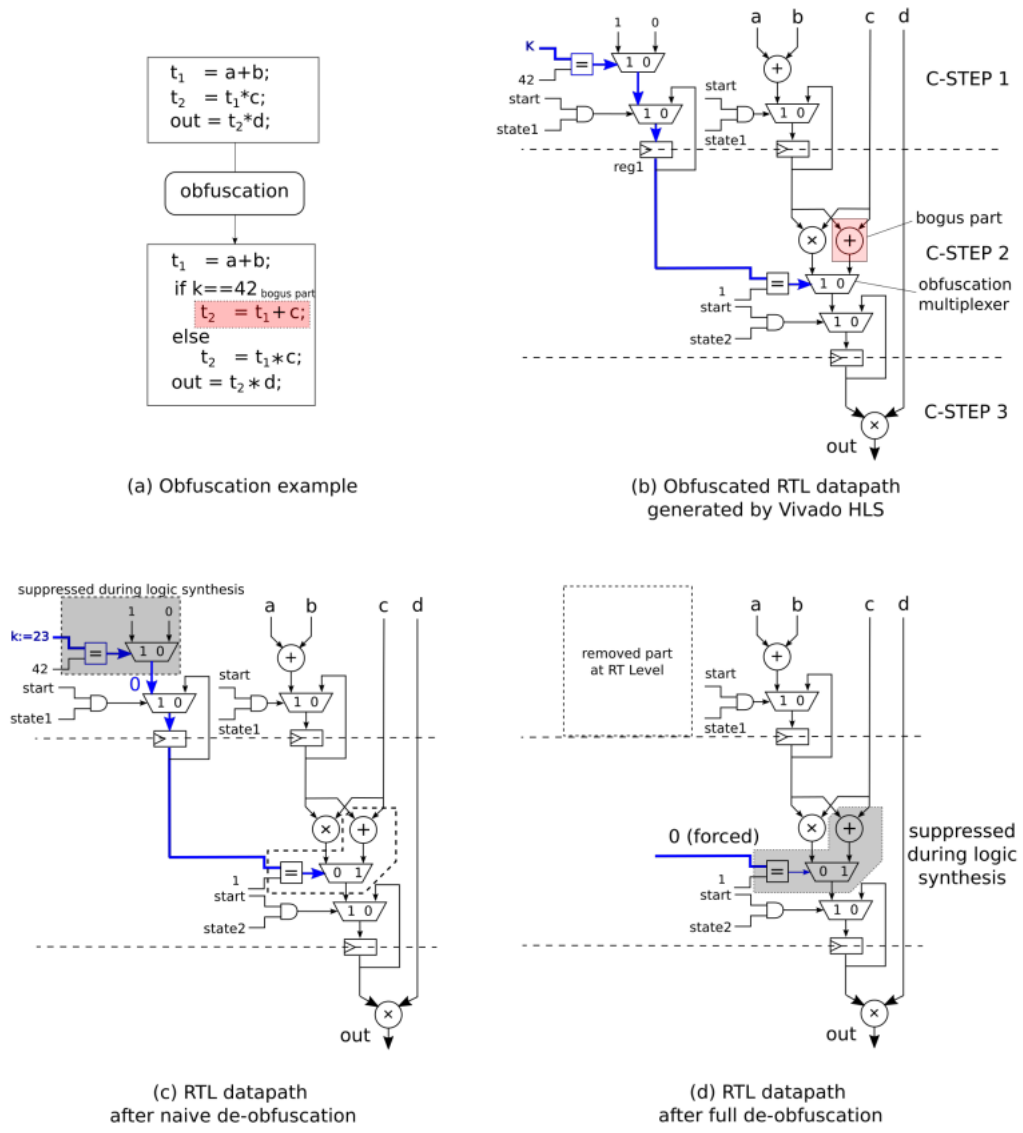


FIGURE 11.8 – Illustration du procédé d'obfuscation appliqué à la HLS.

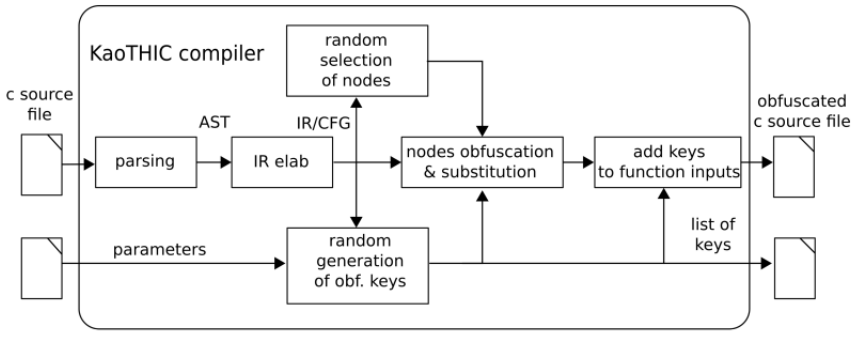


FIGURE 11.9 – Architecture générale du compilateur KaoTHIC.

11.2.5 Résultats de désobfuscation

Mesure de la désobfuscation sur expression Les tableaux 11.10 et 11.11 présentent les résultats en surface et délais d'un ensemble de benchmarks HLS connus synthétisés respectivement sur ASIC et

FPGA à l'aide de notre procédé, pour un degré de branchement de 1 (pour chaque expression, une seule expression factice ajoutée). Les premières colonnes de gauche recensent les résultats dans le cas d'une synthèse logique sans application de désobfuscation. Deux désobfuscations sont par ailleurs étudiées : la désobfuscation naïve et une désobfuscation en profondeur. De manière évidente, la présence de code inutile dans les sources induit des overheads importants : jusqu'à 12% en ASIC et plus de 200% en FPGA. Les résultats montrent que le procédé de désobfuscation réduit fortement l'overhead (notamment 0% pour AES sur ASIC) : nous sommes ainsi à même de supprimer, en tout ou partie, les artefacts de l'obfuscation rendant ainsi le procédé d'obfuscation transitoire applicable.

Benchmark	No Deobfuscation		Naive Deobfuscation		Full Deobfuscation	
	Area(%)	Delay(%)	Area(%)	Delay(%)	Area(%)	Delay(%)
Adpcm	2.61	-0.06	2.29	-0.08	-0.08	-0.01
AES	0.3	<0.01	0.31	<0.01	0.03	2.72
Merge Sort	0.07	<0.01	0.07	<0.01	0.03	0.01
Mips	1.14	0.5	1.45	0.5	-1.12	-7.47
Needwun	9.01	11.69	8.39	11.69	0.26	<0.01
Stencil3d	9.38	0.73	9.2	0.73	1.54	-0.14

FIGURE 11.10 – Résultats de synthèse HLS sur technologies ASIC avec obfuscation (branching degree=1) : sans désobfuscation, puis avec les deux techniques de désobfuscation.

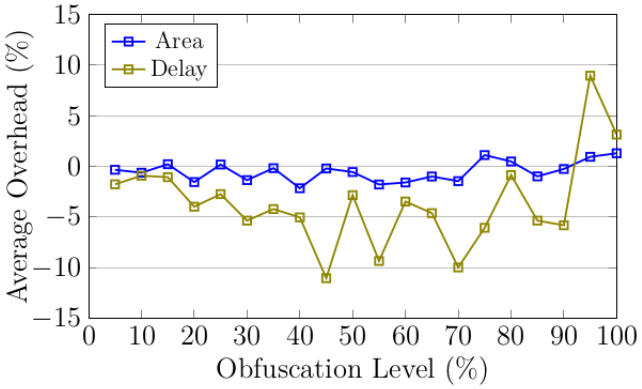
Benchmark	No Deobfuscation			Naive Deobfuscation			Full Deobfuscation		
	LUT(%)	FF(%)	Delay(%)	LUT(%)	FF(%)	Delay(%)	LUT(%)	FF(%)	Delay(%)
Adpcm	5.82	28.9	15.9	-0.08	-0.2	10.85	-0.12	-0.2	11.06
AES	29.0	20.0	1.98	7.05	0.42	3.35	8.97	0.42	4.49
Merge Sort	-1.37	18.7	2.57	-6.18	4.08	-1.27	-5.99	4.08	-3.77
Mips	45.9	210	7.51	1.46	-1.39	4.51	1.78	-1.39	4.61
Needwun	23.9	33.5	-4.22	0.79	0.01	-2.78	-0.6	-0.25	-1.98
Stencil3d	20.7	22.4	-9.44	1.38	4.56	-3.78	-1.25	5.57	-0.86

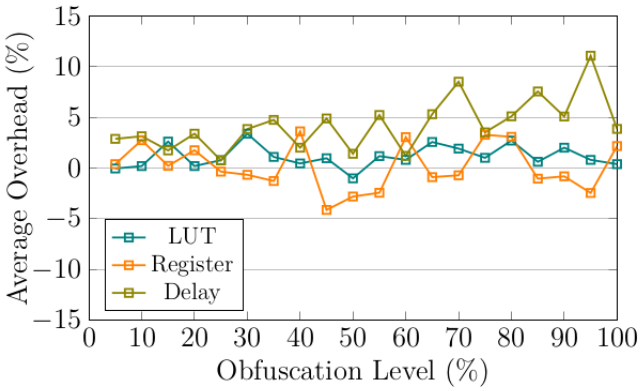
FIGURE 11.11 – Résultats de synthèse HLS sur technologies FPGA avec obfuscation (branching degree=1) : sans désobfuscation, puis avec les deux techniques de désobfuscation.

Overheads proches de zéro et overheads négatifs On note également dans les tableaux précédents un certain nombre d'overheads proches de zéro, mais non nuls, y compris après désobfuscation complète. Il s'agit selon nous des effets du partage de ressources réalisé par l'outil de HLS. On peut également incriminer les effets de la variance syntaxique sur les heuristiques de l'outil. De manière plus suprenante, on note également dans les tableaux précédents un certain nombre d'overheads négatifs (MIPS pour ASIC et Sort pour FPGA par exemple) : là encore, nous supposons qu'il s'agit de choix d'heuristiques différentes selon la complexité du code source en entrée de la HLS. Ces choix internes à l'outil conduisent à des optimisations plus poussées, et notamment à des chemins critiques inférieurs à ceux de la synthèse des sources initiales non-obfusquées !

Effet du niveau d'obfuscation Nous avons également étudié l'impact du niveau d'obfuscation, comme défini précédemment, en le faisant varier de 5% à 100%. Nous cherchons ici en particulier à répondre à la question suivante : existe-t-il une corrélation entre le niveau d'obfuscation et l'overhead ? Les graphes suivants illustrent nos résultats : cette corrélation semble inexistante. C'est un résultat intéressant car il indique que notre procédé de désobfuscation est effectif, quelle que soit la quantité de code qui a

été obfusqué. Cela pousse par exemple à choisir un niveau d'obfuscation maximum, à 100%. D'autres résultats sur le degré de branchement pourront été consultés dans la thèse de Hannah Badier.





11.3 Détection de la propriété intellectuelle : *birthmarking*

Nous présentons ici le principe de *birthmarking* que nous avons élaboré et publié à Date. L'idée est la suivante : les résidus non-nuls de la désobfuscation précédente sont à même de constituer une *signature* qu'il est possible de détecter dans le RTL finalement délivré (et vendu à un tiers).

11.3.1 Modèle d'attaque

Le modèle d'attaque est représenté sur la figure 11.12. Un concepteur réalise l'achat légal d'une IP RTL ou gate-level. On rappelle que ces IP sont sujettes à des règles de réutilisation à usage limité. Un attaquant de type *insider* vole cette IP. Son utilisation ultérieure peut être variée : simple revente, incorporation illicite directe dans un produit ou encore *modifications* pour revente etc. Il y a lieu de chercher à appliquer une marque de propriété à l'IP créée, qui non seulement restera facilement revendicable dans le cas d'un réutilisation illicite directe, mais également détectable après modifications. Ce marquage doit être discrèt et en particulier non localisé sur une portion seule du code. Il ne doit pas entraver le fonctionnement normal de l'IP. La marque participe au procédé de *watermarking* fondé sur deux fonctions inverses :

1. *Embed* : cette fonction prend une IP et ajoute ladite marque w, en utilisant une clé secrète *key* : $embed(IP, w, key) \rightarrow IP_w$
2. *Extract* : cette fonction prend une IP marquée ainsi que la clé secrète *key* et restitue la marque insérée : $extract(IP_w, key) \rightarrow w$

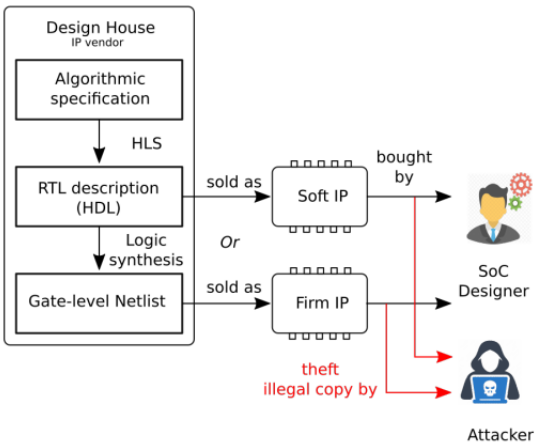


FIGURE 11.12 – Modèle d'attaque justifiant le *birthmarking*

11.3.2 Procédé de birthmarking

Comme indiqué en introduction de la section, le procédé que nous avons élaboré repose sur l'imperfection de la désobfuscation. Cependant, sur le schéma, nous présentons cette opération sur les sources comme une transformation de modèle anonyme, car nous estimons que d'autres transformations *quasi-réversibles* peuvent être imaginées : la chose importante réside dans l'existence d'un procédé inverse sur le RTL. C'est ce dernier qui est vendu et "circule" donc dans la nature. Sur le schéma de principe, on peut voir que le propriétaire de ce code conserve également, en interne, le RTL transformé. Notre papier démontre que la comparaison de ces deux RTL permet de détection une "filiation de conception" entre les deux codes et donc prouver la détention légale d'un code. A l'inverse, un attaquant qui ne possède que le RTL final ne pourra pas prouver la pleine propriété de cette IP.

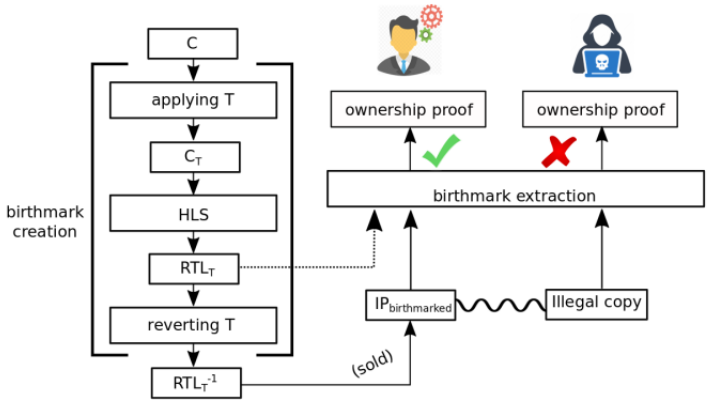


FIGURE 11.13 – Flot de marquage et suivi de *birthmarking*

11.3.3 Aperçu du principe technique

Le principe technique ne sera que survolé ici afin d'en donner l'intuition. Le lecteur pourra se référer à la thèse de Hannah Badier (chapitre 3) et à la publication [C2] pour plus de détails.

Rappels sur l'Obfuscation-desobfuscation Soit C un design originel, écrit en langage C. On lui applique une obfuscation Obf fonction d'une clé K^i, et un ensemble de paramètre P^i. Appelons alors C^i_{obf} le code source résultant. Il subit une opération HLS classique qui conduit au code RTL^i_{obf}. Par la suite,

on applique la transformation inverse Obf^{-1} conduisant au RTL supposé quasi-épuré RTL^i_{deobf} :

$$C^i_{obf} = Obf(C, P^i, K^i)$$
$$RTL^i_{obf} = HLS(C^i_{obf})$$
$$RTL^i_{deobf} = Obf^{-1}(RTL^i_{obf}, K^i)$$

Inclusion et similarités (*containment* et *similarity*) Le procédé d'extraction de la *birthmark* s'appuie sur la mesure de deux métriques applicables sur différents paramètres du design : ces deux métriques quantifient respectivement l'*inclusion* et la *mesure de similarité* de deux design RTL. Pour chacun des paramètres mesurés sur les deux designs, il faudra ainsi être en mesure d'en exprimer ces "distances" que constituent l'*inclusion* et la *similarité*.

Cas de l'ordonnancement A titre d'exemple, observons le cas de la comparaison d'*ordonnancement* présentée sur la figure 11.14. On note l'ensemble des pas de contrôle $S = \{s_j \mid 1 \leq j \leq r\}$

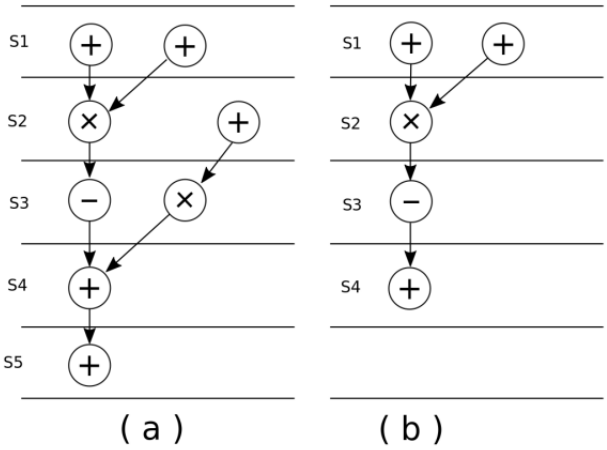


FIGURE 11.14 – Ordonnancements de deux design RTL dont on cherche une mesure d'*inclusion* et de *similarité*

On se rend compte que certains états présentent strictement les mêmes opérations. Par exemple, la similarité de s^a_1 et de s^b_1 est de 100% (ou 1). Par contre, on voit que celle de s^a_3 et de s^b_3 n'est que de 50%, etc. En s'inspirant de travaux dûs Broder [], on établit ainsi que :

$$similarity(s^a, s^b) = \frac{\left|OP(s^a) \cup OP(S^b)\right|}{\left|OP(s^a) \cap OP(S^b)\right|} \quad containment(s^a, s^b) = \frac{\left|OP(s^a) \cup OP(S^b)\right|}{|OP(s^a)|}$$

Par exemple, on a :

$$similarity(s^a_3, s^b_3) = \frac{|\{-,*\} \cup \{-\}|}{|\{-,*\} \cap \{-\}|} = \frac{|\{-\}|}{|\{-,*\}|} = \frac{1}{2}\text{etc}$$

A partir de ces mesures locales sur les états, il devient possible de comparer les deux *ordonnancements* complets S^a et S^b peuvent alors, en reposant sur une formule identique pour les deux métriques :

$$f(S^a, S^b) = \frac{\sum_{j=1}^{r_{max}} f(s^A_j, s^B_j)}{r_{max}} \quad \text{où } r_{max} = max(r_A, r_B)$$

Cas des autre paramètres Nous avons procédé de même pour d'autres paramètres extraits des deux design RTL à comparer :

— Inclusion et similarité de *datapath* : une option *naïve* a été retenue, qui consiste à calculer tous les chemins possibles entre les entrées et les sorties et représenter l'enchaînement des éléments de manière compacte par un mot composé à partir d'un alphabet, chaque lettre étant associée à un type d'opérateurs (MUX $\rightarrow A$, etc).

— Similarité de *flot-de-données* à l'aide d'une *distance d'édition de graphe* (GED) : en l'occurence la similarité de Levenshtein.

11.3.4 Setup expérimental et résultats

A l'aide de ces métriques, nous avons appliqué un classifieur binaire : on classe des paires de design selon qu'elles présentent ou non le tatouage (leur filiation). Le détail de ce travail minitieux, ainsi que de nombreux résultats intermédiaires, peut être trouvé dans [C2]. Notons que ces expériences requièrent un nombre important de codes C, qui ont été générés à l'aide de mon outil *Crokus* (parseur d'un sous-ensemble de C, IR et différentes passes de transformation de code). Les résultats ont démontré qu'il est ainsi possible de détecter les *birthmark* et prouver la "dérivation" illégale d'un design à partir d'une source légale. Le taux de détection est supérieur à 96%, avec un overhead en ressources en dessous de 5 %, ce qui paraît très acceptable.

11.4 Détection des chevaux de Troie

Nous avons proposé une troisième et dernière utilisation du procédé d'obfuscation transitoire, mais elle est elle-même double : d'une part, la suppression automatique d'un cheval de Troie (HT Hardware Trojans), et d'autre part la détection d'un tel cheval de Troie. Nous allons présenter ici succintement quelques travaux menés sur le sujet. La figure 11.15 cherche à expliciter la menace associée aux chevaux de Troie dans un contexte EDA : on suppose l'existence d'un attaquant au sein d'une entreprise developpant des outils de synthèse HLS. Cette hypothèse a été discutée et illustrée dans la littérature [M55] (au titre explicite *"Black-Hat High-Level Synthesis : Myth or Reality ?"*). En utilisant cet outil, un concepteur (utilisateur de l'outil) injecte sans le savoir un tel cheval de Troie dans une IP qui peut être intégrée à des SoC plus volumineux, eux mêmes en fonctionnement dans des environnements variés. Par une activation organisée (non discutée ici) ou de manière sporadique, ce cheval de Troie peut se réveiller et causer différents troubles allant de la fuite d'information, à la défaillance catastrophique, en passant par des ralentissements inexpliqués.

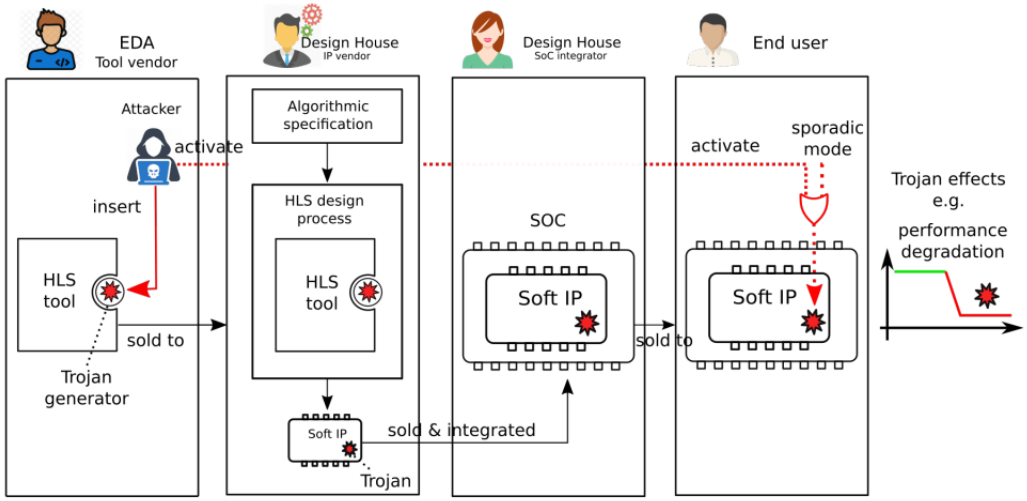


FIGURE 11.15 – Menace liée à une corruption maveillante d'un outil de HLS et propagation d'un Cheval de Troie jusqu'à l'utilisateur final.

11.4.1 Suppression par désobfuscation

L'idée que nous avons explorée est la suivante : un code hautement obfusqué présente une surface d'attaque plus grande. Le cheval de Troie peut ainsi s'installer dans une région de code factice. Lors de la désobfuscation, il existe une probabilité qu'il soit purement et simplement supprimé par notre mécanisme. Nous avons calculé une équation qui donne cette probabilité $P(\lambda, \delta)$ de suppression, où

λ et δ représentent le niveau d'obfuscation et le degré de branchement. Avec un niveau d'obfuscation maximum (auquel la désobfuscation est insensible), on trouve finalement :

$$P(\delta) = 1 - \frac{1}{\delta + 2}$$

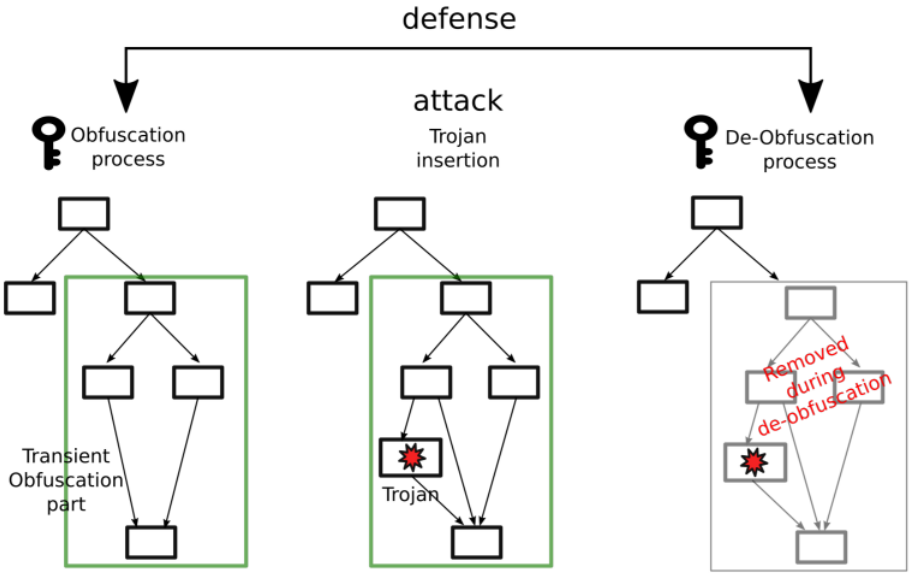


FIGURE 11.16 – Le procédé de désobfuscation qui suit une obfuscation transitoire est susceptible de supprimer également un cheval de Troie inséré dans un CFG.

11.4.2 Résultats de suppression

A l'aide de mon outil `Crokus`[3], nous avons inséré un cheval de Troie de notre cru : Syracuse. Il s'agit d'un code qui ne modifie pas le résultat d'exécution, mais ralentit de manière aléatoire l'exécution par un calcul inutile, difficile à analyser en tant que code mort par un compilateur C. Il est basé sur la conjecture mathématique de Collatz (ou problème "3n+1"). Les code C ainsi infectés sont soumis à KaOTHIC, qui applique l'obfuscation. Il est ici toutefois possible de bypasser une véritable synthèse HLS : nous avons simplement harcodé les bonnes clés dans le code C, puis utilisé `Clang` afin d'optimiser le code. En particulier l'option `-ipsccp` (*interprocedural sparse conditional constant propagation*) réalise une opération similaire à notre injection de clé. Une expression régulière permet de vérifier si le cheval de Troie est toujours présent ou non après cette passe. Nous avons répété l'expérience avec un vaste jeu de paramètres et d'insertion aléatoire, soit environ 1600 codes C infectés. La synthèse des résultats expérimentaux est présentée sur la figure 11.17. Les résultats théoriques et expérimentaux de taux de suppression sont très proches : le taux de corrélation est de 0.93.

11.4.3 Détection par désobfuscation

La détection par désobfuscation est une autre utilisation de la désobfuscation. Ici, le principe est le suivant : dans les cas non infectés, notre procédé de désobfuscation est capable de supprimer non seulement le code factice, mais également la dépendance à la clé (placée en entrée de fonction). Toutefois, si le cheval de Troie utilise *également* d'utiliser cette clé, notre procédé de désobfuscation *conservera* la dépendance à la clé ! Notre outil a été modifié pour indiquer de tel cas de non suppression de clé, qui indique par conséquent la présence d'un cheval de Troie. Cette idée est illustrée sur la figure 11.18.

3. https ://github.com/JC-LL/crokus

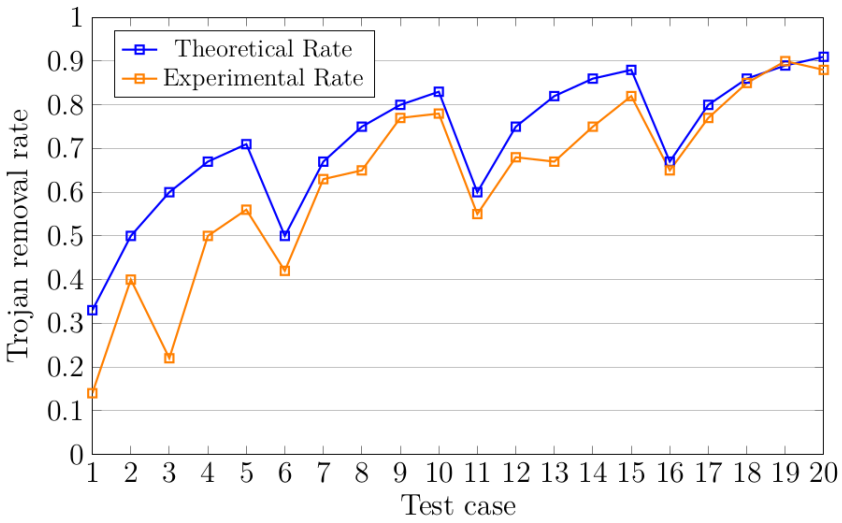


FIGURE 11.17 – Comparaison entre les taux théoriques de suppression d'un cheval de Troie, par obfuscation-désobfuscation, et les valeurs mesurées expérimentalement.

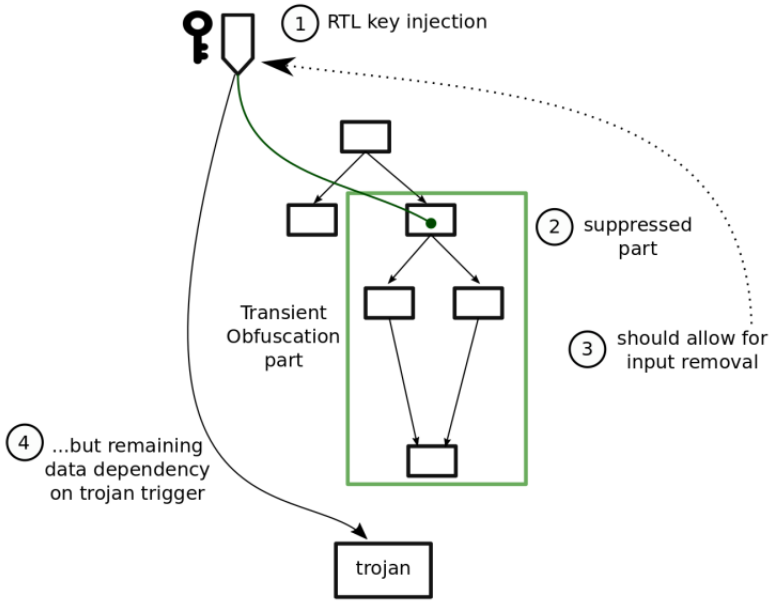


FIGURE 11.18 – Illustration du principe de détection par désobfuscation.

11.5 Conclusion

Ce chapitre a présenté trois utilisations d'un même procédé d'*obfuscation transitoire* des sources lors de la synthèse comportementale. Le premier vise à permettre une protection de la propriété intellectuelle d'un code source, lors d'une HLS réalisée à distance, dans le cloud. La deuxième des utilisations est relative à la présence d'une marque de naissance (*birthmark*) qui résulte de l'opération d'obfuscation-désobfuscation : ce forçage de la HLS induit de petites modifications spécifiques et non locales du RTL qu'il est possible d'identifier a posteriori à l'aide du *machine learning*. Enfin, la suppression et la détection de chevaux de Troie se révêle une utilisation innattendue, mais belle et bien effective et mesurée, de cette même technique d'*obfuscation transitoire*.

Chapitre 12

Détection de Trojans par analyse transitoire

> On entend le bruit du monde, mais rarement sa musique.
>
> Victor Hugo

Publications et projets associés

— Thèse de Quentin Tual (en cours)
— Collaboration avec DGA-MI

Sommaire

12.1 Introduction

Comme indiqué dans le chapitre précédent, la détection de Chevaux de Troie dans les circuits est devenue un enjeu majeur du fait du nombre croissant d'acteurs intervenant dans des chaînes de conception qui ne cessent de s'allonger : le risque de présence d'acteurs mal-intentionnés de type *insiders* (ennemis de l'intérieur) est désormais une réalité. De même, les tensions géo-stratégiques autour de la conception des SoC fait craindre des consignes étatiques poussant concepteurs, fondeurs et fournisseurs d'outils EDA à introduire de tels nouvelles menaces au sein des SoC, pilliers de la société numérique. Il y a donc lieu de s'intéresser au problème d'un point de vue conceptuel et scientifique. Est-il possible de détecter de telles insertions, faites à notre insu ? Une première réponse favorable a été donnée au chapitre précédent, dans le contexte très particulier de la HLS couplée à notre procédé d'obfuscation-désobfuscation. Dans ce chapitre, je présente les idées clés d'un procédé tout autre : il s'agit d'observer le circuit suspecté, dans son régime de stabilisation combinatoire.

12.2 Modèle de chevaux de Troie : discrétion synchrone

Définition et illustration Fidèle à l'image issue de la mythologie grecque, le cheval de Troie (*"Hardware Trojans"* ou "HT") est un dispositf en apparence inoffensif, qui contient pourtant inrinsèquement une menace, déclenchable à tout moment. La figure 12.1 illustre un exemple de cheval de Troie matériel. Ce HT présente deux parties distinctes :

— Le déclencheur ou *trigger* : c'est la fonction booléenne qui permet d'activer le cheval de Troie. Cette fonction booléenne a vocation à n'être déclenchée que lors de la conjonction de conditions très particulières, imaginées par l'attaquant et à l'inverse difficilement imaginables par le concepteur du circuit initial. Comme ce dernier est en général responsable de la définition des tests fonctionnels exigibles après réception des échantillons de fonderie, l'activation d'un éventuel cheval de Troie est très peu probable, ce qui assure sa diffusion à plus large échelle.

— La charge ou *payload* : c'est la fonction exécutée par le cheval de Troie. Elle peut avoir des répercussions diverses, allant de l'apparition de bugs intermittents à la destruction de circuits complets, en passant par la fuite de données sensibles, etc. Dans nos hypothèses, il ne s'agira que de provoquer des erreurs fonctionnelles, dont on ne cherche pas à mesurer les conséquences.

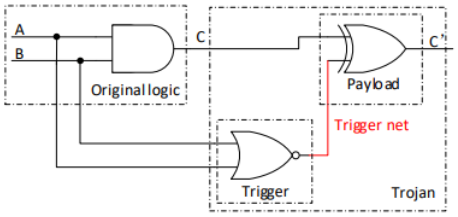


FIGURE 12.1 – Exemple de chevaux de Troie combinatoires. On précise que la condition de déclenchement est supposée rare ou sans signification (et donc omise) dans un plan de test fonctionnel.

Sur la figure 12.1, on constate que le HT inséré vise à inverser le signal initial C, grâce à une porte `xor` présente dans la payload : il peut s'agir d'une prise de décision cruciale dans un processus complexe, etc. Cette inversion se produit sur la condition booléenne $\overline{a+b}$, soit la conjonction $a = 0$ et $b = 0$.

Discrétion synchrone La condition nécessaire au déclenchement du HT est ici très simple, mais elle est supposée par ailleurs difficile à obtenir ou omise du plan de test final du circuit. Le HT se veut ainsi difficile à détecter en *fonctionnement nominal*. Il est très important de noter que lorsque le trigger est par contre *inactif*, tout se passe comme si le seul circuit original fonctionnait (ici une porte `and`). Cela signifie qu'en dehors d'une activation du trigger, le comportement du circuit synchrone initial n'est pas modifié : on parlera de *discrétion synchrone*.

12.3 Principe suggéré

Nous avons cherché à observer les circuits *en dehors de leur fonctionnement nominal* et plus particulièrement dans le régime d'instabilité des signaux. En effet, les signaux d'entrée dans un circuit combinatoire ne se propagent évidemment pas de manière instantannée à travers les portes logiques. Chaque porte est caractérisée par un ensemble de paramètres physiques, dont un délai de propagation associé : il s'agit du temps nécessaire pour que son entrée affecte ses sorties. Ce délai est caractérisé par le fondeur, qui est à même de mesurer cet ensemble de paramètres (dont effets capacitifs) qui participent au comportement global de la porte. Avant que tous les signaux n'aient fini de se propager, les sorties intermédiaires peuvent varier de manière en apparence imprévisible : pendant la phase transitoire, des glitches ou impulsions temporaires non désirées de courte durée peuvent notamment se produire sur les sorties du circuit. Cela survient lorsque certaines branches du circuit atteignent leurs nouvelles valeurs avant d'autres. Ces valeurs intermédiaires peuvent causer des changements rapides et temporaires de la sortie qui ne seront plus présents une fois le circuit stabilisé. Cette zone particulière d'oscillations

transitoires est vue comme une zone de "turbulences", qui ne présente pas d'intérêt pour le concepteur [1].

Idée

L'idée que nous explorons est que la présence d'un cheval de Troie peut être détectée lors de ce régime transitoire par des moyens appropriés. Notamment, la présence *structurelle* de ces portes étrangères à une netlist initiale engendre des modifications subtiles des temps de propagation des signaux, que nous cherchons à détecter.

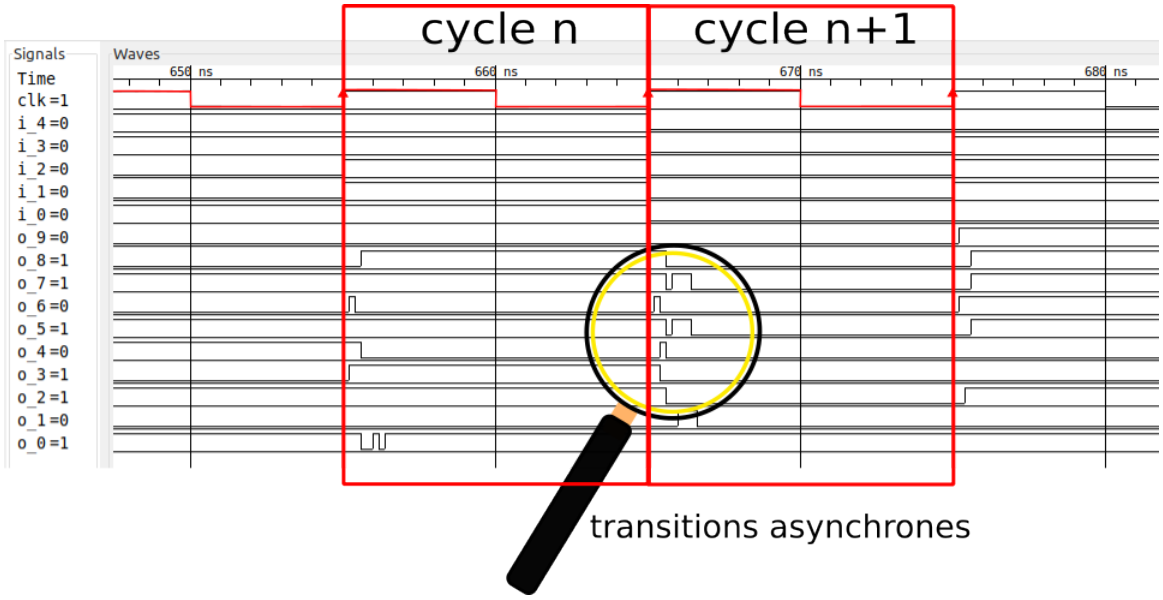


FIGURE 12.2 – Illustration du régime transitoire sur une sortie d'un circuit : cette zone, généralement sans intérêt, contient suffisamment d'information pour révéler la *présence strucurelle* d'un HT.

12.4 Anomalies du comportement transitoire

12.4.1 Définition

Observons les deux circuits de la figure 12.3, que nous suivrons comme fil rouge. Leurs fonctions booléennes respectives sont les mêmes de part et d'autre : $y = (a + b).c$. Le circuit de droite présente par contre un buffer supplémentaire (porte "yes") qui reflète une modification de structure, impliquée par la présence du HT : le signal initial c est considéré comme la cible d'implantation du trojan.

En faisant l'hypothèse d'un modèle de délai unitaire et uniforme (toutes les portes ont un délai de 1 unité temporelle), on peut écrire deux nouvelles équations, qui sont désormais différentes l'une de l'autre.

$$C : y(t) = (a(t-2) + b(t-2)).c(t-1)$$

$$C' : y(t) = (a(t-2) + b(t-2)).c(t-2)$$

Ces équations, qui se nomment *boolean differential equations* (BDE) dans la littérature, permettent désormais de définir la notion d'anomalie de comportement transitoire ou ATB (*anomaly of transient behavior*) : c'est l'occurence, à un instant t_α, de deux valeurs $y(t_\alpha) \neq y'(t_\alpha)$ où $t_0 < t_\alpha < t_0 + \gamma$ et où t_0 représente l'instant d'une mise à jour synchrone de l'ensemble des entrées du circuit (front montant d'une horloge).

1. On pourrait penser qu'il y a lieu de chercher à minimiser ces transitions à des fins d'optimisation de consommation. Cependant, la synthèse logique multi-niveaux ne permet plus ce contrôle ultime

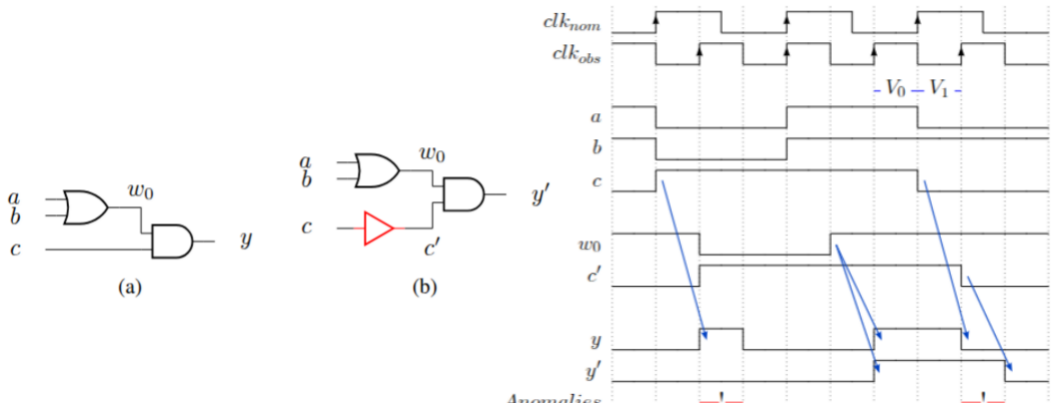


FIGURE 12.3 – Exemple : un circuit C et sa version infectée C'. Les deux circuits exhibent le même comportement synchrone fonctionnel, mais des différences de comportement transitoire apparaissent.

12.4.2 Vecteur générateur d'anomalies

Position du problème La question est désormais de pouvoir exciter le circuit de manière à *engendrer* de telles anomalies ATB en *sortie* du circuit. Il est en effet inconcevable de considérer l'ensemble des combinaisons de valeurs possibles sur les entrées : il faut déterminer quels stimuli particuliers déclenchent ces anomalies.

Avant d'aller plus loin, il est toutefois nécessaire de préciser que l'ensemble des circuits que nous considérons sont des circuits synchrones, dont les entrées sont échantillonnées dans des registres (bascules D). Aussi, si l'on cherche à engendrer des ATB en sortie d'une logique combinatoire, il faudra générer une *paire* de vecteurs, les deux vecteurs étant appliqués sur deux cycles d'horloge consécutifs.

Système de contraintes Dans l'exemple fil rouge précédent, l'observation d'une anomalie en sortie ne peut s'effectuer que dans des conditions précises : la fonction de la porte `and`, qui intervient avant la sortie y, force la présence d'une valeur à 1 sur son entrée supérieure, *afin* que la présence du HT à son entrée inférieure puisse être observée. Ces conditions nécessaires à la détection en sortie peuvent s'écrire comme un système de contraintes équationnelles, judicieusement posées sur les entrées :

$$y(t_\alpha) \neq y'(t_\alpha) \Leftrightarrow \begin{cases} a(t_\alpha - 2) + b(t_\alpha - 2) = 1 \\ c(t_\alpha - 1) \neq c(t_\alpha - 2) \end{cases}$$

Information

La possibilité de poser un tel système d'équations ouvre la voie à l'automatisation. On peut calculer les valeurs aux entrées, qui provoquent de telles différences de comportement en sortie.

12.4.3 Analyse de la propagation

Rappelons que $w_0(t) = a(t-1) + b(t-1)$ est le signal interne tel que $y(t) = w_0(t-1) \cdot c(t-1)$ Ce système indique que, pour propager une transition du signal cible c vers la sortie, le signal interne $w_0(t)$ doit être mis à 1. Dans le diagramme temporel présenté à la figure 12.3, les deux anomalies observées correspondent précisément à cette description mathématique. La seconde anomalie est générée par la paire de vecteurs de test suivante : $(\vec{V_0}, \vec{V_1}) = (111, 010)$. Cette paire introduit une transition de front descendant à l'entrée c tout en assurant sa propagation vers la sortie y. Pour ce faire, le signal w_0 est mis à 1 pendant les deux cycles concernés. Cette valeur spécifique, qui permet à la porte ET de toujours propager la valeur du signal c, est appelée "valeur non contrôlante", par opposition à une "valeur contrôlante", qui forcerait la sortie de la porte ET à 0, indépendamment de c. De cette façon, le front descendant à l'entrée c est propagé à la sortie y sur les deux circuits C et C'. Ainsi, la différence de retard entre c et c' causée par la falsification structurelle est visible aux sorties, et ceci de manière systématique.

Résolution par *SAT solver* Notre objectif est d'exciter notre circuit à l'aide d'une paire de vecteurs, qui provoque l'ATB en sortie. Comment trouver automatiquement cette paire ? Il est assez naturel de recourir à un solveur SAT : dans notre cas, `z3` a été retenu [2]. On sait que de l'amélioration continue des capacités de tels *solvers* sont les enjeux d'une âpre compétition internationale : on peut bénéficier directement, y compris pour des problèmes de grande taille. La formulation précédente du système de contraintes équationnelles booléennes présente cependant une difficulté inhabituelle pour les SAT solvers : comment représenter le décalage temporel présent dans ces équations BDE ? La solution, très élégante, consiste à considérer les signaux retardés comme de simples nouvelles variables libres, correctement distinguées par un nommage approprié. Ainsi un signal $x(t-d)$ est renommé x_t_d, etc. Il est par ailleurs nécessaire d'encoder le fait que les deux vecteurs de la paire sont appliqués à des cycles d'horloge différents.

12.5 Observation à fréquence décalée

Notre procédé impose de réfléchir non seulement sur la génération des vecteurs déclencheurs d'anomalie, mais également sur l'observation de transitions en dehors de leur zone de stabilité. Si cela ne pose pas de difficulté en simulation pure, la conception d'un banc de test physique doit être soigneusement réfléchie. Il est basé sur les principes suivants :

— **Génération d'une horloge légèrement décalée en fréquence**. Si la fréquence nominale du circuit est f, on cherchera à réaliser les observations à une fréquence $f_{obs} = f \times (1+\epsilon)$. Ce décalage en fréquence, permet de réaliser une *observation continue* d'un cycle d'horloge : à chaque cycle, le front montage de l'horloge d'observation se décale dans le cycle nominal, ce qui permet d'obtenir un "scanning" progressif de l'ensemble du cycle. Cette observation peut se faire également par une modification dynamique de la phase de l'horloge.

— **Recours à des synchroniseurs de domaines d'horloges** : étant donné que les deux horloges en présence sont asynchrones l'une de l'autre, il existe un risque de *métastabilité* de la valeur échantillonnée. Il est donc nécessaire de réaliser l'échantillonnage de la logique combinatoire comme il se doit, en utilisant des synchroniseurs (par exemple 2 à trois bascules D cadencées par la fréquence d'observation). Ces dispositifs permettent d'éviter la propagation de valeurs métastables en aval. Elles assurent de manière probabiliste la présence d'une valeurs stable `0` ou `1` en sortie. Ces valeurs ne doivent toutefois pas être considérées comme porteuse d'information fonctionnelle. Pour rappel, le MTBF (temps moyen entre deux défaillances) pour un synchroniseur de base peut être estimé à l'aide de la formule suivante :

$$\text{MTBF} = \frac{e^{(t_r/\tau)}}{f_c \cdot f_d \cdot \lambda}$$

où :
— t_r : Temps de résolution de la bascule (temps disponible pour que la métastabilité se résolve).
— τ : Constante de temps de métastabilité de la bascule.
— f_d : Fréquence initiale du signal d'entrée.
— f_{obs} : Fréquence d'horloge du système d'observation.
— λ : Taux d'échec de la bascule (généralement fourni par le fabricant).

12.6 Questions ouvertes et expériences en cours

Expériences en cours Les expériences en cours de soumission à publication (cible ICCAD 2025) visent à présenter la méthode, ainsi que des résultats préliminaires. Ces expériences, menées à la fois sur des benchmarks internationaux et des netlists synthétiques générées, consistent à :

— Infecter un circuit initial C par un HT. Un circuit C' est obtenu.
— A partir d'une analyse du circuit C, on détermine (avec `z3`) un couple de vecteurs qui permet d'engendrer une ATB sur une sortie particulière, si une altération structurelle est réalisée sur le circuit.
— Excitation des deux circuits C et C' à l'aide de ces vecteurs. Cette stimulation doit se faire de manière répétée, afin d'avoir une chance d'échantillonner l'ATB provoquée.

2. `z3` est en réalité un solveur SMT, qui inclut un moteur SAT.

Cette expérience est encore en cours, mais jusqu'ici 100% des circuits infectés ont pu être ainsi détectés.

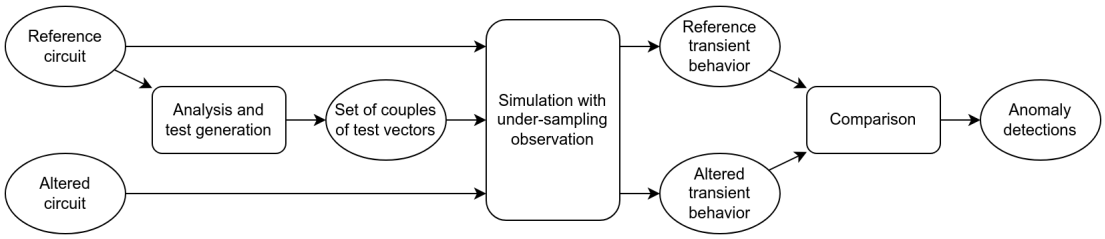


FIGURE 12.4 – Setup de l'expérience en cours, permettant de détecter une altération structurelle due à la présence d'un cheval de Troie.

Impact du modèle de timing sur la méthode Les expériences précédentes ont été menées à l'aide d'un modèle de timing volontairement simpliste : chaque porte est simplement caractérisée par un délai entier. On peut d'ailleurs noter que l'ensemble du comportement d'évolution des signaux transitoires peut se représenter comme un automate d'état finis sur lequel des expériences de *model checking* ont également été entamées. Dans la réalité, le timing des bibliothèques fondeur est évidemment bien plus complexe : lors d'une transition, les signaux peuvent ne pas transiter entre les valeurs 0 à 1 de manière franche. Nous devons donc étudier l'impact d'un modèle de temps plus réaliste sur notre détection. Notamment les temps de montée et descente des signaux (*slew rate*) doivent être pris en considération. Ils incorporent notamment différents *effets capacitifs*.

— Nous cherchons à vérifier si notre méthode résiste à l'augmentation de précision du modèle de portes.
— Nous pensons que les vecteurs déclencheurs d'ATB, constitués sur des hypothèses de délai très simples, conservent en partie leurs effets lorsqu'un modèle de timing plus réaliste est adopté. Ce dernier impacte par contre la durée d'observation nécessaire à la détection des HT : il s'agira pour nous de mesurer le facteur de répétition des observations.

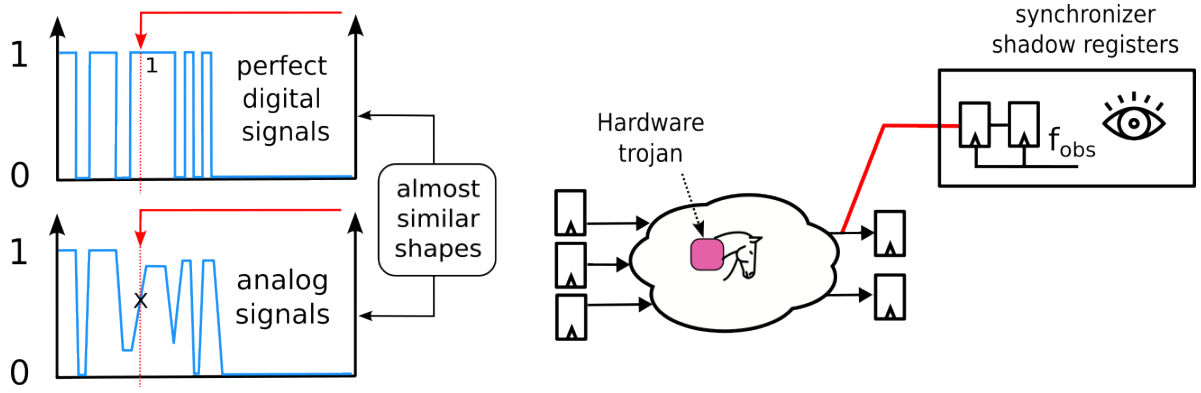


FIGURE 12.5 – Question ouverte : notre méthode resiste-t-elle à un modèle de timing plus réaliste ?

12.7 Conclusion

Ce chapitre présente une activité de recherche entamée dans la thèse actuelle de Quentin Tual, supervisée par Philippe Coussy et moi-même. La thèse est supportée par Creach Labs, qui résulte d'un partenariat institutionnel entre l'État français, le Conseil Régional de Bretagne et différentes institutions académiques bretonnes. Le sujet porte sur la détection de Chevaux de Troie insérés dans une netlist : cette attaque peut être conduite par des attaquants de type *insiders*, mais on peut souligner que des travaux (cite Pilato) ont également démontré que des outils de CAO bien évidemment sont en mesure de réaliser également de tels insertions. Ces Chevaux de Troie sont par définition difficiles à détecter : ils

résistent notamment à des batteries de tests classiques, orientés par des préoccupations de vérification de comportements fonctionnels nominaux. Nous défendons l'idée que l'insertion de tels Chevaux de Troie doit pouvoir se détecter dans le régime particulier –et généralement peu étudié– de transitions rapides qui agitent les circuits combinatoires avant leur stabilisation. De premières idées ont été proposées ici : on cherche en particulier à trouver des vecteurs particuliers qui déclenchent des transitions anormales en sortie, révélatrices d'une altération structurelle du circuit étudié. L'observation conjointe de telles anomalies pose des challenges nouveaux qu'il s'agira de relever.

Cinquième partie

Perspectives de Recherche
&
Conclusion

Chapitre 13

Emulation matérielle de Systèmes Cyber-Physiques

There's plenty of room at the bottom.

Richard P. Feynman
Prix Nobel de Physique 1965

Publications et projets associés

— [C14] **Newcas'22** HLS-based Accelerated Simulation of Large Scale Cyber-Physical Systems on FPGAs.
— [Z4] Thèse de Maélic Louart. Conception d'un récepteur AIS détectant les falsifications de messages : développement de stratégies et prototypage sur FPGA. Cyber-Physical Systems on FPGAs.
— Thèse de Pierre Filiol. Accélération matérielle du calcul par intervalles pour la Robotique mobile.
— Projet AID Tectonic.

Sommaire

13.1 Introduction

Dans ce chapitre, j'esquisse un projet de recherche qui ambitionne de faciliter la **modélisation conjointe de systèmes embarqués et de leur environnement physique** à l'aide du concept d'acteurs.

Du System-on-Chip au Système Cyber-Physique : un nouveau pas à franchir Comme précédemment rappelé, l'avénement des System-on-Chip a permis l'intégration d'un grand nombre de composants qui,

dans des systèmes embarqués précédents, se présentaient comme des composants discrets. L'usage du terme *"Système"* est donc approprié de ce point de vue. Mais il ne s'agit effectivement que d'un point de vue : le mot *"Système"* peut se référer à des systèmes plus vastes, dépassant *a priori* largement le cadre d'une seule puce, aussi vaste soit-elle. Cela était déjà vrai dans mon domaine métier initial de le compression vidéo : la lentille de la caméra, la fiabilité de la transmission, la mobilité de l'ensemble du dispositif, jusqu'aux effets psycho-visuels d'une image trop parfaite, faisaient déjà partie intégrante des préoccupations essentielles à la réussite industrielle d'un projet de tels SoC. Le **couplage fort** de tous ces éléments doivent être pris en compte lors de la conception. Cette conception globale (*holistique*) dépasse évidemment les seules préoccupations électroniciennes et couvre de nombreuses disciplines. C'est le domaine des systèmes dits "Cyber-Physiques" ou CPS (cyber physical systems).

Systèmes Cyber-Physiques Un système embarqué peut se définir comme un système en interaction forte avec son environnement physique. A l'inverse un système cyber-physique (CPS) *inclut* de tels systèmes embarqués, ainsi que l'environnement lui-même. Au sein d'un tel CPS, les systèmes embarqués sont généralement en relation : soit à travers un réseau informatique traditionnel, soit par l'entremise de leurs actions respectives sur l'environnement (systèmes hybrides). Alors que le système embarqué trouve sa raison d'être dans des applications spécifiques et localisées, le thème des CPS trouve toutefois sa justification dans des applications plus larges, comme les véhicules autonomes, les réseaux de distribution d'énergie ou les systèmes de transport intelligents. Le CPS permet la collecte, le traitement et l'analyse de grandes quantités de données provenant de multiples sources multiples à des fins de prise de décisions globales. Les CPS participent ainsi à l'émergence du *Edge Computing*.

Un problème ouvert La mise en oeuvre concrète de ces approches multi-disciplinaires reste un problème ouvert. En particulier, le **maquettage de ces systèmes**, à des fins d'analyse préliminaire, est délicat :

— Nécessité de trouver des abstractions satisfaisantes, autant temporelles que fonctionnelles.
— Nécessité de faire cohabiter des simulateurs variés.
— Nécessité de pouvoir incorporer des composants clés (*hardware-in-the-loop*).

Emulation matérielle L'outillage en matière de conception de CPS reste très parcellaire, essentiellement dominé par des solutions propriétaires (Matlab/Simulink, Labview, etc). Ces solutions, qui faconnent le paysage du **model-based design** peuvent, selon moi, être en grande partie substituées par l'idée centrale d'**émulation matérielle**.

Idée

L'idée centrale est de créer des **laboratoires In Silico** pour la simulation des CPS, à des fins de mise au point. A l'aide du concept d'acteur exposé précédemment, je cherche à incorporer dans cette simulation à la fois des éléments du futur système embarqué, mais également des modèles d'environnement physiques, tout en profitant du parallélisme naturelle des plateformes FPGA et de leur capacité d'accélération.

Détournement de l'EDA La possibilité de concevoir les SoC modernes a démontré la capacité des outils EDA/ESL à absorber des systèmes de grande taille. L'idée centrale défendue et illustrée ici est que les difficultés rencontrées dans le domaine de la modélisation et la simulation de CPS peuvent trouver une solution dans les technologies de l'EDA/ESL . L'originalité de l'approche réside dans le détournement de ces techniques au profit de nouvelles applications : il s'agit de bénéficier de l'ensemble de la démarche de mise au point de tels SoC pour la simulation des CPS.

13.2 Les Mockups de l'UC Riverside : une source d'inspiration

Des travaux très intéressants ont été conduits dès 2010-2011 par l'équipe de Vahid, Givargis, Huang et Miller à l'Université de Californie à Riverside [M50],[M51][M37], [M36],etc. Ces travaux visaient à la mise au point d'appareils médicaux comme des respirateurs artificiels ou des pacemakers : cette mise au point est traditionnellement complexe car elle nécessite la présence de patients aux affections variées.

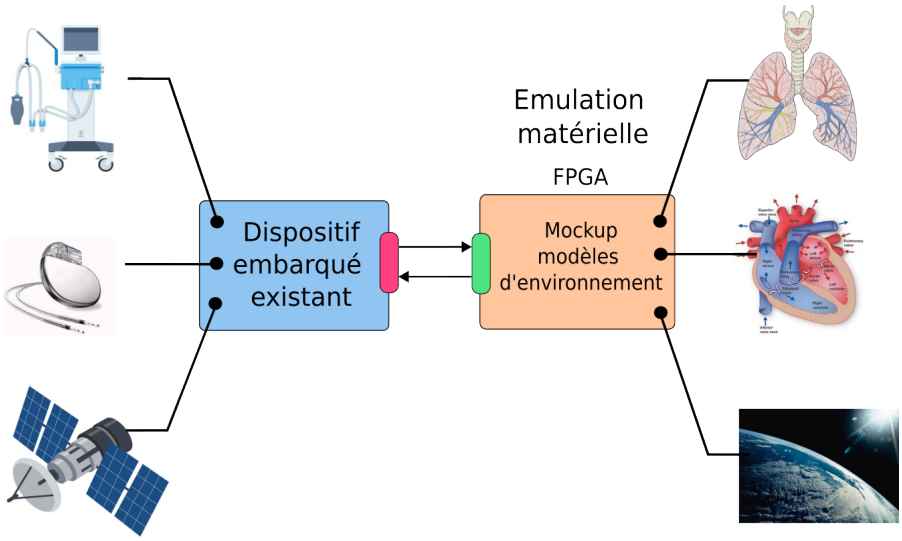


FIGURE 13.1 – Digital Mockups de Vahid, Givargis, Huang et Miller à UC Riverside : dispositifs (médicaux, télécom,...) pré-existants testés à l'aide de modèles d'environnement émulés sur FPGA

Les auteurs ont eu l'idée de substituer ces patients par des modèles des organes concernés, synthétisés sur FPGA : ils dénomment ces modèles des "mockups" (maquettes).

Avantages De tels mockups présentent plusieurs intérêts :

- Disponibilité permanente de l'organe à tester.
- Création possible de situations inhabituelles ou extrêmes.
- Reproductibilité
- Randomisation
- etc.

Outillage On peut noter par exemple que ces auteurs ont cherché à explorer différents styles de conception matérielle : ils ont ainsi bénéficié de la HLS, mais ont également développé, au niveau RTL, des accélérateurs dédiés (programmables ou non). Ainsi, dans le cas de la modélisation du poumon, ils ont développé un solveur de systèmes d'équations différentielles ordinaires (ODE) à l'aide d'un ensemble de PE (processing element), le tout synthétisé sur Xilinx Virtex6 130T. Un SoC approprié, basé sur Microblaze, permet l'exploitation de ce système d'émulation physique. Si les auteurs sont effectivement parvenus à démontrer la viabilité de l'approche mockup, ils soulignent dans [**?**] la nécessité de bénéficier dans le futur d'une chaîne d'outils de co-design hw/sw dédiée à la création et l'exploitation de tels mockups.

Opportunités Le thème de recherche esquissé pour la suite de mes travaux peut ainsi s'appuyer en grande partie sur mes travaux passés dans le domaine de l'EDA/ESL :

- La création de DSLs (chapitre 4,5,6 et 7)
- La modélisation dataflow par acteurs et leur synthèse (chapitre 6 et 8).
- Le raffinement incrémental (chapitre 9).
- La conception de dispositifs reconfigurables (chapitre 10).

Précision

A la différence des travaux menés à Riverside, le projet d'émulation matérielle de CPS proposé ici passe par la synthèse complète du système embarqué et de son environnement sur un même FPGA.

13.3 Exemple 1 : Récepteur AIS détecteur d'anomalies.

J'ai pu arborder ce nouveau thème à travers des premiers projets financés et des thèses. Dans la thèse de Maélic Louart [Z4] (Prix Daveluy 2024), effectuée en collaboration avec l'Ecole Navale [1], nous avons abordé ce problème de manière concrète. Il s'agissait de concevoir un récepteur AIS détecteur d'anomalies. Nous avons mis au point ce système embarqué, en le plongeant d'emblée dans un environnement physico-informatique semi-réaliste [C14] : l'ensemble de ce CPS a été modélisé, synthétisé puis simulé sur FPGA. Cette mise au point du système complet *in silico* a permis d'extraire par la suite le récepteur seul, qui a fonctionné immédiatement dans le cas de signaux réels (rade de Brest).

L'AIS : principe, failles et enjeux Pour rappel, l'AIS (Automatic Identification Systeme) est un système de suivi et d'identification automatique utilisé par les navires et les services de trafic maritime. Il permet aux navires de partager des informations avec d'autres navires et stations côtières, telles que leur position, leur vitesse, leur cap, leur port de destination, la nature de leur cargaison autres données pertinentes . L'Organisation maritime internationale (OMI) exige que tous les navires de commerce d'une jauge brute supérieure à 300 tonnes et tous les navires à passagers, quelle que soit leur taille, soient équipés d'un système AIS. L'AIS est également très répandu dans le domaine de la pêche côtière et de la plaisance. Ce système est toutefois la cible de différentes fraudes : les informations transmises par l'AIS dépendent de la saisie correcte des données par l'équipage. Des erreurs ou des falsifications sont possibles, allant de la simple modification de ces informations en cours de traversée à la création radio-fréquence de véritables bateaux fantômes. Ces failles sont notoirement mises à profit dans différents traffics, allant de pêches occasionnelles mais illicites en zones naturelles protégées, à des organisations mafieuses à l'échelle internationale : traffic d'armes, contournement d'embargo, fraude à l'assurance, échange d'identité, approche de sites interdits, etc.

Emulation d'un système complet AIS sur FPGA La figure 13.2 illustre un système réalisé sur FPGA lors de la thèse de Maélic Louart. Ce système qui a été synthétisé sur un unique FPGA inclut à la fois des émetteurs AIS, un récepteur AIS (objectif de conception), mais également un modèle de canal physique de communication. Nos objectifs étaient multiples :

- Bénéficier d'un modèle de base décodeur AIS, afin d'en comprendre l'algorithme central.
- Test systématique en présence de plusieurs émetteurs, ce qui reflète l'environnement radio-fréquence (ici en bande de base).
- Raffinement progressif du décodeur détecteur d'anomalies, au fur et à mesure de l'élaboration de nouvelles stratégies algorithmiques.
- Possibilité de piloter différentes expériences à partir d'un jeu de paramètres envoyés par un ordinateur connecté au système.

Comme on peut le voir sur la figure, il s'agit ici d'un `CPS complet` : comme indiqué il incorpore en particulier un premier modèle du canal de communication. Ce dernier émule trois effets physiques importants :

- Un décalage en fréquence
- Un ajout de bruit.
- Un affaiblissement du signal dû à la propagation : distance, obstacle, etc

L'affaiblissement est calculé avec une loi physique connue sous le terme d'*Equation des Télécommunications* ou équation de Friis.

$$\frac{P_r}{P_t} = G_t G_r (\frac{\lambda}{4\pi R})^2$$

où P et G représentent de repectivement les puissances (émise et reçues) et les gains (émetteur et récepteur), R la distance entre l'émetteur et récepteur et λ est la longueur d'onde de travail. Cette équation permet d'obtenir un ordre de grandeur de la puissance radio collectée par un récepteur situé à une certaine distance d'un émetteur en espace libre, avec une polarisation supposée correcte.

Retour d'expérience L'ensemble émetteur unique et récepteur représente environ une vingtaine d'algorithmes (blocs), qui ont été synthétisés avec Vitis HLS puis assemblés (avec une directive `dataflow`) pour former le SoC complet. Il nous a été possible d'ajouter jusqu'à 10 émetteurs sur un FPGA Artix7 100T. Le calcul est parfaitement **spatialisé**. Plusieurs constats peuvent être réalisés :

1. Collaboration avec Abdel Boudraa, Jean-Jacques Szkolnik (IRENAV) et Frédéric Le Roy

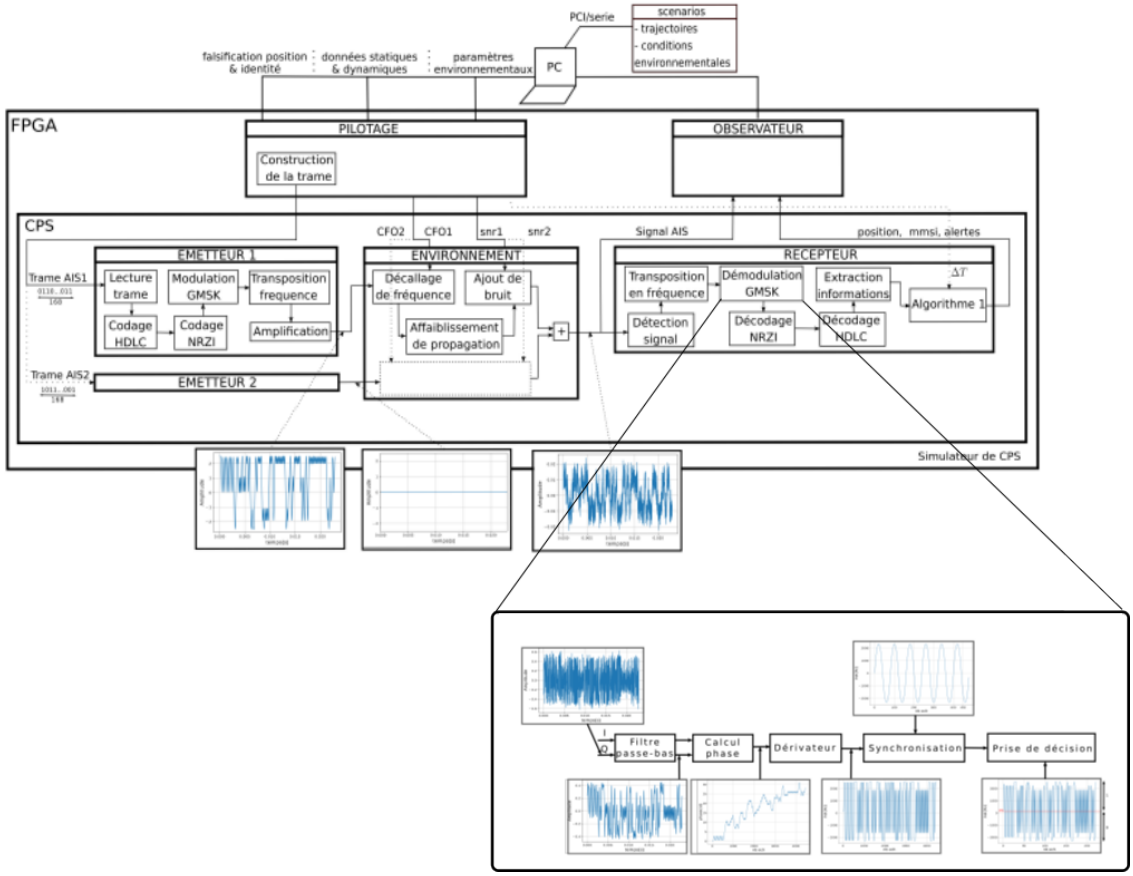


FIGURE 13.2 – Architecture détaillée du CPS : émetteur(s), environnement et récepteur(s) sont co-émulés sur un même FPGA. L'ensemble a été synthétisé à l'aide de directives `dataflow` à l'aide de l'outil Xilinx Vitis (HLS).

1. Aucune perte de performance lors de l'ajout progressif d'un nombre croissant d'émetteurs n'a été constatée, alors que ce n'est absolument pas le cas avec une approche logicielle, où le système est simulé sur PC.
2. Le ratio d'accélération par rapport à une exécution purement logiciel (`c++`) de cet ensemble algorithmique est d'environ **x1000** [C14].
3. Il existe de nombreux codes `c++` qui modélisent tel ou tel phénomène, tel ou tel procotole, etc : ces modèles, rendus synthétisables avec un minimuml d'effort, grâce aux progrès de la HLS, permettent de **bypasser les outils "model-based" traditionnels**.
4. Les outils HLS autorisent un non-expert RTL à élaborer de tels modèles complexes.

Ces constats concourent à suggérer l'émulation matérielle comme :
— une alternative de choix à la simulation logicielle classique : accélération très importante (2 ordres de grandeur visés).
— une alternative aux approches model-based telles que suggérées par les constructeurs d'outils : disponibilité des modèles `c/c++` (notamment issus du HPC), au lieu de modèles propriétaires Matlab ou Simulink.

13.4 Exemple 2 : Navigation autonome en Robotique (2 cas)

La mise au point de systèmes en robotique mobile marine présente des difficultés organisationnelles évidentes : il est notamment compliqué d'organiser le test de dispositifs in situ (tests en mer ou bassins d'essai, autorisations spécifiques, etc). Cette mise au point requiert l'existence de maquettes physiques déjà très abouties et robustes (marinisation etc), bien éloignées des aspects algorithmiques le

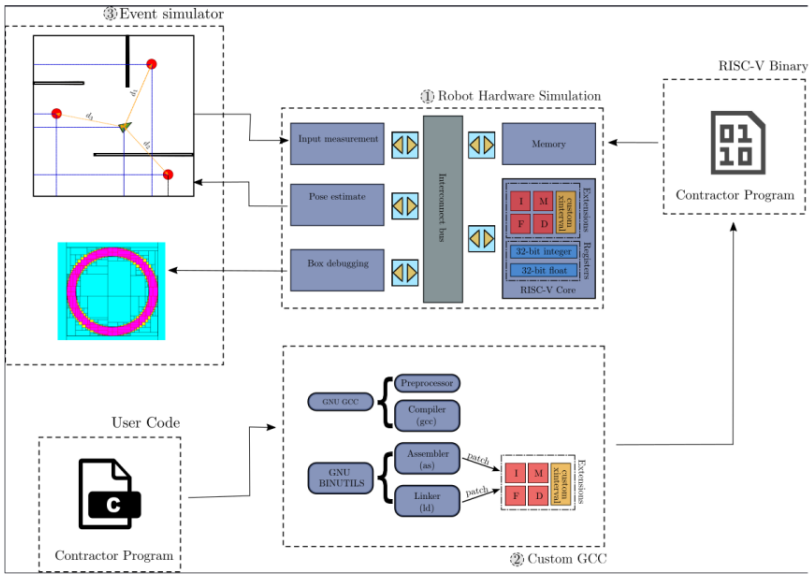


FIGURE 13.3 – Dans la thèse de Pierre Filiol, on s'intéresse à la conception de systèmes embarqués pour la robotique autonome, augmentés d'un accélérateur pour le calcul par intervalles porté sur SoC FPGA. La question de la mise au point de tels systèmes CPS se pose : ici le robot cherche à se localiser grâce à le mesure incertaine de distances à 3 amers (en rouge).

plus souvent à l'épreuve. Là encore, il est intéressant de pouvoir disposer de *prototypes virtuels émulés matériellement*, et sans les complexités logicielles habituelles qui leur sont liées.

13.4.1 Cas de la thèse de Pierre Filiol : système basé RISC-V et calcul par intervalles

Ainsi, dans la thèse de Pierre Filiol, une des expériences en cours porte sur la localisation d'un engin flottant à partir de mesures incertaines de distances d'amers figure 13.3. Si l'outil théorique central est la théorie des intervalles développée par Luc Jaulin (et qui dépasse le cadre du présent manuscrit), tout un ensemble d'outils informatiques relativement complexes sont développés dans le projet :

— Coprocesseur matériel dédié au calcul par intervalles et contracteurs, synthétisé sur FPGA.
— Emulateur logiciel de processeur RISCV.
— Simulateur de robot dans son environnement.
— DSL de formulation du problème de calcul par intervalles
— etc

La mise à l'épreuve de cet ensemble immanquablement hétérogène est à nouveau complexe, mais se présente comme un cas d'étude pour l'émulation de CPS.

13.4.2 Cas du projet AID Tectonic : navigation de drône sans GPS

Le projet AID[2] Ecoles appelé Tectonic, constitue un second cas lié à la robotique mobile. Nous cherchons à concevoir le système embarqué d'un drône longue distance devant se déplacer d'un point A à un point B sans l'aide de GPS. La solution retenue actuellement est essentiellement basée sur une caméra embarquée, et la reconnaissance de points d'intérêt le long d'un parcours. Cette reconnaissance permet in fine la correction de trajectoires. Là encore, la mise au point d'un tel système embarqué nécessite une prise en compte d'un très grand nombre de paramètres que seule une simulation ou une émulation rapide permettent d'explorer : parmi ces paramètres on peut imaginer des conditions météorologiques variées, mais également la détection erronnée d'un point d'intérêt, etc.

2. Agence d'Innovation de Défense

13.5 Structure du programme de recherche

13.5.1 Quatre thèmes

Les premières expériences, dont celles évoquées ici, conduisent à quatre thèmes de travail :

Idée

1. Application pratique du modèle d'acteurs à des fins d'émulation matérielle de CPS.
2. Mécanismes d'abstraction et de raffinement entre les abstractions RTL et comportementales.
3. Modélisation spécifique à l'émulation de phénomènes physiques :
   — Modélisation du temps
   — Prise en compte de modèles *acausaux*.
4. Stratégies d'interconnexion entre les modèles de transducteurs et les modèles d'environnement.

13.5.2 Exemples de questions soulevées

Plusieurs questions doivent être soulevées avant d'envisager une telle émulation. Ces questions permettent de réfléchir à la structure d'un programme de recherche.

1. **Synthèse HLS de dispositifs classiquement conçus au niveau RTL** Le cas des ISS (instruction set simulator) est intéressant. Une question est alors la suivante : est-il possible de synthétiser de tels ISS, de manière à obtenir des processeurs efficaces ? Cette éventualité permet d'envisager de créer des acteurs modélisant des processeurs comme de simples interpréteurs d'instructions, et de bénéficier de leur synthèse matérielle. Il se trouve que la question de la synthèse HLS d'ISS est une question récemment posée dans la communauté EDA et Architecture des Ordinateurs. Ainsi la thèse de Jean-Michel Gorius [M28] (thèse dirigée Steven Derrien) à l'IRISA Rennes traite de ce sujet. Une des problématiques scientifiques abordées dans cette thèse est celle de la synthèse automatique d'un processeur *correctement pipeliné*, à l'aide de nouvelles techniques de HLS, dont la spéculation avancée. Au dela des seuls ISS, la généralisation de la modélisation et synthèse comportementales à des dispositifs traditionnellement conçus au niveau RTL se pose à bien d'autres circuits : contrôleurs mémoires de type SDRAM, bridges entre protocoles de bus, etc.
2. **Emulation des capteurs et actionneurs** Le degré de réalisme de l'émulation de tels systèmes robotique n'est pas exclusivement lié à l'exécution du logiciel embarqué (par exemple sur RISCV), mais également à la *réplique des capteurs* : il est important, durant l'émulation du CPS, de pouvoir répliquer des flux de données capteurs relativement proches de la réalité. Sans cela par exemple, une régulation émulée a des fortes chances de n'être d'aucune utilité pour la mise au point du système final. Une des solutions envisageable est l'injection, dans l'émulation, de flux capteurs réels *pré-enregistrés*. Ce pré-enregistrement et cette restitution doivent être automatisés à l'aide d'outils adéquats qu'il reste à faire émerger.
3. **Harmonisation de rythmes caractéristiques** Emuler un système hétérogène comme un CPS sur un FPGA passe par la synthèse matérielle de sous-parties aux caractéritiques temporelles très variées : par exemple phénomène mécanique lents versus processeur haute-fréqunce. Pourtant, la synthèse logique repose sur les mêmes principes, quels que soient les modèles : il se peut par exemple que la fréquence de fonctionnement atteinte en synthèse soit sensiblement la même pour deux composants qui émulent des phénomènes aux caractérisitiques temporelles très différentes l'une de l'autre. Il y a donc lieu de chercher à *composer* a posteriori ces sous-systèmes en respectant leur fréquence propre. Ceci peut se faire à l'aide de PLL savamment programmées, mais il est vraisemblable que des mécanismes plus abstraits permettent l'expression, puis l'exécution de telles *harmonisation de rythmes*. Ce sujet délicat a été abordé par un grand nombre de travaux autour des langages synchrones : Polychrony ou CCSL en particulier considèrent la notion d'horloges logiques comme les fondations de cette expression, à travers des systèmes de contraintes équationnelles.

13.6 La place centrale d'`Archipel`

Dans ces travaux futurs, la place d'`Archipel` est centrale.

1. Il doit me permettre de continuer à étudier des mécanismes de synthèse HLS au plus près, sans dépendances à des outils tiers.
2. Il me permet l'adjonction de nouveaux mots clés reflétant des concepts indispensables aux CPS :
   — Déclaration d'horloges logiques et leur synchronisation.
   — Modélisation *acausale*, naturelle pour les systèmes physiques (équations différentielles).
   — etc.
3. Constitution de bibliothèques d'acteurs :
   — Acteurs jouant le rôle d'interfaces classiques : accès SDRAM, PCI, etc.
   — Acteurs calculatoires classiques et récurrents : FFT ou réseau de neurones,etc.
   — etc

13.7 Conclusion

Mon projet de recherche propose de développer des technologies EDA/ESL qui permettent d'émuler matériellement des systèmes cyber-physiques complets. Le concept d'acteurs flot-de-données est placé au centre de la méthodologie associée.

Ces acteurs modélisent le comportement attendu de chacun des éléments du système final : il peut s'agir de futurs logiciels embarqués, de matériel (futur ou existant) ou d'un modèle d'environnement physique. L'ensemble de la démarche vise à *plonger le futur système embarqué* dans un environnement semi-réaliste, et ceci *au plus tôt* dans le flot de conception : on peut parler de *laboratoires In Silico*.

Les avantages liés à cette co-modélisation se présentent selon différents axes.
— **Analyse amont** : dès les premiers stades de la conception, il est possible de simuler et d'appréhender le comportement d'ensemble du CPS et du système embarqué en particulier.
— **Conception incrémentale** : l'approche combinée de la modélisation par acteurs et l'aide de la HLS permet d'envisager des itérations courtes, avec une montée en complexité progressive de chacun des acteurs.
— **Vérification continue** : le projet de recherche revient à réaliser une **extension des bancs de tests classiques** de circuits : alors que dans l'approche traditionnelle les stimuli sont généralement figés et stockés dans un système de fichiers externes, il est ici question de modéliser des interactions plus riches du système embarqué avec son environnement. Cette approche doit permettre d'explorer une variété de situations jusqu'alors hors de portée d'une approche traditionnelle.

Chapitre 14

Conclusion

> Roads ? Where we're going, we don't need roads
>
> Dr. Emmett L. Brown
> (Retour vers le Futur)

Ce manuscrit a présenté un ensemble de travaux autour de la conception des SoCs.

Mes contributions portent en premier lieu sur la création de langages spécialisés (DSL) permettant une première capture du futur système. Cette démarche s'inscrit dans un mouvement de fond en Informatique [M49] qui permet une modélisation plus directe que ne le permettent des langages plus généralistes. A travers cette nouvelle forme de modélisation, c'est une meilleure appropriation des problèmes auquels font face les concepteurs qui devient ainsi possible.

La seconde partie du manuscrit illustre mes activités en matière d'architecture de ces SoCs. A travers la synthèse automatique à haut niveau et la création d'overlays FPGA, le concepteur est, là aussi, en mesure de gagner en flexibilité : tandis que la HLS promet une meilleure exploration des solutions architecturales, les overlays lui offrent à la fois de gagner en programmabilité et en indépendance technologique.

Ces deux premières parties ont donc en commun l'idée centrale d'une plus grande *liberté* dans la conception. L'idée de démocratiser la conception, en créant ainsi de tels DSL et DSA (domain specific architectures) *à façon*, semble désormais crédible à l'échelle d'une entreprise, et peut se révéler comme un véritable *game changer*.

La troisième partie du document a illustré mon implication dans le domaine de la sécurité des circuits numériques. Des enjeux importants se jouent autour des SoCs : ils vont du respect de la propriété intellectuelle jusqu'à des questions de souveraineté encore plus critiques. Deux thèses dans lesquelles je suis impliqué concernent la sécurisation du flot de conception lui-même.

Enfin, mon projet de recherche qui se dessine à l'avenir est celui de la mise au point de ces SoCs : la nécessité, en phase de vérification, de "faire vivre" ces futurs SoC dans des environnements dynamiques, réalistes ou semi-réalistes me parait un terrain de jeu fantastique. Ce projet d'émulation électronique (et massivement parallèle) de systèmes cyber-physiques complets est aussi une occasion intéressante de chercher à appliquer les techniques de conception –y compris traditionnelles– des SoCs à des domaines éloignés, qui en ignoraient a priori l'existence.

Bibliographie

Thèse de doctorat

[T1] **Le Lann** Jean-Christophe. Simulation et synthèse de circuits s'appuyant sur le modèle synchrone, 2002. Thèse de doctorat d'Informatique. Thèse dirigée par Le Guernic, Paul . IRISA et Universié de Rennes 1.

Thèses supervisées

[Z1] Hannah Badier. *Transient obfuscation for HLS security : application to cloud security, birthmarking and hardware Trojan defense*. PhD thesis, 2021. Thèse de doctorat dirigée par Gogniat, Guy et Coussy, Philippe et **Le Lann, Jean-Christophe**. Brest, École nationale supérieure de techniques avancées Bretagne.

[Z2] Théotime Bollengier. *Du prototypage à l'exploitation d'overlays FPGA*. PhD thesis, 2018. Thèse de doctorat dirigée par Lagadec, Loïc et **Le Lann, Jean-Christophe**. École nationale supérieure de techniques avancées Bretagne.

[Z3] Nader Khammassi. *Modèle de programmation de haut niveau pour la parallélisation expicite et automatique : application aux architectures multicoeurs. (High-level structured programming models for explicit and automatic parallelization on multicore architectures)*. PhD thesis. Thèse de doctorat dirigée par Diguet Jean-Philippe et **Le Lann, Jean-Christophe**. Université de Bretagne Sud 2014.

[Z4] Maelic Louart. *Conception d'un récepteur AIS détectant les falsifications de messages : développement de stratégies et prototypage sur FPGA*. Theses, ENSTA Bretagne, July 2023. Thèse de doctorat dirigée par Abdel-Ouahab Boudraa, Jacques Szkolnik and Jean-Christophe **Le Lann**.

Chapitres de livres

[B1] Hannah Badier, **Le Lann** Jean-Christophe, Philippe Coussy, and Guy Gogniat. Protecting behavioral IPs during design time : Key-based obfuscation techniques for HLS in the cloud. In Srinivas Katkoori and Sheikh Ariful Islam, editors, *Behavioral Synthesis for Hardware Security*. Springer, 2022.

[B2] Nader Khammassi and **Le Lann** Jean-Christophe. XPU : A C++ metaprogramming approach to ease parallelism expression : Parallelization methodology, internal design and practical application. In Mikhail S. Tarkov, editor, *Parallel Programming : Practical Aspects, Models and Current Limitations*, chapter 8, pages 175–198. Springer, 2014.

[B3] Ali Koudri, Joel Champeau, **Le Lann** Jean-Christophe, and Vincent Leilde. Mopcom methodology : Focus on models of computation. In Thomas Kühne, Bran Selic, Marie-Pierre Gervais, and François Terrier, editors, *Modelling Foundations and Applications*, pages 189–200, Berlin, Heidelberg, 2010. Springer.

Brevets

[P1] Théotime Bollengier, **Le Lann** Jean-Christophe, Loïc Lagadec, and Ciprien Teodorov. Procedé de configuration d'un circuit logique programmable., jan 2022. Brevet Français FR3115134.

[P2] **Le Lann** Jean-Christophe, Christophe Jollivet, Gildas Cocherel, and Mickael Fossard. Arithmetic decoding method and device, jan 2011. US Patent 7,876,240.

[P3] **Le Lann** Jean-Christophe and Pierre-Laurent Lagalaye. Procédé de synthèse de haut niveau d'une application, 2011. Brevet Français FR1156501A.

Revues internationales

[R1] Pierre Filiol, Théotime Bollengier, Luc Jaulin, and Jean-Christophe **Le Lann**. A new interval arithmetic to generate the complementary of contractors. *Acta Cybern.*, 26(4) :817–838, 2024.

[R2] Pierre Filiol, Théotime Bollengier, Luc Jaulin, and Jean-Christophe **Le Lann**. RISC-V based hardware acceleration of interval contractor primitives in the context of mobile robotics. *Acta Cybern.*, 26(4) :889–912, 2024.

[R3] Apostolos A. Kountouris, Christophe Wolinski, and Jean-Christophe **Le Lann**. High-level synthesis using hierarchical conditional dependency graphs in the Codesis system. *Journal of Systems Architecture*, 47(3) :293 – 313, 2001. Modern methods and tools in digital system design.

[R4] Loïc Lagadec, Ciprian Teodorov, **Le Lann** Jean-Christophe, Damien Picard, and Erwan Fabiani. Model-Driven Toolset for Embedded Reconfigurable Cores : Flexible Prototyping and Software-like Debugging. *Science of Computer Programming*, page 1, March 2014.

[R5] Paul Le Guernic, Jean-Pierre Talpin, and **Le Lann** Jean-Christophe. Polychrony for system design. *Journal of Systems Architecture*, 12 :261–304, 2002.

[R6] Maelic Louart, Jean-Jacques Szkolnik, Abdel-Ouahab Boudraa, Jean-Christophe **Le Lann**, and Frédéric Le Roy. Detection of ais messages falsifications and spoofing by checking messages compliance with tdma protocol. *Journal of Digital Signal Processing*, page 103983, 2023.

[R7] Maelic Louart, Jean-Jacques Szkolnik, Abdel-Ouahab Boudraa, Jean-Christophe **Le Lann**, and Frédéric Le Roy. An approach to detect identity spoofing in AIS messages. *Expert Syst. Appl.*, 252 :124257, 2024.

[R8] Mohamad Najem, Théotime Bollengier, **Le Lann** Jean-Christophe, and Loïc Lagadec. Extended overlay architectures for heterogeneous FPGA cluster management. *Journal of Systems Architecture*, 78 :1–14, 2017.

Conférences internationales avec comité de lecture

[C1] I. Ashraf, K. Bertels, N. Khammassi, and J.C. **Le Lann**. Communication-aware parallelization strategies for high performance applications. In *2015 IEEE Computer Society Annual Symposium on VLSI (IVLSI'15)*, pages 539–544, July 2015.

[C2] Hannah Badier, Christian Pilato, **Le Lann** Jean-Christophe, Philippe Coussy, and Guy Gogniat. Opportunistic ip birthmarking using side effects of code transformations on high-level synthesis. In *2021 Design, Automation and Test in Europe Conference (DATE'21)*, pages 52–55, 2021.

[C3] Hannah Badier, Jean-Christophe **Le Lann**, Philippe Coussy, and Guy Gogniat. Transient key-based obfuscation for HLS in an untrusted cloud environment. In *Proc. Design, Automation and Test in Europe Conference (DATE '19)*, Florence, Italy, March 2019. IEEE.

[C4] Théotime Bollengier, Loïc Lagadec, Mohamad Najem, **Le Lann** Jean-Christophe, and Pierre Guilloux. Soft timing closure for soft programmable logic cores : The ARGen approach . In *ARC 2017 - 13th International Symposium on Applied Reconfigurable Computing*, Delft, Netherlands, April 2017. Delft University of Technology.

[C5] Gilbert Edelin, Philippe Bonnot, Waelle Gouja, Koen K. Bertels, Axel Schneider, J Knablein, Bernard B. Pottier, and Jean-Christophe **Le Lann**. A programming toolset enabling exploitation of reconfiguration for increased flexibility in future system-on-chips. In *Proc. Design, Automation and Test in Europe Conference (DATE '07).*, Nice, France, April 2007. IEEE.

[C6] Julien Heulot, Karol Desnos, Jean François Nezan, Maxime Pelcat, Mickaël Raulet, Hervé Yviquel, Pierre-Laurent Lagalaye, and **Le Lann** Jean-Christophe. An experimental toolchain based on high-level dataflow models of computation for heterogeneous mpsoc. In *DASIP*, Karlsruhe, Germany, October 2012.

[C7] **Le Lann** Jean-Christophe, Theotime Bollengier, Mohamad Najem, and Loic Lagadec. An integrated toolchain for overlay-centric system-on-chip. In *13th International Symposium on Reconfigurable Communication-centric Systems-on-Chip, ReCoSoC 2017, July 9–11, 2018*, Lille, France, July 2018. IEEE.

[C8] **Le Lann** Jean-Christophe and Christophe Wolinksi. Load balancing and functional unit assignment in high-level synthesis. In *SCI'99/ISAS'9*, Orlando, Floride, U.S.A, March 1999.

[C9] Nader Khammassi and **Le Lann** Jean-Christophe. Design and implementation of a cache hierarchy-aware task scheduling for parallel loops on multicore architectures. In *PDCTA'14, Parallel, Distributed Computing Technologies and Applications*, Sydney, Australia, February 2014.

[C10] Nader Khammassi, **Le Lann** Jean-Christophe, Jean-Philippe Diguet, and Alexandre Skrzyniarz. MHPM : Multi-scale hybrid programming model : A flexible parallelization methodology. In *2012 IEEE 14th International Conference on High Performance Computing and Communication (HPCC)*, pages 71–80, 2012.

[C11] Ali Koudri, Denis Aulagnier, Didier Vojtisek, Philippe Soulard, Christophe Moy, Joël Champeau, Jorgiano Vidal, and Jean-Christophe **Le Lann**. Using MARTE in a Co-Design Methodology. In *MARTE UML profile workshop co-located with DATE'08*, page 6 pages, Munich, Germany, March 2008.

[C12] Xuan Sang Le, Luc Fabresse, Jannik Laval, **Le Lann** Jean-Christophe, and Loïc Lagadec. Speeding Up Robot Control Software Through Seamless Integration With FPGA . In *SHARC'16*, Brest, France, June 2016.

[C13] Xuan Sang Le, **Le Lann** Jean-Christophe, Lagadec Loïc, Luc Fabresse, Noury Bouraqadi, and Jannik Laval. CaRDIN : An Agile Environment for Edge Computing on Reconfigurable Sensor Networks. In *3rd IEEE International Conference on Computational Science and Computational Intelligence (CSCI 2016)*, Las Vegas, United States, December 2016.

[C14] Maelic Louart, **Le Lann** Jean-Christophe, Frédéric Le Roy, Abdel Boudraa, and Jean-Jacques Szkolnik. HLS-based accelerated simulation of large scale cyber-physical systems on FPGAs. In *2022 20th IEEE Interregional NEWCAS Conference)*, pages 332–336, 2022.

[C15] M. Najem, T. Bollengier, J.C **Le Lann**, and L. Lagadec. A cost-effective approach for efficient time-sharing of reconfigurable architectures. In *2017 International Conference on FPGA Reconfiguration for General-Purpose Computing (FPGA4GPC)*, pages 7–12, May 2017.

[C16] Muhammad Rashid, **Le Lann** Jean-Christophe, and Koen Bertels. Video encoding analysis for parallel execution on reconfigurable architectures. In *DASD'08, 6th Symposium on Design, Analysis, and Simulation of Distributed Systems*, Edinburgh, UK, 2008.

[C17] Jean-Philippe Schneider, Zoé Drey, and **Le Lann** Jean-Christophe. Early exploring design alternatives of smart sensor software with Model of Computation implemented with actors. In *ESUG 2013 - 21th International Smalltalk Conference*, page xx, Annecy, France, September 2013.

[C18] Jean-Christophe **Le Lann**. An HLS algorithm for the direct synthesis of complex control flow graphs into finite state machines with implicit datapath. In *27th Euromicro Conference on Digital System Design, DSD 2024, Paris, France, August 28-30, 2024*, pages 227–233. IEEE, 2024.

Workshop internationaux

[W1] Jalil Boukhobza, Loïc Lagadec, Alain Plantec, and **Le Lann** Jean-Christophe. CDFG Platform in MORPHEUS. In *Workshop AMWAS'07*, Paris, France, October 2007.

[W2] **Le Lann** Jean-Christophe. Operand isolation using signal clock calculus. In *1st Workshop on Low Power Design.*, Sheffield, United Kingdom, March 1998.

[W3] **Le Lann** Jean-Christophe, Hannah Badier, and Florent Kermarrec. Towards a Hardware DSL Ecosystem : RubyRTL and Friends. In *OSDA'2020 Open Source Hardware Design, colocated with DATE'20*, Grenoble, France, March 2020.

[W4] Florent Kermarrec, Sebastien Bourdeauducq, Jean-Christophe **Le Lann**, and Hannah Badier. LiteX : an open-source soc builder and library based on migen python DSL. In *Proc. Design, Automation and Test in Europe Conference (DATE '19). Open Source Design Automation Workshop.*, Florence, Italy, March 2019. IEEE.

[W5] Nader Khammassi and **Le Lann** Jean-Christophe. Tackling real-time signal processing applications on shared memory multicore architectures using XPU. In *ERTS 2014*, page xx, Toulouse, France, February 2014.

[W6] Ali Koudri, **Le Lann** Jean-Christophe, and Joël Champeau. UML/Marte process for SoC/SoPC. In *ERTS'10 - Embedded Real Time Software and Systems Symposium*, pages 201–209, 2010.

Conférences nationales

[N1] **Le Lann** Jean-Christophe. Génération automatique de code vhdl à partir de signal. In *Journees AAA98 - Adequation-Algorithme-Architecture, CEA*, Saclay, Franc, January 1998.

[N2] **Le Lann** Jean-Christophe and Philippe Dhaussy. Synthèse de controleurs numeriques par composition de contraintes applicatives et temporelles. In *Gretsi'13 Colloque Francophone de Traitement du Signal et des Images*, Brest, France, Sep 2013.

[N3] **Le Lann** Jean-Christophe, Joel,Champeau, Papa,Issa Diallo, and Pierre-Laurent, Lagalaye. From system-level models to heterogeneous embedded systems. In *RITF'12 Recherche et Innovation pour les Transports du Futur*, Paris, nov 2012.

[N4] **Le Lann** Jean-Christophe, Pierre-Laurent Lagalaye, and Philippe Dhaussy. Modélisation algorithmique et synthèse d'architectures assistees par model-checking. In *CAL'12- Conference sur les Architectures Logicielles*, Montpellier, France, mai 2012.

[N5] Maelic Louart, **Le Lann** Jean-Christophe, , Jean-Jacques Szkolnik, Abdel-Ouahab Boudraa, and Frédéric Le Roy. Émulation de systèmes cyber-physiques sur fpga. In *Gretsi'22 Colloque Francophone de Traitement du Signal et des Images*, Nancy, France, September 2022.

[N6] Maelic Louart, Jean-Jacques Szkolnik, Abdel-Ouahab Boudraa, **Le Lann** Jean-Christophe, and Frédéric Le Roy. Stratégie de détection des Falsifications des Positions des Messages AIS Basée sur l'Application du Filtre IMM. In *Gretsi'22 Colloque Francophone de Traitement du Signal et des Images*, Nancy, France, September 2022.

[N7] Bollengier Theotime, Najem Mohamad, **Le Lann** Jean-Christophe, and Lagadec Loic. Zeff : Une plateforme pour l'intégration d'architectures overlay dans le cloud. In *COMPAS'16- Conférence d'informatique en Parallélisme, Architecture et Système*, Lorient, France, 2016.

Présentations en GDR

[G1] **Le Lann** Jean-Christophe and Nader Khammssi. A high-level hybrid programming model for heterogeneous multicore architectures. In *GDR SoC-Sip*, Paris, juin 2012.

[G2] **Le Lann** Jean-Christophe and Loïc Lagadec. Overlays FPGA pour le Cloud. In *GDR ISIS*, Paris, juin 2012.

[G3] Lagadec Loïc and **Le Lann** Jean-Christophe. End-to-end environment for overlays. In *GDR SoC SiP'16*, Paris, Juin 2018.

[G4] Olivier Reynet, **Le Lann** Jean-Christophe, and Benoît Clément. JOG : une approche haut niveau des systèmes embarqués via armadeus et java. In *Journées Démonstrateurs en robotique*, Angers, juin 2010.

[G5] LE Xuan Sang, Fabresse Luc, Laval Jannik, **Le Lann** Jean-Christophe, Lagadec Loïc, and Bouraqadi Noury. Dynamic distributed programming on reconfigurable ip-based smart sensor network. In *GDR SoC-SiP*, Nantes, 2016.

Rapports techniques notables

[Z1] **Le Lann** Jean-Christophe, Pottier Bernard, Godet Matthieu, and Keryell. Ronan. Loosely coupled accelerators for reconfigurable SoC. In *Technical Report, ENST Bretagne*, May 2007.

[Z2] Allemand Michel, Bodin François, Kountouris Apostolos, Le Guernic Paul, **Le Lann** Jean-Christophe, Seznec André, and Wolinksi Christophe. A synchronous approach for hardware design. In *Technical Report 1131, IRISA*, Octobre 1997.

Bibliographie générale

[M1] Ieee standard for vhdl language reference manual. *IEEE Std 1076-2019*, pages 1–673, 2019.

[M2] Samar Abdi, Dongwan Shin, and Daniel Gajski. Automatic communication refinement for system level design. In *Proceedings of the 40th Design Automation Conference, DAC 2003, Anaheim, CA, USA, June 2-6, 2003*, pages 300–305. ACM, 2003.

[M3] Rick Bahr, Clark Barrett, Nikhil Bhagdikar, Alex Carsello, Ross Daly, Caleb Donovick, David Durst, Kayvon Fatahalian, Kathleen Feng, Pat Hanrahan, Teguh Hofstee, Mark Horowitz, Dillon Huff, Fredrik Kjolstad, Taeyoung Kong, Qiaoyi Liu, Makai Mann, Jackson Melchert, Ankita Nayak, Aina Niemetz, Gedeon Nyengele, Priyanka Raina, Stephen Richardson, Raj Setaluri, Jeff Setter, Kavya Sreedhar, Maxwell Strange, James Thomas, Christopher Torng, Leonard Truong, Nestan Tsiskaridze, and Keyi Zhang. Creating an agile hardware design flow. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.

[M4] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. *Proc. IEEE*, 91(1) :64–83, 2003.

[M5] Gérard Berry and Georges Gonthier. The esterel synchronous programming language : design, semantics, implementation. *Science of Computer Programming*, 19(2) :87–152, 1992.

[M6] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, 39(2) :1–7, 2011.

[M7] Nicola Bombieri, Nicola Deganello, and Franco Fummi. Integrating rtl ips into tlm designs through automatic transactor generation. In *2008 Design, Automation and Test in Europe*, pages 15–20, 2008.

[M8] Sébastien Bourdeauducq. Milkymist, un system-on-chip libre et orienté vidéo temps réel. *GNU/-Linux Magazine*, (124), février 2010.

[M9] Sébastien Bourdeauducq. Migen, une « boîte à outils » en python pour concevoir des circuits logiques complexes. *GNU/Linux Magazine*, (149), mai 2012.

[M10] Chad P. Bown. How the united states marched the semiconductor industry into its trade war with china. *East Asian Economic Review*, 24(4) :349–388, December 2020. Posted : 15 Jan 2021 ; Last revised : 28 Jan 2021.

[M11] Clair Brown and Greg Linden. *Chips and Change : How Crisis Reshapes the Semiconductor Industry*. The MIT Press, 2009.

[M12] Mario Bucev, Samuel Chassot, Simon Felix, Filip Schramka, and Viktor Kuncak. Formally verifiable generated ASN.1/ACN encoders and decoders : A case study. *CoRR*, abs/2412.07235, 2024.

[M13] Jean Paul Calvez. A codesign case study with the MCSE methodology. *Des. Autom. Embed. Syst.*, 1(3) :183–212, 1996.

[M14] Yuze Chi, Weikang Qiao, Atefeh Sohrabizadeh, Jie Wang, and Jason Cong. Democratizing domain-specific computing. *Communications of the ACM*, 66(1) :74–85, 2022.

[M15] Christian Collberg and Jasvir Nagra. *Surreptitious Software : Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional, 1st edition, 2009.

[M16] Benoit Combemale, Robert France, Jean-Marc Jézéquel, Bernhard Rumpe, James Steel, and Didier Vojtisek. *Engineering modeling languages : Turning domain knowledge into tools*. CRC Press, 2016.

[M17] Katherine Compton and Scott Hauck. Reconfigurable computing : a survey of systems and software. *ACM Computing Surveys (csuR)*, 34(2) :171–210, 2002.

[M18] Philippe Coussy, Daniel D. Gajski, Michael Meredith, and Andres Takach. An introduction to high-level synthesis. *IEEE Design and Test of Computers*, 26(4) :8–17, 2009.

[M19] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4) :451–490, October 1991.

[M20] J. Dekeyser, P. Boulet, P. Marquet, and S. Meftali. Model driven engineering for soc co-design. In *The 3rd International IEEE-NEWCAS Conference, 2005.*, pages 21–25, 2005.

[M21] Stephen A. Edwards. *The Specification and Execution of Heterogeneous Synchronous Reactive Systems*. PhD thesis, Berkeley University, 1997.

[M22] Umer Farooq, Roselyne Chotin-Avot, Moazam Azeem, Maminionja Ravoson, and Habib Mehrez. Novel architectural space exploration environment for multi-fpga based prototyping systems. *Microprocessors and Microsystems*, 56 :169–183, 2018.

[M23] Håkan Forsberg, Kristina Forsberg, and Joakim Lindén. The importance of a system-level approach when bringing in new technologies in avionics. In *2024 AIAA DATC/IEEE 43rd Digital Avionics Systems Conference (DASC)*, pages 1–11. IEEE, 2024.

[M24] Daniel D. Gajski, Nikil D. Dutt, Allen C.-H. Wu, and Steve Y.-L. Lin. *High-level synthesis : introduction to chip and system design*. Kluwer Academic Publishers, USA, 1992.

[M25] Abdoulaye Gamatié. *Designing Embedded Systems with the SIGNAL Programming Language - Synchronous, Reactive Specification*. Springer, 2010.

[M26] Abdoulaye Gamatié, Sébastien Le Beux, Éric Piel, Rabie Ben Atitallah, Anne Etien, Philippe Marquet, and Jean-Luc Dekeyser. A model-driven design framework for massively parallel embedded systems. *ACM Trans. Embed. Comput. Syst.*, 10(4), November 2011.

[M27] Frank Ghenassia. *Transaction level modeling with SystemC*. Springer, 2005.

[M28] Jean-Michel Gorius. *Synthèse de haut niveau de processeurs à jeu d'instructions*. PhD thesis, 2024. Thèse de doctorat dirigée par Derrien, Steven Informatique Université de Rennes.

[M29] David Greaves and Myoung Jin Nam. Synthesis of glue logic, transactors, multiplexors and serialisors from protocol specifications. volume 2010, pages 1 – 7, 10 2010.

[M30] Zhenghua Gu, Wenqing Wan, Jundong Xie, and Chang Wu. Dependency graph-based high-level synthesis for maximum instruction parallelism. *ACM Trans. Reconfigurable Technol. Syst.*, 14(4), September 2021.

[M31] David Harel. Statecharts : a visual formalism for complex systems. *Science of Computer Programming*, 8(3) :231–274, 1987.

[M32] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis – the symta/s approach. *IEE Proceedings - Computers and Digital Techniques*, 152 :148–166, 2005.

[M33] John L Hennessy and David A Patterson. A new golden age for computer architecture. *Communications of the ACM*, 62(2) :48–60, 2019.

[M34] Mokrani Hocine, Rabéa Ameur-Boulifa, and Emmanuelle Encrenaz. *Assisting Refinement in System-on-Chip Design*, volume 311, pages 21–42. 08 2014.

[M35] Denis Hommais. *Une méthode d'évaluation et de synthèse des communications dans les systèmes intégrés matériel-logiciel*. Theses, Sorbonne Université, 1999. Thèse de doctorat dirigée par Frédéric Pétrot et Alain Greiner.

[M36] Chen Huang, Bailey Miller, Frank Vahid, and Tony Givargis. Synthesis of custom networks of heterogeneous processing elements for complex physical system emulation. In Ahmed Jerraya, Luca P. Carloni, Naehyuck Chang, and Franco Fummi, editors, *Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2012, part of ESWeek '12 Eighth Embedded Systems Week, Tampere, Finland, October 7-12, 2012*, pages 215–224. ACM, 2012.

[M37] Chen Huang, Bailey Miller, Frank Vahid, and Tony Givargis. Synthesis of networks of custom processing elements for real-time physical system emulation. *ACM Trans. Design Autom. Electr. Syst.*, 18(2) :21 :1–21 :21, 2013.

[M38] Hamoudi Kalla, Jean-Pierre Talpin, David Berner, and Loïc Besnard. Automated translation of C/C++ models into a synchronous formalism. In *13th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS 2006), 27-30 March 2006, Potsdam, Germany*, pages 426–436. IEEE Computer Society, 2006.

[M39] Nachiket Kapre and Samuel Bayliss. Survey of domain-specific languages for fpga computing. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–12, 2016.

[M40] Alan C. Kay. The early history of smalltalk. In *The Second ACM SIGPLAN Conference on History of Programming Languages*, HOPL-II, page 69–95, New York, NY, USA, 1993. Association for Computing Machinery.

[M41] Sébastien Le Nours. Mémoire d'habilitation à diriger des recherches : Contributions to system-level modelling and simulation of hardware-software architectures of embedded systems., 2022.

[M42] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proc. IEEE*, 75(9) :1235–1245, 1987.

[M43] Zongwei Liu, Wang Zhang, and Fuquan Zhao. Impact, challenges and prospect of software-defined vehicles. *Automotive Innovation*, 5(2) :180–194, 2022.

[M44] Frédéric Mallet. *Temps Logique pour l'ingénierie dirigée par le modèles (Logical Time in Model-Driven Engineering)*. 2010.

[M45] Florence Maraninchi. *Modélisation et validation des systèmes réactifs : un langage synchrone à base d'automates*. Habilitation à diriger des recherches, Grenoble 1 UJF - Université Joseph Fourier, May 1997.

[M46] Florence Maraninchi and Yann Rémond. Argos : an automaton-based synchronous language. *Computer Languages*, (27) :61–92, 2001.

[M47] Kevin JM Martin. Twenty years of automated methods for mapping applications on cgra. In *2022 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*, pages 679–686. IEEE, 2022.

[M48] Marco Mattavelli, Jörn W. Janneck, and Mickaël Raulet. MPEG reconfigurable video coding. In Shuvra S. Bhattacharyya, Ed F. Deprettere, Rainer Leupers, and Jarmo Takala, editors, *Handbook of Signal Processing Systems*, pages 43–67. Springer, 2010.

[M49] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4) :316–344, December 2005.

[M50] Bailey Miller, Frank VahicK, and Tony Givargis. Application-specific codesign platform generation for digital mockups in cyber-physical systems. In *2011 Electronic System Level Synthesis Conference (ESLsyn)*, pages 1–6, 2011.

[M51] Bailey Miller, Frank Vahid, and Tony Givargis. MEDS : mockup electronic data sheets for automated testing of cyber-physical systems using digital mockups. In Wolfgang Rosenstiel and Lothar Thiele, editors, *2012 Design, Automation & Test in Europe Conference & Exhibition, DATE 2012, Dresden, Germany, March 12-16, 2012*, pages 1417–1420. IEEE, 2012.

[M52] Jean-François Nezan. *Mémoire d'habilitation à diriger des recherches : Prototypage rapide d'applications de traitement des images sur systèmes embarqués. (Rapid prototyping of image processing applications on embedded systems)*. 2009.

[M53] Veronica Opranescu and Anca Daniela Ionita. Review of cyber-physical systems modeling with uml, sysml, and marte. *IEEE Access*, 13 :47132–47145, 2025.

[M54] Maxime Pelcat, Karol Desnos, Julien Heulot, Clément Guy, Jean-François Nezan, and Slaheddine Aridhi. Preesm : A dataflow-based rapid prototyping framework for simplifying multicore dsp programming. In *2014 6th european embedded design in education and research conference (EDERC)*, pages 36–40. IEEE, 2014.

[M55] Christian Pilato, Kanad Basu, Francesco Regazzoni, and Ramesh Karri. Black-hat high-level synthesis : Myth or reality ? *IEEE Trans. Very Large Scale Integr. Syst.*, 27(4) :913–926, 2019.

[M56] Katalin Popovici, Xavier Guerin, Frederic Rousseau, Pier Stanislao Paolucci, and Ahmed Amine Jerraya. Platform-based software design flow for heterogeneous mpsoc. *ACM Trans. Embed. Comput. Syst.*, 7(4), August 2008.

[M57] Dumitru Potop-Butucaru, Stephen A. Edwards, and Gerard Berry. *Compiling Esterel*. Springer Publishing Company, Incorporated, 1st edition, 2007.

[M58] Fabrice Rastello. *SSA-based Compiler Design*. Springer Publishing Company, Incorporated, 1st edition, 2016.

[M59] Samuel Rogers, Joshua Slycord, Mohammadreza Baharani, and Hamed Tabkhi. gem5-salam : A system architecture for llvm-based accelerator modeling. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 471–482, 2020.

[M60] Alberto Sangiovanni-Vincentelli. Quo vadis, sld ? reasoning about the trends and challenges of system level design. *Proceedings of the IEEE*, 95(3) :467–506, 2007.

[M61] Alberto L. Sangiovanni-Vincentelli. The tides of EDA. *IEEE Des. Test Comput.*, 20(6) :59–75, 2003.

[M62] Phillippe Sauter, Thomas Benz, Paul Scheffler, Hannah Pochert, Luisa Wüthrich, Martin Povišer, Beat Muheim, Frank K. Gürkaynak, and Luca Benini. Croc : An end-to-end open-source extensible risc-v mcu platform to democratize silicon, 2025.

[M63] Lui Sha, Tarek Abdelzaher, Karl-Erik Årzén, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and Aloysius K Mok. Real time scheduling theory : A historical perspective. *Real-time systems*, 28 :101–155, 2004.

[M64] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Cheddar : a flexible real time scheduling framework. In *Proceedings of the 2004 Annual ACM SIGAda International Conference on Ada : The Engineering of Correct and Reliable Software for Real-Time & Distributed Systems Using Ada and Related Technologies*, SIGAda '04, page 1–8, New York, NY, USA, 2004. Association for Computing Machinery.

[M65] Stephen Y. H. Su. The current status and future work of design automation in japan. *SIGDA Newsl.*, 4(1) :44–47, January 1974.

[M66] Walter HW Tuttlebee. *Software defined radio : enabling technologies*. John Wiley & Sons, 2002.

[M67] Daniel C. Wang, Andrew W. Appel, Jeffrey L. Korn, and Christopher S. Serra. The zephyr abstract syntax description language. In Chris Ramming, editor, *Proceedings of the Conference on Domain-Specific Languages, DSL'97, Santa Barbara, California, USA, October 15-17, 1997*, pages 213–228. USENIX, 1997.

[M68] Zheyao Wang. 3-d integration and through-silicon vias in mems and microsensors. *Journal of Microelectromechanical Systems*, 24(5) :1211–1244, 2015.

[M69] T. Wild, A. Herkersdorf, and R. Ohlendorf. Performance evaluation for system-on-chip architectures using trace-based transaction level simulation. In *Proceedings of DATE Conference*, volume 1, pages 1–6, 2006.

[M70] Matthieu Wipliez, Ghislain Roquier, and Jean-François Nezan. Software code generation for the RVC-CAL language. *J. Signal Process. Syst.*, 63(2) :203–213, 2011.

[M71] Wayne Wolf, Ahmed Amine Jerraya, and Grant Martin. Multiprocessor system-on-chip (mpsoc) technology. *IEEE transactions on computer-aided design of integrated circuits and systems*, 27(10) :1701–1713, 2008.

[M72] Christophe Wolinski and Apostolos A. Kountouris. Hierarchical Conditional Dependency Graphs for Conditional Resource Sharing . In *EUROMICRO Conference*, volume 2, page 10313, Los Alamitos, CA, USA, August 1998. IEEE Computer Society.

[M73] Henry Wai-chung Yeung, Shaopeng Huang, and Yuqing Xing. From fabless to fabs everywhere ? semiconductor global value chains in transition. 2023.

[M74] Huafeng Yu, Jean-Pierre Talpin, Loïc Besnard, Thierry Gautier, Frédéric Mallet, Charles André, and Robert de Simone. Polychronous analysis of timing constraints in UML MARTE. In *13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops, ISORC Workshops 2010, Carmona, Sevilla, Spain, May 4-7, 2010*, pages 145–151. IEEE Computer Society, 2010.

DOCTORAT BRETAGNE LOIRE / MATHSTIC

Université de Bretagne Occidentale

Titre : Contributions aux méthodologies et outils de conception des system-on-chips : capture applicative, architecture et sécurité

Mot clés : Electronique au niveau système, ESL, SoC, conception de systèmes numériques, codesign hardware-software, FPGA, compilation, modélisation, DSL

Résumé : Ce manuscrit présente une synthèse de mes activités de recherche, en grande partie réalisées à l'ENSTA Bretagne (désormais ENSTA) et au sein du laboratoire Lab-STICC (UMR6285) depuis 2009. Cette activité s'est nourrie d'une expérience industrielle préalable d'une dizaine d'années en tant qu'ingénieur de la société Thomson R&D France, acteur mondial de la compression vidéo. J'ai ainsi eu la chance de participer activement à la conception de circuits intégrés de grandes dimensions : les System-on-Chip (SoC). A travers ce domaine de la compression, j'ai ainsi pu me plonger au coeur de la problématique des SoCs en général : comment, à partir de comportements réputés complexes, faire émerger une structure matérielle performante qui réalise ces calculs ? Une grande partie de la réponse à cette question repose sur les capacités des outils logiciels utilisés lors de la conception. Après mon activité d'architecte et concepteur, où j'utilisais de tels outils, mon activité de recherche s'est donc logiquement recentrée sur l'étude et le prototypage de ces outils. Cette recherche s'est déroulée à la fois dans le domaine académique mais également par le montage d'une start-up (Modaë Technologies) dans ce même domaine de l'ESL (Electronic System Level). Cet historique assez long m'amène à en couvrir, dans ce mémoire, plusieurs aspects : création de langages spécialisés (DSL domain specfic languages), techniques de compilation vers le matériel, ainsi que des propositions liées à la sécurité de ces "puces", désormais cibles de différentes attaques de type "cyber".

Title: Contributions to system-on-chip design methodologies and tools: application capture, architecture and security

Keywords: Electronic System Level, ESL, Digital System design, Hardware-software codesign, modeling, compilers, DSLs

Abstract: This manuscript presents a summary of my research activities, most of which were carried out at ENSTA Bretagne (now ENSTA) and within the Lab-STICC laboratory (UMR6285) since 2009. This activity was informed by ten years of industrial experience as an engineer at Thomson R&D France, a global player in video compression. I was fortunate to be actively involved in the design of large-scale integrated circuits: System-on-Chip (SoC). Through this field of compression, I was able to immerse myself in the heart of the problem of SoCs in general: how, from behaviours known to be complex, can we develop a high-performance hardware structure that can perform these calculations? Much of the answer to this question lies in the capabilities of the software tools used during design. After working as an architect and designer, where I used such tools, my research naturally refocused on the study and prototyping of these tools. This research took place both in academia and through the creation of a start-up (Modaë Technologies) in the same field of ESL (Electronic System Level). This fairly long background leads me to cover several aspects in this thesis: the creation of specialised languages (DSL domain-specific languages), compilation techniques for hardware, as well as proposals related to the security of these 'chips', which are now the target of various types of cyber attacks.