UNIVERSITÉ DU QUÉBEC À MONTRÉAL

ASSISTANCE À LA CRÉATION DE LISTES D'INTERCONNEXIONS DE COMPOSANTS ÉLECTRONIQUES

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

MOHAMED BADREDDINE

MARS 2014

REMERCIEMENTS

Je tiens à remercier sincèrement mes directeurs de recherche, les professeurs Yves Blaquière et Mounir Boukadoum, pour leurs conseils, leurs supports enthousiastes et les longues heures d'instruction qui nous ont permis de réaliser ce projet. Je remercie aussi nos partenaires industriels Gestion Technocap inc. et M. Richard Norman pour leur support financier et leurs conseils professionnels. Mes remerciements s'adressent aussi à CRSNG, PROMPT Québec, MITACS et surtout à la Faculté des sciences de l'Université du Québec à Montréal pour leur support financier.

Je remercie aussi les professeurs Roger Villemaire et Étienne Gagnon ainsi que mesdames Louise Tremblay et Lise Arsenault pour leur aide et leurs conseils aux étudiants étrangers.

Je désire remercier tous les professeures et professeurs qui ont contribué à ma formation pendant tout le long de mes études.

Mes remerciements particuliers à ma famille qui m'a donné du courage et de l'aide à tous les niveaux durant tout mon parcours académique.

TABLE DES MATIÈRES

LISTE DES FIGURES

RÉSUMÉ

L'investissement commercial et le temps de conception des circuits électroniques jouent un rôle critique dans la croissance économique et technologique. Les circuits électroniques peuvent être des cartes sur lesquelles se trouvent des composants ayant des dizaines, voire plusieurs centaines de broches connectées électriquement par des traces conductrices. Les composants électroniques se connectent ensemble pour devenir un système électronique tel un ordinateur, une caméra, etc. Cependant, les ingénieurs sont toujours confrontés à des difficultés de conception de circuits telles que la création de leurs dessins physiques, leurs tests et leurs fichiers de description (physiques et logiques) appelés *netlists*. Ce mémoire présente une méthodologie de création intelligente de *netlist* qui assiste le concepteur dans la conception d'un système électronique.

À partir des concepts et propriétés prédéfinis, la méthodologie proposée consiste à créer, dans une première étape, une représentation hiérarchique des broches d'un composant électronique qui répondent aux besoins du concepteur. Les broches d'un composant sont regroupées selon leurs propriétés fonctionnelles en des ensembles appelés *interfaces*. Puis, un algorithme identifie les interfaces compatibles entre les différents composants du circuit et les suggère à l'utilisateur dans une deuxième étape. Ensuite, dans la troisième étape, à partir des interfaces sélectionnées, un autre algorithme interconnecte automatiquement les broches des composants en respectant leurs compatibilités électriques. Un logiciel, nommé « *Intelligent Netlist Creator* » (INC), a été implémenté suivant la méthodologie proposée. Les résultats montrent que l'utilisation des interfaces facilite et accélère la création des *netlists* par rapport à leur création usuelle.

Une élaboration supplémentaire a été explorée dans le cadre de cette recherche pour suggérer, à l'aide d'un raisonnement à base de modèle, des formules mathématiques pour des connexions électriques qui rendent les circuits fonctionnels (ou contraintes comportementales). Cette exploration préliminaire montre que la faisabilité de l'apprentissage automatique des contraintes comportementales peut être appliquée à la méthodologie proposée. Au meilleur de notre connaissance, le logiciel INC serait le premier logiciel de type *conception assistée par ordinateur* qui montre la faisabilité d'utiliser l'apprentissage automatique pour assister le concepteur à la création d'un *netlist* de système électronique.

Mots-clés : conception assistée par ordinateur, *netlist*, forage de données, apprentissage automatique, raisonnement à base de modèle.

INTRODUCTION

Depuis les années 1950, l'électronique fait des progrès constants tant au niveau de la conception qu'à celui de la technologie. Avec cette évolution, une carte entière fabriquée quelques années plus tôt peut aujourd'hui se retrouver intégrée dans un circuit intégré [1, 2]. Cela est devenu possible surtout grâce à l'usage des programmes informatiques qui automatisent la plupart des tâches de conception ; plus connu sous l'expression *conception assistée par ordinateur*. Par exemple, grâce aux simulateurs de circuits *d'intégration à très grande échelle*, il est possible de simuler le comportement d'un circuit électronique intégré pouvant comprendre des millions de transistors [1, 2]. En même temps, l'investissement commercial et le temps de conception des circuits jouent un rôle critique [3, 4]. La microélectronique, l'un des domaines où les erreurs de conception coutent le plus cher, ne peut se permettre de fabriquer un circuit intégré, puis en cas de problème effectuer des modifications [4]. Subséquemment, les circuits intégrés reconfigurables qui ont été développés pour atténuer le problème de fabrication de circuits statiques. D'autres méthodes ont aussi été introduites comme l'usage de simulateurs de circuits. L'utilisation des simulateurs devient essentielle dans les méthodologies de conception des circuits intégrés pour détecter les erreurs de conception (logiques et électriques) et les corriger avant la production [4]. Les circuits intégrés sont ainsi validés et fabriqués dans une étape antérieure avant leurs interconnexions sur des cartes de circuits imprimés pour constituer un système électronique remplissant des fonctions définies. Cependant tout comme pour les circuits intégrés, une erreur de conception peut demander la reprise totale d'une carte de circuits imprimés et induire un retard sur l'échéancier de livraison des appareils (systèmes électroniques) qui les intègrent. Si une automatisation de création des circuits intégrés est faisable, est-il possible d'en faire autant pour les circuits imprimés afin d'accélérer leur fabrication ? Ce mémoire tente de répondre à cette question en proposant une solution.

Les composants électroniques possèdent chacun des broches (billes ou *pins*) qui leur permettent de se connecter électriquement les uns aux autres par des traces ou fils conducteurs afin de former un circuit. Une collection de broches connectées au même fil conducteur est appelée *nœud électrique* ou *signal net* ou simplement *net*. Un circuit est décrit par une liste de *nets* appelée *liste d'interconnexions* ou *netlist*. Autrement dit, un *netlist* est une représentation fonctionnelle d'une spécification donnée [16]. Le *netlist* d'un circuit imprimé décrit les connexions qui se basent sur la structure externe des composants en plus des caractéristiques électromagnétiques de chaque broche. La structure interne des composants n'est pas considérée dans l'approche proposée.

Compte tenu de la grande complexité des contraintes et de l'évolution rapide des technologies, comme la résolution des procédés de connexions, les nouveaux circuits intégrés et leurs interfaces, les développeurs de logiciels de conception assistée par ordinateur ne sont toujours pas parvenus à faciliter la conception d'un *netlist* malgré le progrès de l'informatique. Car, ils se sont souciés de représenter formellement et physiquement les composants électroniques tels que les concepteurs de circuits les conçoivent. Ce flot de conception est très rigide et gruge les ressources mentales des concepteurs de circuits qui consacrent un temps important sur les détails des composants tels que la gestion de leurs configurations, leurs connexions et les contraintes à respecter. Ces détails occupent leur attention pendant des semaines, voire des mois. Certaines contraintes électriques et physiques (dites *comportementales*) s'imposent durant la création d'un circuit imprimé afin que celui-ci devienne fonctionnel. Selon leur expérience, les concepteurs doivent déterminer manuellement les contraintes comportementales en considérant qu'il peut exister des composants de plusieurs centaines de broches dans un *netlist*. Aussi, la pratique avec les outils existants veut que l'utilisateur reprenne le même travail toutes les fois qu'il crée un *netlist*, même si les mêmes composants ont été utilisés dans des *netlists* antérieurs. Au plus, les logiciels les plus performants sauvegardent un schéma du circuit comme patron afin de le réutiliser ultérieurement. Cependant, lors de la réutilisation, l'utilisateur doit revoir les interconnexions et leurs contraintes. À cela s'ajoute aussi la qualité de la conception qui se détermine par le degré de conformité aux contraintes d'optimisation qui ont été satisfaites [16]. On peut faire l'analogie avec un programmeur qui utilise le langage

assembleur plutôt qu'un langage de plus haut niveau tel que C++ ou Java. La conception assistée par ordinateur est fondamentalement limitée par les approches théoriques actuelles des procédés de conception de *netlist*.

L'objectif de ce mémoire est de développer un logiciel qui aide l'utilisateur à créer plus efficacement un *netlist*, comparativement aux approches actuelles. Le travail que présente ce mémoire s'insère dans un projet de recherche appelé *DreamWafer*™.[1] Dans un cadre de recherche qui regroupe un ensemble de chercheurs universitaires (École Polytechnique de Montréal, Université du Québec à Montréal, Université du Québec en Outaouais) en collaboration avec le partenaire industriel Gestion TechnoCap inc. et l'appui des organismes subventionnaires (CRSNG, PROMPT Québec, MITACS), le projet DreamWafer™ a été initié pour développer une plateforme de prototypage rapide de systèmes électroniques appelée *WaferBoard*™. Le projet DreamWafer™, ayant un budget de plusieurs millions de dollars, a pour objectif d'accélérer le prototypage de systèmes électroniques pour permettre une itération en quelques minutes plutôt que des mois. Dans la plateforme WaferBoard™, l'utilisateur y place des composants électroniques qui sont analysés ; puis, il fournit un *netlist* à un outil qui crée le prototype du système électronique en configurant les connexions entre les composants électriques. Si le *netlist* n'existe pas, un logiciel assiste l'utilisateur pour le créer.

Advenant l'absence de *netlist* fourni par l'utilisateur, la création (du *netlist*) devient une étape significative dans le processus d'implémentation du prototypage avec le WaferBoard™. D'où l'importance de développer un logiciel qui assiste à la création de *netlist*s afin d'atteindre l'objectif par le projet DreamWafer™. Pour que cette assistance soit la plus optimale que possible, le logiciel vise des automatisations qui interviennent à toutes les étapes de création de *netlist*s. En effet, il est possible d'appliquer des techniques de l'informatique qui pourraient assister à la conception de *netlists*. À cet égard, l'utilisation de l'apprentissage automatique est un champ à explorer afin d'assister (ou même de remplacer si possible) les experts du domaine. L'apprentissage automatique, une branche de l'intelligence artificielle, consiste à développer des algorithmes informatiques capables de s'autoaméliorer dans

[1] Le site de DreamWafer™ : www.dreamwafer.com, septembre 2013.

l'exécution d'une tâche [6, 7]. Les algorithmes de l'apprentissage automatique estiment un modèle d'un problème du monde réel avec une certaine probabilité établie d'un ensemble de données empiriques [8].

Une problématique de la création d'un *netlist* de circuits imprimés provient majoritairement des contraintes comportementales. Jusqu'à ce jour, il n'existe aucun outil de conception assistée par ordinateur qui soit supporté par l'apprentissage automatique des contraintes comportementales d'un *netlist* de circuits imprimés. Ce mémoire fait une preuve de concept par la définition des infrastructures logicielles pour faire la connexion automatique des composants, assistée par l'utilisateur. L'usage de l'apprentissage automatique des contraintes comportementales accélère davantage la création d'un *netlist*, car l'utilisateur n'a pas à repartir de rien pour établir les formules.

L'objectif de ce mémoire consiste à proposer une méthodologie qui aide le concepteur à créer un *netlist* dans le but de réduire le temps de conception d'un circuit en général, particulièrement dans le cadre du projet DreamWafer™. Cette assistance, qui se discerne à trois niveaux, révèle les objectifs suivants[2] :

- **Objectif 1 – Simplification des modèles de composants :** certains composants (tels que les *Field Programmable Gate Array*s ou FPGA, les processeurs) peuvent avoir plusieurs centaines de broches. Leur usage devient complexe et surchargé au niveau de leur représentation conceptuelle. Il serait donc intéressant de donner une nouvelle représentation aux composants afin de simplifier leur usage et de faciliter leurs connexions.
- **Objectif 2 – Interconnexion automatique des composants** : pour créer le *netlist*, l'utilisateur doit connecter les composants. À l'aide de la nouvelle représentation élaborée dans la première étape et des valeurs connues des contraintes, des processus connectent automatiquement les composants compatibles après vérification ou non par l'utilisateur.

[2] On réfère des fois par « étape 1 » (respectivement 2 et 3) pour désigner l'objectif 1 (respectivement 2 et 3).

- **Objectif 3 – Suggestion de contraintes comportementales** : l'utilisateur doit définir les contraintes comportementales afin que le *netlist* devienne fonctionnel. Un processus supporté par l'apprentissage automatique assiste l'utilisateur à la définition formelle des contraintes comportementales. Un cycle, existant entre les étapes de l'objectif 2 et l'objectif 3, permet de raffiner les résultats avant la création des *nets*.

Pour atteindre l'objectif 1, chaque composant et ses broches sont représentés respectivement par une *configuration* et des *interfaces*. Une configuration contient des interfaces qui contiennent d'autres interfaces (tel un arbre où les feuilles sont les broches). Une interface peut être aussi un regroupement de broches ayant une *même connectivité fonctionnelle*[3]. Des propriétés formelles sont associées aux configurations et interfaces afin de préserver leurs conformités aux composants et au *netlist*. La configuration allège la vue des composants grâce à la structure hiérarchique des interfaces. L'utilisateur peut voir le composant par ses connectivités fonctionnelles hiérarchiques. Un composant peut avoir plusieurs configurations. Les configurations sont réutilisables, ce qui évite donc de les recréer pour chaque nouveau *netlist*.

Quant à l'objectif 2, étant donné que des valeurs prédéfinies de contraintes sont associées aux broches et aux composants (et donc aux interfaces et aux configurations), alors des processus élaborés permettent de détecter les interfaces compatibles et de les connecter automatiquement sous la supervision de l'utilisateur. Cette partie se divise en deux modules, dont l'un détecte la compatibilité des interfaces (appelé module de *suggestion des interfaces compatibles*) et l'autre interconnecte celles sélectionnées pour la création des connexions (appelé module d'*intercorrespondance automatique*). Le module de suggestion d'interfaces trouve les interfaces qui sont compatibles au niveau des contraintes, puis assume que celles-ci sont connectables et les présente à l'utilisateur. Quant au module d'intercorrespondance automatique, il crée des intercorrespondances énumérées entre les interfaces compatibles (à connecter), mais sans créer les *nets* entre les composants. Une intervention de l'utilisateur (à une étape dite *mise en veille*) lui permet d'approuver ou de modifier les paramètres de connexion. En fournissant les détails des broches à connecter, l'utilisateur peut définir plus

[3] Une *même connectivité fonctionnelle* signifie que le courant électrique transmit par ce groupe de broches a un sens opérationnel.

facilement les contraintes comportementales. Donc, pour le soutenir dans sa conception, la suggestion automatique des formules de contraintes comportementales intervient dans la troisième étape.

Les travaux effectués par l'objectif 1 ainsi que le concept de connexion automatique ont fait l'objet d'une publication à *International Symposium on Circuits and Systems* [15]. Une méthode de suggestion de connexions à l'utilisateur fut également élaborée ainsi qu'une infrastructure qui permettra éventuellement d'interconnecter automatiquement les composants.

Les formules des contraintes comportementales, telles que le *délai*, sont exprimées par des systèmes d'équations. Du point de vue informatique, seule leur structure mathématique nous intéresse ; leur sens revient aux experts du domaine. Pour atteindre l'objectif 3, on suppose qu'il existe des *netlists* dans une base de données où les contraintes comportementales sont exprimées formellement (et sont valables seulement dans leurs *netlists*). La solution proposée consiste à représenter les équations en tant que motifs séquentiels afin de faire usage d'un algorithme de forage de données. Plus précisément, l'algorithme *Apriori* [9, 10, 11, 12] (modifié selon les concepts proposés) fait une recherche de motifs séquentiels fréquents où des formules générales sont extraites et intégrées dans les interfaces. Une *formule générale* est une formule qui est apprise après à une répétition de celle-ci (en tant que motif séquentiel fréquent dans la base de données) et peut être réutilisée dépendamment des paramètres du *netlist* en cours de conception. La réutilisation d'une formule générale est possible, car les variables des équations sont représentées par les identifiants des interfaces. Lorsqu'une formule générale est utilisée dans un *netlist* lors de la troisième étape, elle devient spécifique à ce *netlist* à la suite de la détection des interfaces disponibles et des connexions des composants. Dans ce cas, on parle d'*instance*s de formule générale. La troisième étape instancie les formules générales puis les suggère à l'utilisateur qui les approuve ou les modifie. Cependant, à ce point de l'élaboration, la validation des formules reste une tâche manuelle. En guise d'exploration de la troisième étape, des tests ont été faits pour prouver la faisabilité de l'approche proposée. Dans ce mémoire, seule la contrainte du délai a été simulée pour la suggestion de contraintes comportementales.

En accord avec la définition de l'apprentissage automatique, le logiciel INC est capable d'apprendre des modèles (de formules) et de les instancier selon le *netlist* en cours de conception. D'habitude (en apprentissage automatique), un seul algorithme effectue la tâche de l'apprentissage tel que les méthodologies de raisonnement à base de cas [13] ou les arbres de décision [28]. Mais la solution proposée englobe la structure des modules, la modélisation des formules et l'extraction de motifs fréquents. Dans un cadre technique plus général, les modules sont conceptualisés relativement aux modèles (à extraire) afin que le logiciel devienne capable d'apprendre certaines tâches. Par contraste au raisonnement à base de cas où les exemples (pris de la base de connaissances) ne sont pas modifiés, la solution proposée fait usage de modèles extraits par forage de données. Dans le raisonnement à base de modèle [14], la tâche principale se concentre à analyser les données afin de trouver un modèle sans nécessairement identifier la technique ou indiquer comment le modèle est utilisé. La technique proposée dans ce mémoire pousse plus loin l'usage des modèles en les modifiant selon des paramètres dans le but d'identifier le cas auquel est confronté l'utilisateur et de donner sa solution.

Les contributions qu'apporte ce mémoire sont :

- Les nouveaux concepts de Configuration et d'Interface permettent une vue hiérarchique des composants et allègent la vue du *netlist*. Ces concepts suivent des propriétés formelles qui doivent être respectées.
- Une nouvelle méthodologie de création de *netlist*s qui se base sur les interfaces. La détection des interfaces compatibles et leurs connexions automatiques accélèrent la création du *netlist*.
- Une nouvelle approche hybride en apprentissage automatique qui combine le génie logiciel et le forage de données. Le forage de données extrait des formules générales représentées par une série d'identifiants d'interfaces. L'usage de ces modèles est possible grâce à la structure des modules.

La structure du mémoire est organisée comme suit. Le chapitre I introduit les notions de base en électronique pour concevoir un *netlist*. Le chapitre II présente un aperçu de l'état de l'art de conception des *netlists* et la problématique du point de vue informatique. Ensuite, le

chapitre III propose des concepts et leurs propriétés à la problématique de l'objectif 1. Le chapitre IV présente l'algorithme de suggestion d'interfaces compatibles et l'algorithme de connexion automatique fixés par l'objectif 2. Pour l'objectif 3, le chapitre V explique la méthode d'extraction des formules des contraintes comportementales et leurs suggestions durant la conception du *netlist*. Les expérimentations et les résultats sont présentés dans le chapitre VI. Le chapitre VII conclut ce mémoire. Et enfin en annexe, l'appendice A présente le manuel d'utilisation du logiciel INC.

CHAPITRE I

NOTIONS DE BASE EN ÉLECTRONIQUE POUR LA CONCEPTION D'UN CIRCUIT ÉLECTRONIQUE

En électronique, la conception des circuits imprimés prend en compte beaucoup de concepts, de paramètres, d'équations et de modèles qui permettent de simuler le comportement des circuits imprimés, à partir des principes issus de la mécanique, de l'électromagnétisme, de la chimie, de la thermodynamique et la science des matériaux. Ils doivent également prévoir la configuration de certains composants avant de les connecter. Ce chapitre présente une brève description des notions de base en électronique pour la conception d'un circuit imprimé. La première section définit techniquement la notion d'un *netlist*. Puis, la deuxième section explique la problématique des contraintes comportementales. Des exemples de formules de contraintes comportementales sont donnés dans la troisième section. La quatrième section explique comment les circuits imprimés sont actuellement développés. Et enfin, la cinquième section montre les limites de cette conception et le besoin de l'améliorer.

1.1 Notion de *netlist*

La figure 1.1 montre un exemple simplifié d'un *netlist*. Par exemple, le *net* 2 indique que la broche 2 du composant *A* est connectée à la broche 3 du composant *B*. Ainsi, les *nets* n1, n2, n3 et n4 représentent les connexions entre les broches 1, 2, 3 et 6 du composant *A* respectivement aux broches 2, 3, 4 et 5 du composant *B*. De même, les broches 4 et 5 du composant A sont connectées respectivement aux broches 3 et 4 du composant C. Chaque *net* contient la liste des broches connectées. Le *netlist* du circuit devient la liste des *nets* n1, n2, n3, n4, n5 et n6.

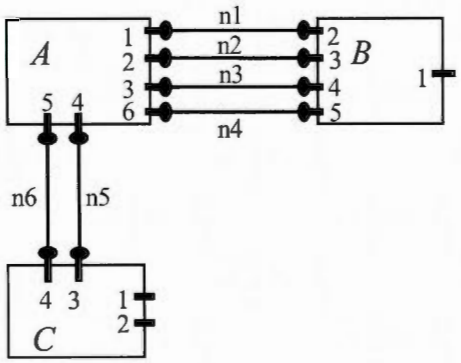


Figure 1.1 Exemple d'un *netlist* schématique.

Un *netlist* contient aussi des descriptions (électriques, mécaniques, mathématiques ou géométriques) des composants, broches et *nets*. Ces descriptions sont des contraintes de conception qui spécifient le circuit. La section suivante présente la notion des contraintes en général et plus particulièrement les contraintes comportementales.

1.2 Notion de contrainte de conception de *netlist*

Des contraintes sont associées aux broches, aux nets et aux composants électroniques. Ces contraintes qui doivent être respectées afin que le *netlist* soit fonctionnellement valide. Par exemple, une broche (d'un composant) qui fonctionne à une tension de 5V ne peut pas être connectée à une source d'alimentation de 10V, car l'opérationnalité du composant ne permet pas de supporter une telle tension. Un autre exemple de contrainte associée à un composant est la température qui affecte le fonctionnement d'un composant, car le matériel qui le construit ne conduit pas le même courant électrique sous différents degrés (de température) et sa fiabilité en dépend. De ceci, une classification de contraintes est faite.

Les contraintes se classent en général dans une des quatre catégories suivantes [16] :

- les contraintes technologiques nécessaires à la fabrication ;
- les contraintes fonctionnelles qui garantissent les fonctionnalités des circuits intégrés ;

- les contraintes de simplification (du désign du circuit) qui émergent à la suite d'une réduction de la complexité du désign du circuit[4] ;
- et les contraintes commerciales qui, par exemple, doivent répondre à certaines obligations d'emballage définies par un dessin physique de circuit.

De plus, les contraintes sont implicites (texte, supposition et algorithme) ou explicites (des valeurs directement accessibles telles que le voltage et la fréquence). Les contraintes explicites sont connues au début du processus de désign alors que les contraintes implicites sont plus difficiles à manipuler à cause de leurs expressions informelles qui se définissent et se valident tout au long de la conception. Chaque contrainte a un type de valeur ; type qui se rapporte à sa propriété et qui se classe dans un domaine (ou groupe de domaines) physique, électrique, mécanique, mathématique ou géométrique. Ces types de contraintes peuvent être simples (c.-à-d. qu'elles sont indépendantes, à une variable) ou complexes (qui dépendent d'autres contraintes, à plusieurs variables). Les contraintes peuvent avoir des valeurs (ou des intervalles de valeurs) nominales, réelles ou statistiques [16]. Les experts du domaine doivent manuellement définir et vérifier les contraintes comportementales des composants selon le contexte du circuit. Les contraintes comportementales sont implicites et peuvent être simples ou complexes dans n'importe quel domaine.

Les broches des composants connectées au même *net* doivent être compatibles électriquement. Le *net* leur permet de communiquer en transmettant des informations grâce à des signaux électriques qui décrivent des comportements ou attributs d'un phénomène [4, 17]. Un signal se définit aussi en tant que quantité physique, variable dans le temps, telle que la température, la pression, le son, la lumière et l'électricité [18]. Un composant électronique prend un signal entrant et produit un signal sortant. Ainsi, les composants connectés communiquent de façon coordonnée des signaux électriques dans un circuit. Mais comme tous les composants ne sont pas fonctionnellement identiques, alors le concepteur contrôle le jeu des contraintes. Ainsi, il influence la transmission des signaux électriques afin que cette transmission ait un sens.

---

[4] Les contraintes de simplification sont applicables seulement pour la synthèse de circuits intégrés et non des circuits imprimés.

Dans la figure 1.2, le *netlist* est représenté avec la contrainte *direction* (ou *type*). Les broches possèdent une direction qui indique les transmissions du signal entrant ou sortant. La valeur *POW*, par exemple, indique que les broches connectées au *net* n4 reçoivent une alimentation de 5V. La broche 1 du composant *A* de type entrant (*IN*) est connectée à la broche 2 de B de type sortant (*OUT*). Le type *OUT* ne peut pas se connecter à un type *OUT* par exemple. Du fait que la quantité des détails surcharge le schéma, alors un même *netlist* peut être représenté dans plusieurs fichiers simultanément (schématiques, textuels et physiques). D'autres contraintes électriques (telles que la fréquence ou l'intensité) pourraient être décrites plus explicitement dans un fichier texte du *netlist*.

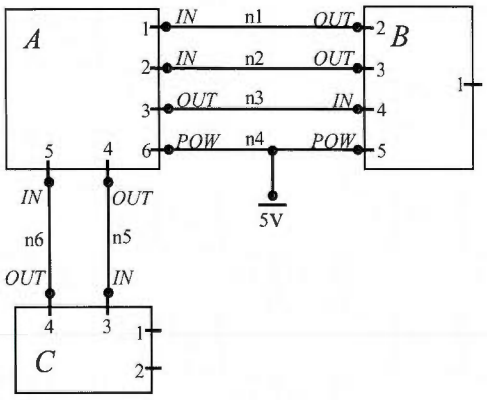


Figure 1.2 Exemple de *netlist* avec des contraintes d'entrées et sorties.

Les informations transmises par les *nets* peuvent devoir respecter des contraintes (comportementales) reliées au temps. Elles peuvent devoir, par exemple, arriver simultanément (en parallèle ou en série) dans un intervalle défini de temps. Par exemple, lorsqu'un processeur transmet des informations en parallèle à des mémoires par un ensemble de *nets* (appelé *bus*), alors le composant mémoire doit respecter une contrainte temporelle (telle que le délai), qui aide à échantillonner l'ensemble de ces signaux dans un intervalle de temps défini par le comportement des composants interconnectés. Ce comportement est généralement extrait des spécifications décrites par le manufacturier de ce composant. Le respect de ces contraintes permet d'assurer la validité des opérations d'écriture et de lecture (dans les mémoires). Si un signal ne respecte pas une contrainte temporelle, les informations sauvegardées dans la mémoire peuvent s'avérer erronées. Les contraintes temporelles

peuvent être dépendantes de certains paramètres tels que la fréquence des signaux électriques, la distance entre les composants, les matériaux des traces conductrices, la distance entre les traces conductrices, le voltage, la température, etc.

1.3 Exemple de contrainte comportementale – le délai

Le signal que les composants connectés transmettent change d'état en passant d'une valeur à une autre. La transition d'un signal spécifique d'un état vers un autre est appelée *évènement* [19, 20]. Du fait que tous les composants ne fonctionnent pas à la même allure de traitement d'un signal électrique, alors un intervalle temporel indique le seuil de flexibilité de transmission dans lequel le circuit est garanti de rester fonctionnel. Soit, la relation entre deux évènements e et e' consiste à spécifier un temps minimum δ et un temps maximum Δ qui indiquent les bornes de l'intervalle temporel (appelé *temps de séparation* ou *délai* noté $[\delta, \Delta]$) [19, 20]. Dans la figure 1.3, un composant transmet un signal pour indiquer l'évènement e (passage de 0V à 2V). Puis, dans l'intervalle de temps $[\delta, \Delta]$, le signal suivant doit être transmis pour indiquer l'évènement e' (passage de 2V à 0V).

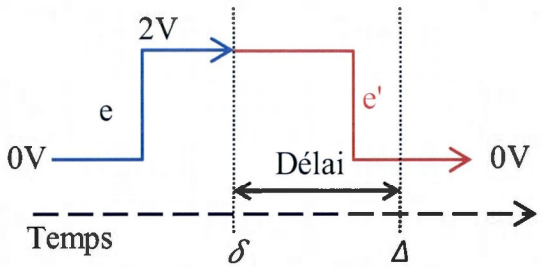


Figure 1.3 Transmission de signal électrique.

La transmission de l'évènement e' débute entre δ et Δ. Soit, t(e) et t(e') les temps initiaux de e et e' respectivement, alors on peut affirmer que la formule suivante [19, 20] :

$$t(e) + \delta \leq t(e') \leq t(e) + \Delta \tag{1.1}$$

Une écriture alternative de l'expression (1.1) est :

$$\delta \leq t(e') - t(e) \leq \Delta \tag{1.1}'$$

Le problème se complique lorsqu'un composant reçoit plusieurs signaux simultanément, car les informations transmises doivent être complètes afin d'être traitées. Par exemple, dans la figure 1.4, les évènements e_1 (avec un délai $T_1 = [\delta_1, \Delta_1]$) et e_2 (avec un délai $T_2 = [\delta_2, \Delta_2]$), transmis par différents composants, amorcent l'évènement e_3. L'évènement e_3 ne peut être activé qu'entre les bornes inférieures et supérieures des intervalles T et T'. Autrement dit, l'évènement e_3 est dépendant de e_1 et e_2 simultanément :

$$max(t(e_1) + \delta_1, t(e_2) + \delta_2) \leq t(e_3) \leq min(t(e_1) + \Delta_1, t(e_2) + \Delta_2) \quad (1.2)$$

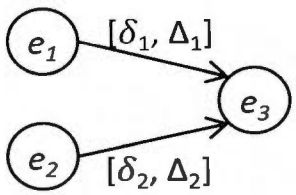


Figure 1.4 Exemple d'un graphe d'évènements.

L'inégalité de l'expression (1.2) peut être plus étendue si l'évènement e_3 dépend d'évènements additionnels ou que d'autres évènements dépendent de e_3. Par exemple, si un évènement e_5 dépend des évènements e_3 et e_4, alors e_5 devient implicitement en relation avec les délais de e_1 et e_2. La tâche du concepteur consiste à trouver les bornes (optimales si possible) des délais consistants avec les prérequis des spécifications ciblées par le circuit. Il doit vérifier qu'il n'y a pas de blocage de délai d'un évènement (par exemple, le déclenchement de e_3 devient impossible si le minimum $t(e_1) + \delta_1$ est supérieur au maximum $t(e_2) + \Delta_2$). Il est à noter aussi que les valeurs δ et Δ peuvent être dépendantes d'autres contraintes telles que la fréquence, la température, etc. Au besoin, le concepteur est contraint à modifier les configurations des composants ou du circuit pour résoudre les impasses ou faire des optimisations.

À titre d'exploration, ce mémoire prend en exemple la contrainte comportementale de délai à déterminer parmi d'autres telles que les aléas de délai (*skew*), les marges de manœuvre de délai (*slack*). Le choix de la contrainte de délai vient du fait qu'elle est importante à définir, car d'autres contraintes comportementales en dépendent.

Pour concevoir un circuit électronique, les concepteurs procèdent par une méthodologie de développement qui définit les spécifications (du circuit) et vérifie les contraintes implémentées. La section suivante présente le cycle de développement d'un circuit électronique.

1.4 Cycle de développement de circuit électronique

La figure 1.5 illustre le flux typique de conception de circuits imprimés où des idées de développement technologique de départ conduisent à la création d'un système électronique [2, 16, 21, 22, 25 et une figure additionnelle[5]]. Les concepteurs examinent dans une première étape les besoins d'un système électronique à fabriquer pour en tirer les prérequis nécessaires (étape de *remue-méninges*). Puis, ces prérequis sont évalués (étape d'*évaluation*) afin de définir des concepts qui assurent de réaliser la fonctionnalité du système électronique à développer selon le budget disponible. À l'étape de la *conception*, les concepts sont traduits en schémas fonctionnels (électriques et électroniques) qui décrivent l'architecture du système. L'étape de *capture* (conception logique) définit les détails (standards) des composants électroniques à utiliser, les schémas logiques et les *netlists* à l'aide des stocks et des librairies disponibles. Les schémas logiques et les *netlists* sont soumis à des simulations et des vérifications de conformité des règles d'interconnexion. Ensuite à l'étape de la conception physique, des plans (physiques) et des fichiers de circuits imprimés sont définis en considérant les optimisations et les simplifications possibles. Une fois de plus, des vérifications et des simulations ont lieu afin d'éliminer les erreurs de conception. Les plans physiques permettent de créer des prototypes de circuits imprimés (étape de *fabrication*) qui sont ensuite testés (étape de *test*) afin de corriger les erreurs s'il y en existe. Lorsque les prototypes finaux sont acquis alors leur production en masse peut être entamée. Ensuite, les circuits imprimés produits sont assemblés en systèmes électroniques. Il faut noter que le flot de conception est un cycle de développement où des ajustements ont lieu à toutes les étapes.

---

[5] La figure additionnelle est tirée du site www.cs.berkeley.edu/~prabal/teaching/cs194-05-s08/cs194-designflow.ppt, 2008, septembre 2013.

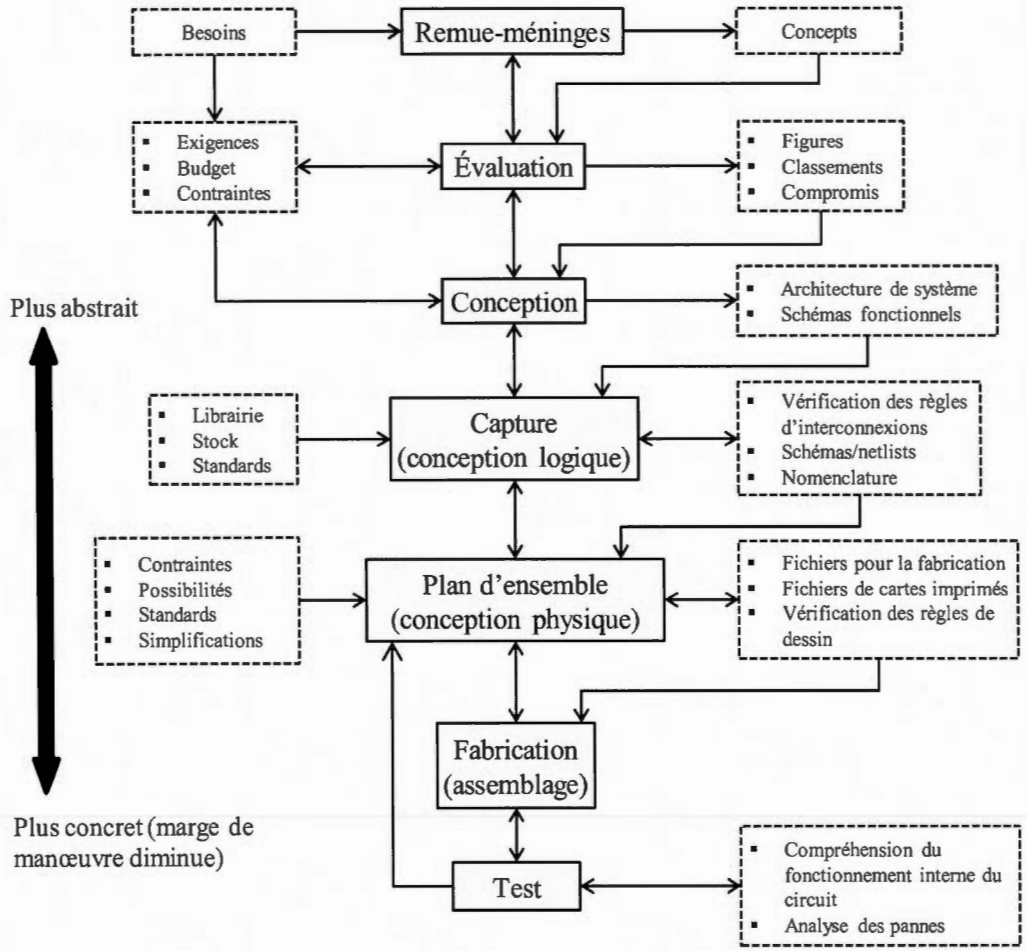


Figure 1.5 Flux de conception des circuits imprimés.

Plus le flux de conception avance vers la fabrication physique du circuit, alors le degré de liberté de conception diminue et donc la solution devient plus restreinte. Les concepteurs réduisent graduellement les marges de manœuvre (c.-à-d. le degré de liberté) afin de restreindre l'espace de la solution dans le contexte du circuit et de ses contraintes selon des stratégies arbitraires [16]. Dans toute conception, les contraintes doivent être respectées. Cependant, le but est de les réaliser avec une optimisation prédéfinie de départ.

Deux importants domaines dans la conception de circuit sont la synthèse et la vérification [19]. La synthèse est le processus de transformation des spécifications abstraites en des implémentations physiques. La vérification est un processus qui assure que les

conceptions formelles ont été proprement implémentées et répondent aux critères requis par les fonctions du circuit. Certaines contraintes temporelles (telles que le délai) sont définies selon un cycle issu de trois principes [19] :

- La vérification par implémentation : qui assure que l'implémentation d'un *netlist* satisfait les contraintes comportementales temporelles de la spécification originale.
- Dérivation des contraintes pour la synthèse : en utilisant les variables des contraintes symboliques δ et Δ, il est possible de déterminer le degré de flexibilité qui est disponible tout en satisfaisant les contraintes comportementales.
- Évaluation de conception : est appliquée en utilisant des variables de contraintes symboliques et en déterminant les bornes des valeurs de ces variables dépendamment des contraintes comportementales.

Quant à la vérification temporelle, elle se spécifie sur trois types de relations [19] :

- Le délai : qui décrit une relation temporelle entre deux évènements.
- Les garanties désignent les relations qui sont assurées d'être maintenues entre les évènements (telles que les configurations des composants et les valeurs de certaines contraintes électriques et physiques déjà établies).
- Les contraintes qui sont des conditions fonctionnelles du circuit.

En général lors de la conception d'un circuit, le concepteur sélectionne et configure les composants (c.-à-d. il initialise les garanties) selon les fonctionnalités du circuit (c.-à-d. les contraintes cibles). Et donc, il ne reste que le calcul des délais. Tout au long de la conception, le concepteur calcule les délais et les vérifie constamment par rapport aux contraintes et aux garanties.

La conception des circuits se fait à l'aide d'outils de conception (graphique et fonctionnelle) assistée par ordinateur qui aident à la génération des *netlists* (des logiciels tels que *DxDesigner* et *PADS* de *Mentor Graphic*[6], *Allegro* et *OrCad* de *Cadence*[7], *ViewLogic* de

[6] Site de Mentor Graphics : www.mentor.com, septembre 2013.
[7] Site de Cadence : www.cadence.com, septembre 2013.

Synopsis[8], *Altium Designer* d'*Altium*[9]). La section suivante présente les limites actuelles des logiciels de conception et le besoin les améliorer.

1.5 Limites actuelles des logiciels de conception de *netlists*

Depuis une trentaine d'années après son introduction, la vérification des modèles pour la conception des circuits a connu un grand progrès [23]. Mais ce progrès a touché en grande partie la vérification algorithmique des contraintes électriques et temporelles (ou *software model checking*). Quant à la vérification de la description matérielle des circuits (ou *hardware model checking*), malgré son progrès, elle reste un problème stagnant et pressant vu les problématiques auxquelles elle est confrontée telles que : la complexité des contraintes comportementales, le développement accru des nouveaux circuits intégrés et leurs interfaces, la qualité des matériels utilisés, l'optimisation des circuits et le temps de mise en marché des systèmes électroniques [16, 23]. Mais, les logiciels de conception de *netlist* n'arrivent pas à suivre le progrès. Le développement des circuits imprimés devient donc dépendant en grande partie sur l'expérience des concepteurs. Vu que la limite des logiciels provient fondamentalement de la stratégie de développement [16], alors il devient nécessaire d'en trouver une nouvelle qui répond aux problématiques de conception.

Le chapitre suivant présente l'état de l'art de la conception de *netlist* du point de vue informatique.

---

[8] Site de Synopsis : www.synopsys.com, septembre 2013.
[9] Site de Altium : www.altium.com, septembre 2013.

CHAPITRE II

ÉTAT DE L'ART DE LA CONCEPTION DE *NETLIST*

Le développement technologique de l'électronique, qui augmente exponentiellement, rend de plus en plus difficile la productivité. Les logiciels de conception assistée par ordinateur de *netlist* n'arrivent pas à suivre à la même allure car ils suivent un flux de développement qui dépend plus sur l'expertise et la connaissance du concepteur. Dans la première section, ce chapitre explique les problématiques du point de vue informatique. Puis, la deuxième section résume les approches théoriques existantes en rapport avec ces problématiques.

2.1 Problématiques de conception de *netlist* du point de vue informatique

Durant la conception d'un circuit, les concepteurs vont dans les deux sens de développement : ascendant (ou *bottom-up*) et descendant (ou *Top-down*) [16]. Le développement ascendant commence par la configuration des structures internes des circuits intégrés (telle que les portes logiques) jusqu'à arriver au dèsign du système électronique. Tandis que le développement descendant commence au niveau système, puis en déduit les composants intégrés à utiliser et leurs configurations. En pratique durant la conception, les concepteurs essayent de joindre les deux bouts ascendant et descendant.

Les contraintes comportementales commencent à se définir dès la configuration interne des circuits intégrés pour ensuite se propager à un plus haut niveau (c.-à-d. conception ascendante). Ce qui rend « presque impossible » de les définir par une conception descendante [16].

En premier lieu, il faut mentionner que ce mémoire se limite à la conception de système électronique jusqu'au niveau des composants et leurs contraintes électriques et fait

abstraction de la structure interne des composants. En plus de cette abstraction, d'autres difficultés de conception au niveau système ne sont pas adressées telles que le routage des *nets* (où la tâche consiste à trouver les chemins sur lesquels seront imprimés les *nets* en respectant les contraintes de conception). Ce mémoire s'adresse à deux problématiques majeures : la connexion des broches des composants et la suggestion des contraintes comportementales où le délai est pris à titre d'exemple.

2.1.1 Connexion des composants

Afin qu'un *netlist* soit généré, tous les éléments qui participent à sa structure doivent également être modelés. Donc, il s'agit des objets tels que les composants, les broches, les contraintes, les *nets* et le *netlist*. Le concepteur ne peut connecter deux broches que si elles sont compatibles électriquement. Chaque broche possède la liste de contraintes. Certaines contraintes peuvent aussi être associées aux composants et aux *nets* car tous deux possèdent des caractéristiques individuelles. La compatibilité consiste à vérifier que toutes leurs contraintes correspondantes sont compatibles électriquement. Par exemple, une broche de direction OUT ayant une tension de [2, 5]V ne peut se connecter qu'à une broche de direction IN ou INOUT de tension [2, 5]V (en supposant que les autres contraintes électriques des broches sont compatibles). On assimile à chaque type de contrainte une fonction de compatibilité qui lui propre. Chaque fonction vérifie la compatibilité subjective à sa contrainte entre plusieurs broches et possiblement une dépendance s'il en existe (par exemple, la tension peut dépendre la résistance et de l'intensité du courant). Les détails de ces fonctions sont fournis par les experts de l'électronique. Donc, la compatibilité entre deux broches consiste à vérifier toutes les fonctions simultanément.

Dans ce mémoire, seule la fonction de la contrainte de direction a été développée en profondeur pour l'algorithme de connexion. Quant aux autres fonctions (plusieurs dizaines), elles sont aussi vérifiées en parallèle (avec la direction). Mais, on se limite à vérifier les bornes des intervalles par l'égalité. Par exemple, on détermine qu'une broche de direction OUT et de tension [3, 4]V n'est pas connectable à une broche IN de tension [3, 5]V car les bornes maximales des intervalles ne sont pas égales (bien que ce soit possible dans certains cas en pratique car il y a une tolérance lorsque le voltage maximale d'une broche OUT est inférieur au voltage maximale d'une broche IN – mais pas supérieure). On se limite à vérifier

par l'égalité les minimums et les maximums des intervalles de chaque contrainte afin d'asserter la compatibilité des broches testées. Un développement plus concret nécessite la prise en compte des détails de chaque fonction lorsque c'est nécessaire.

D'un autre côté, même si le logiciel à développer suggère les broches compatibles, le concepteur aura toujours de la difficulté à identifier celles dont il a besoin car il peut exister plusieurs centaines voire milliers de broches dans un *netlist*. Autrement dit, il y a de fortes chances qu'une broche puisse avoir plusieurs dizaines de broches qui lui sont compatibles électriquement, cependant, une seule serait nécessaire à la conception. Bien que la tâche de la recherche manuelle devienne réduite aux broches compatibles, l'utilisateur reste confronter à la tâche de connexion broche-par-broche pour tout le circuit. Donc, il devient important de trouver une stratégie de représentation des composants qui permettrait d'alléger la vue du *netlist* et de connecter un lot de broches plutôt qu'une à la fois. Ce mémoire répond à cette problématique en introduisant de nouveaux concepts qui permettent non seulement la suggestion des broches compatibles, mais aussi la connexion automatique par groupes compatibles de broches.

La deuxième problématique à laquelle s'adresse ce mémoire est la suggestion des contraintes comportementales qui est couverte dans la section suivante.

2.1.2 Suggestion de contraintes comportementales

La contrainte comportementale, prise en exemple dans ce mémoire, est le délai qui s'exprime par des formules mathématiques. Du point de vue informatique, le sens des formules n'est pas pris en considération. En d'autres termes, il ne s'agit pas de faire des vérifications mais plutôt de suggérer des formules à l'utilisateur selon la conception du circuit. Mais comme les formules sont souvent subjectives au désign du circuit, il devient difficile de trouver avec exactitude celles dont l'utilisateur a besoin. Jusque à ce jour, il n'existe aucun logiciel qui suggère la contrainte de délai même au niveau de la conception des circuits intégrés [16, 24]. Pour résoudre une problématique qui se base sur l'expérience du concepteur, des techniques de l'informatique telles que l'apprentissage automatique peuvent être utilisées.

Afin de mieux analyser la problématique de la contrainte du délai dans ce mémoire, on réduit l'expression des formules à des équations incluant les connecteurs des base =, < et > ; les opérations + et - ; et les opérateurs * et /. À partir de l'expression (1.2), une formule est une expression dépendante de δ et Δ qui sont aussi fonctions d'autres contraintes. On supposera que δ et Δ sont des fonctions polynomiales à plusieurs variables[10] (c.-à-d. $f(d) = a_1 x_1^{p1} y_1^{q1} \dots z_1^{r1} + a_1 x_2^{p2} y_2^{q2} \dots z_2^{r2} + \cdots + a_n x_n^{pn} y_n^{qn} \dots z_n^{rn}$ avec $f(d) = \delta$ ou Δ; $x, y, \dots, z \in \mathbb{R}$ sont des valeurs réelles des contraintes de dépendance; $a_1, \dots, a_n \in \mathbb{R}$ sont des constantes réelles; p, q, r et $n \in \mathbb{N}$). Alors on peut écrire que :

$$a_1 x_1^{p1} y_1^{q1} \dots z_1^{r1} + a_1 x_2^{p2} y_2^{q2} \dots z_2^{r2} + \cdots + a_g x_g^{pg} y_g^{qg} \dots z_g^{rg} \leq t(e') - t(e) \leq b_1 r_1^{\alpha 1} s_1^{\beta 1} \dots t_1^{\gamma 1} + b_2 r_2^{\alpha 2} s_2^{\beta 2} \dots t_2^{\gamma 2} + \cdots + b_h r_h^{\alpha h} s_h^{\beta h} \dots t_h^{\gamma h} \quad (2.1)$$

avec $\delta = a_1 x_1^{p1} y_1^{q1} \dots z_1^{r1} + a_1 x_2^{p2} y_2^{q2} \dots z_2^{r2} + \cdots + a_g x_g^{pg} y_g^{qg} \dots z_g^{rg}$ et $\Delta = b_1 r_1^{\alpha 1} s_1^{\beta 1} \dots t_1^{\gamma 1} + b_2 r_2^{\alpha 2} s_2^{\beta 2} \dots t_2^{\gamma 2} + \cdots + b_h r_h^{\alpha h} s_h^{\beta h} \dots t_h^{\gamma h}$.

De plus, on peut reformuler l'expression (1.2) comme une suite d'expressions telle que :

$$(t(e_1) + \delta_1 \leq t(e_3) \leq t(e_1) + \Delta_1) \wedge (t(e_2) + \delta_2 \leq t(e_3) \leq t(e_2) + \Delta_2). \quad (2.2)$$

Ou encore :

$$(e_{1+}\delta_1 \leq e_3) \wedge (e_3 \leq e_{1+}\Delta_1) \wedge (e_2 + \delta_2 \leq e_3) \wedge (e_3 \leq e_2 + \Delta_2) \quad (2.2)'$$

Les expressions (1.2) et (2.2)' sont équivalentes car e_3 doit vérifier tous les termes simultanément par un ET logique. Autrement dit, on déduit un système d'équations E tel que :

$$E = \begin{cases} (e_3 - e_1) \geq \delta_1 \\ (e_3 - e_2) \geq \delta_2 \\ (e_3 - e_1) \leq \Delta_1 \\ (e_3 - e_2) \leq \Delta_2 \end{cases} \quad (2.3)$$

[10] À ce stade de développement, on ne considère pas des fonctions complexes telles que trigonométriques, logarithmiques, etc.

En utilisant les expressions (2.1) et (2.3), on déduit un nouveau système E' équivalent à E :

$$E' = \begin{cases} (e_3 - e_1) \geq a_{1,1} x_{1,1}^{p1,1} y_{1,1}^{q1,1} \dots z_{1,1}^{r1,1} + a_{1,1} x_{1,2}^{p1,2} y_{1,2}^{q1,2} \dots z_{1,2}^{r1,2} + \dots + a_{1,g} x_{1,g}^{p1,g} y_{1,g}^{q1,g} \dots z_{1,g}^{r1,g} \\ (e_3 - e_1) \leq b_{1,1} r_{1,1}^{\alpha 1,1} s_{1,1}^{\beta 1,1} \dots t_{1,1}^{\gamma 1,1} + b_{1,2} r_{1,2}^{\alpha 1,2} s_{1,2}^{\beta 1,2} \dots t_{1,2}^{\gamma 1,2} + \dots + b_{1,h} r_{1,h}^{\alpha 1,h} s_{1,h}^{\beta 1,h} \dots t_{1,h}^{\gamma 1,h} \\ (e_3 - e_2) \geq a_{2,1} x_{2,1}^{p2,1} y_{2,1}^{q2,1} \dots z_{2,1}^{r2,1} + a_{2,1} x_{2,2}^{p2,2} y_{2,2}^{q2,2} \dots z_{2,2}^{r2,2} + \dots + a_{2,g} x_{2,g}^{p2,g} y_{2,g}^{q2,g} \dots z_{2,g}^{r2,g} \\ (e_3 - e_2) \leq b_{2,1} r_{2,1}^{\alpha 2,1} s_{2,1}^{\beta 2,1} \dots t_{2,1}^{\gamma 2,1} + b_{2,2} r_{2,2}^{\alpha 2,2} s_{2,2}^{\beta 2,2} \dots t_{2,2}^{\gamma 2,2} + \dots + b_{2,h} r_{2,h}^{\alpha 2,h} s_{2,h}^{\beta 2,h} \dots t_{2,h}^{\gamma 2,h} \end{cases} \quad (2.4)$$

Il est à noter que le nombre des équations et le nombre de leurs variables peuvent être théoriquement indéterminés dans le système E'. Donc, la tâche du concepteur revient à trouver les valeurs optimales pour lesquelles E' reste vrai. Il est possible qu'une même contrainte x de E' puisse exister dans un autre système E". La variable x doit être vrai dans les deux.

On généralise le système E' par un système S d'équations selon les connecteurs, les opérations et les opérateurs pris en compte dans ce mémoire sur un nombre inconnu de variables, tel que :

$$S = \begin{cases} d_1(=,<,>)(+,-)a_{1,1}x_{1,1}(+,-,*,/)a_{1,2}x_{1,2}(+,-,*,/) \dots (+,-,*,/)a_{1,n_1}x_{1,n_1} \\ d_2(=,<,>)(+,-)a_{2,1}x_{2,1}(+,-,*,/)a_{2,2}x_{2,2}(+,-,*,/) \dots (+,-,*,/)a_{2,n_2}x_{2,n_2} \\ \dots \\ d_m(=,<,>)(+,-)a_{m,1}x_{m,1}(+,-,*,/)a_{m,2}x_{m,2}(+,-,*,/) \dots (+,-,*,/)a_{m,n_2}x_{m,n_m} \end{cases} \quad (2.5)$$

Les connecteurs, les opérations et les opérateurs entre parenthèses signifient qu'un seul de ces éléments est sélectionnable pour la formulation de l'équation. Les valeurs de n et m sont théoriquement inconnues. Donc, pour définir une contrainte de délai, il revient à suggérer à l'utilisateur un système d'équations tel que S selon la conception du circuit où les variables sont les connecteurs, les opérations, les opérateurs, les constantes a_{ij} et les valeurs x_{ij} des contraintes de dépendance. On se limite à suggérer seulement les équations $f(d)$ et non à formuler le système[11]. La valeur de $f(d)$ dans S ne représente pas forcément $(e_p - e_q)$ mais peut être n'importe quelle contrainte qui s'exprime par une formule. Autrement dit, on peut

[11] Bien qu'il aurait été possible de suggérer le système d'équations S, ce mémoire s'est limité à suggérer seulement les équations individuelles à l'utilisateur. Pour un développement plus concret, il faut nécessairement prendre en compte la suggestion des systèmes (incluant leur vérification). Il est à noter que ce mémoire a pris en considération seulement les opérations de base. Il n'est donc pas impossible d'en déduire les fonctions complexes ou, par exemple, d'utiliser un tier module (tel que *mathlab* : www.mathworks.com/products/matlab/, sept. 2013) pour résoudre le système d'équations.

avoir la valeur 0 à gauche du connecteur et toutes les variables à droite (étant donné que c'est un système). Donc, l'exercice consiste à résoudre le système en trouvant les variables correctes. Ce mémoire propose une approche en apprentissage automatique qui tente de répondre à cette problématique.

2.2 Approches théoriques existantes de conception de système électroniques

Tout comme l'indiquent [16, 23], il n'y a pas beaucoup de travaux qui ont été faits dans la conception de système électronique ces dernières années, que ce soit au niveau de la vérification ou aux problématiques auxquelles s'adresse ce mémoire. La première section qui suit couvre les approches existantes qui tentent de faire usage de la connexion automatique. Puis, la deuxième section résume les approches de conception faisant usage de l'apprentissage automatique des contraintes comportementales en générale.

2.2.1 Approches existantes pour la connexion automatique

Pour la connexion automatique des broches au niveau système électronique, il n'existe pas de référence (au meilleur de notre connaissance) à part la publication [15] qu'a faite ce mémoire. Dans les logiciels de conception assistée par ordinateur tel qu'*Altium*[12], l'utilisateur sélectionne manuellement un nombre n de broches pour les connecter à un même nombre n de broches (ou soit à un bus). Une broche se connecte à celle qui se trouve en face d'elle par un « glisser-déposer » (*drag-and-drop*). La technique est plus graphique et n'a aucun fond qui se base sur les contraintes. De plus, il n'y a pas de suggestion de broches compatibles électriquement. De même pour les circuits intégrés, il existe des travaux tels que [26] où la technique n'est pas plus différente que celle d'*Altium* et fait une correspondance entre les broches qui se trouvent face à face.

2.2.2 Approches existantes pour la suggestion de contraintes comportementales

Les approches théoriques existantes qui font usage de l'apprentissage automatique utilisent les machines à vecteurs de support (ou *support vectors machines*) pour déterminer, par exemple, l'emplacement physique des transistors dans un circuit intégré [24]. Au meilleur de

[12] Une démonstration de connexion est disponible sur le site d'Altium http://videos.altium.com/trainingcenter/od_player.html, septembre 2013.

notre connaissance, il n'y a pas de référence sur la suggestion de la contrainte de délai pour les circuits de systèmes électroniques. Ce mémoire fait usage de nouveaux concepts qui rendent possible la suggestion de formules et donc présente une solution originale.

L'astuce qu'utilise ce mémoire pour la suggestion des formules consiste à décomposer la contrainte comportementale du délai en des expressions régulières. Dans ce cas, il existe des travaux qui portent sur l'extraction des expressions régulières tels que l'algorithme *Apriori*, *FP-growth* [29], algorithme d'apprentissage d'expressions régulières de données positives [30]. Cependant, l'extraction des expressions régulières n'est pas un apprentissage automatique. Ce mémoire utilise les expressions régulières comme modèles afin de les reformuler plus tard lors de la conception du *netlist*. Autrement dit, nous présentons une nouvelle approche hybride en apprentissage automatique basée sur un raisonnement à base de modèle qui fait usage des expressions régulières.

Le chapitre suivant propose une nouvelle représentation de modèle des composants électroniques grâce à laquelle la connexion automatique et la suggestion des formules deviennent réalisables.

CHAPITRE III

DÉFINITIONS DES CONCEPTS ET PROPRIÉTÉS DE REPRÉSENTATION DE MODÈLE DE COMPOSANTS ÉLECTRONIQUES

Ce chapitre présente de nouveaux concepts pour la représentation de modèle de composants. Cette représentation permet d'adopter une organisation hiérarchique des broches des composants afin de faciliter leurs connexions.

La première section définit plus en détails le concept d'interface et de configuration. Puis, la deuxième section présente des postulats qui garantissent une représentation fidèle aux composants. La troisième section explique la gestion des contraintes et des connexions. Ensuite, la quatrième section décrit les relations entre les différents objets du *netlist* par un réseau sémantique. Et enfin, un exemple pratique est donné à titre de validation des concepts.

3.1 Définitions de termes

Dans l'approche proposée, certains termes utilisés ont des sens spécifiques auxquels s'attachent des propriétés et des remarques. Ils sont définis dans les sections qui suivent.

3.1.1 Interface

Une *interface* représente les connexions externes d'un composant et l'accessibilité à ses comportements [27]. En restant fidèle à cette définition, une caractéristique supplémentaire peut être ajoutée. Une interface d'un composant est un ensemble de broches nécessaires et suffisantes pour une connexion fonctionnelle [15]. Le concepteur peut regrouper d'avance l'ensemble des broches qui constitue la fonctionnalité d'une interface. Autrement dit, les interfaces peuvent être créées avant les connexions des broches. Par exemple, les broches d'un composant qui se connecteront à un bus d'adresse forment une interface ; ou encore, les

broches d'une même banque d'un FPGA peuvent former une interface. Les interfaces deviennent des parties intégrantes du composant. La figure 3.1 montre une représentation des broches d'un composant A par des interfaces. Par exemple, l'interface i_1 contient les interfaces i_3 et i_4. Les interfaces de C_A de la figure 3.1 et de la figure 3.2-(b) sont identiques sauf que les broches sont cachées dans la figure 3.2-(b). Dans les figures 3.2-(a) et 3.2-(a'), le composant A se connecte par les broches 4 et 5 (notés *A.4* et *A.5*) respectivement aux broches *C.3* et *C.4* du composant C. Le concepteur sachant à l'avance que *A.4* et *A.5* fonctionnent ensemble, alors il peut les regrouper ensemble sous un même objet (c.-à-d. interface) avant la conception du circuit. Telles qu'illustrées par la figure 3.2-(b), toutes les broches des composants A, B et C sont représentées par leurs interfaces. Dans la figure 3.2-(b'), l'utilisateur connecte l'interface i_2 du composant A (i_2 contient les broches *A.4* et *A.5*) à l'interface i_1 du composant C (i_1 contient *C.3* et *C.4*). Le circuit de la figure 3.2-(b') est équivalent à celui de la figure 3.2-(a').

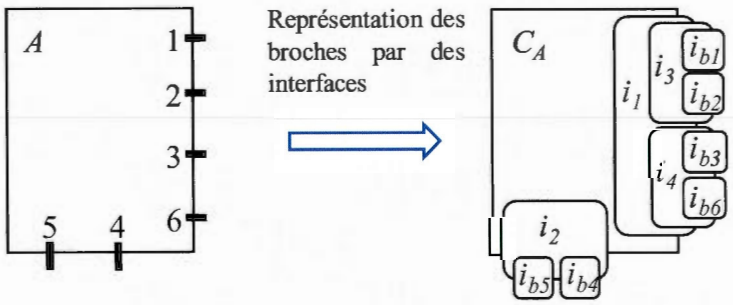


Figure 3.1 Exemple d'une représentation hiérarchique des broches du composant A par des interfaces.

Chaque broche est représentée par une interface initiale appelée *interface basique* (noté i_b). Une interface peut représenter aussi un ensemble d'interfaces existantes [15]. Dans ce cas, il y a une hiérarchie d'interfaces où les interfaces basiques sont les feuilles. Ceci implique qu'une interface peut contenir plusieurs niveaux de hiérarchie d'interfaces. Par exemple, dans la figure 3.2-(b), l'interface i_1 du C_A contient les interfaces i_3 et i_4 qui contiennent les interfaces basiques. Une interface qui regroupe plusieurs autres interfaces est dite *interface*

mère ou *interface parent* (notée i^+)[13]. À la suite de sa structure hiérarchique, l'interface mère a des interfaces enfants directes (du premier niveau de la hiérarchie) et interfaces-descendants (tous les niveaux de hiérarchie sauf le premier niveau). La cardinalité d'une interface i (notée $|i|$) est définie par le nombre de ses interfaces-enfants directes.

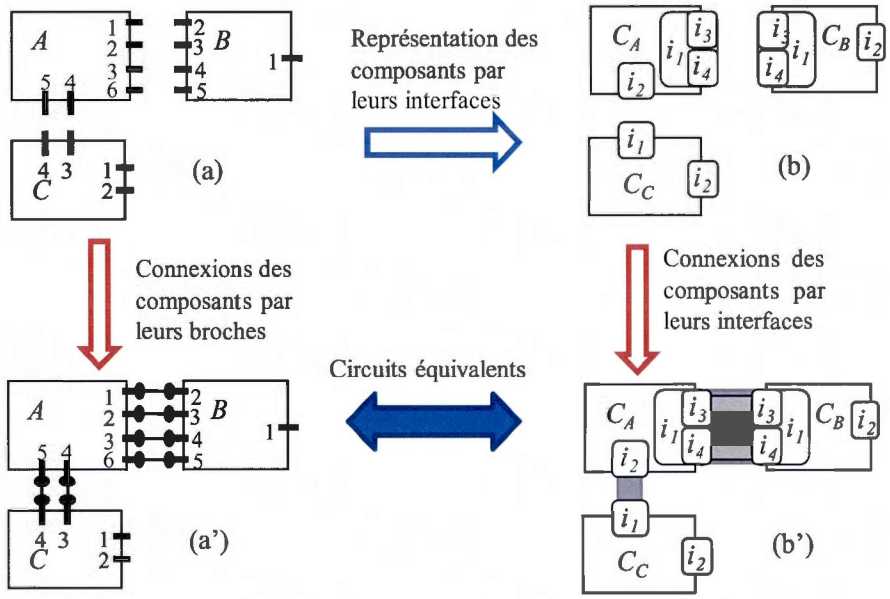


Figure 3.2 Représentations équivalentes d'un circuit schématique et d'un circuit créé à l'aide d'interfaces.

3.1.2 Configuration

Suite à la définition d'une interface, un composant a en fin de compte toutes ses broches représentées par des interfaces. Un ensemble d'interfaces qui contient exactement toutes les broches d'un composant A est appelé *configuration* et est noté C_A. Une configuration qui ne contient que des interfaces basiques (sans hiérarchie) est appelée *configuration basique*. Tout composant a primitivement une configuration basique. Un composant peut avoir plusieurs configurations selon les combinaisons des interfaces, mais il ne fait usage que d'une seule (configuration) lors de la conception d'un *netlist*. Toutes les configurations subséquentes (appelées *configurations avancées*) à la configuration basique ont des interfaces qui sont

[13] On utilisera des fois la notation i^+ pour insister sur le rôle d'une interface en tant que mère. Quand il n'est pas nécessaire, on négligera l'exposant +.

structurées à la base des interfaces basiques[14]. De plus, une configuration est toujours spécifique à un composant, car elle contient des caractéristiques exclusives à ce composant (telles que des contraintes physiques et électriques qui décrivent le composant). À l'étape de la conception logique (Capture) d'un circuit imprimé, l'utilisateur connait d'avance tous les composants sélectionnés (pour le *netlist*) et donc toutes les connexions qui auront lieu, mais sans nécessairement connaitre leurs détails. En d'autres termes, il est possible de créer les configurations avant l'usage des composants dans le *netlist*. Comparativement à la représentation usuelle d'un composant où les broches sont indépendantes les unes des autres, la configuration représente l'ensemble des connectivités fonctionnelles (d'un composant) dans un *netlist* selon les intentions du concepteur. On note par $C_A.i$ l'interface i d'une configuration C d'un composant A. La cardinalité d'une configuration C (noté $|C|$) est le nombre d'interfaces qui la constituent sans compter les interfaces des encapsulations (par abus de langage, on y réfère par les interfaces enfants directes de la configuration C).

Conformément aux définitions des termes présentés dans cette section, la section suivante établit des postulats qui s'appliquent aux interfaces et configurations. Ces principes permettent d'assurer la consistance des concepts établis avec les représentations réelles des composants dans un circuit.

3.2 Postulats

Des postulats sont définis formellement pour compléter les définitions établies. Ils sont partie intégrante de l'approche proposée et *doivent être respectés* tout au long du développement afin d'assurer l'intégrité des *netlists*.

3.2.1 Atomicité d'interface

La propriété d'*atomicité* consiste à représenter chaque broche par une interface basique dans un *netlist*. Une broche d'un composant programmable peut avoir différentes fonctionnalités (dépendamment de ses propriétés programmées) et donc différentes interfaces basiques, car elle change de connectivités fonctionnelles. Mais lors de son instanciation dans un *netlist*, une

[14] La configuration basique est automatiquement générée et est par défaut la configuration de tout composant. Pour les configurations avancées, elles sont manuellement conçues.

seule interface basique est utilisée pour chaque broche. Soit, une broche p d'un composant A, la fonction $g(p)$ est la relation de p avec une ou plusieurs interfaces basiques i_b telle que :

$$g(p) = \{i_{b1}, i_{b2}, \dots, i_{b\alpha}\} \tag{3.1}$$

où α est le nombre de propriétés possibles que la broche p peut avoir.

La fonction $g(p)$ retourne l'ensemble des interfaces basiques que la broche p peut avoir. Si le composant A n'est pas programmable, alors $g(p)$ retourne l'ensemble qui contient un singleton.

Dans un *netlist*, chaque broche p doit être représentée par une seule interface basique qui appartient à l'ensemble solution de $g(p)$. Soit, la fonction $g'(p)$ est la relation d'une broche p avec l'interface basique i'_{bk} qui la représente dans le *netlist* telle que :

$$g'(p) = instance\big(g(p)\big) = i'_{bk} \tag{3.2}$$

avec $i'_{bk} \in \{i_{b1}, i_{b2}, \dots, i_{b\alpha}\}$ et $1 \leq k \leq \alpha$.

L'instance d'une interface basique (en résumé) consiste à assimiler les valeurs de la broche p à l'interface i'_{bk}. Les détails de l'instance d'une interface basique sont couverts au chapitre V car il y a des concepts qui doivent être introduits et pris en compte.

3.2.2 Existence d'interface

Toute interface mère est un ensemble d'interfaces formé d'une combinaison d'interfaces basiques ou d'autres interfaces mères existantes (voir la figure 3.3). Il faut au moins deux interfaces pour en concevoir une nouvelle. Soit :

$$\begin{aligned} &\forall\, i^+ \in I, \exists i_{bk}, \dots, i_{bm}, i_r^+, \dots, i_s^+ \in I \text{ tel que} \\ &i^+ = \{i_{bk}, \dots, i_{bm}, i_r^+, \dots, i_s^+\} \end{aligned} \tag{3.3}$$

avec $|i^+| \geq 2$; I est l'ensemble des interfaces existantes d'un composant A ; $i_{bk}, \dots, i_{bm}$ sont des interfaces basiques avec $k \leq n$, $m \leq n$; n est le nombre total d'interfaces basiques du

composant A ; $i_r^+, \ldots, i_s^+$ sont des interfaces mères existantes avec $r \leq t$, $s \leq t$; t est le nombre total des interfaces mères existantes dans I.

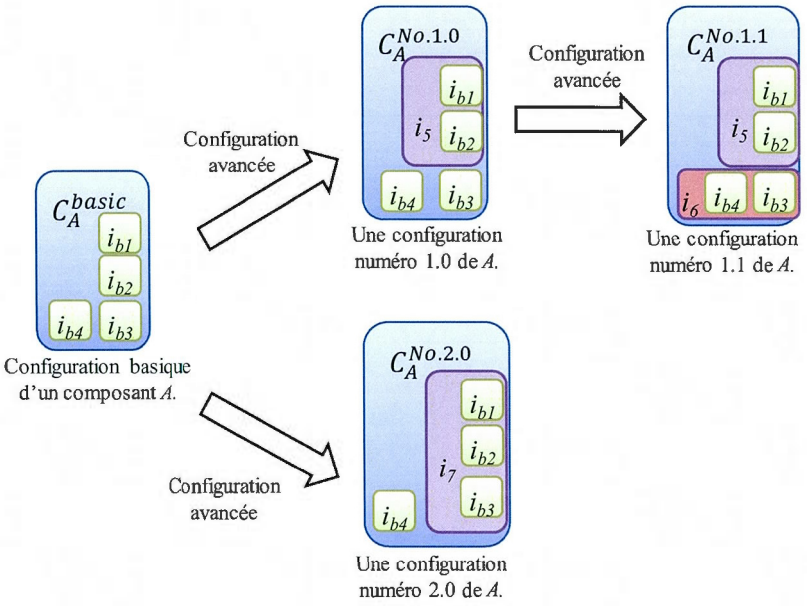


Figure 3.3 Formation de plusieurs configurations à partir de la configuration basique d'un composant A.

Dans la figure 3.3, plusieurs versions de configurations avancées peuvent être construites à l'aide d'interfaces nouvelles ou existantes. La configuration de numéro 1.0 n'est pas forcément une étape intermédiaire entre la configuration basique et celle de numéro 1.1. L'utilisateur pourrait directement former la configuration de numéro 1.1 à partir de la configuration basique.

Toutes les interfaces mères existantes utilisent la même expression (3.3). Pour concevoir les encapsulations, l'ensemble de départ I des interfaces existantes est initialement composé d'interfaces basiques. Toute nouvelle interface (mère) est ajoutée à l'ensemble I. Soit, I augmente d'un élément à la création d'une nouvelle interface.

Avec $|i| \geq 2$, toute interface mère est contrainte de regrouper plus qu'une seule interface enfant, empêchant ainsi d'avoir une boucle infinie d'interfaces vides de sens. On veut éviter des exemples triviaux tels que $i_1=\{i_{b1}\}$, $i_2=\{i_1\}$, $i_3=\{i_2\}$; par hiérarchie, on obtient i_3 contenant

plusieurs niveaux d'interfaces pour représenter inutilement l'interface basique i_{bl}. De plus, une interface triviale ne représente aucune connectivité fonctionnelle, car elle n'apporte aucune nouvelle information différente de son interface enfant.

Il y a une différence entre $i \in j^+$ et $i \subset j^+$. L'appartenance signifie que i est un élément de j^+ (c.-à-d. j^+est parent ou ancêtre de i). Alors que l'inclusion signifie que tout élément de i est un élément de j^+ sans nécessairement que la structure de i se retrouve dans j^+. Autrement dit, $i \in j^+ \Rightarrow i \subset j^+$ est vrai dans un seul sens et pas forcément dans l'autre. Par exemple, dans la figure 3.3, l'interface i_5 de $C_A^{No.1.1}$ est incluse dans i_7 de $C_A^{No.2.0}$mais i_5 n'appartient pas i_7.

La relation + des interfaces est la sommation d'interfaces enfants pour la création d'une interface mère. Dans l'expression (3.3), une écriture alternative de i serait $i = i_{bk} + \cdots + i_{bl} + i_r^+ + \cdots + i_s^+$. On n'utilise pas la relation d'union ∪ car elle crée une confusion avec l'union ensembliste où les interfaces enfants des interfaces ajoutées deviendraient les enfants directs de la nouvelle interface. Ici, on veut créer un nouvel ensemble où les interfaces ajoutées sont des éléments et non des ensembles.

Avec la propriété d'existence, une interface appartient toujours à une configuration. Physiquement, cela assure l'intégrité des interfaces avec les broches d'un composant.

3.2.3 Équivalence d'interfaces

Dans un circuit, toute broche est représentée par une interface basique, mais toute interface basique n'est pas forcément une broche. Une interface mère peut être déclarée basique lorsque tous ses enfants sont conceptuellement équivalents. Deux interfaces sont dites *équivalentes* lorsque les broches qu'elles représentent se connecteront à une même broche d'un autre composant. Sachant leur usage à l'avance, le concepteur regroupe les interfaces équivalentes sous une même interface mère basique.

Plus formellement, soient un composant A ayant un nombre n de broches p, $g'()$ une fonction qui retourne l'interface basique d'une broche à utiliser dans la configuration de A et une relation d'équivalence ≡ tels que :

$$\forall p_1, p_2, \ldots, p_m \in A, m \leq n,$$

$$si\ g'(p_1) \equiv g'(p_2) \equiv \cdots \equiv g'(p_m)$$
$$alors\ g'(p_1) + g'(p_2) + \cdots + g'(p_m) = i_b^+ \quad (3.4)$$

avec i_b^+ une interface mère basique.

La relation d'équivalence $g'(p_k) \equiv g'(p_l)$ signifie que :

1. Les broches p_k et p_l appartiennent au même composant.
2. Les valeurs des contraintes de $g'(p_k)$ sont équivalentes à celles de $g'(p_l)$ (avec $k \leq m$, $l \leq m$).
3. Leur usage dans le *netlist* les connecte à une même interface basique d'un autre composant.

Dans la figure 3.4, un composant peut avoir plusieurs de ses broches de type POWER connectées à un même *net*. Conceptuellement, ce groupe de broches serait équivalent à une seule broche, car ce sont les sorties fonctionnelles qui sont considérées et non des broches individuelles (en fait, le manufacturier aurait pu les connecter à l'intérieur du composant et ne montrer qu'une seule broche). À ce point de développement, la création d'une interface mère basique est manuelle. Une configuration basique qui utilise des interfaces mères basiques (et des interfaces basiques uniquement) est dite configuration basique avancée par abus de langage.

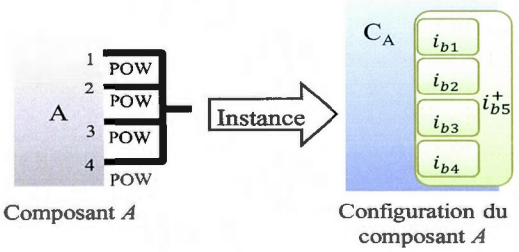


Figure 3.4 Exemple d'équivalence d'interfaces dans un FPGA.

Dans la figure 3.5-(a), les broches 1 et 2 du composant *A* sont connectées à la même broche 1 du composant *B*. Conceptuellement, on représente les interfaces équivalentes $C_A.i_{b1}^{(POW)}$ et $C_A.i_{b2}^{(POW)}$ par une interface mère basique $i_{b5}^{+(POW)}$. L'utilisateur interagit seulement avec l'interface mère basique et ne peut pas accéder les enfants pour des connexions individuelles.

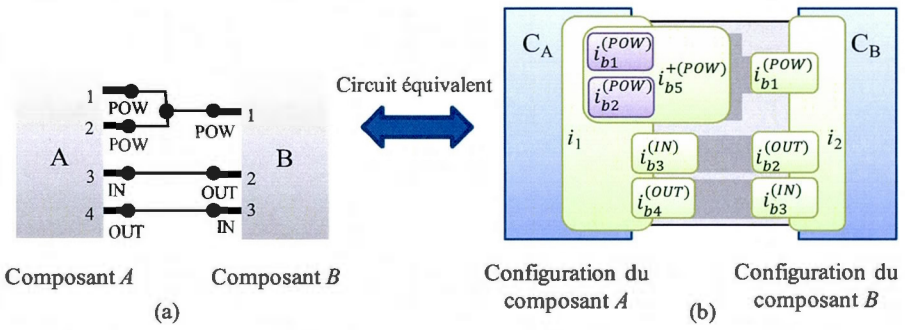


Figure 3.5 Exemple de connexion de l'interface mère basique.

Pour que deux interfaces i et j se connectent, elles requièrent une précondition qu'elles aient le même nombre d'interfaces-enfants directs (c.-à-d. $|i|=|j|$). En utilisant la propriété d'équivalence dans la figure 3.5-(b), l'utilisateur connecte les interfaces enfants de $C_A.i_1$ à celles de $C_B.i_1$ en faisant une correspondance bijective. Ce qui n'aurait pas été possible si l'interface mère basique n'existait pas.

Les valeurs des contraintes d'une interface mère basique ne sont pas forcément définies (par exemple, sa contrainte de direction est de type *NONE*)[15]. Il revient à l'utilisateur de donner ses valeurs. Dans une interface mère basique, les enfants ne sont pas accessibles pour des connexions individuelles. Cependant, par conformité à la création d'un *netlist*, on supposera que tous les enfants sont déjà interconnectés dans leur interface mère basique avant de connecter celle-ci aux autres interfaces. De cette façon, toute connexion de l'interface mère basique impliquera la connexion immédiate de tous ses enfants (telle que requérait l'intention planifiée de leur connexion dans un *netlist* usuel). Aussi, pour qu'une interface mère soit déclarée comme basique, il faut que tous ses enfants soient basiques.

La propriété d'équivalence résout une problématique de la connexion automatique, car elle rapporte les interfaces compatibles à un même nombre d'interfaces enfants. Il devient plus facile de les connecter par une correspondance bijective. Aussi, une autre raison est qu'on veut éviter de suggérer automatiquement les interfaces compatibles à une interface venant

[15] Le type est référé par *mode* dans le jargon de conception matériel. Le type NONE, tel qu'introduit dans ce mémoire, il peut se connecter à tous les types sans exception. Il est par défaut le type de toutes les interfaces sauf pour les interfaces basiques associées à des broches.

d'une même configuration (mais c'est possible de les connecter manuellement). Lorsqu'une configuration est utilisée pour représenter un composant, elle ne montre que les interfaces connectables aux autres composants (suivant la définition d'une interface). Avec cette stratégie, la représentation d'un composant est allégée. Cependant, il existe des astuces qui permettent de contourner ce point si plus tard l'utilisateur ne veut pas modifier les configurations déjà établies (durant la conception du circuit). À supposer, par exemple, dans la figure 3.5-(b), l'utilisateur avait utilisé les configurations basiques, alors les interfaces qui seront suggérées à $C_B.i_{b1}^{POW}$ seront $C_A.i_{b1}^{POW}$ et $C_A.i_{b2}^{POW}$ simultanément (dans la direction du composant B vers le composant A). Mais dans la direction de A vers B, seule l'interface $C_B.i_{b1}^{POW}$ sera suggérée pour une connexion avec $C_A.i_{b1}^{POW}$ ou $C_A.i_{b2}^{POW}$.

3.2.4 Fermeture de configuration

Quelle que soit la configuration d'un composant A, elle doit contenir les interfaces basiques de toutes les broches de A (voir la figure 3.3). Toutes les configurations sont équivalentes à la configuration basique lorsqu'elles sont ramenées à un niveau plat (c.-à-d. sans hiérarchie). La relation + des interfaces est une *relation fermée* sur toute configuration C d'un composant A. Car, quelle que soit l'interface i obtenue par l'addition de plusieurs interfaces, alors $i \subset C_A$. Soit :

$$\forall i_r, \dots, i_s \in C_A, alors\ i_r + \cdots + i_s = i \subset C_A \tag{3.5}$$

où $r \leq n, s \leq n$; n est le nombre d'interfaces dans C_A.

Il y a une différence entre $i \in C_A$ et $i \subset C_A$. L'appartenance $i \in C_A$ signifie que l'interface i est une interface (ou descendant d'une interface) de C_A. Alors que l'inclusion $i \subset C_A$ signifie que $\forall i' \in i$ alors $i' \in C_A$ sans nécessairement avoir $i \in C_A$. Quelle que soit une interface $i \subset C_A$, une nouvelle configuration $C_A^{'}$ peut être formée où $i \in C_A^{'}$.

3.2.5 Partage d'interface

Dans une configuration, une interface s est dite *partageable* lorsqu'elle peut être une interface enfant de plusieurs interfaces mères simultanément ; sinon s est une *interface non*

partageable[16]. Par exemple, dans la figure 3.6, l'interface basique i_{b3} représente une horloge qui doit appartenir à deux groupes d'interfaces ayant différentes connectivités fonctionnelles. Une interface mère doit être créée pour chaque groupe. Dans ce cas, l'interface basique i_{b3} est déclarée *partageable* (notée *i.partageable = vrai*) afin qu'elle existe dans les deux interfaces mères. L'intersection ∩ des interfaces qui ne sont pas encapsulées (l'une n'est pas l'ancêtre de l'autre) est l'ensemble des interfaces partageables qu'elles ont en commun. Soit, une configuration C d'un composant A et i et j des interfaces :

$$\forall i,j \in C_A, si\ i \notin j\ et\ j \notin i\ alors\ i \cap j = \{s/s.partageable = vrai\}$$
$$avec\ s \in i\ et\ s \in j\ et\ s \in C_A. \quad (3.6)$$

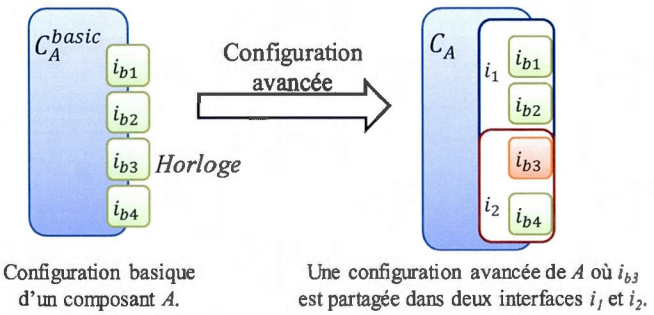


Figure 3.6 Exemple de partage d'interface.

Bien qu'une interface partageable puisse exister dans plusieurs interfaces mères, il s'agit d'une même instance. Lorsqu'elle subit un changement, alors toutes ses interfaces mères doivent être mises à jour. Lorsqu'une interface est déclarée partageable, alors récursivement tous ses enfants le deviennent aussi.

Suite à la définition des propriétés des interfaces, la gestion des contraintes et des connexions est introduite dans la section qui suit.

3.3 Gestion des contraintes et des connexions

Les concepts d'interface et configuration introduisent de nouvelles approches de gestion des contraintes et des connexions. La conception du *netlist* devient différente de l'approche

[16] Par défaut, toute interface est non-partageable.

usuelle. La première section présente la gestion des contraintes puis la deuxième section présente la gestion des connexions.

3.3.1 Gestion des contraintes

Dans le développement actuel, certaines contraintes prennent des valeurs numériques (telles que le voltage, la fréquence) et d'autres prennent des valeurs énumérées (tel que le type). En fait, chaque contrainte numérique peut être spécifiée sous forme d'intervalle de variations autorisées pour un signal électrique, telles que des valeurs minimale (Min), maximale (Max) typiques et pire (Pire). Soit, chaque contrainte numérique possède les valeurs Min, Max et Pire. Dans la pratique, un signal qui passe dans un *net* doit être compris dans l'intervalle de valeurs seuils [Min, Max]. À ce point de développement, seules les valeurs Min et Max sont prises en compte pour déterminer la compatibilité électrique.

Les valeurs des contraintes ne sont pas associées aux broches et au composant, mais plutôt aux interfaces et à ses configurations. L'avantage de cette stratégie permet qu'un composant soit représenté par plusieurs configurations différentes où chacune d'elles possède ses propres valeurs de contraintes. Les valeurs des contraintes dans une configuration (et ses interfaces) sont modifiables à volonté sans pour autant affecter les autres configurations. Il peut arriver qu'une même interface avec les mêmes valeurs soit requise dans différentes configurations. Pour éviter que l'utilisateur ne reprenne le même travail de création et d'affectation de valeurs, il est possible d'utiliser une copie conforme d'une interface (provenant d'une configuration existante). Cette copie garde les mêmes valeurs de contraintes pour servir à la création d'une nouvelle interface ou une nouvelle configuration.

Une interface mère peut hériter une valeur d'une contrainte de ses interfaces enfants sous la condition qu'elles aient toutes la même valeur pour cette contrainte. Par exemple, si tous les enfants ont la contrainte voltage de valeur 3.3V, alors l'interface mère peut avoir la même valeur. De même, si l'utilisateur affecte une valeur de contrainte à une interface mère, alors il peut décider dans la même opération que tous les enfants de celle-ci soient affectés aussi par la même valeur. Subséquemment, un processus récursif d'assignation se déclenche de mères à enfants jusqu'aux interfaces basiques et vice-versa. L'utilisateur peut décider qu'une contrainte particulière (ou un ensemble de contraintes) soit affectée par héritage.

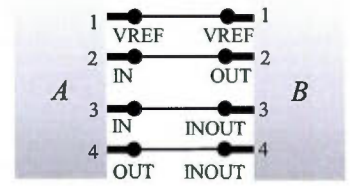
3.3.2 Gestion des connexions

Pour être en mesure de connecter deux composants A et B par leurs interfaces $C_A.i$ et $C_B.j$, il faut nécessairement qu'elles aient le même nombre d'interfaces enfants directs (c.-à-d. $|C_A.i|=|C_B.j|$). Cette condition préalable de connexion est dite *compatibilité de cardinalité*. Chaque interface enfant direct de i devient connectée à une interface enfant direct (appelée un *correspondant*) de j. Leur connexion devient telle qu'une relation bijective entre deux ensembles cartésiens où chaque élément de l'ensemble A de départ a une seule image dans l'ensemble B d'arrivée et vice-versa. Par itération de mères à enfants directs, la connexion bijective s'étant jusqu'aux interfaces basiques. Par exemple, dans la figure 3.7-(a), un composant A se connecte à un composant B. Des circuits équivalents (mais avec différentes configurations) sont montrés dans la figure 3.7-(b) et la figure 3.7-(c). Toutes les interfaces basiques de $C_A.i_1$ sont connectées avec les interfaces basiques de $C_B.j_1$. Il est à remarquer que le nombre d'interfaces basiques de $C_A.i_1$ est égal au nombre d'interfaces basiques de $C_B.j_1$. Soit, pour être en mesure de connecter deux interfaces i et j, il faut nécessairement qu'elles aient le même nombre d'interfaces basiques.

Dans une interface mère, chaque interface basique a un paramètre *rang* qui détermine l'ordre de l'inter-correspondance avec son correspondant. Autrement dit, une interface enfant et son correspondant ont le même rang de part et d'autre des interfaces mères (voir la figure 4.5). Donc, pour avoir une connexion entre deux interfaces compatibles, il suffit de déterminer le rang de leurs interfaces basiques.

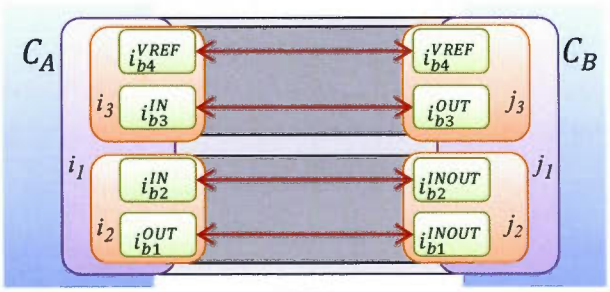
Les interfaces ayant le même rang sont connectées au même *net* du bus. Il est à noter que les rangs n'ont plus d'importance une fois que les *nets* sont créés. Donc, même si une interface partageable change de rang lors d'une seconde connexion, alors la première connexion n'est pas affectée.

Deux interfaces à connecter doivent aussi être compatibles sur toutes les contraintes électriques. Cette deuxième condition préalable de connexion est dite de *compatibilité de contraintes*. Soit, pour que deux interfaces soient connectables (compatibles), il faut nécessairement que les deux conditions préalables soient vérifiées.

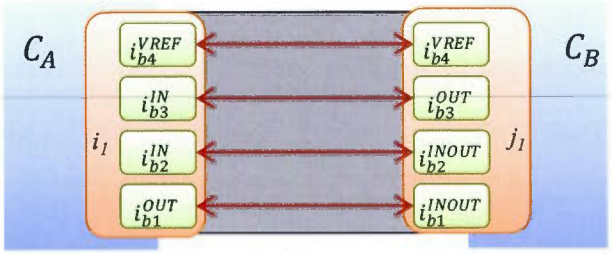


Composant A Composant B

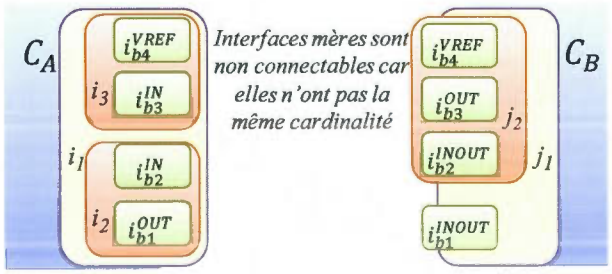
(a)



(b)



(c)



Configurations du composant A

(d)

Configurations du composant B

Figure 3.7 Connexions d'interfaces mères entre différentes configurations de A et B.

La propriété de partage permet de résoudre une problématique de connexion d'interfaces. Lorsqu'une interface mère se connecte à une autre, alors leurs enfants se correspondent dans un certain ordre de connexion. Une interface enfant non partageable qui se connecte indépendamment de son interface mère implique la perte de la structure de l'interface mère et de tous ses ancêtres. La propriété de partage autorise plusieurs connexions d'une interface enfant sans forcément connecter l'interface mère. Une interface non partageable peut aussi se connecter à plusieurs interfaces, mais les connexions doivent se faire simultanément (c.-à-d. en une seule opération). La connexion d'une interface non partageable est dite *connexion fermée*. Par opposition à *connexion ouverte*, une interface partageable peut avoir différentes connexions à différents moments. La propriété de la connexion fermée est importante pour réduire la liste des interfaces compatibles. Car, au fur et à mesure qu'une interface se connecte, la liste des interfaces disponibles devient plus courte.

3.4 Réseau sémantique

Pour mieux comprendre les relations des termes définis, un réseau sémantique est établi tel que présenté dans la figure 3.8. Un *netlist* est décrit par un ensemble de composants qui sont connectés par des bus. Un composant et ses broches sont respectivement représentés par une configuration et les interfaces qui les représentent. Une interface peut être composée d'une encapsulation d'interfaces. La configuration et les interfaces possèdent des contraintes. Un *net* connecte plusieurs interfaces basiques tout en prenant en compte leurs contraintes (dont certaines peuvent être associées au *net*). Un bus regroupe plusieurs *nets*.

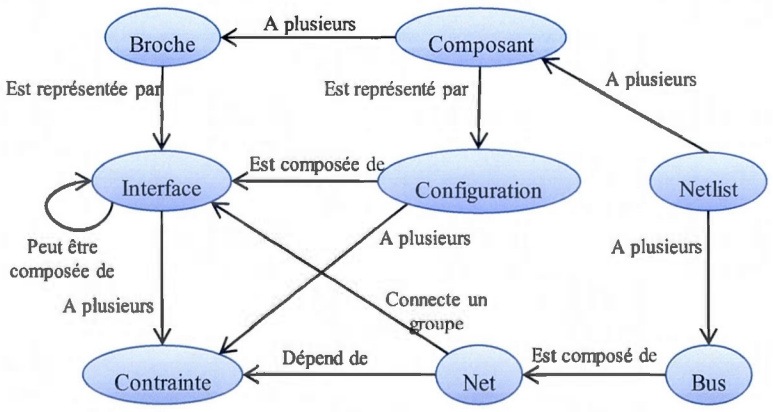


Figure 3.8 Réseau sémantique des objets d'un *netlist*.

3.5 Exemple pratique

L'exemple pratique suivant montre partiellement les connexions d'un processeur E3 et d'une mémoire RAM E7 à la figure 3.9.

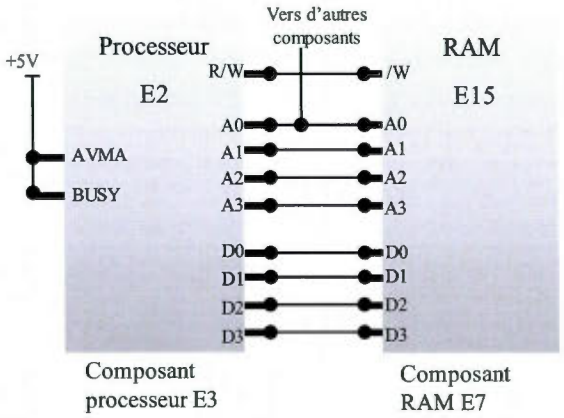


Figure 3.9 Exemple pratique de connexion partielle d'un processeur et d'une mémoire.

Un bus de données (D0-D3) et un bus d'adresse (A0-A3) connectent C3 à C7. Aussi, la broche (R/W) de lecture et d'écriture de E3 est connectée à la broche (/W) de E7. En appliquant les propriétés établies, la transformation obtenue est montrée à la figure 3.10.

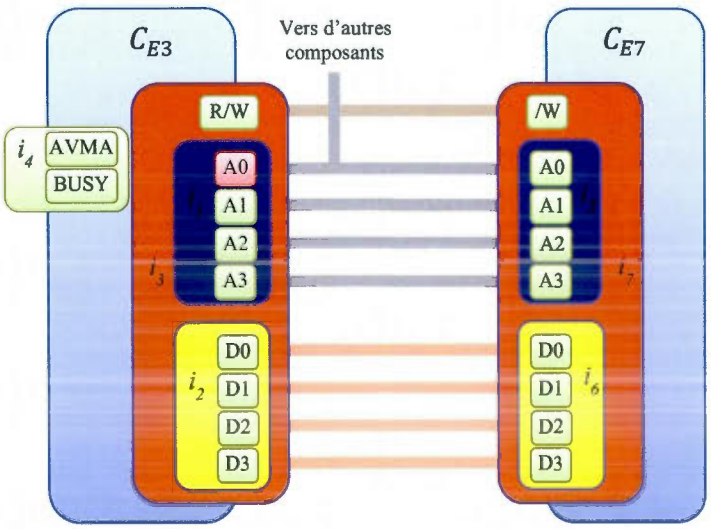


Figure 3.10 Représentation de l'exemple de la figure **3.9** par des interfaces.

Les configurations avancées obtenues sont dépendantes du choix du concepteur. Les interfaces qui connectent C_{E3} à C_{E7} sont $C_{E3}.i_3=\{i_{R/W},i_1,i_2\}$ et $C_{E7}.i_7=\{i_{/W},i_5,i_6\}$. Les transformations imposées par chacune des propriétés (postulats) sont décrites comme suit :

- Atomicité : La représentation des composants E3 et E7 en configurations basiques sont respectivement C_{E3}^{basic} et C_{E7}^{basic}. Chaque broche est associée à une interface basique qui hérite des mêmes contraintes électriques. À partir de la configuration basique obtenue, les configurations avancées C_{E3} et C_{E7} peuvent être créées en appliquant les autres propriétés.
- Existence : toutes les interfaces mères i_1, i_2, i_3, i_4, i_5, i_6 et i_7 ont plus que 2 interfaces enfants.
- Équivalence : les interfaces basiques $C_{E3}.i_{AVNA}$ et $C_{E3}.i_{BUSY}$ sont connectées ensemble à une même tension +5V. Ces deux interfaces sont équivalentes et regroupées en une seule interface basique qui est $C_{E3}.i_4$.
- Fermeture : les configurations avancées C_{E3} et C_{E7} sont fermées, car toutes les interfaces basiques (des configurations basiques C_{E3}^{basic} et C_{E7}^{basic}) sont présentes et aucune interface externe (provenant de la configuration d'un autre composant) ne participe aux nouvelles interfaces de C_{E3} et C_{E7}.
- Partage : l'interface $C_{E3}.i_{A0}$ est partageable, car elle participe toute seule à une connexion avec un autre composant. Du fait que la connexion de $C_{E3}.i_{A0}$ est devenue indépendante de celle de $C_{E3}.i_1$, alors cette situation force le concepteur à la déclarer partageable pour qu'un autre composant puisse avoir accès.

Il faut remarquer que $C_{E3}.i_{A0}$ est déclarée partageable dans $C_{U2}.i_1$ mais n'est pas incluse dans aucune autre interface mère. La propriété de partage lui a permis d'être partagée et de participer à des connexions différentes de $C_{U2}.i_1$. D'un autre côté, $C_{E7}.i_{A0}$ n'est pas partageable bien qu'elle soit connectée à $C_{E3}.i_{A0}$ (et donc dans le même processus aux autres composants connectés à $C_{E3}.i_{A0}$). Si une nouvelle interface basique doit être ajoutée à ce *net*, alors l'utilisateur se sert de la connexion ouverte de $C_{E3}.i_{A0}$.

La vue simplifiée de la figure 3.10 est présentée à la figure 3.11. L'utilisateur se sert des configurations avancées C_{U2} et C_{U15} pour connecter le processeur E3 et la RAM E7 à l'aide

de leurs interfaces respectives $C_{U2}.i_3$ et $C_{UI5}.i_7$ sans se soucier des détails internes. Dans le chapitre suivant, un algorithme permet d'interconnecter automatiquement les interfaces internes sans intervention de l'utilisateur.

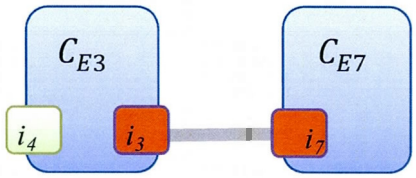


Figure 3.11 Vue simplifiée de la figure **3.10** (vue utilisateur).

À supposer que, dans un autre circuit, l'utilisateur se sert d'autres configurations similaires à C_{E3} et C_{E7} où les interfaces $C_{U2}.i_3$ et $C_{UI5}.i_7$ n'existent pas, mais juste leurs interfaces enfants. Dans ce cas, l'utilisateur connecte C_{E3} à C_{E7} par trois connexions $C_{E3}.i_1$, $C_{E3}.i_2$ et $C_{E3}.i_{R/W}$ respectivement à $C_{E7}.i_5$, $C_{E7}.i_6$ et $C_{E7}.i_{/W}$.

Grâce aux définitions et propriétés formellement définies dans ce chapitre III, la représentation des composants est simplifiée. Le premier objectif de ce mémoire est atteint. Le chapitre IV présente le processus de connexion automatique des interfaces compatibles.

CHAPITRE IV

SUGGESTION D'INTERFACES COMPATIBLES ET CONNEXION AUTOMATIQUE

L'utilisateur connecte les interfaces pour créer le *netlist*. Afin de l'assister dans la conception, ce chapitre présente l'algorithme de suggestion des interfaces compatibles dans la première section. Puis, la deuxième section présente l'algorithme de connexion automatique. Et enfin, la troisième section explique la direction de développement des algorithmes en rapport avec la conception de *netlist*.

4.1 Algorithme de suggestion d'interfaces compatibles

La compatibilité entre deux interfaces est évaluée sur deux critères : la compatibilité de cardinalité et la compatibilité des contraintes. Dans la figure 4.1, l'algorithme de vérification de compatibilité consiste à évaluer les deux critères entre deux interfaces i et j qu'il prend en entrée. La sortie de l'algorithme est un booléen qui confirme si i et j sont compatibles ou non. On prend pour hypothèses que les propriétés des interfaces sont respectées.

Chaque interface possède la liste de n contraintes qui s'applique à une broche. On note par $i.c_k$ une contrainte donnée c_k d'une interface i avec $1 \leq k \leq n$. À ce point de développement, la compatibilité des contraintes consiste à vérifier que chaque contrainte c_k a un même intervalle de valeurs dans i et j (sauf pour certaines contraintes telles que le type, la position). Quel que soit une contrainte donnée c_k de i et j compatibles, alors $i.c_k.Min = j.c_k.Min$ et $i.c_k.Max = j.c_k.Max$. Aussi, on considère que la valeur indéfinie est une valeur acceptable pour la compatibilité, même si le Min ou le Max de l'autre interface est différent. Il peut arriver des fois que l'utilisateur n'ait pas encore défini certaines contraintes, mais il veut quand même établir les connexions.

Algorithme de compatibilité entre deux interfaces *i* et *j*

```
1:  Boolean compatible (Interface i, Interface j)
2:  Begin
3:      if ( compatibleType(i, j) ) then    //Vérifier le type avec la compatibilité de caridnalité
4:          for each constaint c
5:              if ( ( i.c.min ≠ j.c.min && i.c.min!=0 && j.c.min ≠ 0 )
6:                  || ( i.c.max ≠ j.c.max && i.c.max. ≠ 0 && j.c.max. ≠ 0 ) ) then
7:              return false  //Au moins une contrainte ne vérifie pas la compatibilité
8:          end for
9:      else return false           //Les nombres des types de i et j ne se correspondent pas
10:     end if
11:     return true                 //Tous les critères de compatibilité sont corrects
12: end compatible
```

Figure 4.1 Algorithme de compatibilité entre deux interfaces *i* et *j*.

D'autre part, la contrainte de type a été explorée plus en détails. Elle consiste à faire une correspondance des valeurs IN (signal entrant), OUT (signal sortant), INOUT (signal entrant et sortant), VREF (signal de référence), POW (courant d'alimentation) et GND (la masse électrique). Plus précisément, chaque contrainte serait en fait un critère de compatibilité et tous les critères doivent être vérifiés pour que deux interfaces soient compatibles ; dont le type qui est pris ici en exemple. Mais comme toutes les contraintes (sauf Type) utilisent similairement l'égalité des Min et Max, alors elles sont évaluées par une même exécution générique (lignes 5 et 6 de la figure 4.1). Celle du type, appelée *compatibleType*(), est développée séparément, car elle utilise une correspondance de valeurs.

Dans la figure 4.2, l'algorithme *compatibleType*(Interface *i*, Interface *j*) vérifie la compatibilité de cardinalité entre deux interfaces *i* et *j* en établissant une correspondance des types. Une correspondance est faite entre les IN de *i* avec les OUT de *j*. Puis, s'il en reste des IN dans *i*, alors ils se font correspondre aux INOUT de *j*. De même, les OUT de *i* se correspondent avec les IN ensuite les INOUT de *j*. Et enfin, le même nombre de INOUT de *i* doit être égal au nombre restant des IN, OUT et INOUT de *j* ; sinon *i* et *j* sont incompatibles. Les types VREF, POW et GND ne peuvent être connectés qu'à leur type identique[17] et donc *i* et *j* doivent aussi

[17] Les autres types peuvent parfois être connectés au GND ou POW dans les circuits réels. Par la propriété d'équivalence, lorsque plusieurs interfaces basiques d'un même composant doivent se connecter au même *net*, alors elles sont regroupées en une interface mère basique. Ce qui résout le problème d'incohérence de types.

avoir le même nombre pour chacun de ces types. Pour que *i* et *j* vérifient la compatibilité de cardinalité, il faut que la correspondance de leurs types soit exacte. Si la fonction *compatibleType*(*i*, *j*) retourne *vrai*, alors *i* et *j* sont compatibles pour la correspondance des types et donc implicitement le même nombre d'interfaces basiques.

Algorithme de compatibilité de types entre deux interfaces

```
1:  Boolean compatibleType(Interface i, Interface j)
2:  Begin
3:      //n (respectivement m) est le nombre de type dans i (respectivement j).
4:      if ( n_VREF == m_VREF && n_POW == m_POW && n_GND == m_GND ) then
5:          if ( n_IN <= m_OUT ) then        //Faire correspondre les IN de i en premier lieu.
6:              m_OUT = m_OUT − n_IN
7:              n_IN = 0
8:          else        //Il y a plus de IN dans i que de OUT dans j.
9:              n_IN = n_IN − m_OUT
10:             m_OUT = 0
11:             if ( n_IN<= m_INOUT ) then
12:                 m_INOUT = m_INOUT − n_IN
13:                 n_IN = 0
14:             else return false     //Impossible, il reste des n_IN sans correspondance.
15:             end if
16:         end if
17:         if ( n_OUT <= m_IN ) then    //Faire correspondre les OUT de i en second lieu.
18:             m_IN = m_IN − n_OUT
19:             n_OUT = 0
20:         else        //Il y a plus des OUT dans i que des IN dans j.
21:             n_OUT = n_OUT − m_IN
22:             m_IN = 0
23:             if ( n_OUT <= m_INOUT ) then
24:                 m_INOUT = m_INOUT − n_OUT
25:                 n_OUT = 0
26:             else return false     //Impossible, il reste des n_OUT sans correspondance.
27:             end if
28:         end if
29:         //Si les INOUT de i ne sont pas égaux aux types restants de j alors
30:         if ( n_INOUT ≠ m_IN + m_OUT + m_INOUT ) then
31:         return false      //Impossible, il reste des broches sans correspondance.
32:         end if
33:     return true       // Les interfaces i et j se correspondent au niveau des types.
34:     end if
35:     return false      //Impossible, pas de correspondance entre Vref, Pow et Gnd.
36: End compatibleType
```

Figure 4.2 Algorithme de correspondance de types entre deux interfaces.

Dans la figure 4.3, deux interfaces *i* et *j* sont détectées compatibles par application de l'algorithme *compatibleType*(*i*, *j*). Chaque interface enfant de *i* trouve un correspondant qui lui est compatible sur la contrainte de type. Le nombre restant de types dans *j* est nul. Donc, la fonction *compatibleType*(*i*, *j*) retourne *vrai*.

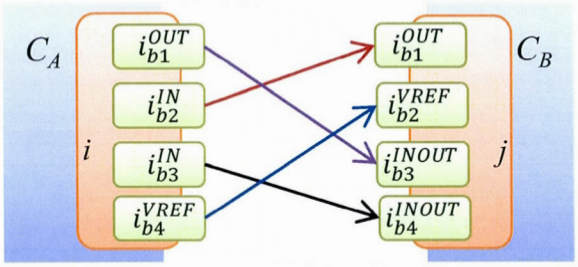


Figure 4.3 Exemple d'application de l'algorithme *compatibleType*(*i*, *j*).

Chaque interface est comparée à toutes les autres interfaces des autres composants (sur toutes les contraintes) et celles qui lui sont compatibles sont sauvegardées dans une liste. Ainsi, un répertoire est utilisé pour suggérer les listes des interfaces compatibles à l'utilisateur. Si l'utilisateur modifie la valeur d'une contrainte d'une interface alors la fonction de suggestion est automatiquement relancée pour une mise à jour. Lorsque l'utilisateur sélectionne une interface, sa liste d'interfaces compatibles lui est alors affichée. Puis, il choisit de la liste suggérée une ou plusieurs interfaces à connecter. La liste des interfaces sélectionnées est acheminée à l'algorithme de connexion automatique.

4.2 Algorithme de connexion automatique

L'algorithme de la connexion automatique prend en entrée une liste d'interfaces assurées d'être compatibles par l'algorithme de suggestion (d'interfaces compatibles) et sa sortie est un ensemble de *nets*. On prend pour hypothèse que le répertoire (des interfaces compatibles) trouvé par l'algorithme de suggestion (d'interfaces compatibles) est disponible.

En fait, l'algorithme de connexion automatique se divise en deux modules. Le premier module, appelé algorithme d'intercorrespondance automatique, détermine le rang des intercorrespondances entre deux ou plusieurs interfaces compatibles à connecter. Le deuxième module, appelé algorithme de création des *nets*, ajoute les interfaces ayant le même

rang à un même *net* pour créer une liste d'interconnexions. Bien que le *netlist* puisse être créé directement par un seul module, on procède par deux modules afin de permettre à l'utilisateur d'intervenir pour des changements. Ces changements lui permettent de vérifier ou modifier les rangs avant la création des *nets* ou encore de changer les valeurs des contraintes pour de nouvelles suggestions d'interfaces compatibles. Cette intervention manuelle entre le calcul des rangs et la création des *nets* est appelée *étape de mise en veille*.

Utilisant le répertoire trouvé par l'algorithme de suggestion (d'interfaces compatibles), l'algorithme d'intercorrespondance automatique *connectInter*(*i*, *j*) consiste à comparer récursivement les enfants des interfaces *i* et *j* jusqu'à arriver aux interfaces basiques pour leur affecter un rang (voir la figure 4.4).

Algorithme d'inter-correspondance automatique

```
1:  Boolean connectInter (Interface i, Interface j, int &rank)
2:  Begin
3:  //hashCompatibleInter est une hash liste <Interface, liste<Interface>> qui associe à
    //chaque interface sa liste d'interfaces compatibles. La fonction
    //hashCompatibleInter[i].contains(j) retourne vrai lorsque j se trouve dans la liste
    //compatible de i sinon faux.
4:      if (hashCompatibleInter[i].contains(j)==true && i.connected==false
5:          && j.connected==false) then
6:          i.connected=true     //L'attribut booléen connected de i et j devient vrai car
7:          j.connected=true     //elles sont compatibles.
8:          if (i.basic==true && j.basic==true) then
9:              i.setRank(rank)
10:             j.setRank(rank)
11:             rank=rank+1
12:         else if (i.basic==false && j.basic==false) then   //i et j ne sont pas basiques.
13:             for each child interface p of i
14:                 for each child interface q of j
15:                     connectInter(p,q,rank)        //Appel récursif.
16:                     //Sortir de la boucle interne si p est connecté.
17:                     if (p.connected) break inner For-loop
18:                 end for
19:             end for
20:         end if
21:     end if
22: End connectInter
```

Figure 4.4 Algorithme d'intercorrespondance automatique.

La *hash* liste *hashCompatibleInter*() est le répertoire trouvé par l'algorithme de suggestion (d'interfaces compatibles) et contient les listes des interfaces compatibles de toutes les interfaces. À la première occurrence de compatibilité, l'exécution récursive continue d'interface mère à interfaces enfants jusqu'à arriver aux interfaces basiques. L'intercorrespondance des interfaces basiques consiste à leur affecter un rang pour désigner leur ordre de correspondance entre les interfaces mères.

Lorsque le rang d'une interface est défini, alors son attribut *connected* est mis à *vrai* bien que le *net* de connexion n'ait pas été créé. D'où, par abus de langage, on dit que l'interface est *préconnectée*. Il aurait été possible de créer immédiatement les *nets* une fois que les rangs sont connus, mais l'exécution de l'algorithme de connexion automatique passe au mode de mise en veille. Les *nets* se créent lorsque les rangs et les contraintes sont approuvés par l'utilisateur. L'algorithme de création des *nets* consiste à créer un *net* pour chaque rang. Ensuite, il ajoute les interfaces basiques qui ont le même rang au même *net* créé pour avoir le bus (voir la figure 4.5).

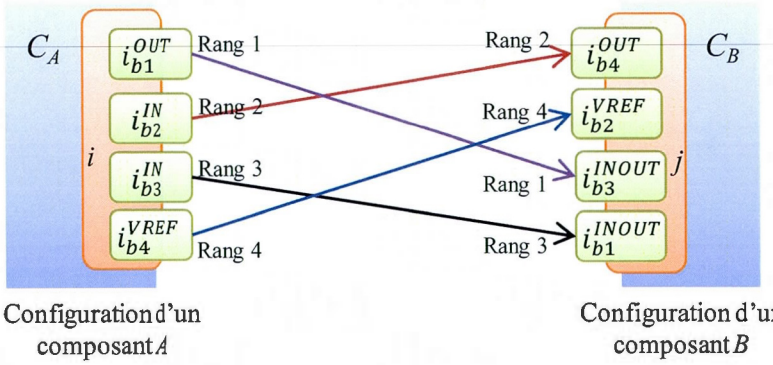


Figure 4.5 Intercorrespondance rangs des interfaces basiques lors de la connexion de $C_A.i$ vers $C_B.j$.

L'ordre des enfants dans une interface intervient seulement lorsqu'un enfant a plusieurs correspondants dans l'autre interface mère. Par exemple, dans la figure 4.5, l'interface OUT de *i* de rang 1 peut se connecter aux interfaces INOUT de troisième et quatrième position dans *j*. Mais comme la troisième vient avant la quatrième dans la liste de *j*, alors la priorité lui est donnée. Alors, INOUT de troisième position de *j* s'affecte le rang 1. Soit, à la première occurrence, l'interface-enfant est préconnectée au premier de la liste des correspondants

d'une même interface mère. Donc, il devient important (lors de la création d'une interface) de sélectionner les interfaces enfants dans un ordre arbitraire pour anticiper les intercorrespondances désirées au cas où un enfant pourrait avoir plusieurs correspondants. Il est à remarquer que plus les contraintes sont définies, alors moins de correspondants une interface aura. Soit, au fur et à mesure l'utilisateur avance dans la conception du *netlist*, alors il devient plus facile de connecter les composants. Les connexions à la figure 4.5 sont obtenues seulement lorsque la contrainte de type est définie et que toutes les autres sont nulles. Par exemple, si l'interface $C_A.i.i_{b1}^{OUT}$ avait une tension de [2,5]V et celle de $C_B.j.i_{b3}^{INOUT}$ est de [3,5]V (c.-à-d. elles ne sont pas compatibles au moins une contrainte), alors $C_A.i.i_{b1}^{OUT}$ se connecterait à $C_B.j.i_{b1}^{INOUT}$ et $C_A.i.i_{b3}^{IN}$ se connecterait à $C_B.j.i_{b3}^{INOUT}$ (même si les valeurs de tension de $C_B.j.i_{b1}^{INOUT}$ et $C_A.i.i_{b3}^{IN}$ sont indéfinies). Si l'utilisateur est déjà arrivé à l'étape de connexion de $C_A.i$ à $C_A.j$, cela veut dire que chaque interface enfant a assurément au moins une correspondance.

4.3 Direction de développement des algorithmes dans la conception de *netlist*

Les prochaines sections présentent les décisions prises pour le développement des algorithmes afin d'aider à la conception de *netlist*.

4.3.1 Développement de l'algorithme de suggestion d'interfaces compatibles

Le choix d'inclure la correspondance de type dans la compatibilité de cardinalité vient du fait que les types sont des informations disponibles dans les interfaces basiques au début de la conception du *netlist*. La contrainte de type est statique comparativement aux autres contraintes qui sont susceptibles de changer lors de la connexion des composants. Puisque l'évaluation de la compatibilité commence par la correspondance des types, il devient rapide de conclure si les interfaces ne sont pas compatibles (car il suffit que le type d'une seule interface n'ait pas un correspondant). Cependant, même si les types se correspondent, la vérification des autres contraintes reste une condition nécessaire pour décider de la compatibilité des interfaces. Autrement dit, tout comme dans la pratique, la compatibilité des types n'est pas assumée vraie tant que les autres contraintes (telles que le voltage) ne sont pas vérifiées.

Il existe des interfaces compatibles et celles dites *potentiellement compatibles*. Deux interfaces sont compatibles lorsqu'elles vérifient les deux critères de compatibilité. Par itération, il faut que les interfaces enfants directes vérifient aussi les deux critères jusqu'aux interfaces basiques. À la fin de l'itération, chaque interface basique doit avoir un correspondant qui lui est compatible électriquement. Deux interfaces sont potentiellement compatibles lorsqu'au moins une interface enfant ne trouve aucun correspondant qui vérifie le critère de cardinalité. Autrement dit, les interfaces mères potentiellement compatibles ont le même nombre d'interfaces basiques, mais pas la même hiérarchie pour les interfaces enfants (en supposant que toutes vérifient la compatibilité des contraintes). Par exemple, dans la figure 4.6, les interfaces i_1={{i_{b1},i_{b2}},{i_{b3},i_{b4}}} et j_1={j_{b1},{j_{b2},j_{b3},j_{b4}}} ont le même nombre d'interfaces basiques et donc auraient pu être connectées. Mais, puisque les interfaces enfants i_1, i_2 et j_1 n'ont pas le même nombre d'enfants, alors i_1 et j_1 devient potentiellement compatibles.

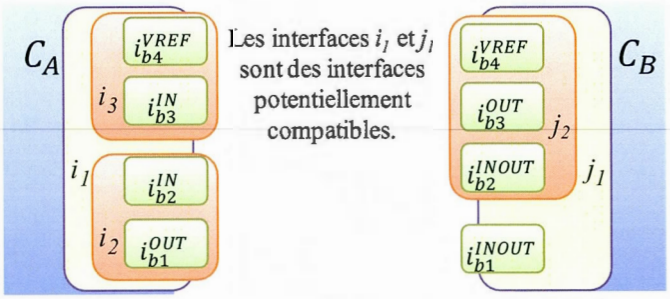


Figure 4.6 Exemple d'interfaces potentiellement compatibles.

La fonction *compatible*() n'est pas récursive en elle-même (contrairement à *compatibleType*()) mais elle est appelée exhaustivement pour chaque interface à tour de rôle avec toutes les autres interfaces (des composants configurés). Si l'exécution de *compatible*() avait été récursive, alors les interfaces enfants devraient être compatibles afin que les interfaces mères le deviennent aussi. Donc, ceci impliquerait que la fonction *compatible*() devrait être vraie pour leurs enfants aussi. Selon le développement suggéré par ce mémoire, lorsque *compatible*(*i*, *j*) retourne *vrai* pour deux interfaces mères *i* et *j*, alors l'exécution de l'algorithme est relancée pour les interfaces enfants par force brute. S'il retourne *faux* pour une interface enfant, alors les interfaces *i* et *j* sont potentiellement compatibles. Si

l'algorithme *compatible*(*i*, *j*) retourne *vrai* pour tous les enfants, alors *i* et *j* sont compatibles. Ce mécanisme est intentionnel, car la détection des interfaces potentiellement compatibles évite que la structure hiérarchique des interfaces ne soit un critère additionnel de compatibilité. En fait, seule la connexion des interfaces basiques est importante. Car, en fin de compte, il s'agit de la connexion des broches. Toutefois, bien qu'elles soient détectées, les interfaces potentiellement compatibles ne peuvent pas être automatiquement connectées. Leur détection sert à informer l'utilisateur de leurs différences structurelles et qu'il peut les connecter manuellement ou les restructurer s'il veut se servir de la connexion automatique. La suggestion additionnelle des interfaces potentiellement compatibles atténue l'impact du choix incorrect des interfaces lors de la création d'une configuration et donc facilite la conception du *netlist*. Néanmoins, il existe des méthodes pour forcer leurs connexions et d'en créer automatiquement de nouvelles interfaces avec les structures correctes[18].

Il est à noter que le nombre des interfaces potentiellement compatibles augmente le nombre d'interfaces à suggérer et donc coute un peu plus de performance dans la recherche exhaustive et dans la mémoire allouée.

4.3.2 Développement de l'algorithme de connexion automatique

L'algorithme *connectInter*() permet à l'utilisateur de connecter (toutefois) les interfaces potentiellement compatibles. Cependant, il devrait exister des interfaces enfants (ou descendants) qui ne seront pas *préconnectées* (*connected=false*) bien que leurs parents soient compatibles et *préconnectés* (*connected=true*). Ceci notifie l'utilisateur, à l'étape de mise en veille, que ces interfaces enfants (ou descendants) ne sont préconnectées à cause de leur structure ou à cause de leurs contraintes électriques (le statut *connected* des interfaces enfants est visible à l'utilisateur). Cette stratégie développée, pour la connexion automatique, détecte les erreurs afin notifier l'utilisateur des problèmes rencontrés (suites à une assignation erronée de contraintes et de choix de structure d'interfaces) et donc une forme assistance à la création de *netlist*s. Si l'algorithme *connectInter*() ne permettait pas une connexion

[18] Par exemple, si une interface *i* est compatible avec une interface *k* mais potentiellement compatible avec *j* (et *i* doit se connecter à *j* et *k*), alors il serait possible de créer automatiquement une nouvelle structure pour *i* qui accepte les structures de *j* et *k* en fusionnant les structures compatibles à *j* et *k* et en appliquant la propriété de partage sur certaines interfaces enfants de *i*.

incomplète des interfaces potentiellement compatibles, alors il reviendrait à l'utilisateur de trouver ses erreurs dans les valeurs des contraintes sans savoir quoi corriger (en considérant qu'une interface pourrait avoir plusieurs dizaines d'interfaces enfants et que chaque interface possède une liste de plusieurs dizaines de contraintes).

4.3.3 Flux de conception de *netlist*

La figure 4.7 montre le flux de conception d'un *netlist* que propose ce mémoire. Dans un premier temps, l'utilisateur décide de connecter une interface *i*. Lorsqu'il la sélectionne, alors une liste d'interfaces compatibles lui est suggérée par le processus de suggestion (d'interfaces compatibles). L'utilisateur en sélectionne une liste d'interfaces à connecter, mais ces interfaces n'ont pas toutes leurs contraintes définies. Certaines pourraient se révéler inutiles ou incohérentes dans une première itération. Ensuite, le module d'intercorrespondance d'interfaces compatibles calcule les rangs des interconnexions puis passe à l'étape de mise en veille. L'utilisateur examine les résultats. Il peut décider de modifier certaines contraintes pour faire une deuxième itération d'interfaces plus compatibles ou faire appel au processus de suggestion de formules de contraintes comportementales. Bien que les *nets* ne soient pas créés, le processus de suggestion de formules suppose que les futures connexions (ou *préconnexions*) seront celles des rangs déjà calculés. Les modèles des formules suggérées dépendent des préconnexions et des interfaces existantes dans le *netlist*. Les valeurs des formules changent en même temps que les valeurs des contraintes de dépendance changent. Si l'utilisateur trouve que les formules suggérées sont très vagues, il peut définir plus de contraintes dans une deuxième itération. Les formules suggérées sont des formules extraites par forage de données. Soit, plus il y a des exemples de formules dans la base de données, plus les modèles de formules deviennent sophistiqués.

Suite à la présentation des algorithmes de suggestion d'interfaces compatibles et de connexion automatique, le deuxième objectif de ce mémoire est atteint. Le chapitre V, présenté dans la section qui suit, introduit de nouveaux concepts qui permettent la suggestion des formules des contraintes comportementales.

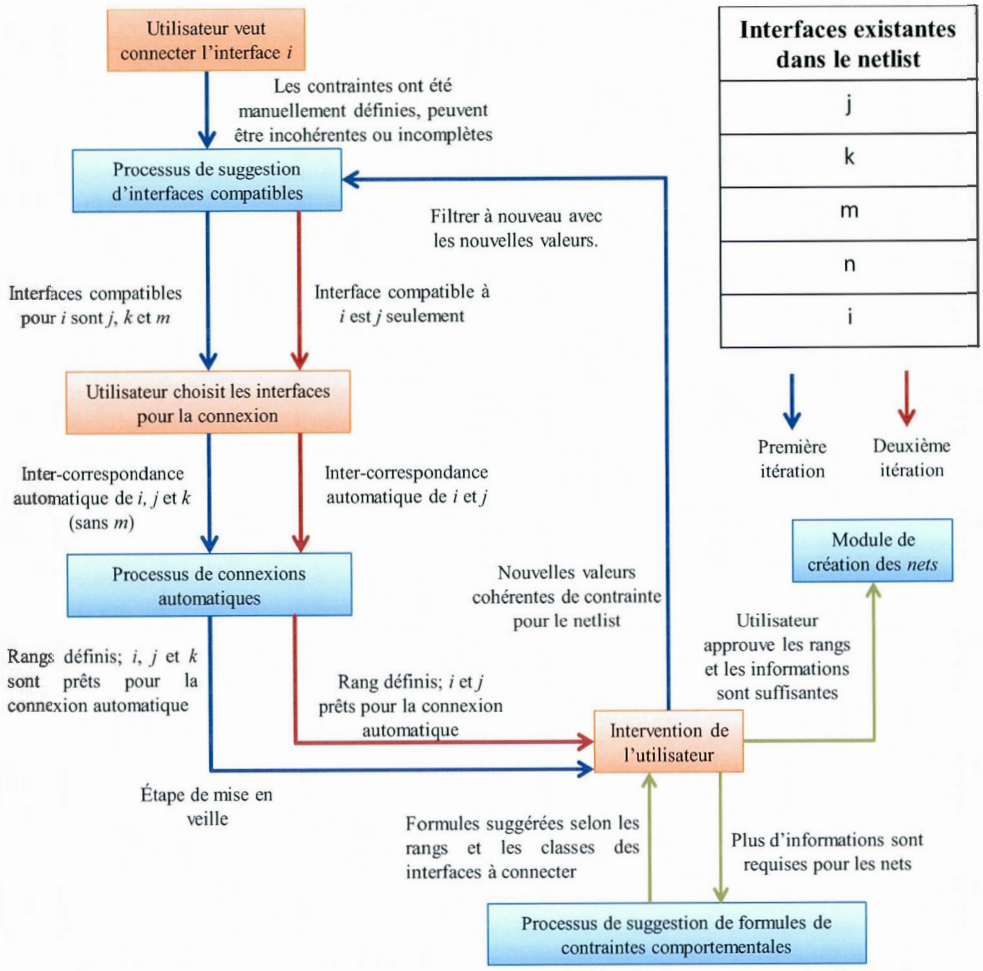


Figure 4.7 Flux de conception d'un *netlist*.

CHAPITRE V

SUGGESTION ET EXTRACTION DES CONTRAINTES COMPORTEMENTALES

Le troisième objectif de ce mémoire consiste à suggérer des formules de contraintes comportementales pour un *netlist* en cours de conception. Dans le chapitre IV, l'algorithme de connexion automatique permet une intercorrespondance entre les interfaces à connecter en définissant les rangs des interfaces basiques. Bien que les *nets* puissent être créés, les informations restent insuffisantes pour que le *netlist* devienne fonctionnel, car des contraintes comportementales doivent être définies. Ce chapitre V propose une méthodologie de suggestion de contraintes comportementales pour un système électronique dans le domaine de conception assistée par ordinateur.

La première section présente les fonctionnalités d'une interface pour la suggestion de formule. Ensuite dans la deuxième section, la représentation considérée des formules est expliquée afin de rendre leurs extractions possibles. Le processus de suggestion de formules de contraintes comportementales est expliqué dans la troisième section. Puis, la quatrième section explique l'usage des interfaces pour la suggestion des formules. La cinquième section présente l'algorithme d'extraction des formules générale et fait une comparaison avec l'algorithme Apriori. La sixième et dernière section fait un aperçu de la méthode proposée par rapport à l'apprentissage automatique et suggère une nouvelle méthodologie pour le raisonnement à base de modèle.

5.1 Interface gabarit – Fonctionnalités d'une interface pour la suggestion de formules

De façon générale, en apprentissage automatique, il est nécessaire d'avoir un certain degré de similitude des caractéristiques d'attributs (telles que qualitatives, quantitatives, de temps, de relations vers d'autres données) entre les objets du domaine afin de faire des

regroupements [13, 19]. Plutôt que de lancer un apprentissage sur toutes les données empiriques en même temps, impliquant le risque d'avoir des résultats qui peuvent se contredire, il est préférable (voir obligatoire) d'effectuer l'apprentissage sur des ensembles de données qui ont des similitudes (ou *classes*) afin d'avoir des résultats cohérents. L'apprentissage se lance sur chaque classe et les résultats obtenus ne concernent que cet ensemble. Ces résultats permettent de prévoir les comportements d'une donnée quelconque appartenant à cette classe. Les résultats sont des règles générales applicables avec une certaine probabilité à toute donnée de la classe. Le regroupement en classes peut être prédéfini, *à priori*, par les experts du domaine ; ou en analysant des données empiriques (existantes) avec certaines méthodes telles que la classification. Les similitudes des caractéristiques des données permettent d'en tirer des attributs qui permettent de classer une nouvelle donnée dans sa classe correspondante. Puisque des règles générales sont applicables à cette classe (grâce à l'apprentissage), alors la nouvelle donnée se comporte très probablement comme les autres données de sa classe. Donc, elle pourrait avoir les mêmes règles générales. Tel est, en général, la technique utilisée pour optimiser l'apprentissage automatique.

Dans ce mémoire, on appelle par *attributs spécifiques* les caractéristiques communes à une classe prises en compte pour la classification.

En référence à la méthodologie mentionnée ci-haut, il devient nécessaire de définir les attributs spécifiques selon les fonctions ciblées (c.-à-d. selon les caractéristiques de la contrainte comportementale du délai) afin de trouver les classes. Cependant, la classification des composants n'est pas suffisante, car il s'agit de la connexion des broches. Autrement dit, on prendra en compte la classification des broches tout en considérant les caractéristiques de leurs composants. Les experts de l'électronique font déjà ce regroupement sans être explicites à son sujet. Par exemple, ils peuvent utiliser un composant tel qu'une mémoire RAM sans être précis sur les détails dans un premier temps ; puis, ils peuvent la remplacer par une autre sans changer (ou en modifiant) très peu les connexions dans le circuit. En outre, une *famille* se définit comme étant un ensemble de composants ayant des caractéristiques similaires au niveau des fonctions globales des broches (ex : la famille des FPGA, des microcontrôleurs,

des RAM, des connecteurs, etc.). Dans ce cas, la famille de composants devient un attribut spécifique additionnel à prendre en compte pour la classification des broches.

Plusieurs possibilités de classifications peuvent être envisagées. Par exemple, si les formules se rapportaient, en plus des familles de composants, à d'autres paramètres (ex : la fréquence, le voltage), alors la classification des broches doit changer relativement. Par exemple, regrouper dans une même classe les composants de même famille qui fonctionne entre 200MHz et 300MHz s'ils utilisent des formules similaires de contraintes comportementales. L'important est que les formules des contraintes comportementales triées par la classification contiennent des données empiriques cohérentes pouvant servir à l'apprentissage. Dans ce mémoire, la classification des broches se limite aux attributs spécifiques de famille et de type, mais elle peut être développée davantage pour inclure d'autres paramètres.

En plus de la classification, la méthodologie proposée crée l'opportunité d'avoir un ensemble de broches (c.-à-d. interface) qui répète les mêmes comportements pour une même classe dans différents *netlists*. Pour s'assurer que des caractéristiques de similitude sont extractibles dans les circuits, une particularité donnée aux interfaces est qu'elles peuvent être utilisées par une même classe de broches. Ceci permet d'examiner les caractéristiques des interfaces dans leur classe. Donc, il devient possible d'effectuer un meilleur apprentissage de formules. Par exemple, lorsqu'une mémoire RAM est remplacée par une autre dans une modification de circuit, alors certaines connexions sont gardées, mais pas d'autres. Ainsi, les interfaces utilisées par les connexions désirées de l'ancienne RAM peuvent être réutilisées par la nouvelle RAM. Ces interfaces sont équivalentes, car elles ont les mêmes fonctionnalités des broches même si leurs contraintes électriques diffèrent légèrement (pourvu que ce changement n'affecte pas les valeurs des attributs spécifiques de la classification). Donc, elles peuvent utiliser des formules similaires de contraintes comportementales.

Une interface utilisable pour une même classe est appelée *interface gabarit*. Plutôt que de recréer la même interface, son gabarit peut être utilisé afin de réduire le temps de création d'interfaces ou de configurations. Dans ce cas, on parle d'*instances de gabarit*[19]. Pour que

[19] Pour la suite du texte, on réfère par interface tout court pour désigner l'instance d'une interface gabarit.

l'utilisation d'une interface gabarit soit possible, il devient nécessaire de la sauvegarder dans une base de données (relationnelle de préférence).

Du fait que deux types d'interfaces existent (c.-à-d. interface basique et interface mère), alors on définit les attributs spécifiques de leurs classes de différente manière. Une *interface basique gabarit* est utilisable par une même classe de broches. Autrement dit, une interface basique gabarit se classe par famille et par type (car le type d'une broche est associé à une interface basique). Le type étant non défini pour une interface mère, alors les attributs spécifiques d'une *interface mère gabarit* sont la famille et la combinaison d'interfaces gabarits qu'elle contient. Par extension de la définition d'une interface, la combinaison des interfaces a une signification fonctionnelle pour les interfaces mères gabarits. Donc, cette combinaison d'interfaces (c.-à-d. la structure hiérarchique) devient un attribut spécifique valide pour sa classe. Suivant la hiérarchie d'une interface mère gabarit, ses interfaces (enfants) gabarits finissent aussi par des interfaces basiques gabarits. Donc, pour instancier une interface mère gabarit, il revient juste à instancier ses interfaces basiques gabarits peu importe le niveau de la hiérarchie (car toutes les interfaces supérieures s'instancient dans le même processus récursif)[20]. Dans l'expression (3.2), la fonction $g'(p)$ de la propriété d'atomicité consiste à identifier l'interface gabarit de p selon ses attributs spécifiques (c.-à-d. la famille du composant et le type de p ; puis, l'interface gabarit correspondante est chargée de la base de données).

Une interface garde toutes les valeurs des contraintes qui lui sont attribuées dans le *netlist* lors de la sauvegarde dans la base de données. Mais, tel n'est pas le cas pour une interface gabarit. Du fait que celle-ci est utilisable par une classe, alors elle ne garde que les valeurs des attributs spécifiques. Les valeurs des autres paramètres sont aussi sujettes à perpétuel changement et donc il n'est pas nécessaire de les archiver. Une interface gabarit doit rester générale à toute sa classe afin qu'elle reste réutilisable. Lors de leurs instances dans une configuration, les interfaces gabarits deviennent plus spécifiques au composant, car elles héritent des valeurs de contraintes des broches de ce composant. Chaque interface gabarit à un numéro d'identification unique dans la base de données.

[20] En utilisant ce processus, on génère automatiquement la configuration basique d'un composant.

5.2 Représentation d'une formule

La contrainte de délai (prise à titre d'exemple dans ce mémoire) se définit par un système d'équations. Chaque équation est une suite de connecteurs, opérations, opérateurs, constantes réelles, valeurs réelles des contraintes de dépendances. Le développement actuel ne prend en compte que les équations et non le système en entier. Dans la suite du texte, on sous-entend par formule une équation du système.

Une contrainte est identifiée de façon unique dans un *netlist* par une série d'identifiants (sID) des objets comme suit :

$$sID = contraintID.interGabaritID.interInstanceID.configID.complnstanceID \quad (5.1)$$

où ID est l'identifiant de (respectivement de la gauche vers la droite) la contrainte (son énumération est unique dans la liste générique des contraintes), l'interface gabarit (unique dans la base de données), l'instance de cette interface gabarit (unique dans sa configuration), la configuration (unique par rapport aux autres configurations du même composant) et l'instance du composant (unique dans le *netlist*)[21].

Conformément à l'expression (2.5), une équation prend la forme :

$$sID_{Di}(= | < | >)(+|-)sID_{x1}\,(+|-|*|/)sID_{x2}\,(+|-|*|/)\ \ldots(+|-|*|/)sID_{xn} \quad (5.2)$$

Par exemple, une équation E d'un système S a une forme $\mathrm{D_i} = \mathrm{X}_{k1} + \mathrm{X}_{k2} * \mathrm{X}_{k3}$ conformément à l'expression (5.2). Supposons que les numéros d'identification des interfaces *i*, *k1*, *k2* et *k3* sont respectivement 21.11.4.2.4, 21.14.1.1.2, 21.42.5.1.9 et 6.31.5.3.7, alors en utilisant l'expression (5.2), on obtient :

$$E : \text{« } 21.11.4.2.4 = 21.14.1.1.2 + 21.42.5.1.9 * 6.31.5.3.7 \text{ »}$$

Pour exprimer les formules générales, l'approche que suggère ce mémoire est la suivante. La formule générale consiste à n'utiliser que les deux premiers identifiants (c.-à-d. *contrainteID* et *interGabaritID*), car les identifiants des instances (c.-à-d. *interInstanceID*, *configID* et

[21] La série d'identifiants est générique et n'a pas besoin d'être entrée manuellement.

compInstanceID) n'ont aucune utilité dans les interfaces gabarits. Les informations importantes sont l'identifiant de la contrainte qui participe dans la formule et l'identifiant de la classe de l'interface utilisée (c.-à-d. l'identifiant du gabarit). Par exemple, la formule générale de E est E': « $21.11 = 21.14 + 21.42 * 6.31$ ». La formule E est dite une instance de la formule générale E'. Les identifiants des instances peuvent varier, mais pas les identifiants de la contrainte ou de la classe. Ainsi, une formule générale peut être instanciée dans différents *netlists* ; mais, et elle garde sa structure de base formée par les identifiants *contrainteID* et *interGabaritID*. Une formule générale devient un modèle dépendant des interfaces gabarits. Les identifiants servent à détecter les contraintes et les instances des interfaces gabarits tels que présentés dans la section suivante.

5.3 Processus de suggestion de formules de contraintes comportementales

Le processus de suggestion de formules prend en entrée une liste d'interfaces préconnectées (ayant leurs rangs définis) et la liste des interfaces existantes dans le *netlist*. On prend pour hypothèse que les interfaces gabarits contiennent des formules générales extraites par forage de données. La sortie du processus est une liste de formules instanciées selon la conception du *netlist*.

Une interface gabarit contient la liste de toutes les formules générales. Pour ne déclencher que les formules intéressantes, la méthode proposée consiste à instancier les formules générales selon les classes des interfaces connectées. Plus précisément, il existe un attribut *typage* dans chaque interface gabarit qui contient une liste de classes (voir la figure 5.2). Chaque classe contient une liste de contraintes. Puis, chaque contrainte possède une liste de formules générales. Par exemple, l'utilisateur veut définir la contrainte de délai d'une interface i de classe G (famille FPGA et type IN) préconnectée à une interface j de classe G' (famille RAM et type OUT). Alors, il sélectionne la contrainte délai puis la liste des formules de la classe G' dans i est instanciée automatiquement (vice-versa pour G dans j). L'utilisateur peut ensuite choisir les instances valides par sa conception.

Pour instancier une formule générale, une recherche exhaustive est faite dans tout le *netlist* afin de trouver toutes les instances des interfaces ayant les mêmes identifiants de gabarits. Dans la figure 5.1, un exemple est donné sur l'instance d'une formule générale dans un *netlist*

où les identifiants des instances sont représentés dans le tableau. L'identifiant de la contrainte du délai est 21 et celle de la fréquence est 6. Les termes de la formule générale contiennent les identifiants des interfaces-gabarits (14, 42 et 31). Ces identifiants servent à détecter les instances ayant la même valeur d'*interGabaritID* dans le tableau. Les *termes applicables* sont suggérés à l'utilisateur qui choisit ceux qui sont convenables pour sa conception. La série d'identifiants est remplacée par la valeur de sa contrainte, ensuite la formule est calculée et le résultat est assigné à la contrainte de l'interface qui a déclenché la formule générale. Les *combinaisons possibles* sont à usage d'exemple pour montrer qu'il est possible de combiner les termes applicables pour avoir plusieurs formules instanciées à partir d'une seule formule générale[22]. Si un identifiant d'un gabarit ne trouve pas d'instance dans le *netlist*, alors sa valeur dans la formule instanciée est remplacée par une valeur neutre. De plus, si les variables d'une formule F sont exprimées par d'autres formules et que les valeurs de celles-ci changent, alors la formule F est automatiquement mise à jour, car elle utilise des identifiants des contraintes et non pas leurs valeurs directes.

La section qui suit présente les étapes d'une interface gabarit lors de la suggestion des formules de contraintes comportementales.

5.4 Usage de l'interface gabarit pour la suggestion des formules

La réutilisation d'une interface gabarit, dans différents *netlists*, permet d'accumuler au fur et à mesure les données nécessaires à l'extraction des formules générales (voir la figure 5.2). Une interface gabarit permet d'identifier les instances qui proviennent d'elle. Toutes les formules utilisées par les instances d'un même gabarit sont regroupées en tant que données empiriques à partir desquelles les formules générales sont extraites. De plus, la tâche principale d'une interface gabarit est de stocker les formules générales. Étant donné qu'une interface gabarit est réutilisable, alors les formules générales qu'elle contient deviennent disponibles à toutes ses instances.

---

[22] Le nombre de combinaisons possibles est le produit des nombres d'instances de chaque terme dans une formule. Par exemple, l'interface gabarit d'ID 14 est instanciée 3 fois, celle d'ID 32 est instanciée 2 fois et celle d'ID 31 est instanciée une fois. Alors, les combinaisons possibles pour avoir des instances de F' sont 3*2*1=6.

Formule générale:
F'= 21.**14**.?.?.? + 21.**42**.?.?.? * 6.**31**.?.?.?

Termes applicables dans le *netlist* N:
21.**14**.3.1.2 21.**42**.2.3.5 6.**31**.5.6.7
21.**14**.3.1.3 21.**42**.2.11.11
21.**14**.2.2.10

Combinaisons possibles des termes applicables pour avoir différentes instances de F':
=21.14.3.1.2 +21.42.2.3.5 * 6.31.5.6.7
=21.14.3.1.2 +21.42.2.11.11* 6.31.5.6.7
=21.14.3.1.3 +21.42.2.3.5 * 6.31.5.6.7
=21.14.3.1.3 +21.42.2.11.11* 6.31.5.6.7
=21.14.2.2.10 +21.42.2.3.5 * 6.31.5.6.7
=21.14.2.2.10 +21.42.2.11.11* 6.31.5.6.7

Des séries d'ID dans un *netlist* N			
interGabaritID	interInstanceID	configID	complnstanceID
12	1	2	1
14	3	1	2
14	3	1	3
25	5	2	4
42	2	3	5
32	3	7	6
31	5	6	7
8	6	4	8
14	2	2	10
42	2	11	11

L'utilisateur choisit les termes applicables (21.14.3.1.2, 21.42.2.11.11, 6.31.5.6.7) pour avoir la formule F= 21.14.3.1.2+ 21.42.2.11.11* 6.31.5.6.7.
La formule F évaluée est F=1.1+0.8*22 = 18.7ns

Les ID 14, 42 et 31 sont des ID d'interfaces gabarits.
Les ID 21 et 6 sont respectivement les ID de délai et de fréquence de l'énumération de la liste générique de contraintes.

Série d'ID	Valeur des contraintes
21.14.3.1.2	0.5 nanoseconde
21.14.3.1.3	1.1 nanoseconde
21.14.2.2.10	2.5 nanoseconde
21.42.2.3.5	1.4 nanoseconde
21.42.2.11.11	0.8 nanoseconde
6.31.5.6.7	22 hertz

Figure 5.1 Exemple de suggestion de formules.

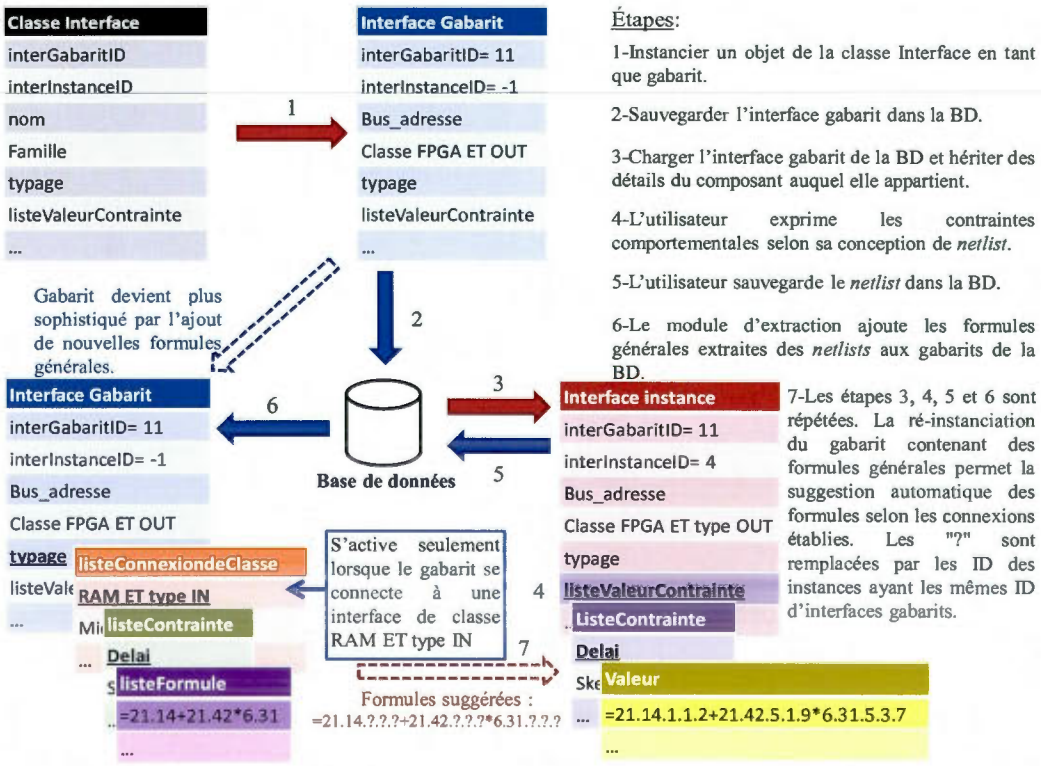


Figure 5.2 Processus de suggestion de formules de contraintes comportementales.

La figure 5.2 illustre le processus de suggestion des formules de contraintes comportementales. Une interface-gabarit est instanciée initialement de la classe Interface (étape 1) et sauvegardée dans la base de données (étape 2). Au besoin, le gabarit est rechargé de la base de données en tant qu'instance dans une configuration (étape 3). L'utilisateur exprime la contrainte de délai de cette instance à l'aide de formules (étape 4) puis la sauvegarde dans la base de données (étape 5). Au fur et à mesure que le gabarit est utilisé, alors il devient plus sophistiqué, car des formules sont ajoutées (étape 6). Lorsque le gabarit est rechargé de la base de données, alors les formules générales peuvent être instanciées (étape 7). L'instance des formules générales dépend des connexions et des interfaces instanciées dans le *netlist*.

5.5 Extraction des formules de contraintes comportementales – Exemple le délai

Le processus d'extraction des formules de délai prend en entrée un groupe filtré de formules empiriques. Ce groupe est obtenu en filtrant la base de données sur deux critères. Le premier critère est l'identifiant de l'interface gabarit i_G qui a instancié les interfaces associées aux formules. Le deuxième critère est l'identifiant de l'interface gabarit qui s'est connectée à i_G lors de la création des formules empiriques. La sortie du processus est une liste des formules générales associées à leur interface gabarit. Les hypothèses considérées sont les suivantes :

- Les contraintes des formules sont représentées par leurs séries d'identification telle que l'expression (5.2).
- Les identifiants des instances sont enlevés (c.-à-d. *interInstanceID*, *configID* et *compInstanceID*).
- Les formules sont arrangées dans un ordre lexicographique sans modification des valeurs des termes (c.-à-d. prendre en compte les propriétés de l'addition, la soustraction, la multiplication et la division).

L'approche proposée consiste à représenter les formules par des chaines de caractères et un arbre syntaxique abstrait (voir la figure 5.3). De cette façon, les formules peuvent être représentées en tant que motifs séquentiels [10, 12] afin de les extraire à l'aide d'une méthode telle que l'algorithme Apriori. Chaque terme devient une liste de chaines de caractères et la formule devient une liste de termes. Chaque terme débute par une opération +

ou – et se termine au niveau du dernier caractère (inclus) avant le début d'un autre terme ou à la fin de la chaine. Exception faite pour la première série d'identifiants de la formule (c.-à-d. l'interface qui utilise cette formule) et le connecteur, chacun étant une liste de chaines de caractères.

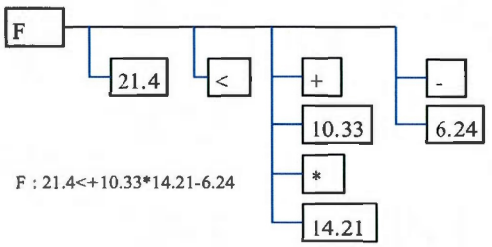


Figure 5.3 Exemple de représentation d'une formule de délai dans un arbre syntaxique abstrait.

5.5.1 Recherche de séquences larges

Une séquence s se définit comme une liste ordonnée d'éléments e_j ($1<j<n$) notée $s=\langle e_1,e_2,e_3,\ldots e_n\rangle$ où e_i est une collection d'un ou plusieurs évènements i_k ($1<k<m$) tel que $e_j=\{i_1,i_2,\ldots,i_m\}$ [9]. Soit une séquence est caractérisée par le nombre d'éléments et le nombre d'évènements qu'elle contient. Une séquence s est contenue dans une séquence q lorsque chaque élément de s est inclus dans un élément de q en suivant le même ordre d'apparition. Par exemple, une personne P_1 qui achète des produits a et d le premier jour, puis b et c le deuxième jour et enfin c et d le troisième jour. Ces transactions effectuées par la personne P_1 sont notées par $s_1=\langle\{a,d\},\{b,c\},\{c,d\}\rangle$. Cette séquence s_1 est différente de $s'=\langle\{b,c\},\{a,d\},\{c,d\}\rangle$ car l'ordre chronologique des achats (un facteur important) est différent. À supposer qu'une personne P_2 effectue des achats de produits comme suit : d le premier jour, c et d le deuxième jour ; soit $s_2=\langle\{d\},\{c,d\}\rangle$. La séquence s_2 se retrouve dans la séquence s_1 car tout élément de s_2 se retrouve chronologiquement dans s_1 (s_2 est aussi incluse dans s'). Une séquence qui se répète un nombre de fois supérieur à la valeur de support dans un ensemble de séquences est dite fréquente (ou motif fréquent). Soit pour chaque inclusion d'une séquence s dans une séquence q, le support de s est incrémenté par 1. Une séquence fréquente est dite large lorsqu'il n'existe pas une séquence fréquente qui la contient. Ainsi, la recherche des motifs fréquents consiste donc à trouver les séquences larges. Cependant, la

comparaison des séquences avec elles-mêmes n'est pas suffisante pour trouver les séquences larges, car une telle approche n'assure pas la couverture totale des possibilités (la base de données ne contient pas tous les cas possibles). Pour cette raison, une approche consiste à générer, par niveau, toutes les combinaisons de séquences possibles dites candidates, puis à chercher leur support dans la base de données. Seules les séquences candidates fréquentes seront prises en compte.

5.5.2 Principe de l'algorithme Apriori

Le principe de l'algorithme Apriori stipule que « si un *itemset*[23] est fréquent, alors tous ses sous-ensembles doivent être aussi fréquents » [10]. Un itemset est dit fréquent lorsqu'il a un support dépassant un seuil minimal *minsup*. Le comptage du support d'un itemset consiste à trouver le nombre de fois que cet itemset se retrouve dans un ensemble d'autres itemsets. Le comptage σ du support s permet de calculer le support d'un itemset en divisant σ par le nombre total d'itemsets N (ce qui signifie que le support s est compris dans un intervalle [0, 1]). Par exemple, on veut trouver le comptage du support du candidat $\{a,c,d\}$ dans les itemsets $\{a,b,c,d\}$, $\{a,c,d,f\}$, $\{a,b,c,d,e\}$ et $\{c,d,e\}$ de la base de données. On vérifie que $\{a,c,d\} \subset \{a,b,c,d\}$, $\{a,c,d\} \subset \{a,c,d,f\}$ et $\{a,c,d\} \subset \{a,b,c,d,e\}$ mais $\{a,c,d\} \not\subset \{c,d,e\}$. Pour chaque inclusion, on incrémente le comptage du support du candidat $\{a,c,d\}$ de +1 et donc son support devient 3. Si le seuil minimal *minsup* est de 2 alors $\{a,c,d\}$ devient un itemset fréquent. Le principe de l'algorithme Apriori indique que si $\{a,c,d\}$ est fréquent alors tous ses sous-ensembles doivent être fréquents, c'est-à-dire que les supports de $\{a\}$, $\{c\}$, $\{d\}$, $\{a,c\}$, $\{a,d\}$ et $\{c,d\}$ sont supérieurs à *minsup* car leurs supports sont supérieurs ou égaux à celui de $\{a,c,d\}$. D'autre part, si un itemset n'est pas fréquent, alors tout itemset qui le contient n'est pas fréquent non plus. Un itemset contenant k items est noté k-itemset. De cela, la génération des candidats se fait par niveau L où le premier niveau L_1 est l'ensemble des singletons de 1-itemsets (un niveau est l'ensemble des itemsets de même taille k ; soit, L_1 contient tous les 1-itemsets, L_2 contient les 2-itemsets…).

L'opération la plus couteuse dans l'exécution de l'algorithme est la comparaison d'un k-itemset candidat avec tous les k-itemsets de la base de données pour trouver son support.

[23] *Itemset* est équivalent à un élément d'une séquence.

Afin d'améliorer la performance d'Apriori, on s'intéresse à générer les candidats qui seraient à priori fréquents. On est certain que les candidats contenant des itemsets non fréquents sont inutiles. Autrement dit, on génère des k+1-itemsets candidats obtenus par la fusion de deux k-itemsets fréquents. Pour générer tous les candidats à priori fréquents de niveau L_{k+1} de façon optimale, une règle de correspondance consiste à vérifier si les k-1 premiers items d'un k-itemset *i* fréquent sont identiques aux k-1 premiers items d'un autre k-itemset *j* fréquent. S'ils sont égaux alors le dernier item de *j* est ajouté à la position k+1 de *i*. Le nouveau k+1-itemset obtenu est un candidat à priori fréquent dont le support sera testé avec le support minimal seuil. S'il est fréquent, alors il est ajouté à L_{k+1} pour continuer la suite de la recherche ; sinon, il n'est pas utile pour la suite et est donc annulé. Ainsi, chaque itemset fréquent est comparé avec tous les autres itemsets fréquents du même niveau pour générer tous les candidats du niveau supérieur. Ce processus récursif s'arrête au plus haut niveau lorsqu'il n'y a plus d'itemsets fréquents pour la génération des candidats à priori fréquents. Par exemple, dans la figure 5.4, si $i=\{\underline{a},\underline{b},c\}$ et $j=\{\underline{a},\underline{b},d\}$ sont fréquents, alors on trouve qu'ils ont en commun la séquence $\{\underline{a},\underline{b}\}$ (dans cet ordre chronologique). Donc, un nouveau itemset candidat $\{a,b,c,d\}$ est créé en ajoutant à la fin $i=\{\underline{a},\underline{b},c\}$ le dernier élément *d* de *j*. L'itemset candidat $\{a,b,c,d\}$ doit avoir son support évalué afin qui soit considéré un itemset fréquent. Si tel est le cas, alors il est ajouté au niveau L_4.

Dans la figure 5.4, si un itemset est fréquent alors tous ses sous-ensembles sont fréquents. Si un itemset est non fréquent, alors tous les ensembles qui le contiennent sont non-fréquents. Quant aux itemsets larges, ils sont $\{\underline{\boldsymbol{a,b,c,d}}\}$ et $\{\underline{\boldsymbol{c,e}}\}$, car ils ne sont contenus dans aucun autre itemset fréquent.

L'approche proposée dans ce mémoire applique le même principe sur les formules, mais avec les changements appropriés. Ces changements sont présentés dans la section qui suit.

5.5.3 Application d'Apriori sur les concepts proposés

L'application d'Apriori sur les concepts proposés nécessite différentes stratégies d'extraction des formules fréquentes. La raison est que les formules en tant que séquences n'ont pas les mêmes caractéristiques des itemsets utilisés par Apriori car les items peuvent se répéter dans les itemsets.

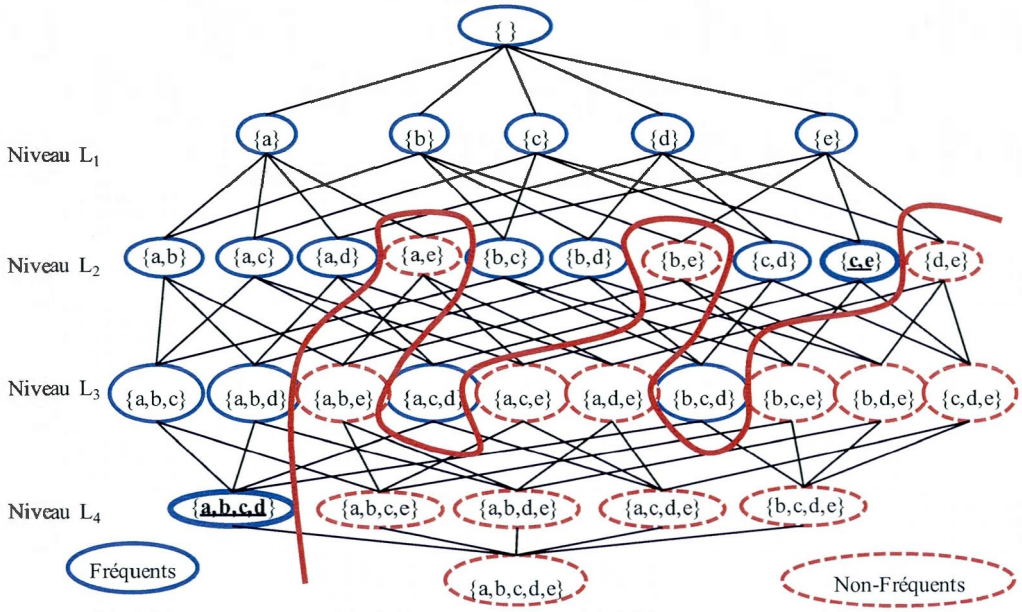


Figure 5.4 Exemple du principe de l'algorithme Apriori.

Pour mieux illustrer la problématique des formules, la figure 5.5 donne un exemple de treillis des formules. On peut avoir une infinité de niveaux tant qu'il y a des itemsets fréquents. Autrement dit, le niveau de la borne supérieure est inconnu même avec un seul itemset (ex : une formule telle que a+a*a/a-a…+a). Cette différence, par rapport à l'approche classique d'Apriori, vient du fait qu'un même terme peut être répété plusieurs fois dans une formule. L'exposant k dans a^k signifie le nombre de fois que a se trouve dans la formule.

De même dans les termes, on peut avoir un nombre inconnu d'items (ex : +a/a*…*a). Si l'on considère aussi les termes comme étant des séquences et les séries d'identifiants et d'opérateurs comme itemsets, alors on obtient un treillis pour les termes de la même forme que la figure 5.5.

Autrement dit, les niveaux du treillis pour les formules générales recherchées prennent des dimensions dépendantes du niveau maximal du treillis des termes multiplié par le niveau maximal du treillis des formules. À cette difficulté s'ajoute la génération des candidats pour les différents niveaux. Elle consiste à prendre en compte la structure lexicographique des formules sans modifier les opérations et opérateurs dans les formules.

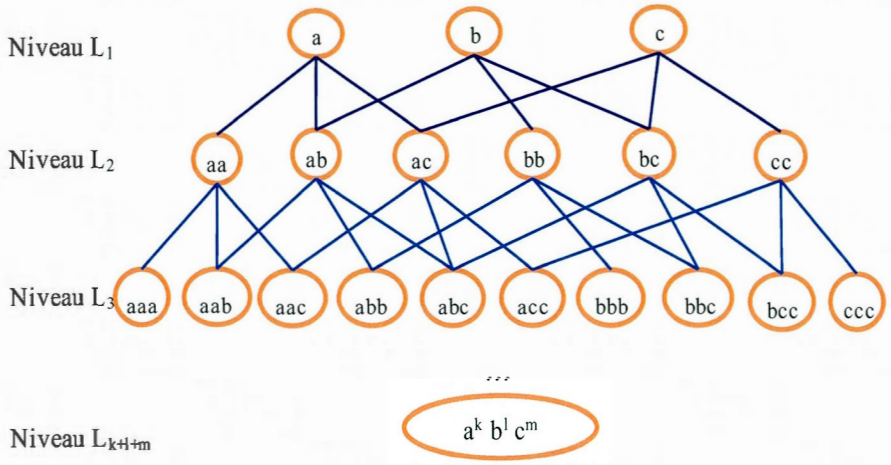


Figure 5.5 Exemple de treillis de formules avec trois termes.

Dans la figure 5.5, par exemple, si *a* est un terme +4.1*5.22 et *b* un terme -3.5/6.8, alors la génération de *aab* au troisième niveau signifie que la formules *F* est +4.1*5.22 +4.1*5.22 -3.5/6.8. Il est à remarquer que lorsque le niveau augmente, alors ses itemsets augmentent aussi. Soit, il y une infinité de niveaux (à la différence de l'approche usuelle d'Apriori où les niveaux sont finis). Dans l'approche usuelle d'Apriori, un item ne peut apparaitre qu'une seule fois dans un itemset, mais tel n'est pas le cas avec les formules. Par exemple, le terme +2.11+2.11 est un itemset candidat valide. Alors que dans l'approche usuelle, l'itemset {a,a} ne l'est pas. Ceci a un impact sur la génération des candidats, car il faut prendre en compte toutes les combinaisons légitimement possibles.

La solution proposée consiste à exécuter l'algorithme Apriori une fois sur les termes en tant que séquences et une fois sur les formules en tant que séquences. Cependant, l'exécution d'Apriori sur les termes (en tant que séquences) ne retourne pas les itemsets larges, mais plutôt tous les itemsets fréquents de tous les niveaux. Cet ensemble de termes fréquents sera considéré le premier niveau L_1 pour l'algorithme Apriori des formules (en tant que séquences). On veut trouver tous les termes fréquents, car une formule de taille 1-itemset contient un terme au niveau L_1. Ensuite, on trouve les formules fréquentes par niveau pour enfin en déduire les formules larges recherchées. Ces formules larges sont les formules générales recherchées pour les interfaces gabarits.

Afin d'améliorer la performance des algorithmes de comparaisons, on fait usage de la réduction de la dimensionnalité des termes. Ceci permet de réduire le nombre de candidats par niveau. Les termes sont accompagnés par une matrice M_{m*n} de supports où m est un nombre énuméré de connecteurs (c.-à-d. m = 3) et n un nombre énuméré de signes (c.-à-d. n = 2) comme suit :

$$M_{3*2} = \begin{bmatrix} & = & < & > \\ + & 0 & 0 & 0 \\ - & 0 & 0 & 0 \end{bmatrix} \tag{5.3}$$

Par exemple, un itemset 3.11 supporté par la matrice $\begin{bmatrix} 2 & 1 & 10 \\ 0 & 8 & 3 \end{bmatrix}$ devient les formules >+3.11 et <-3.11 si le support minimal est 5. Il est à noter que le signe de la division fait partie intégrante de la série d'identifiants, mais tel n'est pas le cas avec +, - et *. Donc, lorsque les termes sont comparés, il faudra prendre en compte ces détails.

L'algorithme d'extraction des termes fréquents se trouve à la figure 5.6 et celui des formules larges à la figure 5.7. L'algorithme d'extraction des termes fréquents prend en entrée le groupe filtré de formules empiriques (par interface gabarit de provenant et interface gabarit de connexion). Du fait que toutes les formules proviennent d'une même interface gabarit, alors la série d'identifiants à gauche du connecteur est la même pour toutes les formules. Raison pour laquelle, l'identifiant de l'interface gabarit (à gauche du connecteur) ne participe pas à l'extraction des termes et des formules. La sortie de l'algorithme d'extraction de termes fréquents est la liste des termes fréquents supportés par leurs matrices de support. Cette matrice permet de les reconstruire en des formules ayant un seul terme. Seules les formules ayant le support supérieur au seuil minimal sont reconstruites. Les formules obtenues sont de niveau L_1. Autrement dit, c'est l'entrée de l'algorithme d'extraction des formules larges. L'algorithme d'extraction des formules larges extrait les formules larges (ou générales). Ensuite, les formules générales sont sauvegardées dans les interfaces gabarits suivant les deux critères qui ont filtré les formules empiriques. C'est-à-dire, elles sont sauvegardées dans le typage de l'interface gabarit qui est associée à l'identifiant du côté gauche du connecteur des formules générales.

La section qui suit discute la méthodologie de suggestion des contraintes comportementales que propose ce mémoire par rapport à sa classification en apprentissage automatique.

Algorithme d'extraction des termes fréquents

```
1:  //Une formule est une liste de termes, terme est une liste de chaînes de caractères,
2:  //formulaDBList contient les formules de la base de données, minsup est le support minimal
3:  //La fonction findFrequentTerms() return tous les termes fréquents dans la base de données
4:  List< Term> findFrequentTerms (List<Formula> formulaDBList, int minsup)
5:  Begin
6:      List<Term> L1=générer le premier niveau à partir des series d'ID des interfaces basiques
7:
8:      //trouver les termes fréquents du premier niveau
9:      findFrequentTrms(L1, formulaDBList, minsup)
10:
11:     //ajouter les termes fréquents trouvés à la liste totale
12:     List< List <Term>>allLevel.append(L1)
13:
14:     //Générer les termes candidats du niveau suivant
15:     //Remarque : le support seuil est envoyé en paramètre car chaque terme fréquent contient une
16:     // matrice de supports dont certains pourraient être inférieurs à minsup. Lorsque des termes
17:     // vérifient la correspondance de k-1 premiers items , il faut vérifier aussi que les supports de
18:     //même coordonnées dans les matrices soient tous deux supérieurs au seuil minimal avant de
19:     //générer le terme candidat.
20:     List<Term> nextLvl = genTrmCandid(L1, minsup)
21:
22:     //trouver les termes fréquent du niveau suivant en éliminant les infréquents
23:     findFrequentTrm (nextLvl, formulaDBList, minsup)
24:
25:     //Exécuter tant que le niveau suivant n'est pas vide
26:     while(nextLvl is not null)
27:         allLevel.append(nextLvl)
28:         nextLvl = genTrmCandid (nextLvl, minsup)
29:         findFrequentTrm (nextLvl, formulaDBList, minsup)
30:     end while
31:
32:     //Copier tous les niveaux dans une seule liste de termes
33:     List<Term> singleList = returnSingleList(allLevel)
34:
35:     //Retourner tous les termes fréquents trouvés
36:     return singleList
37: end findFrequentTerms
```

Figure 5.6 Algorithme d'extraction des termes fréquents.

5.6 Aperçue de la méthodologie proposée par rapport à l'apprentissage automatique

Cette section analyse les caractéristiques de la méthodologie de suggestion de contraintes comportementales par rapport aux autres méthodes de l'apprentissage automatique afin de mieux évaluer sa classification.

Algorithme d'extraction des formules larges

```
1:  //Une Formule est une liste de Terme, Terme est une liste de chaînes de caractères, formulaDBList
2:  //contient les formules de la BD, minsup est le support minimal
3:  //La fonction findLargeFormula () return toutes les formules larges
4:  List<Formula> findLargeFormula (List<Formula> formulaDBList, int minsup)
5:  Begin
6:      //Appeler la fonction findFrequentTerms() pour trouver tous les termes fréquents
7:      List<Term> totalFrequentTermList= findFrequentTerms(formulaDBList, minsup)
8:
9:      // Restaurer la forme originale des termes à partir de la matrice pour avoir le premier niveau.
10:     //Remarque : seuls les termes fréquents sont retourné par restaure()
11:     List<Formula> L1= restaure(totalFrequentTermList)
12:
13:     //Ajouter les formules du premier niveau à la liste totale des formules fréquentes.
14:     //Remarque : les formules du premier niveau sont fréquentes
15:     List< Formula>allFrequentFormulas.append(L1)
16:
17:     //Assigner le premier niveau en tant que niveau suivant pour régler la condition de la boucle
18:     List<Formula> nextLvl = L1
19:
20:     //Exécuter tant que le niveau suivant n'est pas vide
21:     while(nextLvl is not null)
22:         //Générer la liste des formules candidates à partir des formules du niveau actuel et du niveau 1
23:         nextLvl = genFmlCandid(L1, nextLvl);
24:
25:         //Trouver les formules fréquentes du niveau suivant en éliminant les infréquentes
26:         findFrequentFml(nextLvl, formulaDBList, minsup)
27:
28:         //Ajouter la liste des formules fréquentes trouvées à la liste totale
29:         allFrequentFormulas.append(nextLvl)
30:     end While
31:
33:     List<Formula> largeFormula          //Liste totale des formules larges
34:
35:     //Vérifier si chaque formule dans la liste des formules fréquentes est contenue dans une autre.
36:     //Si aucune formule fréquente contient fmlIsLarge alors fmlIsLarge est une formule large
38:     For each Formula fmlIsLarge in allFrequentFormulas
39:         bool isNotLarge= false
40:         For each Formula tempFml in allFrequentFormulas
41              If (fmlIsLarge is included in tempFml) true
42                  isNotLarge =true
43                  break inner For loop
44              end If
45          end For
46
47          //Si fmlIsLarge est large alors ajouter à la liste totale des formules larges
48          If(isLarge is false) then largeFormula.append(fmlIsLarge)
49      end For
50      //Retourner tous les formules larges trouvées
51      return largeFormula
52  End
```

Figure 5.7 Algorithme d'extraction des formules larges.

Dans l'introduction, ce mémoire a couvert une partie de cette analyse en disant que la méthodologie proposée pourrait se classer dans le raisonnement à base de modèle. En effet, la méthodologie proposée trouve des modèles de formules. Mais le raisonnement à base de modèle n'a pas un algorithme particulier qui identifie les modèles. Il est souvent euristique puis demande une profonde connaissance de la structure des modèles afin de les utiliser.

En ce qui concerne la création du *netlist*, ce mémoire suggère que la méthodologie proposée, à ce point de développement, est une méthode d'apprentissage semi-automatique, car elle demande des interventions de la part de l'utilisateur. En fait, tel que la méthodologie est développée, l'interaction avec l'utilisateur est intentionnelle, car il doit décider de la conception du circuit après chaque étape automatisée. Par exemple, si l'on enlève l'étape de mise en veille entre la suggestion d'interfaces compatibles et la création automatique des *nets*, on aurait eu le cas d'utilisation suivant. Le logiciel INC simule dans un premier temps les préconnexions de toutes les interfaces compatibles. Puis, il instancie les formules générales des contraintes comportementales qui ont le meilleur support afin de déterminer le reste des variables inconnues du *netlist*. Après plusieurs itérations, les préconnexions deviennent des connexions. Cependant, le point central de cette section concerne la suggestion des contraintes comportementales et non la création du *netlist*.

En ce qui concerne la suggestion des contraintes comportementales, à ce point de développement, la méthodologie proposée permet en effet d'apprendre des formules de contraintes comportementales telles que l'illustre la figure 5.8. Tout comme les autres méthodes d'apprentissage (arbre de décision, raisonnement à base de cas, etc.), l'ajout de nouvelles données empiriques implique un développement plus sophistiqué des modèles de formules. De même, la mise à jour de la base de connaissances (c.-à-d. des modèles de formules) peut être déclenchée automatiquement après que la base de données augmente d'un certain pourcentage de données empiriques. En regardant de cette perspective, on peut déduire que ce mémoire introduit une nouvelle approche en apprentissage automatique qui peut se généraliser au raisonnement à base de modèle. Ce mémoire se garde de dire que la méthodologie proposée pourrait être le premier algorithme général pour le raisonnement à base de modèle (qui, jusque-là, n'avait pas une méthodologie générique de développement). Mais, on peut déclarer que si la méthodologie proposée est applicable dans un domaine aussi

compliqué que l'électronique, alors elle peut être applicable pour des problèmes de complexité similaire.

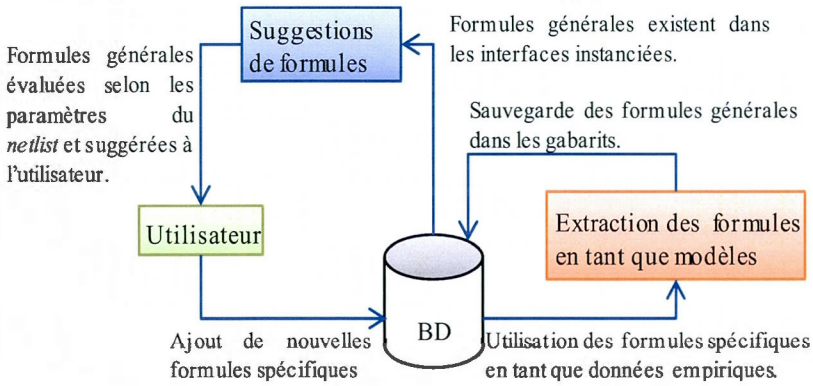


Figure 5.8 Cycle de l'apprentissage de modèles de formules

Dans la figure 5.9, l'algorithme général de la méthodologie proposée consiste à utiliser une technique d'extraction de motifs fréquents pour avoir des modèles. Au début, les classes (en matière de génie logiciel telles que la classe Interface) permettent de générer les instances qui sont les données empiriques. D'un autre côté, les spécifications des actions ou des fonctions recherchées (tel que l'intérêt de trouver les formules de la contrainte du délai) permettent de regrouper les données empiriques par classification ou *clusterisation*. Les classes (ou *clusters*) obtenues sont représentées par des instances spéciales dites gabarits auxquels on associe des identifiants uniques. Toute action de l'utilisateur est enregistrée sous forme d'expressions dans les instances (données empiriques). Les expressions sont une suite des identifiants des classes (ou des *clusters*). Ensuite, l'extraction des expressions régulières d'identifiants (ou modèles) est appliquée sur chaque classe (ou *cluster*). Les identifiants font référence à des gabarits d'objets et permettent d'identifier les instances correspondantes. Les modèles extraits sont sauvegardés dans leurs gabarits respectifs. L'usage d'un gabarit doit s'étendre seulement sur sa classe. Un nouveau cas possède des attributs qui identifient sa classe et donc son gabarit. Lorsque l'utilisateur cherche la solution du nouveau cas, alors les modèles (contenus dans le gabarit) sont déclenchés suite à une action ou à une détection de valeurs ciblées. Puis, les modèles évalués sur les valeurs des instances présentes grâce aux caractéristiques de la problématique dans son environnement en temps réel.

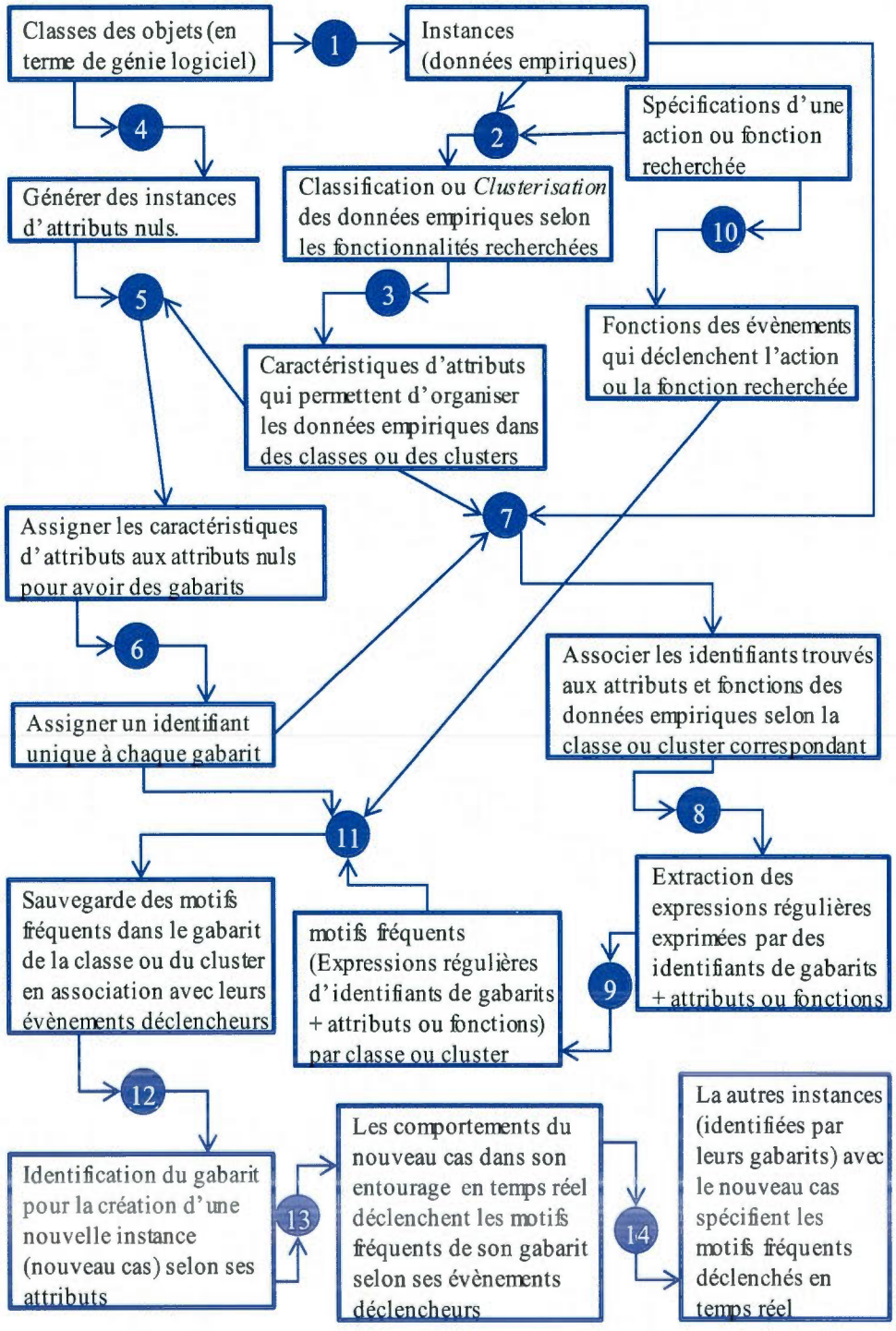


Figure 5.9 Méthodologie de développement et usage des modèles.

Quant au développement du logiciel, les modules sont conceptualisés relativement aux modèles à extraire afin que le logiciel devienne capable d'apprendre certaines tâches. Cette conception peut être étendue vers une automatisation complète et optimale du logiciel afin qu'il devienne complètement indépendant. Dans le cas de INC, sa version complète serait qu'il soit capable de construire des systèmes électroniques de lui-même. Par exemple, dans INC, il est possible (avec la méthodologie proposée) d'inclure des *modèles de graphes* (trouvés avec la même méthodologie des modèles de formules) comme déclencheurs de formules. Ils identifieraient le réseau de connexion des composants du circuit en cours afin d'instancier les formules générales plus correctes. Dans le cas actuel, on se contente d'instancier une formule juste pour un graphe à un arc (c.-à-d. le déclenchement de la formule a lieu lorsqu'un composant est connecté à un autre). Le travail futur prévoit une identification de réseau de composants connectés grâce à des modèles de graphe.

On peut soutenir que si la méthodologie proposée est applicable dans un domaine aussi compliqué que l'électronique, alors elle peut être adaptée pour des problèmes de complexité similaire. Par exemple, les jeux de simulation peuvent utiliser des objets gabarits qui contiennent des expressions régulières composées de séries d'identifiants. Les séries d'identifiants représentent les actions entrées par l'utilisateur à des moments donnés T à la suite d'un évènement E. Les expressions régulières sont extraites en considérant le temps T et l'évènement E. Lors de la simulation, lorsque l'utilisateur rencontre l'évènement E, alors le simulateur anticipe l'action de l'utilisateur au temps T en instanciant les séries d'identifiants qui font références aux classes d'objets disponibles dans l'environnement E. L'anticipation de l'action consiste par exemple, à contrer l'action usuelle ou à supporter la décision de l'utilisateur.

À la suite des nouvelles approches proposées dans ce chapitre, le troisième objectif du mémoire a couvert une partie de la problématique des contraintes comportementales. Ces approches peuvent être développées davantage pour améliorer la suggestion des formules de contraintes comportementales. Les expérimentations et les résultats obtenus sont discutés dans le chapitre suivant.

CHAPITRE VI

EXPÉRIMENTATIONS ET RÉSULTATS

Ce chapitre présente les expérimentations et les résultats obtenus par des jeux de tests avec INC. Des tests sont effectués pour montrer la validité des concepts et la méthodologie de suggestion de formules de contraintes comportementales. Dans l'appendice A, un exemple pratique de *netlist* est a été effectué dans les interfaces utilisateurs graphiques en même temps que INC est présenté.

Pour les tests, plusieurs *netlists* réels ont été reproduits afin de valider les concepts et leurs propriétés. En particulier, les tests sur le *netlist* « *Bottom PCB* » du WaferBoard™ révèlent que les concepts peuvent effectivement aider à la conception. Le *netlist* « *Bottom PCB* » du WaferBoard™ contient plusieurs dizaines de composants électroniques. Pour le *netlist* de test, sept composants importants ont été utilisés : un microcontrôleur (voir la figure 6.1), deux *I2C Mux* (voir la figure 6.2), deux *Fan Tray* (voir la figure 6.3), un *Power Supply* (voir la figure 6.4) et la banque No.2 d'un *FPGA* (voir la figure 6.5). Le *netlist* obtenu est montré dans la figure 6.6. Un groupe de connexions (plusieurs dizaines) de broches a été créé avec INC en un seul clic de souris.

La taille du fichier (produit par INC) de l'exemple pratique de la figure A.1 est d'environ 6.5 Mégaoctets (Mo) et celle de la figure 6.6 est d'environ 30Mo. La taille d'un fichier de *netlist* augmente en fonction des connexions créées et des configurations existantes dans les composants. En fait, chaque objet composant contient la liste totale de ses configurations ; donc, lorsque les configurations deviennent plus structurées, alors la taille du fichier augmente.

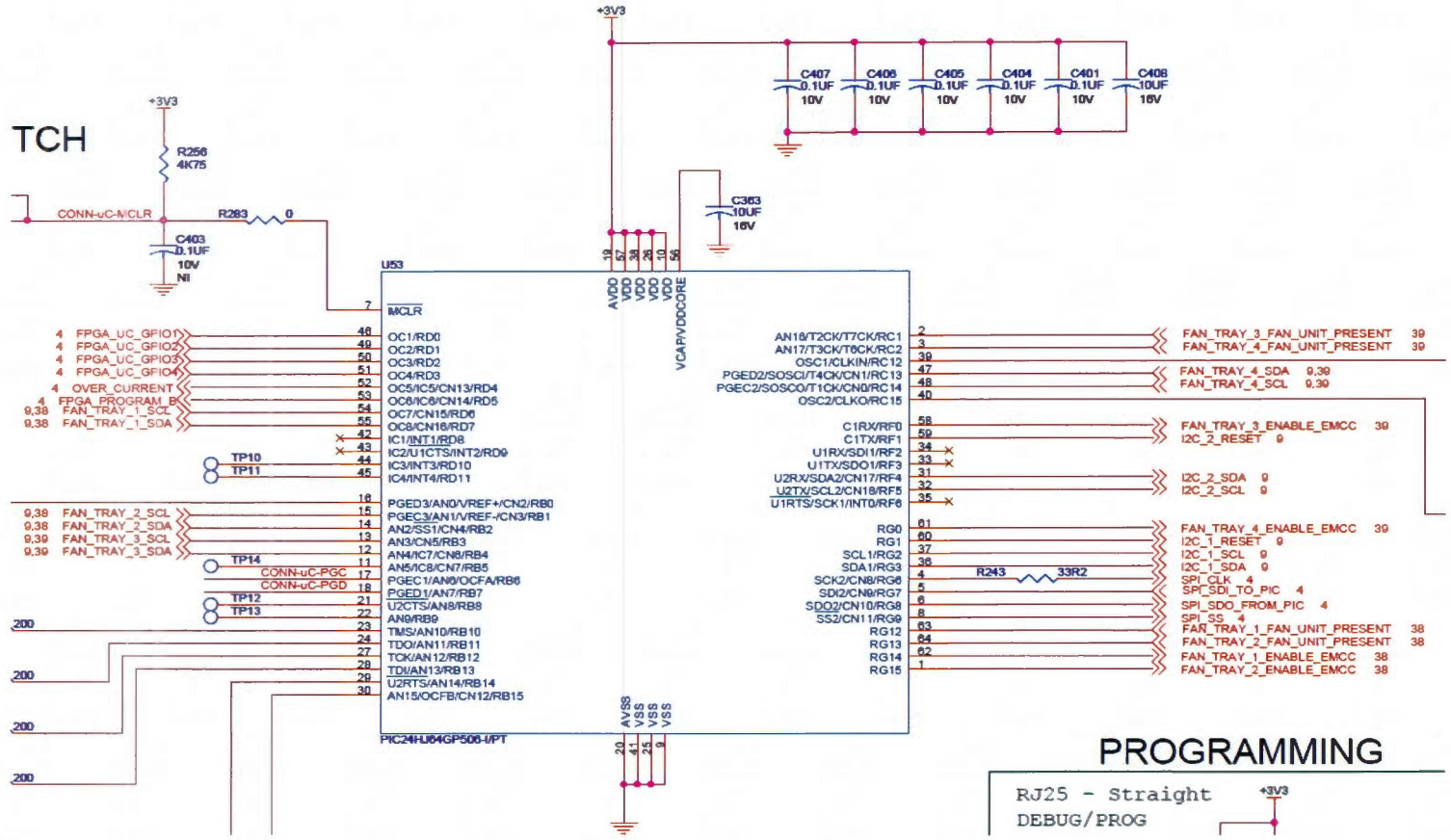


Figure 6.1 Microcontrôleur d'un *netlist* de test.

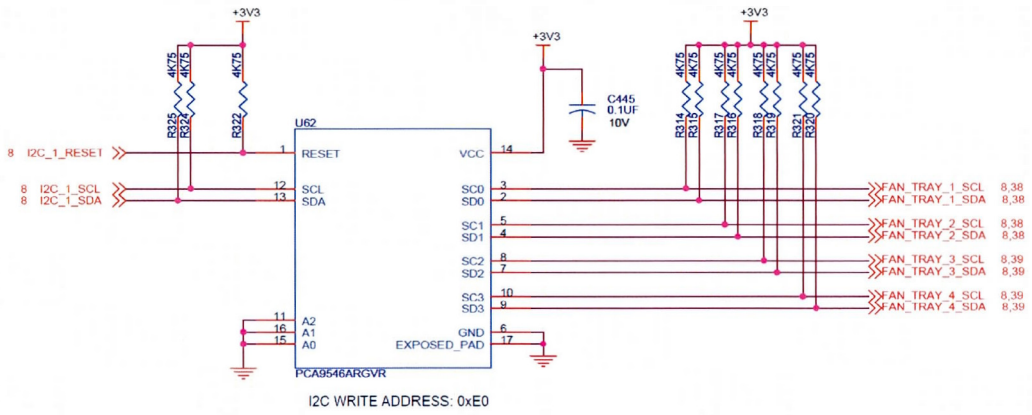


Figure 6.2 Composant *I2C Mux* d'un *netlist* de test.

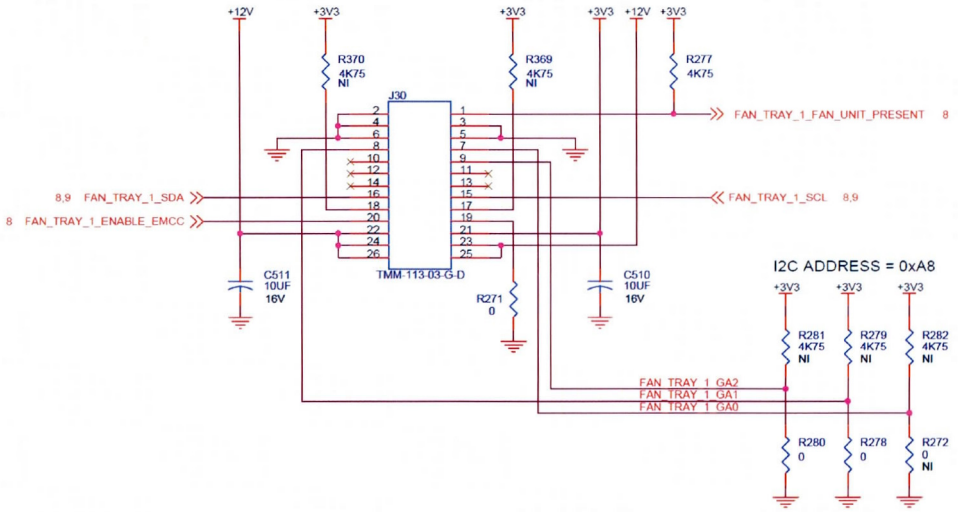


Figure 6.3 Connecteurs *Fan Tray* d'un *netlist* de test.

Dans les deux *netlists* de test, chaque composant avait au moins deux configurations, mais les grands composants tels que le FPGA avait quatre configurations (car ils sont le point central de plusieurs tests). Le fait de sauvegarder un composant avec toutes ses configurations dans un *netlist* donne l'avantage à l'utilisateur de réutiliser ce composant et ses configurations dans d'autres *netlists*. L'inconvénient est que la sauvegarde de toutes les configurations requière plus de mémoire. Il est mieux de supprimer, si possibles, les configurations non nécessaires.

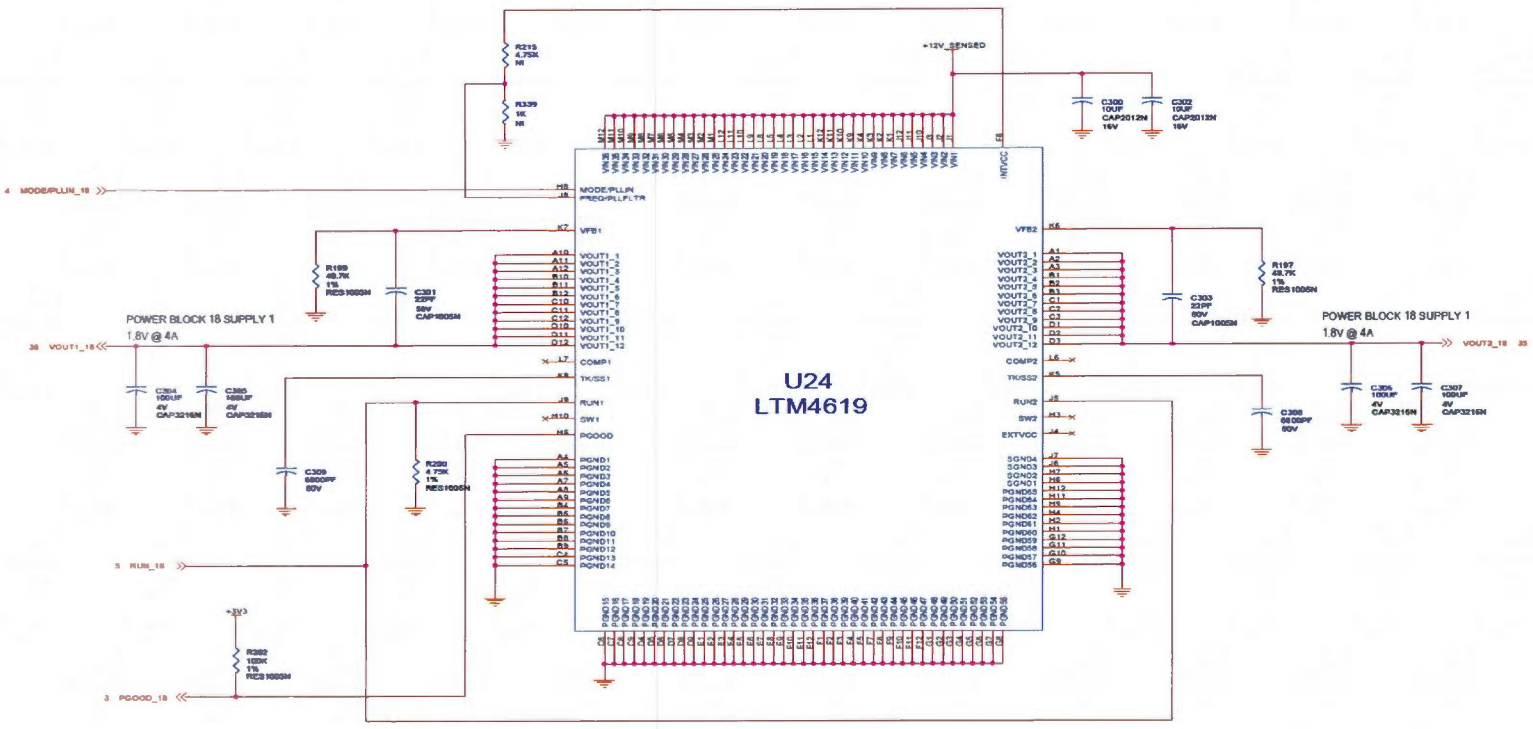


Figure 6.4 Composant *Power Supply* d'un *netlist* de test.

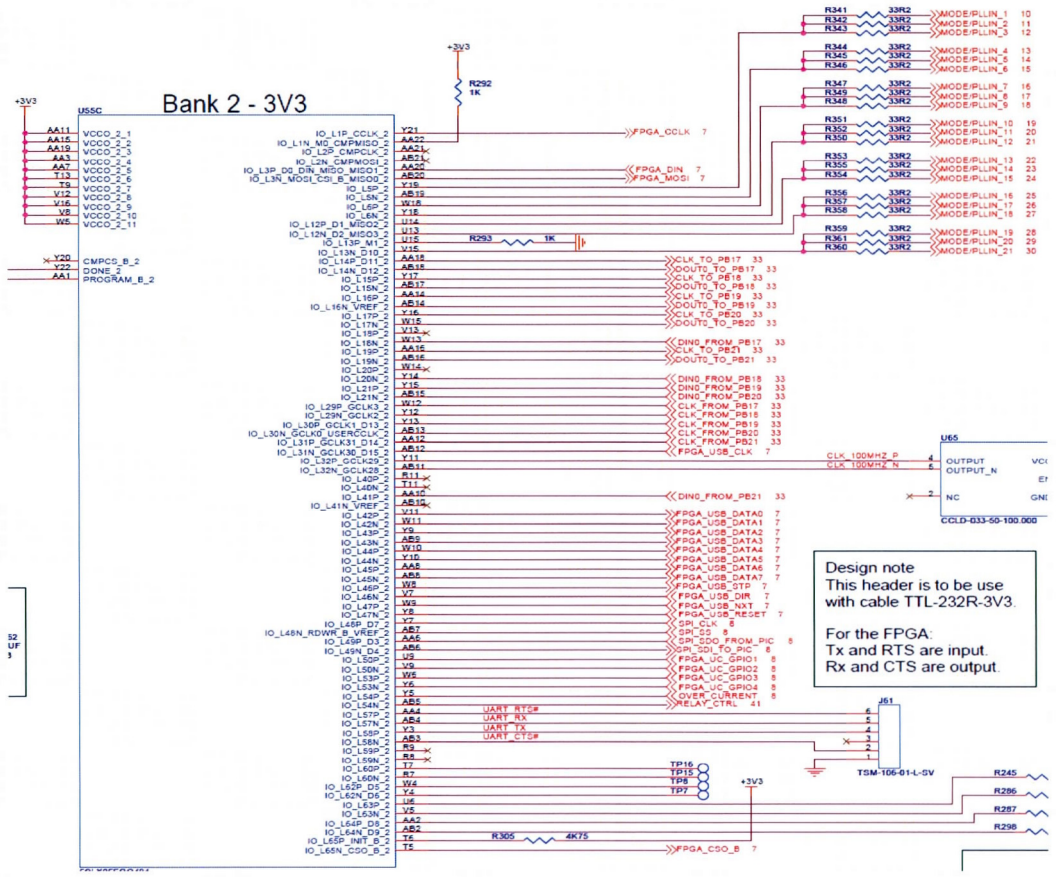


Figure 6.5 Bank No 2 d'un FPGA d'un *netlist* de test.

Afin de s'assurer que les concepts de configurations et interfaces sont valides par ingénierie inverse, des fichiers de *netlists* de format EDIF[24] ont été effectivement convertis au modèle de INC. L'exemple d'un *netlist* de trois composants connectés est montré à la figure 6.7. Le *netlist*, présenté dans la figure A.2, a été converti en configuration et interfaces connectées. Les configurations qui peuvent être obtenues sont les configurations basiques et une seule version de configuration avancée. Le processus de création des configurations avancées consiste à regrouper les interfaces basiques qui se connectent à un même composant dans une interface mère.

[24] Le format EDIF 2.0 a été pris en exemple, car c'est le format le plus standard pour la description de *netlist*. Le point de ce développement est de prouver qu'il est possible de *parser* des *netlists* existants en des structures de *netlists* basées sur les configurations et interfaces telles que proposées dans ce mémoire.

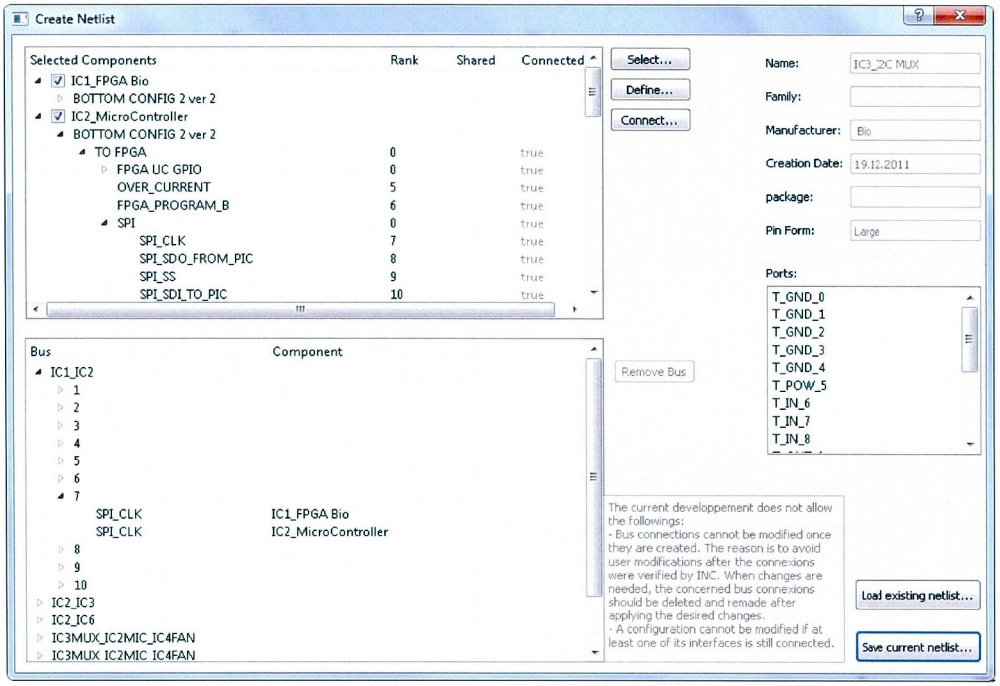


Figure 6.6 *Netlist* « *Bottom PCB* » du Waferboard™ faite avec INC.

Toutefois, il existe des techniques pour avoir des interfaces plus développées. Par exemple, trouver dans la base de données toutes les interfaces mères qui possèdent le même nombre d'interfaces basiques que l'interface mère obtenue (ceci fait partir du développement futur). La conversion des fichiers n'a pas été développée plus en profondeur (bien qu'il soit important), car il sort de la portée de l'objectif de ce mémoire.

Quant à l'extraction des formules du délai, le format *EDIF* ne contient malheureusement aucune information sur les contraintes comportementales. Donc, ce point reste à être appliqué sur des *netlists* qui contiennent les informations nécessaires et qui proviennent de projets réels. D'un autre côté, il est difficile de trouver de tels fichiers, car ce sont des fichiers avancés dans la conception ; donc, ils portent des droits d'auteurs. Pour ces raisons, une base de données de formules aléatoires a été générée en respectant la formulation de l'expression (5.2). Suite à la déduction établie par l'expression (5.2), les formules de la contrainte du délai provenant de projets réels auraient une formulation structurellement similaire aux formules générées. Ce qui assure que les formules générées sont représentatives

aux cas réels. À l'aide de la base de données simulée, les tests qui portent sur l'extraction des formules sont devenus possibles. Les tests qui simulent la base de données et l'extraction des formules sont présentés dans la figure 6.8.

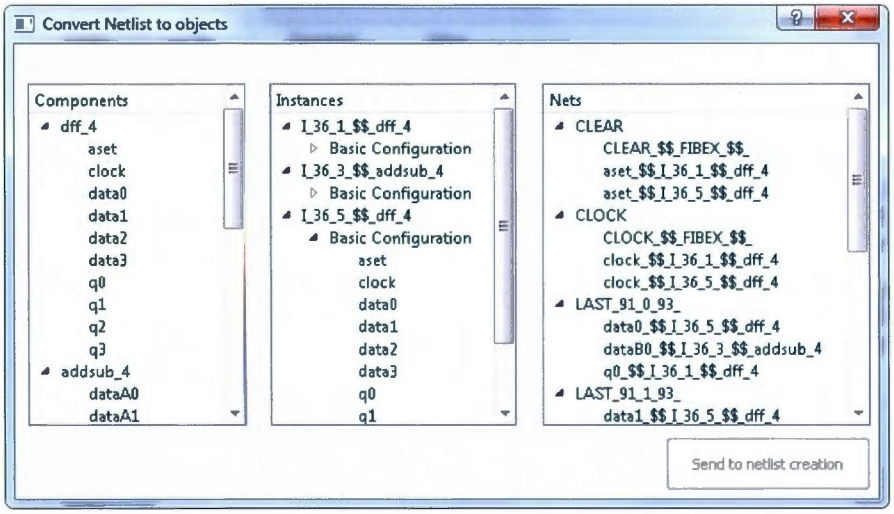


Figure 6.7 Conversion du *netlist* de format EDIF de la figure A.2 en configuration et interfaces dans INC.

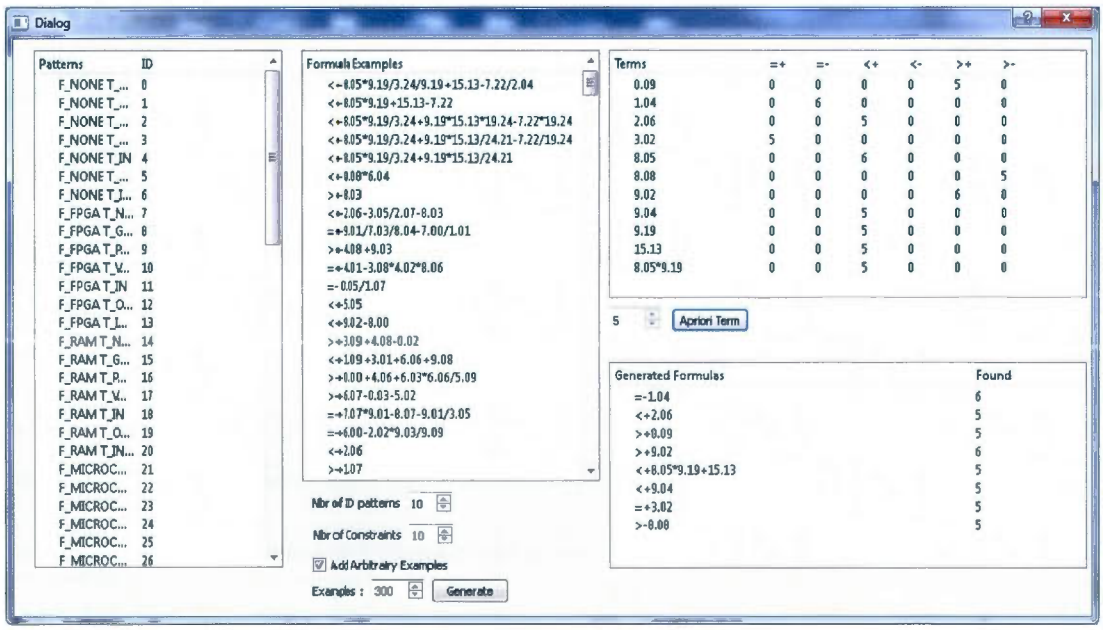


Figure 6.8 Simulation de la base de données et de l'extraction des formules.

Dans la figure 6.8, 300 exemples ont été générés pour simuler les données empiriques dans une base de données. La génération des formules consistait à utiliser 10 interfaces gabarits et 10 contraintes afin que les combinaisons des séries d'identifiants se retrouvent assez fréquemment. On suppose que ces formules proviennent d'une même interface gabarit. On suppose aussi que toutes ces formules ont été utilisées lors de la connexion du gabarit à une classe de composants (c.-à-d. on prend un groupe de formules filtrées tel que requis par l'algorithme d'extraction). Les formules générées peuvent contenir les connecteurs (=, < et >), les opérations (+ et -) et les opérateurs (* et /) avec 5 séries d'identifiants au maximum (les mêmes séries peuvent revenir dans une même formule). Cinq formules arbitraires (en tant que données empiriques) ont aussi été ajoutées au début de la liste des formules simulées.

Tous les termes fréquents de tous les niveaux sont trouvés dans la figure 6.8. Ensuite, les termes fréquents sont reconstruits pour devenir des 1-itemsets formules. Les 1-itemsets formules permettent la suite de l'extraction des formules larges. Les formules larges obtenues sont sauvegardées dans le typage de leur interface gabarit correspondant.

La figure 6.9 montre un exemple de suggestion d'une formule générale. Lorsqu'une interface *i* se connecte à une interface *j*, alors les formules générales contenues dans *i* sont suggérées à la suite de l'identification de la classe de *j*.

Une simulation de suggestion de la formule de la figure 6.9 est présentée à la figure 6.10. Dans la figure 6.10, les composants existant dans le circuit sont trois FPGA *A*, *B* et *C* où *A* est connecté à *B* mais pas *C*. Les composants *A*, *B* et *C* sélectionnés sont identiques et utilisent aussi des configurations identiques. On veut tester intentionnellement un exemple où l'on a plusieurs duplications d'identifiants de gabarits. On veut aussi tester le comportement des mêmes interfaces gabarits existant dans un même *netlist*. On peut remarquer que, pour chaque terme de la formule, trois instances d'une même interface gabarit (d'identifiant 88 et d'identifiant 89) sont suggérées, mais avec des valeurs différentes. La formule générale est contenue dans l'interface de *A*. Elle a été suggérée avant la création des *nets*, car les rangs ont préconnectés les interfaces. Donc, la formule générale se déclenche et les valeurs des contraintes des interfaces existantes sont suggérées tel que prévu.

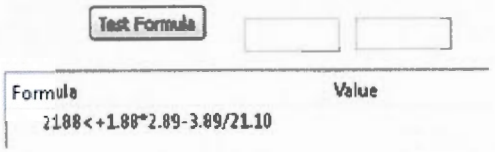


Figure 6.9 Exemple de suggestion d'une formule générale.

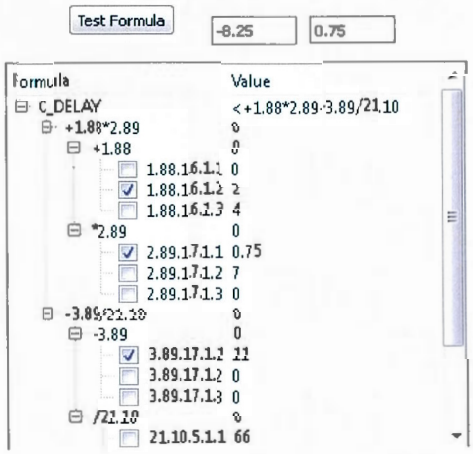


Figure 6.10 Exemple d'instanciation de la formule de la figure 6.9.

Les configurations basiques sont automatiquement générées lorsque l'utilisateur indique la famille du composant et le nombre de types des broches. Les gabarits correspondants sont identifiés et instanciés pour hériter des autres valeurs des broches. L'utilisateur peut se servir des configurations basiques pour créer un *netlist* sans avoir besoin de créer des interfaces mères. La suggestion des contraintes comportementale ne nécessite pas l'existence de hiérarchie des interfaces. Autrement dit, les méthodologies de suggestion de compatibilité et de suggestion de formules présentées dans ce mémoire pouvaient se passer des concepts des interfaces. On aurait pu appliquer ces méthodologies seulement au niveau des broches. Du fait que les interfaces mères ne sont pas obligatoires, alors on pouvait aussi se passer des configurations et les propriétés à appliquer. Mais dans ce cas, l'utilisateur doit connecter toutes les broches une par une à celles qui sont compatibles. Ce qui n'est pas pratique.

L'usage des interfaces mères permet des connexions automatiques d'un groupe de plusieurs dizaines de broches pour créer un bus telles que les tests l'ont montré. Dans l'exemple

pratique de la figure A.1, la connexion du processeur U2 à la RAM U15 se fait en un clic de souris. Donc, le nombre d'interfaces à connecter n'est plus important une fois que les composants sont configurés. On peut noter aussi que ces configurations sont portables d'un utilisateur à un autre et d'un projet à un autre. Cela simplifie les tâches de création de configurations.

L'usage des interfaces mères permet aussi d'affecter des valeurs par héritage. Les tests ont montré que l'utilisateur est assisté lors de l'affectation des valeurs de contraintes aux interfaces. Par exemple, l'affectation d'une norme standard des signaux (tel que HSTL) se fait en un clic de souris. Toutes les interfaces basiques prennent les valeurs attribuées de la norme standard.

Les formules suggérées peuvent effectivement reproduire les valeurs appropriées du circuit (mais dans une certaine limite). Ceci est devenu possible, car la suggestion des formules a pris en compte la connexion des composants comme évènement déclencheur. Cependant, les résultats de la suggestion sont dépendants de la classification considérée. Par exemple, si les broches des composants ont plusieurs attributs spécifiques de classification, alors les formules deviennent plus précises, car les interfaces gabarits deviennent plus spécifique un groupe de composants. Les interfaces gabarits rencontrées durant la conception du *netlist* deviennent nombreuses. Donc, l'instanciation des formules n'est plus vague, car elle vise des groupes précis de composants. Dans le cas contraire, si la classification regroupe beaucoup de catégories de composants par classe, alors les formules deviennent vagues. Les interfaces gabarits seront instanciées pour tout composant de la classe, peu importe son type. Autrement dit, la performance de la méthodologie dépend de la classification des composants.

Le logiciel INC est toujours en voie de développement pour une éventuelle production. Le chapitre suivant présente la conclusion du mémoire.

CHAPITRE VII

CONCLUSION

Ce mémoire a présenté une méthodologie de création intelligente de *netlist*s qui assiste le concepteur dans la conception de système électronique. Trois objectifs ont été fixés :

- Objectif 1 – Simplification des composants.
- Objectif 2 – Connexion automatique.
- Objectif 3 – Suggestion de formules de contraintes comportementales.

Pour le premier objectif, ce mémoire a suggéré une nouvelle méthodologie de représentation de modèle de composants. Des concepts et des propriétés formels ont été proposés afin de garder une représentation fidèle aux composants et à la conception du *netlist*. La méthodologie proposée consiste à représenter un composant par une configuration. Une configuration est un ensemble d'interfaces. Une interface est une structure hiérarchique d'autres interfaces. Cette représentation arborescente est finie par des feuilles appelées interfaces basiques. À chaque interface basique est associée une broche. Soit, toute interface est structurée à la base d'interfaces basiques. Le regroupement des interfaces basiques en une interface mère nécessite que ce groupe ait une connectivité fonctionnelle assumée vraie par l'utilisateur. Autrement dit, la structure de la configuration est dépendante de l'intention de conception du *netlist*. Cinq propriétés ont été formellement définies : l'atomicité, l'existence, l'équivalence, la fermeture et le partage. Ces propriétés doivent être respectées obligatoirement lors de la création des interfaces et des configurations. La représentation hiérarchique des broches simplifie la vue du composant, car l'utilisateur peut voir un composant par ses connectivités fonctionnelles sans les détails des broches. L'utilisateur se sert des connectivités fonctionnelles pour créer le *netlist*.

Pour le deuxième objectif, ce mémoire propose deux algorithmes : la suggestion d'interfaces compatibles et la connexion automatique des interfaces compatibles. L'algorithme de suggestion d'interfaces compatibles consiste à trouver par force brute toutes les interfaces connectables des composants du *netlist*. La compatibilité des interfaces prend en compte les des contraintes électriques et leurs structures hiérarchiques. Les interfaces compatibles sont assurées d'être connectables. À chaque interface du *netlist* est associée sa liste d'interfaces compatibles. Lorsque l'utilisateur veut connecter une interface alors la liste des interfaces compatibles lui est présentée. La liste sélectionnée des interfaces compatibles (à connecter) passe à l'algorithme de connexion automatique. L'algorithme de connexion automatique se divise en deux modules : la préconnexion des interfaces et la connexion des interfaces. La préconnexion consiste à définir un rang de correspondance entre les interfaces basiques selon leurs compatibilités électriques. Une étape dite de mise en veille permet à l'utilisateur d'apporter des modifications aux préconnexions ou aux valeurs des contraintes. Cette étape permet aussi de lancer une itération de réglage des contraintes. Une fois approuvés, les rangs sont utilisés pour créer automatiquement les *nets*. Cette tâche est faite par le module de connexion automatique qui consiste à ajouter au même *net* toutes les interfaces basiques de même rang. Si deux interfaces mères sont compatibles, alors elles sont connectées automatiquement, peu importe le nombre de leurs interfaces basiques. Des connexions de plusieurs dizaines de broches peuvent être créées en un clic de souris.

Pour le troisième objectif, la contrainte comportementale prise à titre d'exemple est le délai. Le délai s'exprime par un système d'équations. Ce mémoire propose une nouvelle méthodologie de raisonnement (à base de modèle) qui se base sur des gabarits d'interfaces et des modèles de formules mathématiques. La méthodologie proposée consiste à classer en premier lieu les broches selon leur usage similaire des contraintes comportementales. Ensuite, les interfaces gabarits sont associées aux classes des broches obtenues. Une équation du système qui exprime la contrainte du délai contient un ensemble d'opérations et d'identifiants associés aux valeurs des contraintes. Ces contraintes peuvent provenir de n'importe quelle interface du *netlist*. Pour trouver les formules générales, ce mémoire propose de transformer les expressions mathématiques en des motifs séquentiels exprimés par des séries d'opérations et d'identifiants (d'interfaces gabarits) afin d'en extraire les motifs fréquents. L'extraction

des formules générales est exécutée par un algorithme à la base de l'algorithme Apriori. Il consiste notamment à trouver tous les termes fréquents dans une première étape, ensuite toutes les formules larges dans une deuxième étape. Une formule étant une suite séquentielle de termes. Les formules générales sont exprimées par des connecteurs (=, <, >), des opérations (+, -), des opérateurs (*, /) et des identifiants d'interfaces gabarits. Les formules générales sont ensuite sauvegardées dans les interfaces gabarits correspondantes à leurs classes.

Pour suggérer des formules de contraintes, les formules générales sont instanciées selon la conception du circuit. L'instanciation d'une formule générale consiste à trouver toutes les interfaces existantes dans le *netlist* suivant deux critères. Le premier critère requiert que les interfaces recherchées aient leur identifiant d'interface gabarit présent au moins une fois dans la formule générale. Le deuxième critère d'instanciation est la préconnexion de l'interface qui contient la formule à une classe d'interfaces. Les interfaces trouvées ont leurs valeurs de contraintes suggérées à l'utilisateur. L'utilisateur choisit les valeurs appropriées des contraintes selon la conception du *netlist*. La suggestion dépend de la présence des composants dans le *netlist*. La suggestion des formules nécessite un module d'extraction qui s'applique à une base de données contenant des formules de contraintes comportementales. Les formules générales deviennent plus précises dans le temps, car elles sont mises à jour au fur et à mesure que la base de données augmente.

Au meilleur de notre connaissance, la méthodologie proposée dans ce mémoire est la première dans le domaine de conception assistée par ordinateur de *netlist* qui soit basée sur l'apprentissage automatique. Elle aide l'utilisateur à créer un *netlist* de système grâce à un raisonnement à base de modèle. Un logiciel, nommé *Intelligent Netlist Creator* (*INC*), a été implémenté pour valider la méthodologie proposée (INC est présenté en appendice A).

Les concepts ont été implémentés dans INC suivant plusieurs patrons de conception tels que la fabrique abstraite (*Abstract Factory Pattern*), le singleton, le contrôleur. Les propriétés et les remarques des concepts sont exécutées automatiquement par INC à l'insu de l'utilisateur. L'utilisateur n'est pas obligé (mais il est recommandé) qu'il comprenne les propriétés de configuration et d'interface.

Le logiciel *Intelligent Netlist Creator*, réalisé dans le cadre de ce mémoire et pour le projet DreamWafer™, applique la méthodologie proposée. L'usage des configurations et des interfaces facilite la création d'un *netlist*. Le logiciel élaboré a été intégré dans le projet DreamWafer™. Cependant, le module de suggestion de contraintes comportementales n'a pas été introduit dans INC mais a été appliqué sur des bancs d'essai typiques. Les causes de détachement du processus (de suggestion de formules) sont l'absence des données empiriques pour valider l'approche et les nombreux détails de développement à considérer (tels que la vérification des systèmes d'équations et la résolution des blocages de dépendance de contraintes). Toutefois, les tests ont montré que la faisabilité de l'apprentissage automatique des formules peut être appliquée à la méthodologie proposée.

Le développement fait dans le cadre de la recherche n'est qu'une preuve de concepts et plusieurs améliorations sont à apporter ; à savoir :

- Dans le processus de suggestion d'interfaces compatibles, la compatibilité des interfaces a été réduite à l'égalité des Min et Max. Cependant, l'implémentation doit être plus formelle et conforme à la pratique de connexion de broches (sous la direction des experts). Chaque contrainte doit avoir une liste de fonctions formelles de compatibilité entre deux interfaces.
- Les attributs spécifiques de la classification considérée des broches dans la suggestion de formules sont la famille de composants et la contrainte de type. Si des tests sur des données empiriques réels montrent que les classes sont très vagues, alors il faut nécessairement redéfinir les attributs spécifiques.
- Le logiciel INC peut être étendu jusqu'à définir des contraintes physiques (pour le désign du *layout*). En effet, il est possible d'automatiser un type de connexion de réseau tel qu'un réseau étoilé en donnant aux connexions des conditions supplémentaires avec des coordonnées physiques. Par exemple, de la liste des interfaces compatibles, connecter chaque interface à cinq de ses interfaces compatibles pour avoir une forme étoilée sous la condition qu'elles soient des interfaces partagées, distantes entre 5mm et 7 mm et ayant un voltage de 1.5Volt.
- Il est possible de substituer une des configurations d'un composant à une autre en plein désign du circuit. Mais le développement actuel ne permet pas une telle action,

car il y a des stratégies de rémodélisation d'interfaces mères à considérer (telles que l'annulation des interfaces basiques mères créées par la propriété d'équivalence). Ce développement dépend des stratégies de conception futures ou même d'une modification des propriétés des concepts établis (si c'est nécessaire).

- Dans l'expression d'une formule de contrainte comportementale, l'utilisation des fonctions complexes (telles que sinus, somme algébrique...) requiert que la représentation d'une formule utilise des graphes directionnels. Ce genre de problématique fait partir de l'ordre de développement futur. Diverses méthodes de représentation de formule peuvent impliquer relativement différentes approches d'apprentissage.
- Un *netlist* peut être transformé en un composant ayant une configuration et des interfaces telles que des circuits intégrés ou des systèmes sur puces (*system-on-a-chip*). Pour cela, certaines interfaces seront déclarées externes et visibles et l'ensemble des composants et des interfaces restantes devient invisible et inaccessible. Dans ce cas, un *netlist* devient telle qu'une *configuration interconnectée* qui a ses interfaces dépendantes les unes des autres.

Et enfin, on peut conclure ce chapitre en indiquant que la méthodologie proposée d'apprentissage à base de raisonnement de modèle dans ce mémoire peut être généralisée pour traiter d'autres problèmes similaires à la création de *netlists*. Ces problèmes peuvent toucher à plusieurs domaines où le comportement d'un tiers partie (c.à.d. un utilisateur ou un autre système) est pris en compte tels que les logiciels de conception en général, les jeux de simulation, l'intelligence artificielle pour des robots indépendants, le traitement de langage naturel. Pourvu que la tâche en question s'exprime par des expressions régulières et pas forcément par des formules mathématiques. Il s'agit notamment de faire une classification de gabarits d'objets (en classes) et associer ces gabarits à des identifiants et à des déclencheurs d'évènements correspondants. L'extraction d'expressions régulières d'identifiants permet d'avoir des instances relatives à l'environnement d'application en temps réel. Si cette approche est applicable dans un domaine aussi complexe que l'électronique, alors elle peut servir dans d'autres domaines de complexité similaire. Le logiciel *Intelligent Netlist Creator* est toujours en voie de développement pour une éventuelle production.

APPENDICE A

PRÉSENTATION DU LOGICIEL *INTELLIGENT NETLIST CREATOR*

À partir des concepts établis dans ce mémoire, l'implémentation et les tests ont été effectués en même temps que le développement des concepts se faisait. Notamment, la représentation des composants par des configurations et interfaces, la suggestion des connexions et la connexion automatique ont été implémentées dans le logiciel *Intelligent Netlist Creator* (INC). Quant à la suggestion des formules et leurs extractions de la base de données, des tests ont simulé des exemples aléatoires (dynamiques) et statiques pour valider la méthodologie proposée, mais ils n'ont pas été introduits dans INC. Les raisons sont l'absence des données empiriques pour valider l'approche, la vérification des systèmes d'équations et la résolution des blocages de dépendance de contraintes. Toutefois, le développement actuel permet une telle extension, car la structure du code source a été implémentée dans un sens qui permet l'apprentissage de modèles.

Le *netlist* est pris en exemple pour la présentation de INC est présenté à la figure A.1. Il servira d'exemple d'application dans la présentation de l'interface graphique utilisateur (IUG).

A.1 Désign et IUG

INC a été implémenté avec la plateforme de développement *Qt*[25]. Sans le code des IUG et sans les commentaires, INC a atteint environ 6.500 lignes de code (environ 15.000 avec les tests de l'algorithme d'extraction et les simulations). Les fenêtres d'interaction avec

[25] Le site de *Qt-Creator* : http://qt-project.org/ , sept. 2013.

l'utilisateur pour créer un *netlist* d'un exemple pratique (voir la figure A.1) sont présentées dans les sections suivantes.

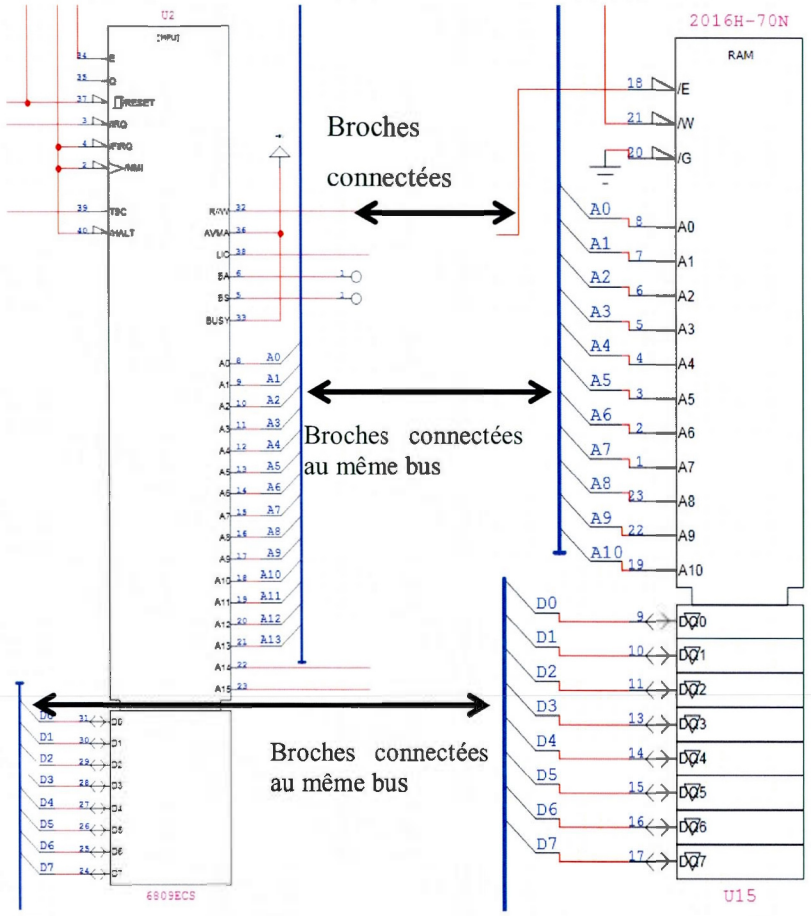


Figure A.1 Exemple pratique[26].

A.1.1 Fenêtre d'entrée

La figure A.2 est une capture d'écran de l'IUG d'entrée de INC qui se présente comme suit :

1. Le bouton « Create… » permet de créer un nouveau *netlist* en ouvrant la fenêtre de « sélection d'un composant » ;

[26] Exemple tiré partiellement de DesignWorks™ : *DesignWorks™ Professional*, version 5 (Windows All). Développeur : Capilano Computing Systems Ltd, October 2009.

2. Le bouton « EDIF Test » permet de charger un fichier de format EDIF existant en une forme arborescente (tel que montre la figure A.2) ;
3. Le bouton « Parsing... » convertit le fichier EDIF chargé en des configurations, interfaces et *nets* connectés (tel que montré dans la figure 6.7).

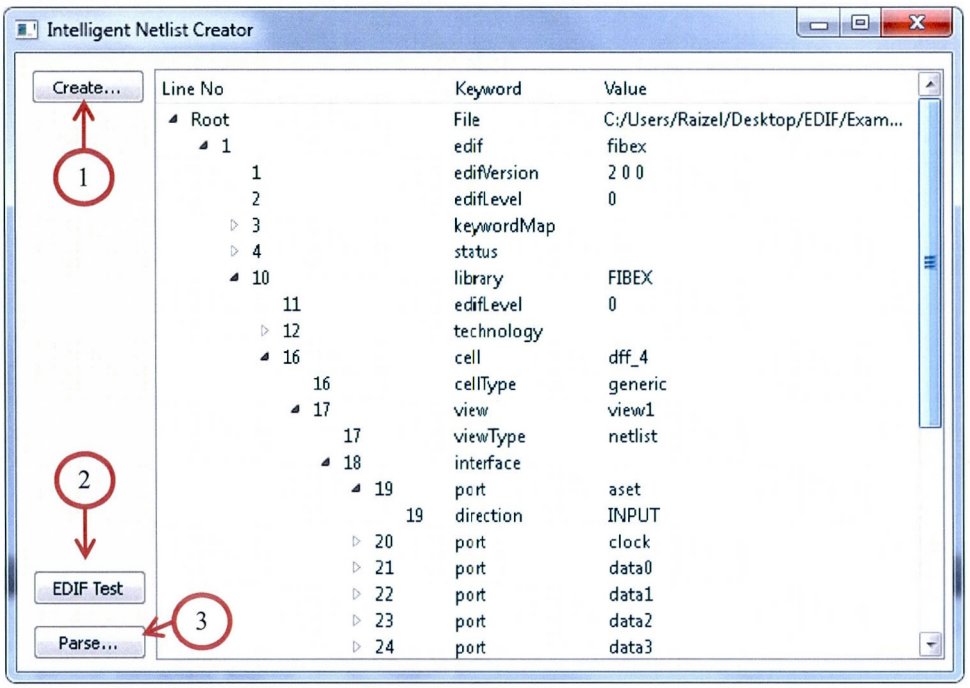


Figure A.2 Fenêtre d'entrée de INC.

Le développement actuel supporte la transformation d'un *netlist* de format EDIF (seulement à titre de preuve de concept). La création d'un nouveau *netlist* mène à une autre fenêtre que la section suivante présente.

A.1.2 Fenêtre de création de *netlist*s

La figure A.3 est une capture d'écran de l'IUG de INC pour la création d'un *netlist* et se présente comme suit :

1. La liste des composants sélectionnés que le *netlist* contient ;

2. Le bouton « Select… » permet de sélectionner les composants du *netlist* en ouvrant la fenêtre de « sélection de composants ». Ce bouton conduit au scénario du processus « Sélectionner Composants » ;
3. Le bouton « *Define…* » permet de définir la configuration d'un composant sélectionné dans le *netlist* en ouvrant la fenêtre de « définition de configuration ». Ce bouton devient disponible seulement lorsque l'utilisateur clique sur un composant du *netlist*. Si l'objet Composant perd l'interaction (*focus*) avec l'utilisateur, alors le bouton « *Define…* » devient indisponible (*disable*). Aussi, si un composant est connecté alors ce bouton est indisponible ; ce qui contraint l'utilisateur à ne pas changer de configuration si ce composant est connecté, car le développement actuel ne supporte pas cette option. Ce bouton mène au scénario du processus « Définir Configuration » ;
4. Le bouton « *Connect…* » permet de connecter deux ou plusieurs composants du *netlist* en ouvrant la fenêtre « Connexion de composant ». Ce bouton devient disponible seulement si l'utilisateur sélectionne deux ou plusieurs composants. Il mène au scénario du processus « Connecter Interfaces » ;
5. Les boutons « Save current *netlist…* » et « Open existing *netlist…* » permettent de sauvegarder et de charger un fichier d'un *netlist* existant ayant un format binaire propre à INC (binaire d'extension INCNetlist)[27].

L'action de « sélection des composants » est la première dans la chronologie. Elle est la seule action d'interaction disponible pour l'utilisateur pour la création d'une nouvelle *netlist* (à moins que l'utilisateur ne charge un *netlist* existant). La sélection des composants est présentée dans la section qui suit.

A.1.3 Fenêtre de sélection de composants

La figure A.4 (« Sélectionner Composants ») est une capture d'écran de l'IUG de INC pour la sélection des composants d'un *netlist* et se présente comme suit :

[27] Un futur développement de INC prévoit de générer des formats textes ou XML. La raison du choix de format binaire est que les fichiers prennent moins de capacité de stockage dans la mémoire (à des fins de tests). Les fichiers binaires d'extensions INCNetlist et INCComp sont spécifiques à INC.

1. La liste des composants de référence qui peuvent être instanciés plusieurs fois dans un *netlist*. Ils sont chargés soit de la base de données ou à partir d'un fichier de format INCComp à l'aide du bouton « Open… »;

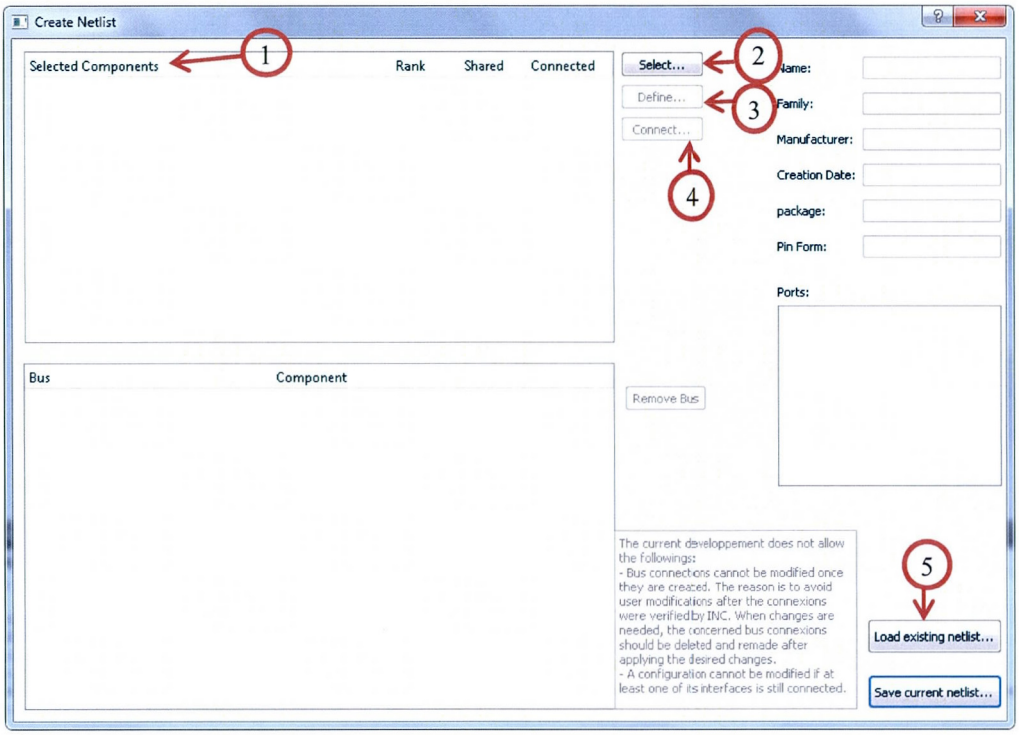


Figure A.3 Fenêtre de INC pour la création de *netlists*.

2. La liste des composants instanciés qui sont actuellement dans le *netlist*. Les boutons « Add » et « Remove » permettent respectivement d'instancier un composant de référence dans le *netlist* ou d'enlever un composant instancié du *netlist*. Le composant instancié prend un nom et un identifiant génériques. Le nom générique est modifiable par l'utilisateur en doublecliquant sur la chaine de caractères. La liste de toutes les configurations et leurs interfaces existantes de chaque composant est montrée en une forme arborescente à l'utilisateur afin qu'il ait une idée du composant instancié ;
3. Le bouton « New… » permet de créer un nouveau composant de référence en ouvrant la fenêtre de « création d'un nouveau composant ».

La liste des composants de la base de données (c.-à-d. de référence) est chargée dans la liste du point 1 de la figure A.4 pour la création d'un *netlist*. Si l'utilisateur ne trouve pas un composant dans la liste des composants de référence, alors il crée un nouveau à l'aide de la fenêtre de « création d'un nouveau composant » présenté dans la section qui suit.

A.1.4 Fenêtre de création d'un nouveau composant de référence

La figure A.5 est une capture d'écran de l'IUG de INC pour la création d'un nouveau composant de référence et se présente comme suit :

1. Les informations du composant sont remplies par l'utilisateur qui donne un nom de référence, la famille énumérée (pour la classe) dans une zone combinée et des informations passives.
2. L'utilisateur indique le nombre de chaque type de broches (attributs spécifiques).

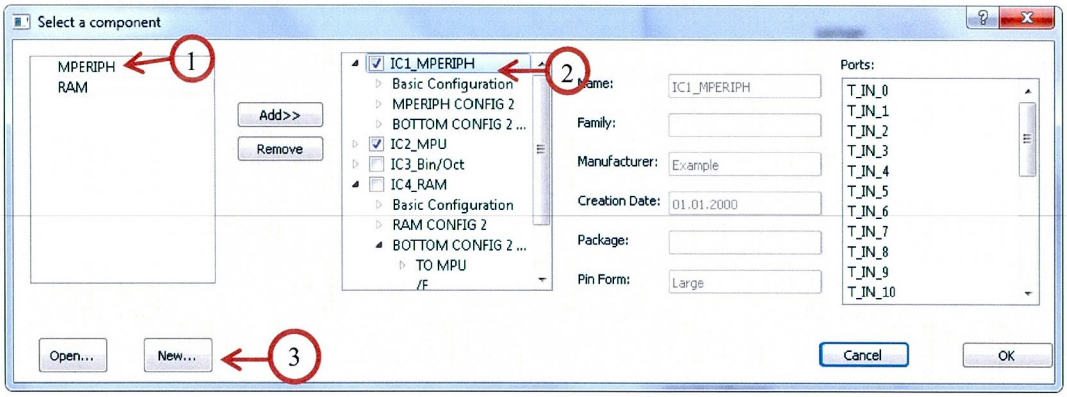


Figure A.4 Fenêtre de INC pour la sélection des composants d'un *netlist*.

Le composant créé est un composant de référence où les identifiants et les noms des broches sont génériques. Le développement actuel ne permet pas de modifier les noms ou les identifiants génériques des broches. Cependant, ce changement (incluant les attributs spécifiques tels que le type) se fait au niveau des interfaces gabarits auxquelles les broches sont associées. Le composant de référence permet de dupliquer plusieurs composants d'instances. Le composant instancié est ajouté au point 2 de la figure A.4.

L'ajout et le chargement des composants de référence permettent de créer la liste des composants à instancier dans le *netlist*. Puis, cette liste de composants instanciés est chargée

dans le point 1 de la figure A.3. Pour chaque composant de la liste, l'utilisateur définit une configuration en ouvrant la fenêtre de « définition de configurations » tel que présenté dans la section qui suit.

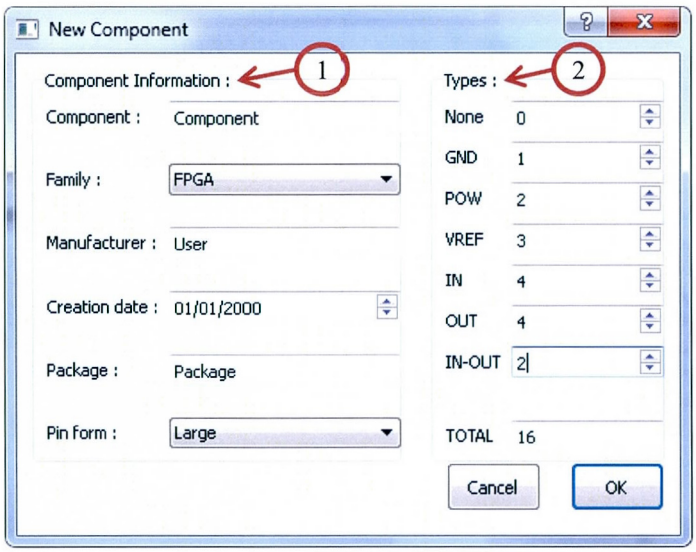


Figure A.5 Fenêtre de INC pour la création d'un nouveau composant de référence

A.1.5 Fenêtre de définition de configurations

Après avoir sélectionné un composant instancié et appuyé le bouton « Define... » de la figure A.3, la fenêtre de définition de configuration s'ouvre en chargeant le composant sélectionné (le bouton devient disponible seulement lorsque l'objet cliqué par l'utilisateur est un composant et que ce composant n'est pas connecté). Cette IUG correspond au processus « Définir Configuration » du chapitre II (section 2.3.2). La figure A.6 est une capture d'écran de l'IUG de INC pour la définition d'une configuration d'un composant et se présente comme suit :

1. À l'ouverture de la fenêtre, la liste des configurations existantes dans le composant est chargée dans « Configuration ». Si le composant a été instancié d'un nouveau composant de référence, il ne contient aucune configuration. Dans ce cas, la configuration basique est automatiquement générée et ajoutée à la liste des configurations du composant. La création automatique de la configuration est basée

sur les propriétés établies dans ce mémoire. Notamment, chaque broche du composant instancie une interface basique pointe sur la broche. À l'instance, toute interface possède la liste de toutes les contraintes et donc les informations telles que le type de la broche ou d'autres contraintes associées sont passées à la nouvelle interface basique. Toutes les nouvelles interfaces basiques sont ajoutées à une nouvelle configuration qui prend aussi les contraintes associées au composant ;

2. La liste « Interface » contient toutes les interfaces gabarits existantes dans toutes les configurations du composant. Chaque interface est représentée une seule fois. La liste « Related Configurations » montre la liste des configurations dans lesquelles se trouve une interface gabarit sélectionnée dans « Interface » ;
3. Le bouton « Save... » permet de sauvegarder le composant et toutes ses configurations et interfaces dans un seul fichier binaire (de format INCComp) indépendamment du *netlist* en cours de conception. Ce fichier peut être rechargé dans un autre *netlist* en tant que composant de référence (à partir du bouton « Open... » de la figure A.4) ;
4. L'utilisateur peut assigner une valeur de contrainte à une configuration ou une interface mère et ses enfants (par héritage récursif) dépendamment de l'objet sélectionné dans la liste « Configuration ». L'utilisateur sélectionne les contraintes par leurs cases à cocher pour désigner les valeurs à faire hériter aux enfants d'une interface mère. Toutefois, l'assignation de valeurs aux contraintes d'une interface est mieux contrôlée dans l'IUG de connexion d'interfaces. Ce point concerne essentiellement les contraintes de configuration ;
5. Le bouton « Add new... » permet de créer une nouvelle configuration et de nouvelles interfaces si requises en ouvrant la fenêtre de « création de nouvelle configuration ». Le bouton « Delete » permet de supprimer une configuration existante avec toutes ses interfaces sauf la configuration basique. Le développement actuel ne permet pas la modification d'une configuration existante.

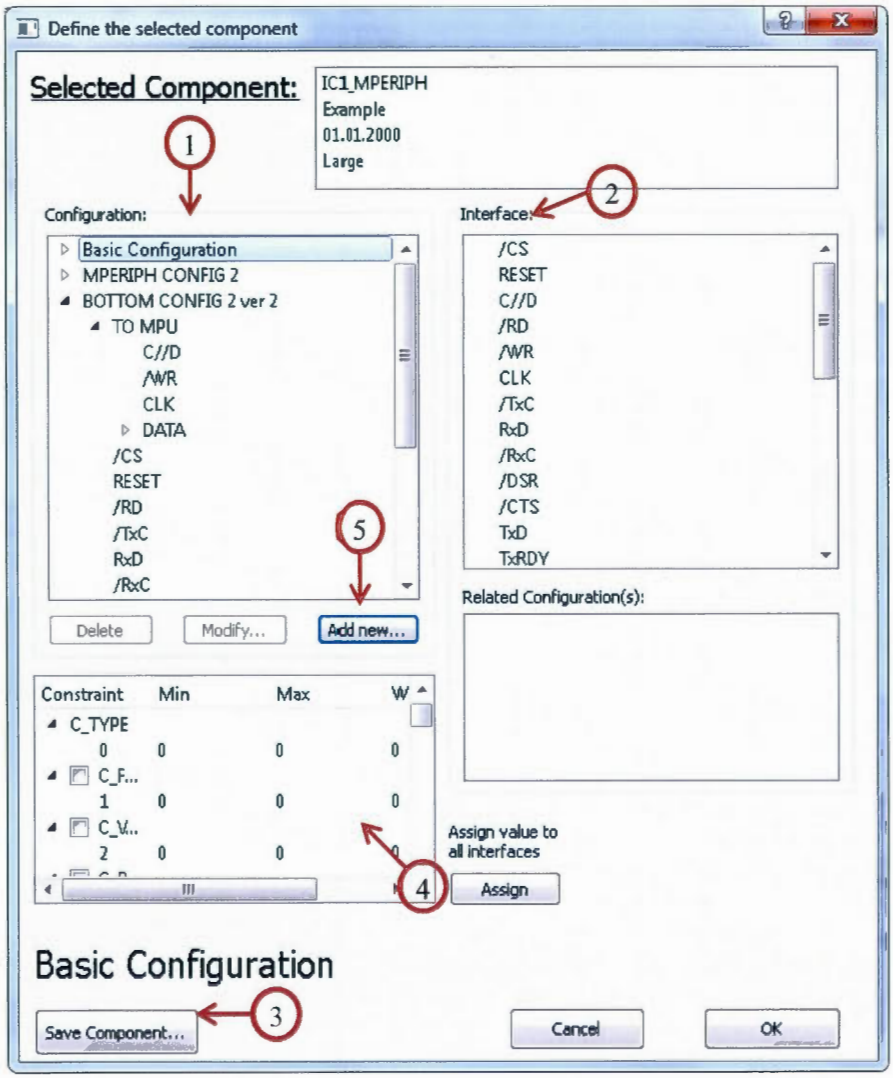


Figure A.6 Fenêtre de INC pour la définition d'une configuration d'un composant.

Si l'utilisateur ne trouve pas la configuration désirée, alors il crée une nouvelle à l'aide de la fenêtre de « création d'une nouvelle configuration » tel que présenté dans la section qui suit.

A.1.6 Fenêtre de création de nouvelle configuration et interfaces

Après avoir cliqué sur le bouton « Add new… » de la figure A.6, la fenêtre de création d'une nouvelle configuration s'ouvre en chargeant toutes les configurations existantes du composant. La figure A.7 est une capture d'écran de l'IUG de INC pour la création d'une configuration du composant sélectionné et se présente comme suit :

1. La liste des configurations et interfaces existantes est chargée lors de l'ouverture de la fenêtre ;
2. Pour créer une configuration, l'utilisateur sélectionne les interfaces existantes du point 1 et les copies d'objets dans la liste du point 2 à l'aide du bouton « >>>> ».
3. Pour créer une interface, l'utilisateur sélectionne les interfaces existantes du point 1 et les copies d'objets dans la liste du point 3 à l'aide du bouton « >> ». La nouvelle interface peut être ajoutée directement dans la liste de la nouvelle configuration (du point 2) ou être ajoutée (temporairement) à la liste des interfaces existantes (du point 1) pour servir dans la hiérarchie d'une autre nouvelle interface ;
4. L'utilisateur ajoute la nouvelle interface du point 3 temporairement à la liste des interfaces existantes pour s'en servir dans la création d'une autre nouvelle interface. Cette option permet à l'utilisateur de créer les encapsulations désirées des interfaces.

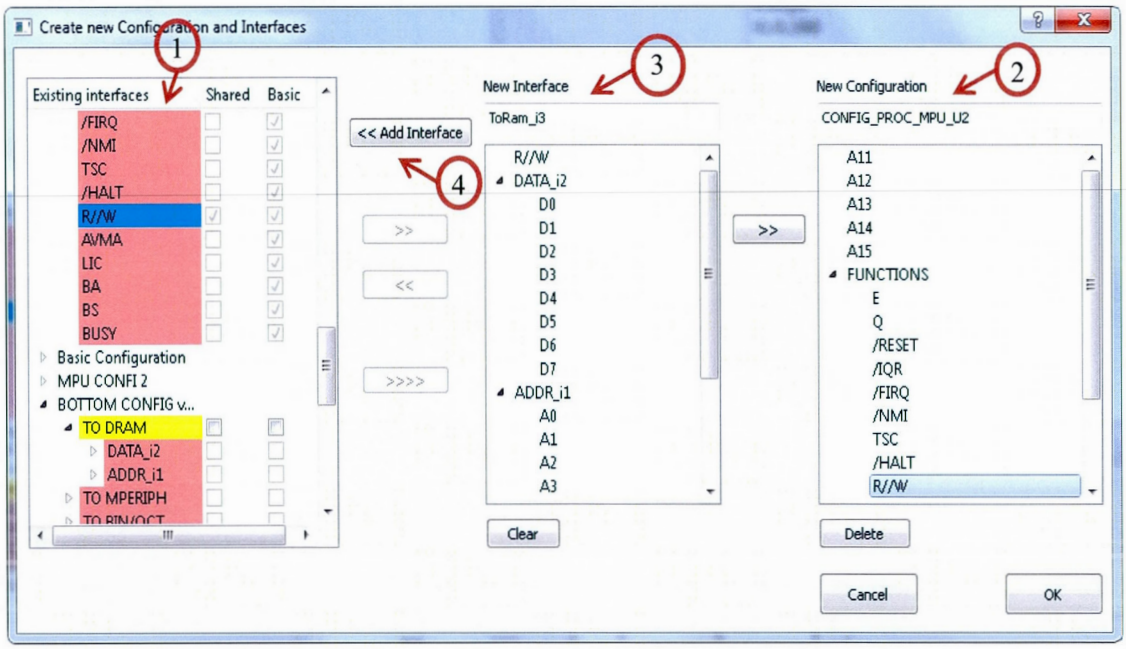


Figure A.7 Fenêtre de INC pour la création d'une nouvelle configuration et de plusieurs interfaces.

Toutes les propriétés des interfaces interviennent dans cette fenêtre (sauf l'atomicité qui a permis de créer automatiquement une configuration basique). Lorsqu'une interface non partageable est ajoutée, elle devient indisponible et de couleur rouge. L'utilisateur ne peut

plus s'en servir une fois ajoutée dans une nouvelle interface ou configuration. Dans la liste du point 1, l'utilisateur peut changer l'attribut partageable d'une interface pour la rendre partageable (ainsi que tous ses enfants automatiquement). Ceci permet à l'utilisateur de l'ajouter dans plusieurs nouvelles interfaces mères (telle que l'interface « R/W » de la figure A.7 présente dans la liste du point 2 et du point 3) ; assurant ainsi la propriété de partage. De plus, la même interface partageable ne peut être ajoutée qu'une seule fois en tant qu'enfant direct à une interface mère ou à la nouvelle configuration pour empêcher l'utilisateur de créer des interfaces vides de sens. L'utilisateur peut aussi appliquer la propriété d'équivalence en changeant l'attribut *basic* dans la liste du point 1. Cette option est disponible seulement pour les interfaces mères. Une interface mère basique connecte tous ses enfants et descendants en une seule connexion. D'où, les enfants et descendants deviennent tous indisponibles (ne peuvent être sélectionnés individuellement) et l'utilisateur ne peut interagir qu'avec l'interface mère basique (incluant les valeurs des contraintes). La propriété d'existence est assurée par le fait que seules les interfaces de même classe sont chargées dans le point 1. Quant à la propriété de fermeture, elle est assurée par le fait que l'utilisateur ne peut sauvegarder la nouvelle configuration que lorsque toutes les interfaces basiques s'y retrouvent au moins une fois. Aussi, dans le point 1 de la figure A.7, une interface en jaune (telle que l'interface « ToDRAM ») signifie que la structure de cette interface n'est plus valide, car au moins une interface enfant (ou descendant) non partageable a été utilisée. Du fait que l'interface enfant n'est pas partageable, alors l'interface mère n'est plus valide et donc devient indisponible. Ce qui notifie l'utilisateur des interfaces valides qui restent pour la nouvelle configuration. À ce point de développement, le IUG des interfaces gabarit pour l'affectation des interfaces basiques n'a pas été développé (mais juste des tests ont été appliqués).

La nouvelle configuration créée est ajoutée à la liste des configurations existantes du point 1 de la figure A.6. Après la définition de sa configuration, le composant est mis à jour dans la liste des composants du point 1 de la figure A.3 (liste de composants du *netlist*). La sélection de plusieurs composants définis active le bouton « Connect… » de la figure A.3. L'utilisateur peut choisir un sous-groupe de composants définis pour les connecter tel que présenté dans la section qui suit.

A.1.7 Fenêtre de connexion d'interfaces

La fenêtre de la connexion d'interfaces est présentée dans les figures A.8, A.9, A.10 et A.11. Elle correspond au processus « Connecter Interfaces » du chapitre II (section 2.3.2) et contient également les sous-processus 3.1 de « suggestion d'interfaces compatibles » et 3.2 de « interconnexions automatiques» des interfaces suggérées. La démonstration est appliquée sur l'exemple pratique de la figure A.1 dans lequel se trouvent un processeur U2 et une RAM U15. De plus, deux autres RAM U16 et U17 identiques à U15 ont été instanciées pour faire une connexion simultanée au processeur U2 (et d'autres tests). Elles sont présentées dans les sections qui suivent.

A.1.7.1 Suggestion de connexions d'interfaces compatibles

Le sous-processus de « suggestion de connexions » d'interfaces compatibles est décrit dans les points des figures A.8 et A.9 comme suit :

1. Le sous-groupe de composants définis est chargé à l'ouverture de la fenêtre dans « la liste composants ». Lorsque l'utilisateur clique sur un objet Interface, alors la liste des interfaces compatibles (et potentiellement compatibles) trouvées est montrée dans le point 2;
2. Dans l'exemple pratique, l'interface $C_{U2}.i_3$ (ToRam_i3 du composant IC1_MPU) contient $C_{U2}.i_1$, $C_{U2}.i_2$ et $C_{U2}.i_{R/W}$. Lorsque l'utilisateur clique sur $C_{U2}.i_3$ alors la liste des interfaces compatibles montre la disponibilité de $C_{U15}.i_7$ (ToMPU_i7) qui contient $C_{U15}.i_5$, $C_{U15}.i_6$ et $C_{U15}.i_{/W}$ de la RAM U15 (composant IC2_RAM). Les interfaces $C_{U16}.i_7$ et $C_{U17}.i_7$ des autres RAM U16 et U17 (composants IC3_RAM et IC4_RAM) sont aussi compatibles, car elles ont la même structure. L'utilisateur peut sélectionner toutes les trois interfaces et les connecter directement. Il peut décider de modifier les valeurs des contraintes pour mieux filtrer les interfaces suggérées ;
3. Dans le point 2 ci-dessus (la suggestion d'interfaces compatibles), les valeurs de contraintes des interfaces $C_{U15}.i_7$ et $C_{U17}.i_7$ sont toutes zéro alors que celle de $C_{U16}.i_7$ a juste la contrainte du voltage à [2; 3]V. L'utilisateur modifie les contraintes de la fréquence et d'alimentation respectivement pour les valeurs [222, 333]MHz et

[3, 4]V pour l'interface $C_{U2}.i_3$. L'utilisateur peut décider aussi de faire hériter les valeurs des contraintes sélectionnées par leur case à cocher à tous ses descendants ;

4. L'utilisateur peut décider de faire passer toutes les valeurs de contraintes sélectionnées (par leur case à cocher) à n'importe quelle interface de n'importe quel composant dans le *netlist* (compatible ou pas). Il peut aussi spécifier les interfaces à affecter selon leur type. Par exemple, dans la figure A.8, l'utilisateur a affecté les valeurs des contraintes de fréquence et d'alimentation de l'interface $C_{U2}.i_3$ à l'interface $C_{U16}.i_7$ et tous ses enfants de type NONE (qui par défaut désigne tous les types). L'affection des valeurs de contraintes aux descendants de $C_{U16}.i_7$ est optionnelle. Cette option permet à l'utilisateur de transférer automatiquement toutes les contraintes sélectionnées d'une interface à une autre (pouvant la rendre directement compatible ou potentiellement compatible si elles ont le même nombre d'interfaces basiques) ;
5. Les normes standards de signaux sont un cas spécial du point 4. Dans ce point 5, HSTL pris en exemple pour les tests. Cependant, l'exemple a été arbitrairement choisi lorsque la valeur du voltage de $C_{U16}.i_7$ a été mise à [2, 3]V au début des connexions. Lorsque l'utilisateur a affecté la norme HSTL à $C_{U2}.i_3$ dans la figure A.8, alors la liste des composants suggérés est mise à jour telle que présentée dans la figure A.9. L'interface $C_{U16}.i_7$ (ToMPU_i7 de IC3_RAM) a été éliminée de la liste des interfaces compatibles de $C_{U2}.i_3$, car celle-ci a pris la valeur [0.75, 0.75]V et donc n'est plus compatible avec [2, 3]V. Suivant cette approche, une mise à jour des interfaces compatibles assiste l'utilisateur à mieux filtrer les interfaces à connecter selon les variations des valeurs des contraintes.

À la suite de la connexion $C_{U2}.i_3$ avec les interfaces compatibles $C_{U15}.i_7$ et $C_{U17}.i_7$ simultanément, les rangs de correspondance des interfaces internes sont affectés automatiquement tels que présentés dans la section qui suit.

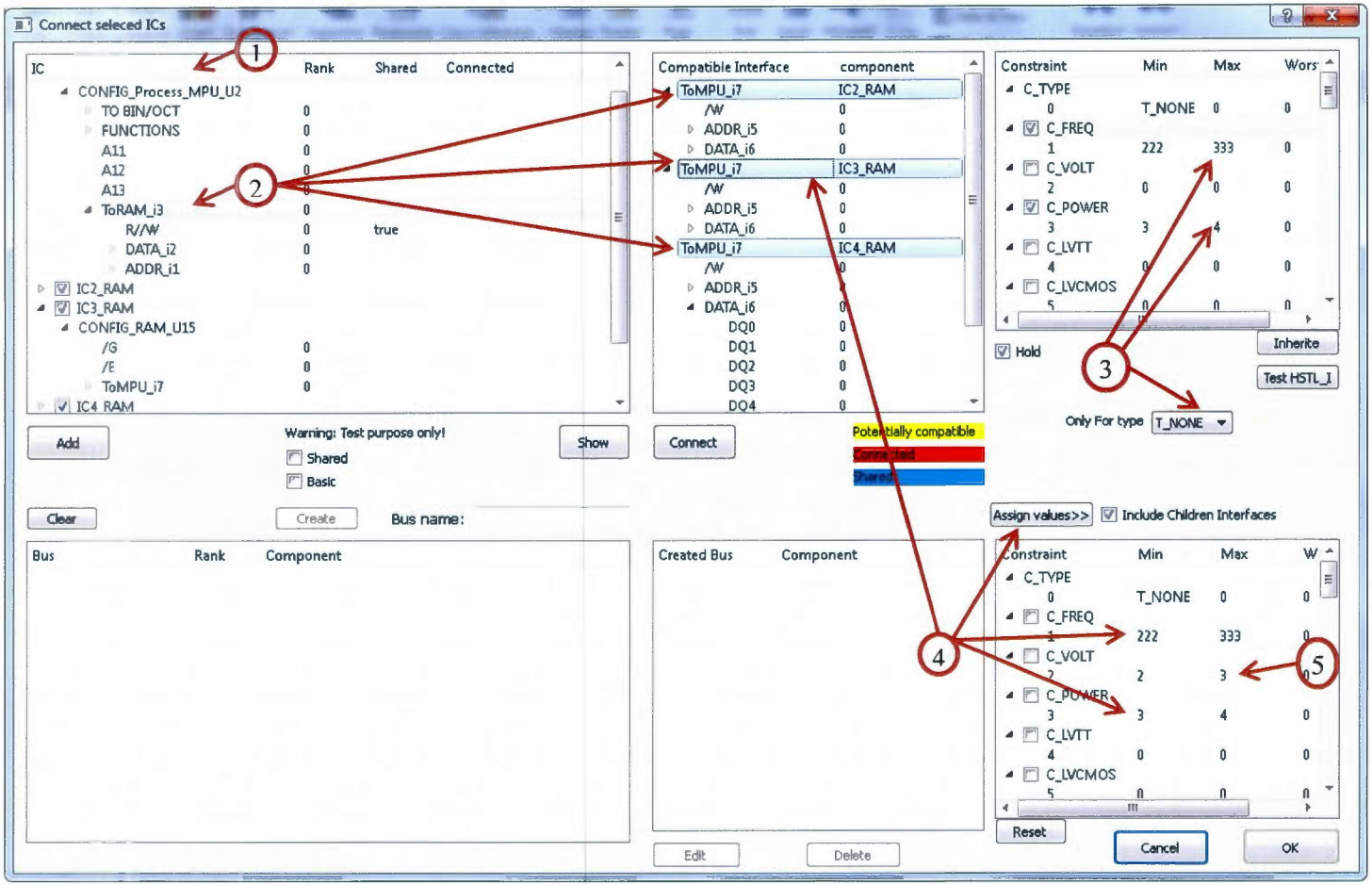


Figure A.8 Fenêtre de INC pour la suggestion de connexions.

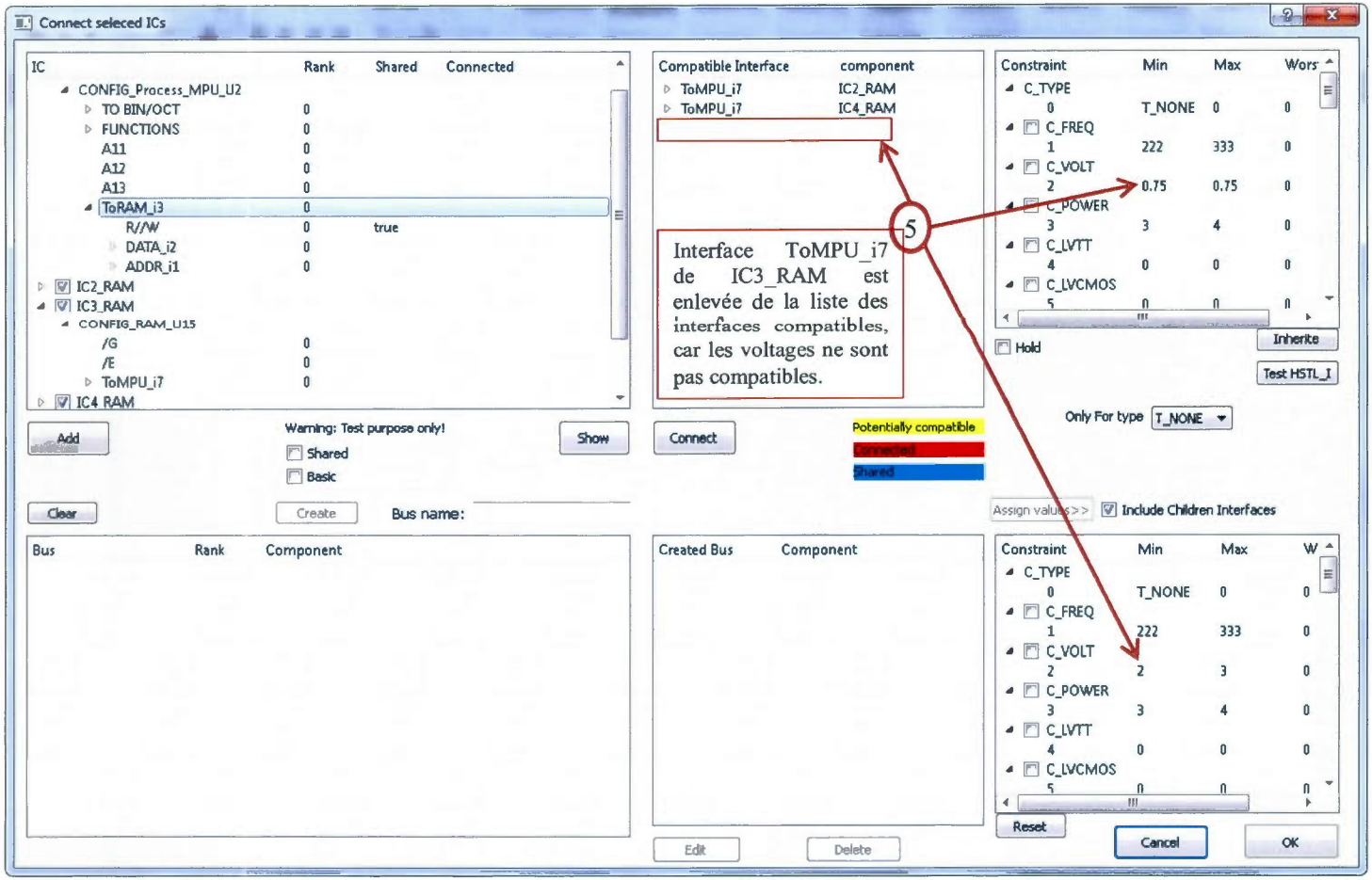


Figure A.9 Mise à jour automatique des interfaces compatibles.

A.1.7.2 Interconnexions automatiques des interfaces compatibles

Le sous-processus des « interconnexions automatiques » des interfaces compatibles (à connecter) est décrit dans les figures A.10 et A.11 comme suit :

1. Dans la figure A.10, le rang est affecté automatiquement aux interfaces basiques tel que décrit dans l'algorithme d'intercorrespondance automatique. Les *nets* ne sont toujours pas créés à ce point et ceci permet à l'utilisateur de faire des modifications manuelles telles que l'ajout de n'importe quelle interface de la liste des composants disponibles, la modification des rangs des interfaces qui créés les *nets*, les valeurs des contraintes[28] ;
2. Dans la figure A.11, après que l'utilisateur décide des rangs et des valeurs de contraintes comportementales des interfaces, alors les connexions sont automatiquement créées dépendamment des rangs. Toutes les interfaces basiques ayant le même rang sont ajoutées au même *net* ; puis, l'ensemble des *nets* est assigné au bus créé.

À la suite de la création des *nets*, de nouvelles connexions peuvent être créées. Cependant, dans la liste des interfaces compatibles d'une interface à connecter, des informations aide l'utilisateur à mieux cerner la situation actuelle du circuit. Par exemple, dans le point 3 de la figure A.10, il y a des interfaces déjà connectées (fond rouge) et des interfaces partageables (fond bleu) dans la liste des interfaces compatibles. Les interfaces partageables (montrées par des flèches bleues) peuvent être ajoutées à de nouvelles connexions, mais pas les interfaces connectées (fonds rouges et indiquées par des flèches noires). Dans le point 4 de la figure A.11, le composant *IC5_MPERIPH* contient une interface *TO_MPU* potentiellement compatible avec l'interface *ADDR_i1* ($C_{U2}.i_1$) de *IC1_MPU* (mais celle-ci étant connectée et potentiellement compatible alors elle est affectée d'un fond jaune et rouge). Si l'utilisateur veut connecter *ADDR_i1* avec *TO_MPU*, alors il peut la déclarer comme partageable ; cependant, cette option est disponible juste dans un cadre de test. De même pour une interface mère basique, l'utilisateur peut déclarer une interface mère en tant qu'une interface

[28] À ce point, le module de suggestion de formules est supposé intervenir, car les connexions sont prêtes à être créer. Le module de suggestion de formules utilise les rangs pour trouver les classes. Un exemple de test de formules est montré dans la section des résultats.

basique où tous ses enfants seront interconnectés. Dans ce cas, les interfaces enfants deviennent inaccessibles.

Après la création des *nets*, la fermeture de la fenêtre de « connexions d'interfaces » retourne l'utilisateur à la fenêtre principale de « création de *netlists* » pour une éventuelle sauvegarde dans un fichier tel que présenté dans la figure A.12. L'utilisateur peut sélectionner un groupe de composants définis pour de nouvelles connexions ou ajouter de nouveaux composants au *netlist.*

La présentation de INC dans cette section est la version introduite dans le projet DreamWafer™. Quant aux tests de simulation de suggestion et d'extraction des formules du délai, ils n'ont pas été introduits dans INC mais sont présentés dans la section des résultats.

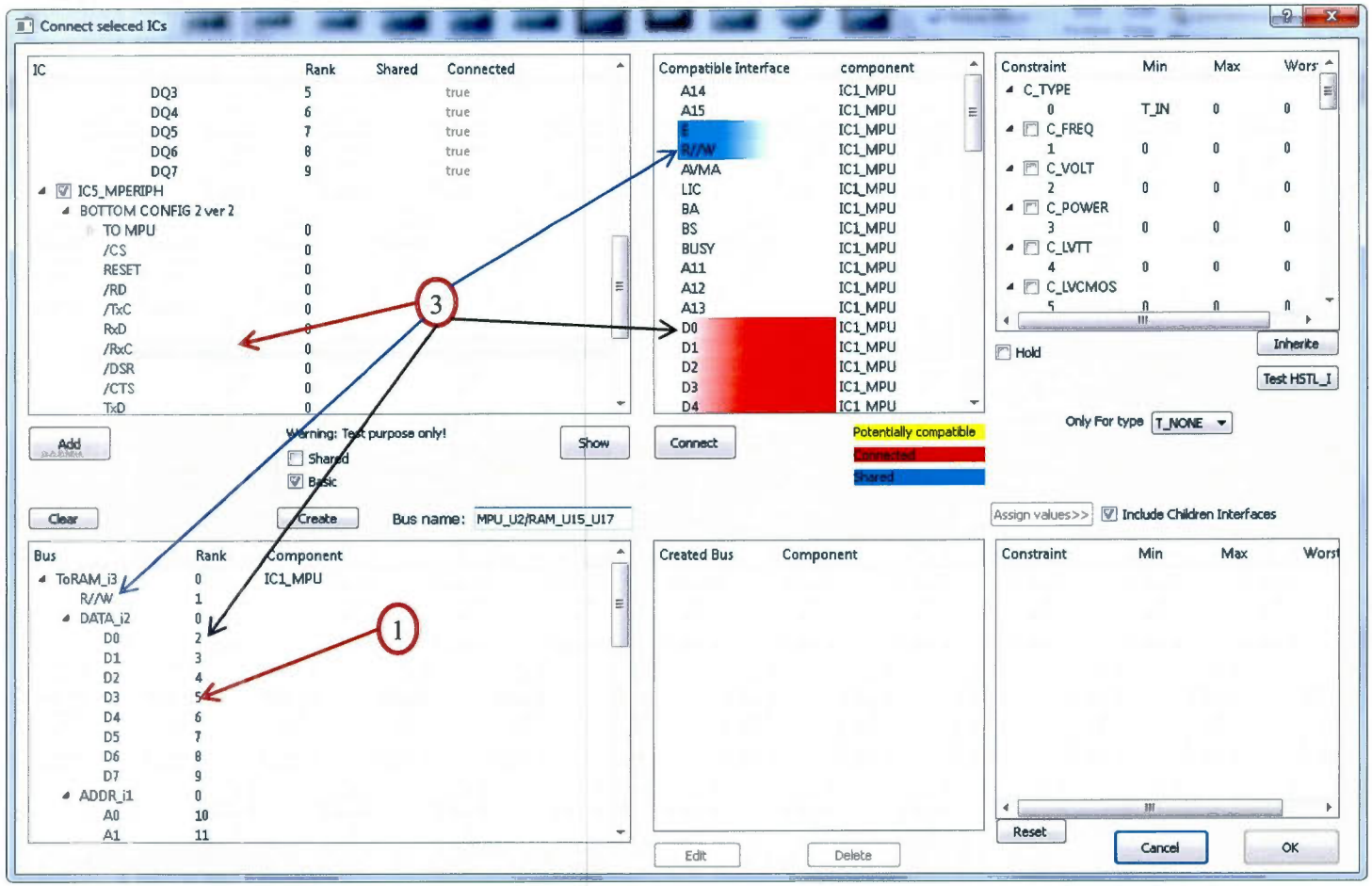


Figure A.10 Affectation automatique des rangs aux interfaces compatibles.

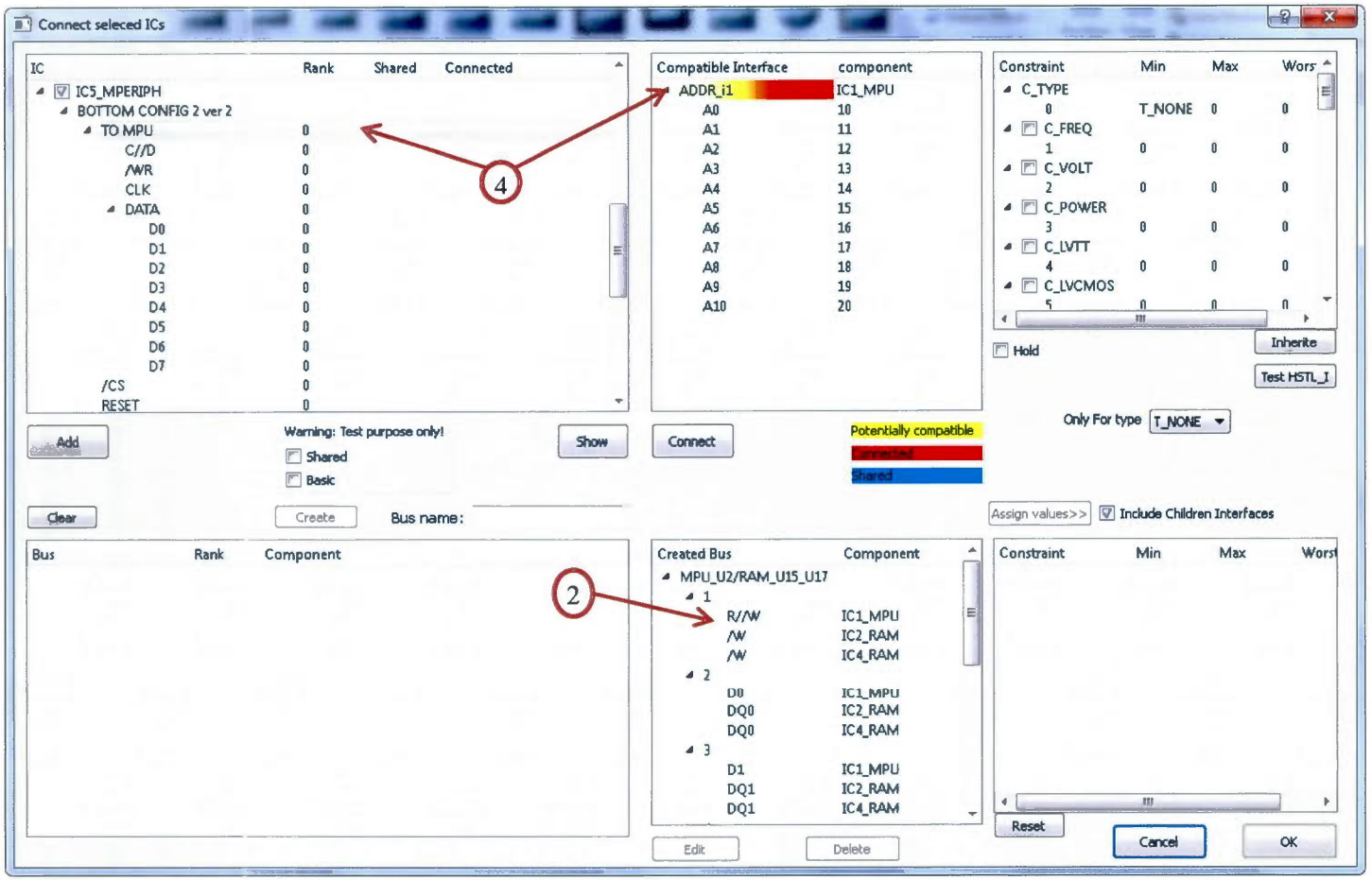


Figure A.11 Création automatique des connexions.

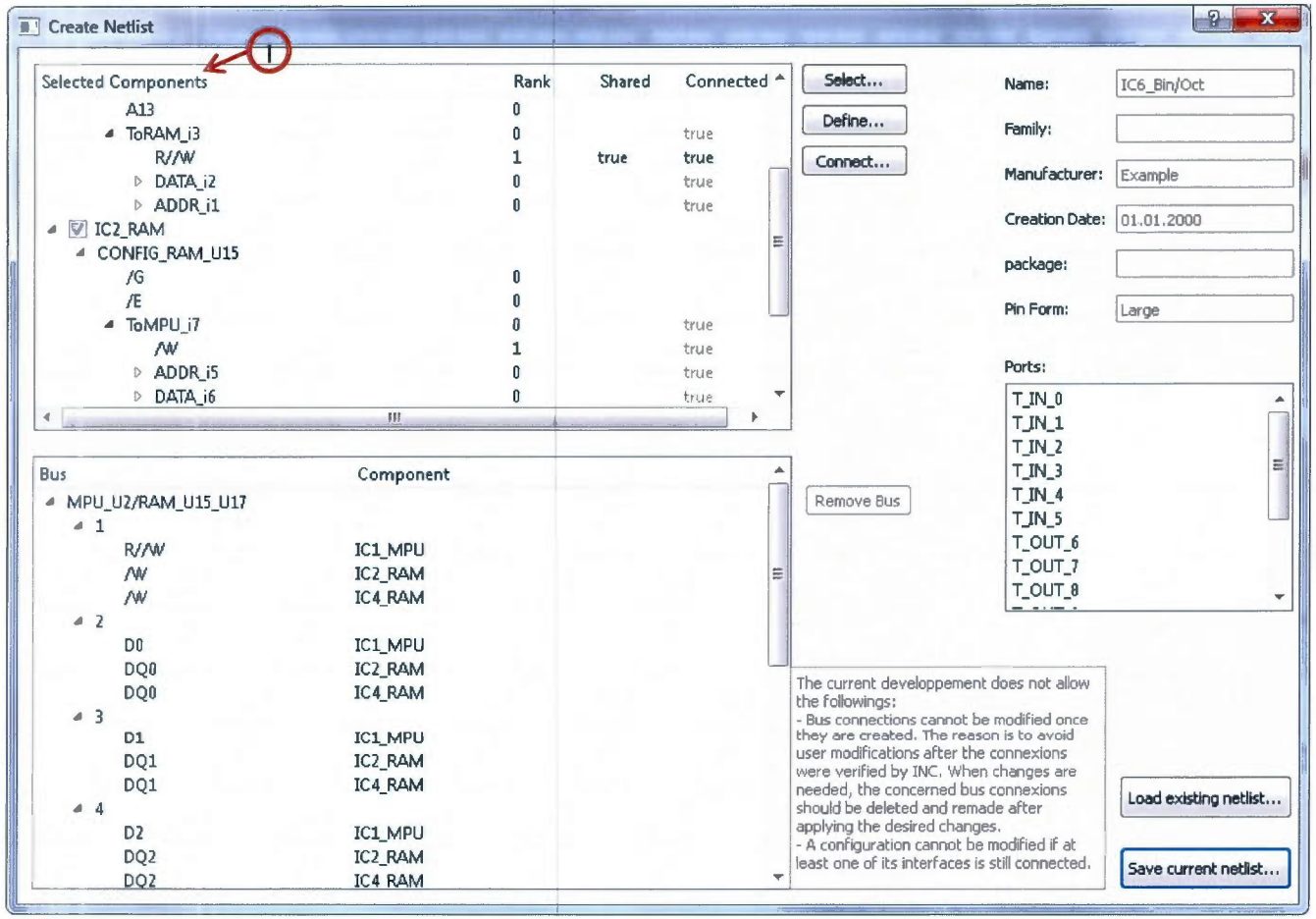


Figure A.12 *Netlist* de l'exemple pratique de la figure A.1 (avec une mémoire RAM additionnelle identique à U1.

BIBLIOGRAPHIE

[1] Sait, Sadiq, et Habib Youssef. 1995. « Introduction ». In *VLSI Physical Design Automation Theory and Practice*, p. 1-2. Berkshire, Maidenhead : McGraw-Hill.

[2] Gielen, Georges, et Rob Runtenbar. 2000. « Computer-aided design of analog and mixed-signal integrated circuits ». In *Proceedings of the IEEE*, vol. 88 (décembre), p. 1825-1854.

[3] Sherwani, Naveed. 1995. « Vlsi physical design automation ». In *Algorithms for VLSI Physical Design Automation*, p. xvii-2. Boston : Kluwer Academic Publishers.

[4] Airiau, Roland, Jean-Michel Bergé, Vincent Olive et Jacques Rouillard. 1998. « En guise de préface ». In *VHDL : langage, modélisation, synthèse*, deuxième édition revue et augmentée, p. 1-3 et p. 546. Lausanne, Suisse : Presses polytechniques et universitaires romandes.

[5] Lepercq, Etienne, Yves Blaquière, Richard Norman et Yvon Savaria. 2009. « Workflow for an electronic configurable prototyping system ». In *International Symposium on Circuits and Systems (ISCAS)*, IEEE, p. 2005-2008.

[6] Mjolsness, Eric, et Dennis DeCoste. 2001. « Machine Learning for Science: State of the Art and Future Prospects ». *Science*, New Series, vol. 293, no. 5537 (Sep. 14), p. 2051-2055.

[7] Jaiswal, Swati, Neeraj Gupta et Hina Shrivastava. 2012. « Enhancing the features of Intrusion Detection System by using machine learning approaches ». *International Journal of Scientific and Research Publications*, vol. 2, no. 2, Février, p.166-170.

[8] Wang, Jue, et Qing Tao. 2008. « Machine learning : state of the art ». *Intelligence Systems*, IEEE, vol. 23, no. 6, p. 49-55.

[9] Tan, Pang-Ning, Michael Steinbach et Vipin Kumar. 2006. « Introduction ». In *Introduction to data mining*, p. 1. Boston; Montreal. Édition : Pearson Addison Wesleyp.

[10] Tan, Pang-Ning, Michael Steinbach et Vipin Kumar. 2006. « Association analysis : basic concepts and algorithms ». In *Introduction to data mining*, p. 333. Boston; Montreal. Édition : Pearson Addison Wesleyp.

[11] Fayyad, Usama, Gregory Piatetsky-Shapiro et Padhraic Smyth. 1996. « From data mining to knowledge discovery in databases ». *AI Magazine*, vol. 17, no 3 : automne, p. 37-54.

[12] Agrawal, Rakesh, et Ramakrishnan Srikant. 1995. « Mining sequential patterns ». In *Proceedings of the Eleventh International Conference on Data Engineering*, p. 3-14.

[13] Aamodt, Agnar, et Enric Plaza. 1994. « Case-based reasoning: Foundational issues, methodological variations, and system approaches ». *AI communications*, vol. 7, no. 1, p. 39-59.

[14] Ifenthaler, Dirk, et Norbert Seel. 2012. « Model-based reasoning ». *Computers & Education*, vol. 64, p. 131-142.

[15] Badreddine, Mohamed, Yves Blaquière et Mounir Boukadoum. 2011. « Machine-learning framework for automatic netlist creation ». *IEEE International Symposium on Circuits and Systems (ISCAS)*, p. 2865- 2868.

[16] Jerke, Goeran, et Jens Lienig. 2009. « Constraint-driven design: the next step towards analog design automation ». In *Proceedings of the 2009 international symposium on Physical design (ISPD '09)*. ACM, New York, NY, USA, p. 75-82.

[17] Priemer, Roland. 1991. « Signal and signal processing ». In *Introductory signal processing*, vol.6, p.1. Singapore, Teaneck, NJ : World Scientific Publishing.

[18] Sinha, Priyabrata. 2010. « Sampling and quantization ». In *Speech processing in embedded systems*, p. 9. London, New York: Springer.

[19] Amon, Tod, Gaetano Borriello, Taokuan Hu et Jiwen Liu. 1997. « Symbolic timing verification of timing diagrams using Presburger formulas ». In *Proceedings of the 34th annual Design Automation Conference*, ACM, p. 226-231.

[20] Hulgaard, Henrik, et Tod Amon. 2000. « Symbolic timing analysis of asynchronous systems ». *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol.19, no. 10, p.1093-1104.

[21] Toumazou, Christofer, and Costas A. Makris. 1995. « Analog ic design automation. i. automated circuit generation: new concepts and methods ». *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, no. 2, p. 218-238.

[22] Sommer, Ralf, et al. 2002. « From system specification to layout: seamless top-down design methods for analog and mixed-signal applications ». In *Proceedings on Design, Automation and Test in Europe Conference and Exhibition*, IEEE, p. 884-891.

[23] Talupur, Muralidhar. 2011. « Hardware Model Checking: Status, Challenges, and Opportunities ». In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, p. 154.

[24] Barros, Manuel, Jorge Guilherme et Nuno Horta. 2010. « Analog circuits optimization based on evolutionary computation techniques ». *Integration, the VLSI Journal*, vol.43, no. 1, p.136-155.

[25] Coussy, Philippe, Daniel Gajski, Michael Meredith et Andres Takach. 2009. « An introduction to high-level synthesis ». *Design & Test of Computers*, IEEE, vol. 26, no. 4, p. 8-17.

[26] Arakawa, Toshio, et al. 2004. « Method and apparatus for automatic wiring design between block circuits of integrated circuit ». *U.S. Patent 6 760 897*, issu le 6 juillet 2004.

[27] Glasser, Mark, et al. 2006. « Cookbook orientation ». *The Verification CookBook: Advanced Verification Methodology*, deuxième édition, p. 22. Mentor Graphics, Cambridge.

[28] Olaru, Cristina, et Louis Wehenkel. 2003. « A complete fuzzy decision tree technique ». *Fuzzy sets and systems*, ScienceDirect, vol. 138, no. 2, p. 221-254.

[29] Borgelt, Christian. 2005. « An Implementation of the FP-growth Algorithm. » In *Proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations*, ACM, p. 1-5.

[30] Fernau, Henning. 2009. « Algorithms for learning regular expressions from positive data. » *Information and Computation*, vol. 207, no. 4, p. 521-541.