Table des matières

Prologue

L'aboutissement de cet ouvrage fut possible grâce au soutien de nombreuses personnes. J'exprime tout d'abord ma sincère gratitude au Prof. Dr. Jürg Kohlas qui m'a permis de réaliser un travail des plus intéressants, en me laissant toujours une large liberté de manoeuvre. Mon intégration à son groupe de recherche a influencé très positivement ce travail.

Je suis également très reconnaissant à Rolf Haenni pour sa constante disposition, son aide et ses conseils sur l'implémentation de DIG. Je remercie également spécialement Norbert Lehmann et Urs Hänni pour les nombreuses discussions qui m'ont aidé à clarifier beaucoup de points.

Enfin, je me dois de remercier ma famille et mes amis, sans lesquels ce travail n'aurait pas été possible.

1 Introduction

Dans le but de généraliser le système ATMS (*Assumption-based Truth Maintenance Systems*) [DeKleer 1986a, DeKleer 1986b], on a été amené récemment à développer le *Clause Management System* (CMS) [Reiter, DeKleer 1987]. Schématiquement, la tâche d'un tel système consiste à accepter en entrée un ensemble de clauses propositionnelles et à redonner en sortie l'ensemble des impliqués premiers correspondants.

Pour réaliser un CMS, Ngair a développé dans [Ngair 1992] une nouvelle méthode de génération d'impliqués premiers ou, dualement, d'impliquants premiers. Cette nouvelle méthode se distingue doublement; tout d'abord parce qu'elle a été motivée entièrement d'un point de vue théorique, se basant sur la théorie des espaces convexes; ensuite parce qu'elle se révèle beaucoup plus flexible en ce qui concerne les entrées, en acceptant une conjonction de formes normales disjonctives, abrégées DNF. Cette deuxième particularité rend l'algorithme présenté particulièrement intéressant. Il peut traiter directement des problèmes se formulant en une conjonction de DNF, sans devoir, comme c'est le cas pour les autres algorithmes, la transformer auparavant dans la forme normale conjonctive (CNF) correspondante; d'ailleurs, une telle transformation est très coûteuse. La plupart des algorithmes de génération d'impliqués premiers sont de ce fait inefficaces pour ce cas concret. L'algorithme de Ngair permet aussi de traiter directement, comme la majorité des algorithmes existants, les problèmes se formulant en conjonctions.

De nombreux problèmes peuvent être formulés à l'aide de formes normales disjonctives. Un exemple typique est la description d'une porte d'un circuit logique (*and*, *nand*, *or*, *nor*, etc.) pouvant se comporter selon différents modes: par exemple selon un mode correct *ok* et deux modes fallacieux *ab1* et *ab2*. On peut décrire cette porte facilement par une conjonction de DNF. Une formule telle que $xor(x_1,\ldots,x_n)$ peut être également naturellement codé par la DNF $(x_1 \wedge \neg x_2 \wedge \cdots \wedge \neg x_n) \vee \cdots \vee (\neg x_1 \wedge \cdots \wedge \neg x_{n-1} \wedge x_n)$. S'il fallait la transformer en une CNF, il faudrait au moins n^2 clauses disjonctives. Il est donc plus simple d'utiliser une formulation intuitive.

La tâche du travail de diplôme fut d'implémenter l'algorithme de Ngair en l'appliquant ensuite à la déduction basée sur suppositions au moyen de formes normales disjonctives. Une application directe fut réalisée en construisant un langage et un système (du nom de DIG) pour la description et le diagnostique de circuits digitaux contenant des erreurs. Les impliqués premiers d'un circuit digital à différents modes de comportement permettent la construction d'un réseau de valuations [Kohlas 1993b, Shenoy 1993], sur lequel on peut réaliser différentes interrogations. Pour la convivialité du système, une interface graphique a été ajoutée à l'environnement. L'utilisateur peut ainsi manipuler et visualiser beaucoup plus aisément les structures de données engendrées.

L'environnement de programmation choisi fut celui de Lisp, plus concrètement celui de Macintosh Common Lisp, Version 2.0, abrégé MCL 2.0. Les avantages de ce langages sont évidents en ce qui concerne la problématique à résoudre, notamment en raison de l'énorme richesse des fonctions offertes en standard. Nous avons également pu disposer du module TEPI (*Theory of Evidence Programming Interface*) [Haenni 1995] —qui a fourni à DIG les structures de données des réseaux de valuations et des des formules logiques— et du système ABL 1.0 (*Assumption-Based Language*) [Lehmann 1994] —dont le langage d'interrogation a été intégré à DIG.

La documentation se compose de cinq chapitres. Après avoir mis au clair quelque notions mathématiques, nous expliquerons l'algorithme de génération d'impliqués premiers de Ngair. L'implémentation choisie sera exposée et son efficacité montrée par divers exemples. Ensuite vient la présentation de DIG qui est l'environnement programmé donnant les outils nécessaires à la description et au diagnostique de circuits digitaux défectueux. Enfin l'implémentation concrète du système DIG est décrite. En annexe se trouvent la description formelle de la grammaire de DIG, la bibliographie et le code complet du système.

2 La génération d'impliqués premiers

Le présent chapitre expose un algorithme de génération d'impliqués premiers proposé par Ngair [Ngair 1992]. La section 2.1 pose les bases théoriques qui sont le fondement de cet algorithme; elle y traite notamment toutes les notions concernant les espaces convexes. Les complexités de différentes opérations seront abordées. La section 2.2 expose l'algorithme en donnant quelques optimisations possibles. Une implémentation en LISP y est proposée.

L'exposé ne serait pas complet sans une série de tests sur différents problèmes. Dans la section 2.3, l'implémentation concrète de *GEN-PI*, intégrée à DIG, est comparée à un autre algorithme de génération d'impliqués premiers défini dans TEPI. La structure de donnée de base reste donc la même. Deux problèmes sont traités: l'un, artificiel, est très bien résolu par GEN-PI; l'autre, une description de circuit digital défectueux, permet de traiter un problème classique. Nous serons amenés à tirer diverses conclusions de ces données empiriques.

2.1 Bases théoriques

L'élaboration de l'algorithme de Ngair pour la génération d'impliqués premiers nécessite certaines notions mathématiques qui sont exposées dans le présent chapitre. Nous nous inspirons directement de la thèse de Ngair dans [Ngair 1992] qui offre une approche beaucoup plus complète et rigoureuse. Nous employons dans tout notre exposé différentes notions de la logique propositionnelle; pour plus de détails, on peut se référer à [Chang 1973].

2.1.1 Quelques notions

Un ensemble partiellement ordonné (ou *poset*) est un ensemble P lié à une relation binaire □ réflexive, antisymétrique et transitive. Il existe, par rapport à cette relation binaire, le plus grand minorant ou infimum (*greatest lower bound*, abrégé *inf* ou *meet*, et écrit 3) et le plus petit majorant ou le supremum (*least upper bound*, abrégé *sup* ou *join*, et écrit £).

Un treillis est un poset dont chaque sous-ensemble fini possède un supremum et un infimum avec des équations décrivant comment les *meet* et *join* se comportent.

Définition: L'ensemble des parties (*powerset*) d'un ensemble X , noté P*(X)*, est l'ensemble de tous les sous-ensembles de X ordonnés selon l'inclusion.

L'ensemble des parties d'un ensemble X est un exemple typique de treillis.

Définition: On dit qu'une fonction f d'un poset (P, □) à un autre poset (P', □') est *monotone* lorsque, pour $x, y \in P$ et $f(x), f(y) \in P'$, la condition suivante est remplie:

$$si\ x \preceq y\ ,\ alors\ f(x) \preceq'\ f(y).$$

Pour une fonction f de P vers P', on dit qu'un $x \in P$ est un *point fixe* de f si $f(x) = x$.

Parmi les fonctions monotones, on en trouve une classe importante qui sont d'un grand intérêt pour notre étude: les opérateurs de fermeture.

Définition: Soient un treillis L et une fonction $f : L \to L$. f est un *opérateur de fermeture* (*closure operator*) s'il satisfait aux propriétés suivantes:

$$C1.\ x \preceq f(x)\ pour\ tout\ x \in L;$$
$$C2.\ f(f(x)) = f(x),\ et$$
$$C3.\ f(x) \preceq f(x),\ si\ x \preceq y.$$

Il est simple de voir que *C2* indique que *f(x)* est un point fixe de *f*.

Définition: Soit un poset P; un sous-ensemble $C \subseteq P$ est dit *fermé inférieurement* (*downward closed)* si $p \in C$ et $p' \preceq p$ implique que $p' \in C$. De plus, pour un sous-ensemble $S \subseteq P$, la *fermeture inférieure* de S est l'ensemble

$$\downarrow S = \{p \in P \mid \exists p' \in S,\ p \preceq p'\}.$$

Dorénavant, nous ne considérerons que des treillis définis comme ensembles des parties.

2.1.2 Espaces convexes

La présente section introduit aux espaces convexes; il s'agit d'une notion qui est analogue au domaine de la géométrie. Nous partons d'un poset P et d'un sous-ensemble $C \subseteq P$. Le sous-ensemble C est un *espace convexe* si

$$p1 \preceq p \preceq p2 \text{ et } p1, p2 \in C, \text{ alors } p \in C.$$

Ce qui rend les espaces convexes particulièrement intéressants, c'est que la théorie des opérateurs de fermeture s'y applique entièrement. Les espaces convexes forment un treillis dont l'ordre est défini par leurs limites (*fringes*), ce qui implique un gain considérable dans la manière de les représenter. La fonction $c : \mathrm{P}(P) \rightarrow \mathrm{P}(P)$ définie par

$$c(C) = \{p \in P \mid \exists p1, p2 \in C, \ t.q. \ p1 \preceq p \preceq p2\}$$

est un opérateur de fermeture.

Définition: Soit un $C \in \mathrm{P}(P)$. Nous définissons *MIN* et *MAX* de la manière suivante (il s'agit en fait des ensembles limites de *C)*:

$$MIN(C) = \{p \in C \mid \forall p' \in C, p' \preceq p \Rightarrow p' = p\};$$

$$MAX(C) = \{p \in C \mid \forall p' \in C, p' \succeq p \Rightarrow p' = p\}.$$

Il faut encore noter que C peut contenir jusqu'à une infinité d'éléments, représentés par les sous-ensembles *MIN(C)* et *MAX(C)*. On peut ainsi se rendre compte que tout sous-ensemble représenté par des ensembles limites est un espace convexe.

A partir de maintenant, nous ne considérons que la relation d'ordre $\preceq$ pour des ensembles C et C' définie par

$$C \preceq C' \Leftrightarrow C \subseteq C'.$$

2.1.3 Algorithme de *join* et *meet* pour les espaces convexes

Les espaces convexes que nous allons considérer sont des espaces pouvant être représentés de manière finie; il s'agit des ensembles fermés supérieurement et inférieurement. Ce sont en fait deux spécialisations d'espaces convexes.

Les ensembles fermés supérieurement et inférieurement d'un poset *P* sont respectivement les points fixes des opérations de fermeture ? et |.

L'intérêt particulier que nous portons à ces espaces convexes est dû au fait qu'ils permettent un gain considérable dans la manière de les représenter. La complexité de mémoire et de temps pour les opérations réalisées sur ces espaces diminue fortement. De plus, certaines de leurs propriétés en augmentent de beaucoup l'intérêt. Nous les exposons dans deux lemmes, démontrés par Ngair.

Lemme 2.1: Soit un poset *P*; chaque collection des ensembles fermés supérieurement $\uparrow_P = \{\uparrow S \mid S \subseteq P\}$ et des ensembles fermés inférieurement $\downarrow_P = \{\downarrow S \mid S \subseteq P\}$ forme un sous-treillis des espaces convexes de *P*. De plus, pour ces sous-treillis, les opérations de *meet* et *join* sont respectivement égales à l'intersection et à l'union.

Le lemme suivant nous fournit également des informations très intéressantes:

Lemme 2.2: Soient deux ensembles fermés inférieurement *C1* et *C2* dans un poset *P*; nous avons:

$$MAX(C1 \wedge C2) = MAX(\{p1 \cup p2 \mid p1 \in L1,\ p2 \in L2\}),$$

$$MAX(C1 \vee C2) = MAX(L1 \cup L2),$$

avec $L1 = MAX(C1)$ et $L2 = MAX(C2)$.

Ces propriétés, comme nous le verrons plus loin, vont simplifier grandement les opérations sur les espaces convexes qui nous intéressent.

2.1.4 Complexité

Nous aimerions aborder la complexité de différentes opérations concernant les espaces convexes. Nous n'allons pas en donner une démonstration; pour cela il faut se référer à [Ngair 1992].

Nous partons d'un poset *P*. Calculer le *MAX* ou le *MIN* d'un $X \subseteq P$ dépend directement de la cardinalité de *X*; la complexité équivaut à $O(|X|^2)$, ce qui est considérable. C'est d'ailleurs cette opération qui conditionne le plus les différentes applications; d'où la nécessité d'optimiser les structures de données. Nous verrons, pour l'environnement développé dans le cadre de ce travail, quelle

Opération	Complexité
meet($\wedge$)	$O\left(c*\left(\text{maximum}\left(\|S1\|*\|S2\|,\|G1\|*\|G2\|\right)\right)^2\right)$
join ($\vee$)	$O\left(c*\left(\text{maximum}\left(\|S1\|+\|S2\|,\|G1\|+\|G2\|\right)\right)^2\right)$
subset ($\preceq$)	$O\left(c*\left(\|S1\|*\|S2\|+\|G1\|*\|G2\|\right)\right)$

Table 2.1: Complexité de différentes opérations sur les espaces convexes.

structure a été utilisée et quels avantages celle-ci a apportés à l'efficacité de calcul.

Nous donnons la complexité pour deux espaces convexes *C1* et *C2,* avec des limites *S1, G1* et *S2, G2*. Les espaces $C_\wedge$ et $C_\vee$ sont les résultats du *meet* et du *join* de *C1* et *C2*; ils sont convexes. Leur limites sont respectivement $S_\wedge, G_\wedge$ et $S_\vee, G_\vee$. Nous supposons que la complexité de l'opération □ est de $O(c)$.

La table 2.1 nous montre quelle sont les complexités des opérations *meet*, *join* et *subset* pour des espaces convexes.

2.2 La génération d'impliqués premiers

Cette section aborde la réalisation de *GEN-PI*, la nouvelle méthode de génération d'impliqués premiers décrite par Ngair. Après avoir clarifié la notion d'impliqué premier, nous allons exposer la formulation formelle de la méthode. Nous présentons un algorithme de l'opération centrale de *GEN-PI*: l'opération *PHI*. Nous en donnons des optimisations et une implémentation concrète. Ensuite vient l'exposé à proprement dit de *GEN-PI* également avec des optimisations, suivi de la présentation de l'implémentation.

L'algorithme de Ngair utilise directement l'opération *F* définie à la section 2.2.2. L'algorithme *PHI* calcule, à partir d'un ensemble d'impliqués premiers et d'une formule DNF, le nouvel ensemble d'impliqués premiers. Comme nous le verrons, pour obtenir l'ensemble complet des impliqués premiers, il faudra appliquer *F* séquentiellement.

2.2.1 Impliqués et impliquants premiers

Notation: Soit θ une formule DNF (forme normale disjonctive); $\mathcal{E}(\theta)$ est l'ensemble des conjonctions de θ. Pour un clause disjonctive ϑ, $\overline{\vartheta}$ est la conjonction des négations des littéraux de ϑ.

A noter qu'une clause disjonctive est un cas spécial d'une formule DNF: chacune de ses conjonctions ne contient qu'un seul littéral.

Définitions:

• Soit Θ un ensemble de formules propositionnelles. Une clause disjonctive ϑ est un impliqué (*implicate*) de Θ si $\Theta \models \vartheta$.

• ϑ est un impliqué premier (*prime implicate*) s'il n'y a pas d'autre clause disjonctive ϑ', tel que $\mathcal{E}(\vartheta') \subset \mathcal{E}(\vartheta)$ et $\Theta \models \vartheta'$.

• Une conjonction $\overline{\vartheta}$ est un impliquant (*implicant*) de Θ, si ϑ est un impliqué de $\neg\Theta$. $\overline{\vartheta}$ est un impliquant premier (*prime implicant*), si ϑ est un impliqué premier de $\neg\Theta$.

2.2.2 L'opération F

Nous définissons un opérateur de fermeture d'une importance centrale. Nous verrons comment l'intégrer dans la génération d'impliqués premiers.

Définition: Soit un treillis $(P, \preceq)$. Pour un sous-ensemble $T \in \mathrm{P}(P)$ l'opération Φ_T est définie sur des ensembles fermés inférieurement $C \in \mathrm{P}(P)$:

$$\Phi_T(C) = \{p \in P \mid \forall t \in T,\ p \wedge t \in C\}.$$

On peut montrer que $\Phi_T(C)$ est à son tour fermé inférieurement et que Φ_T est un opérateur de fermeture [Ngair 1992].

Ce qui rend F intéressant, ce sont les propriétés qu'il possède et qui sont démontrées par Ngair.

1. $\Phi_A \circ \Phi_B = \Phi_T$ où $T = \left\{t \in P \;\middle|\; \exists t_A \in A,\ \exists t_B \in B,\ t = t_A \wedge t_B\right\}$.

2. $\Phi_A \circ \Phi_B = \Phi_B \circ \Phi_A$.

3. Soient $\Phi_{T_1},\ldots,\Phi_{T_n}$ et un ensemble C fermé inférieurement, $\Phi = \Phi_{T_1}\circ\cdots\circ\Phi_{T_n}(C)$ est le plus petit point fixe commun de chaque Φ_{T_i}, $1 \le i \le n$, sur C.

2.2.3 L'algorithme de génération d'impliqués premiers

L'algorithme développé par Ngair est de nature duale. Cela signifie qu'il est apte, en vertu des lois de de Morgan, de générer des impliqués premiers à partir d'une conjonction de DNF, ou des impliquants premiers à partir d'une disjonction de CNF. Il accepte des clauses contenant des littéraux opposés, ce que la plupart des algorithmes de génération d'impliqués premiers ne permettent pas.; ces clauses sont alors interprétées comme des tautologies pour des impliqués premiers et des inconsistances pour des impliquants premiers. La description de l'algorithme sera celle de la génération d'impliqués premiers; Θ est donc considéré comme une conjonction de formules DNF.

Nous supposons tout d'abord un treillis P qui est l'ensemble des parties de l'ensemble des littéraux $L = \{x_1, \neg x_1, \ldots, x_n, \neg x_n\}$. Les symboles $x_1, x_2, \ldots, x_n$ sont en nombre fini et appartiennent à Θ. L'ensemble $\{\{x_i, \neg x_i\} \mid 1 \le i \le n\}$ définit la fermeture inférieure C_P.

Les éléments de P, que l'on nomme *environnements*, sont ordonnés par l'inverse de l'inclusion, c'est-à-dire que $\preceq$ est identique à $\supseteq$. Un environnement $\{l_{i_1},\ldots,l_{i_k}\}$, avec $l_{i_1},\ldots,l_{i_k} \in L$, est interprété comme une conjonction de littéraux $l_{i_1}\wedge\cdots\wedge l_{i_k}$. De là, on peut déduire que C_P représente l'ensemble de toutes les conjonctions inconsistantes de P.

Soit $p = \{l_{i_1},\ldots,l_{i_k}\}$ un environnement de P. Si p est inconsistant, alors $l_{i_1}\wedge\cdots\wedge l_{i_k} \models \bot$, ce qui implique que $\models \neg l_{i_1}\vee\cdots\vee\neg l_{i_k}$, c'est-à-dire que $\bar{p}$ est une tautologie.

On dit qu'un environnement p est inconsistant avec Θ, si $\Theta \wedge p \models \bot$.

Lemme 2.3: Un environnement p est inconsistant avec Θ si et seulement si $\bar{p}$ est un impliqué de Θ. De plus, p est maximal si et seulement si $\bar{p}$ est un impliqué premier.

On voit simplement que pour trouver les impliqués premier*s* de Θ, il suffit de trouver les environnements maximaux inconsistants avec Θ, qui sont en fait

les négations des impliqués premiers. Nous rappelons qu'un environnement maximal équivaut au plus petit ensemble possible.

Pour expliquer la méthode de Ngair de générations d'impliqués premiers, nous procédons par étapes. Considérons tout d'abord Θ avec une seule DNF $\theta = f_1 \vee \cdots \vee f_t$. Chaque f_i est une conjonction. ϑ est un impliqué de θ, seulement si $f_i \wedge \overline{\vartheta} \models \perp$ pour chaque $f_i \in \mathcal{E}(\theta)$. $f_i \wedge \overline{\vartheta} \models \perp$ est vrai dans ce cas uniquement si f_i et $\overline{\vartheta}$ (qui sont des conjonctions) possèdent une paire de littéraux opposés: *p* et ¬*p*. L'ensemble des impliqués premiers devient donc

$$\Phi_\theta(C_P) = \left\{\overline{\vartheta} \ : \ f \wedge \overline{\vartheta} \in C_P, \ \textit{pour chaque } f \textit{ dans } \mathcal{E}(\theta)\right\}.$$

Mais si Θ est une conjonction de plusieurs DNF (par exemple $\theta_1 \wedge \cdots \wedge \theta_k$), alors ϑ est un impliqué de Θ si et seulement si $\Theta \cup \left\{\overline{\vartheta}\right\} \models \perp$. Il est alors possible de démontrer le théorème central suivant:

Théorème 2.4: Pour une conjonction de DNF $\Theta = \theta_1 \wedge \cdots \wedge \theta_k$, ϑ est un impliqué de Θ si et seulement si $\overline{\vartheta}$ est un élément de $C'_P = \Phi_{\varepsilon(\theta_k)} \circ \cdots \circ \Phi_{\varepsilon(\theta_1)}(C_P)$.

Ce théorème nous montre qu'il suffit d'appliquer séquentiellement l'opération *F* pour obtenir l'ensemble des impliqués. Le théorème 2.5 nous indique en plus une manière de calculer *F*, ce qui rend très intéressante la méthode.

Théorème 2.5: Soit un treillis *P*; pour un ensemble *C* fermé inférieurement avec $MAX(C) = \{s_1, \ldots, s_m\}$, et pour $T = \{t_1, \ldots, t_n\}$, nous avons:

$$\Phi_T(C) = \bigwedge_{1 \le j \le n} \bigvee_{1 \le i \le m} E_j^i,$$

avec $E_j^i = \left\{p \in P \ \middle| \ p \wedge t_j \preceq s_i\right\}$. De plus, E_j^i est un ensemble fermé inférieurement avec une limite supérieure unique: $e_j^i = s_i - t_j$.

Il reste néanmoins un problème: nous n'obtenons comme résultat que l'ensemble des impliqués, et non des impliqués premiers. Cela signifie que cet ensemble est d'ordinaire très grand. Pour pouvoir traiter des ensembles maximaux, on peut utiliser le lemme 2.2 et ainsi déduire que

$$MAX(\Phi(C)) = \bigwedge_j MAX\left(\bigcup_i \left\{e_j^i\right\}\right).$$

La méthode pour obtenir les impliqués premiers est maintenant complète. Il reste à décrire dans les prochaines sections quelques optimisations de *F* et de la méthode de génération d'impliqués premiers, et à présenter les algorithmes respectifs de ces opérations.

2.2.4 Quelques optimisations de F. Les algorithmes *PHI* et *P-PHI*

D'importantes optimisations peuvent être apportées à l'opération *F*. Le point de départ est le fait qu'on ne doit pas remplir obligatoirement toute la matrice. On peut se restreindre à certaines conditions qui sont les suivantes:

1. Condition *rowdone*: S'il existe un i' et un j' tel que $s_{i'} \succeq t_{j'}$, alors $s_{i'} - t_{j'} = \{\}$, ce qui implique que $\bigvee_i E_j^i = \{\{\}\}$. La colonne $j=j'$ peut être ignorée dans le calcul de $\bigwedge_j \bigvee_i E_j^i$, si $\{\}$ est généré dans toute la colonne.

2. Condition *matrixdone*: S'il existe un j' tel que pour tous les i, $s_i - t_{j'} = s_i$, alors $\Phi_T(C) = C$. Il n'y pas besoin de continuer les calculs dans la matrice.

3. Condition *emptycell*: Si, pour un i' et un j', $s_{i'} - t_{j'} = s_{i'}$, c'est-à-dire $e_{j'}^{i'} = s_{i'}$, alors $E_{j'}^{i'} \subseteq C$. C'est clair que $E_{j'}^{i'}$ ne va pas apporter de nouvelles informations pour le calcul de $\Phi_T(C)$. On ne va donc pas le prendre en compte.

Avec ces optimisations, nous pouvons décrire l'algorithme *PHI*. En entrée, il accepte l'ensemble *S* maximal qui est celui des impliqués premiers de départ, et l'ensemble *T* des conjonctions (il s'agit en fait d'une formule DNF).

```
FUNCTION PHI(S,T)
Initialize
  for 1 • j • n and 1 • i • m
    e[j, i] := NIL;
    Y[j] := NIL;
    rowdone(j) := FALSE;
    matrixdone := FALSE;
    Result := NIL;
  end for;
repeat for each tj in T
  repeat for each si in S
    if rowdone-condition(i,j) then
      rowdone(j) := TRUE;
```

```
    elsif not emptycell(i,j) then
      e[j,i] := {s_i - t_j};
    end if;
  until rowdone(j);
  if matrixdone-condition then
    matrixdone := TRUE;
  end if;
until matrixdone;
if matrixdone then
  return Result := S;
end if;
Y[j] := MAX(″_i e[j,i]), 1•j•n and not rowdone(j);
return Result := #_j Y[j]£S, not rowdone(j);
end function PHI.
```

Il est encore possible d'apporter quelques optimisations supplémentaires pour le cas où nous savons que les t_j sont tous disjoints entre eux, c'est-à-dire si $t_{j_1} \cap t_{j_2} = \{\}$, pour $1 \le j_1 \ne j_2 \le m$.

4. Si nous avons, pour i', j_1, et j_2, $e_{j_1}^{i'} \wedge e_{j_2}^{i'} = \left(s_{i'} - t_{j_1}\right) \wedge \left(s_{i'} - t_{j_2}\right) = s_{i'} - \left(t_{j_1} \cap t_{j_2}\right) = s_{i'}$, c'est-à-dire $E_{j_1}^{i'} \wedge E_{j_2}^{i'} \subseteq C$, alors $\wedge_j E_j^{\sigma(j)}$ peut donner de nouveaux environnements uniquement si σ est bijectif (*one-to-one*).

5. Soient i', j_1, et j_2 avec $t_{j_1}, t_{j_2} \succeq s_{i'}$. Pour un ensemble $A = \left\{E_j^{\sigma(j)} \mid 1 \le j \le n\right\}$ avec σ bijectif, alors au moins $E_j^{\sigma(j_1)}$, ou $E_j^{\sigma(j_2)}$ n'appartient pas à A. On suppose que c'est $E_j^{\sigma(j_1)}$ qui n'est pas dans A. S'il existe j_3 avec $\sigma\left(j_3\right) = i'$, alors $e_{j_3}^{i'} \preceq t_{j_1}$, puisque t_{j_1} et t_{j_3} sont disjoints. Soit $p = e_{j_1}^{\sigma(j_1)} \wedge e_{j_3}^{i'}$; étant donné que $e_{j_1}^{\sigma(j_1)} = s_{\sigma(j_1)} - t_{j_1}$, nous avons $p \preceq t_{j_1} \cup \left(s_{\sigma(j_1)} - t_{j_1}\right) = s_{\sigma(j_1)}$. Donc $\wedge_{1 \le j \le n} e_j^{\sigma(j)} \preceq s_{\sigma(j_1)}$, c'est-à-dire $\wedge A \subseteq C$. Cela signifie que l'on peut ignorer chaque A qui contient $E_j^{i'}$ pour un j quelconque; cela signifie que, pour $1 \le j \le n$, $E_j^{i'}$ peut être ignoré.

En résumé, l'on peut dire que, pour le cas où les éléments de la disjonction sont disjoints et forment des singletons, nous ne devons considérer que les $\wedge_{1 \le j \le n} E_j^{\sigma(j)}$ qui satisfont aux critères suivants:

- σ est bijectif (*one-to-one*);
- $\forall j, s_{\sigma(j)} - t_j \ne s_{\sigma(j)}$;

- $\forall j', j' \neq j, s_{\sigma(j')} - t_j = s_{\sigma(j)}$.

De plus, pour les $\wedge_{1 \leq j \leq n} E_j^{\sigma(j)}$ qui nous intéressent, nous ne devons ajouter que les environnements $\wedge_j e_j^{\sigma(j)} = \bigcup_j \left(s_{\sigma(j)} - t_j \right)$.

Si nous prenons en compte la cinquième remarque, nous avons un nouvel algorithme optimisé de F où T est une clause de n littéraux, et S l'ensemble maximal, c'est-à-dire l'ensemble des impliqués premiers de départ. Cet algorithme est nommé *P-PHI*.

```
FUNCTION P-PHI(S,T)
Initialize
  for 1 • j • n and 1 • i • m
    e[j,i] := NIL;
    Y[j] := NIL;
    rowdone(j) := FALSE;
    matrixdone := FALSE;
    Result := NIL;
  end for
repeat for each s_i in S
  if exactly one literal b of s_i occurs as t_j in T then
    if rowdone-condition(i,j) then
      rowdone(j) := TRUE;
    else
      e[j,i] := s_i - {b};
    end if;
  end if;
  if matrixdone-condition then
    matrixdone := TRUE;
  end if;
end repeat
if matrixdone then
  return Result := S;
end if;
Y[j] := MAX(″_i e[j,i]), 1•j•n and not rowdone(j);
return Result := #_j Y[j]£S, not rowdone(j);
end function P-PHI.
```

Il nous semble encore intéressant d'indiquer les complexités des algorithmes proposés. Nous supposons que la complexité d'une comparaison d'ensemble est

de $O(c)$. Ngair démontre que pour une formule DNF $T = \{t_1, \ldots, t_k\}$ et $S = \{s_1, \ldots, s_n\}$, la complexité, pour le pire des cas, de *PHI(S,T)* est de $O(c*n^{2k})$; celle de *P-PHI(S,T)* est de $O(c*(n/k)^{2k})$.

2.2.5 Implémentation de *PHI* et de *P-PHI*

Le langage de programmation utilisé est Lisp. Si des doutes apparaissent durant la lecture du code, on peut consulter [Steele 1992], le manuel de référence, deuxième édition.

En tenant compte de toutes les optimisations présentées à la section 2.2.4, il a été possible de réaliser une implémentation très efficace des algorithmes *PHI* et *P-PHI*. Pour l'exposé de la réalisation des algorithmes en question, nous supposons que les opérations de *meet* et de *join* ont déjà été implémentées. Nous indiquerons plus loin différentes méthodes pour réduire fortement la complexité de *meet*.

La fonction implémentée, que nous appelons `nf-PHI`, doit tout d'abord être en mesure de reconnaître les cas où les éléments de la disjonction sont disjoints et forment des singletons. Pour cela nous définissons les fonction `nf-disjoint?` et `nf-only-literals?` qui sont des prédicats sur la disjonction `T-nf`.

```
(defun nf-disjoint? (T-nf)
  (loop for ls1 in T-nf
        for x = 1 then (1+ x)
        always (loop for ls2 in T-nf
                     for y = 1 then (1+ y)
                     always (if (ls-empty? (ls-intersection ls1 ls2))
                                T
                                (and (= x y)
                                     (ls-equal? ls1 ls2))))))


(defun nf-only-literals? (T-nf)
  (loop for ls in T-nf
        always (= (ls-member ls)
                  1)))
```

Selon la disjonction présente, l'on peut appeler les fonctions `nf-PHI-for-lit-and-dis` (pour *P-PHI*) ou `nf-PHI-normal` (pour *PHI*).

```
(defun nf-PHI (S-nf T-nf)
  (if (and (nf-only-literals? T-nf)
           (nf-disjoint? T-nf))
    (nf-PHI-for-lit-and-dis S-nf T-nf)
    (nf-PHI-normal S-nf T-nf)))
```

Les deux algorithmes implémentés possèdent une structure analogue. Les itérations sur la disjonction T-nf et sur l'ensemble actuel des impliqués premiers sont réalisées au moyen de l'utilité du `loop`. Selon les différentes conditions, on accumule ou non les résultats des différences entre les éléments S-nf et T-nf (`ls-difference si-ls tj-ls`). Sur ces listes, on réalise un *join*, et sur la liste résultante un *meet*.

```
(defun nf-PHI-for-lit-and-dis (S-nf T-nf)
  (let* ((matrixdone T)
         (hnf2 (loop for tj-ls in T-nf
                     for hnf1 =
                     (progn
                       (setf matrixdone T)
                       (loop for si-ls in S-nf
                             for cell-ls = (ls-difference si-ls tj-ls)
                             when (ls-row-done? cell-ls)
                               do (setf matrixdone NIL)
                               and return '()
                             else
                               when (ls-empty-cell? si-ls cell-ls)
                                 do (setf matrixdone (and matrixdone T))
                               else
                                 do (setf matrixdone NIL)
                                 and
                                 when (ls-subset? tj-ls si-ls)
                                   collect (list cell-ls)))
                     when matrixdone return matrixdone
                     else
                     when hnf1
                     collect (hnf-join hnf1))))
    (if matrixdone
      S-nf
      (if hnf2
        (nf-special-join S-nf (hnf-meet hnf2 T-nf S-nf))))))


(defun nf-PHI-normal (S-nf T-nf)
  (let* ((matrixdone T)
         (hnf2 (loop for tj-ls in T-nf
                     for hnf1 =
                     (progn
                       (setf matrixdone T)
                       (loop for si-ls in S-nf
                             for cell-ls = (ls-difference si-ls tj-ls)
                             when (ls-row-done? cell-ls)
                               do (setf matrixdone NIL)
                               and return '()
                             else
                               when (not (ls-empty-cell? si-ls cell-ls))
                                 do (setf matrixdone NIL)
                                 and collect (list cell-ls)
                               else
                                 do (setf matrixdone (and matrixdone T))))
                     when matrixdone return matrixdone
                     else
                       when hnf1
                         collect (hnf-join hnf1))))
    (if matrixdone
      S-nf
      (if hnf2
        (nf-special-join S-nf (hnf-meet hnf2 T-nf S-nf))))))
```

Une implémentation de `nf-PHI-for-lit-and-dis` est également possible à l'aide d'un tableau. On remplit uniquement les éléments du tableau `A` qui sont nécessaires au calcul final. Ensuite, à l'aide de différents index et d'un vecteur `rowdone` qui contrôle la condition du même nom, on peut accumuler les éléments pertinents que l'on aura inscrits dans le tableau `A` et réaliser par après les fonctions *join* et *meet*.

```
(defun nf-PHI-for-lit-and-dis (S-nf T-nf)
  (let* ((X-nf T-nf)
         (xi (length S-nf))
         (yj (length T-nf))
         (A (make-array (list xi yj)))
         (Y (make-array yj))
         (rowdone (make-array yj))
         (matrixdone T))
    (loop for tj-ls in T-nf
          for j from 0 to (1- yj)
          do (loop for si-ls in S-nf
                   for i from 0 to (1- xi)
                   for cell-ls = (ls-difference si-ls tj-ls)
                   when (ls-row-done? cell-ls)
                     do (progn (setf matrixdone NIL)
                               (setf (aref rowdone j) T))
                   else
                     when (ls-empty-cell? si-ls cell-ls)
                       do (setf matrixdone (and matrixdone T))
                     else
                       do (setf matrixdone NIL)
                       and
                       when (ls-subset? tj-ls si-ls)
                         do (setf (aref A i j) (list cell-ls))
                   until (aref rowdone j))
          until matrixdone)
    (if matrixdone
      S-nf
      (loop
        for j from 0 to (1- yj)
        when (not (aref rowdone j))
        do (setf (aref Y j)
                 (hnf-join (loop for i from 0 to (1- xi)
                                 for elt = (aref A i j)
                                 when elt collect elt)))

        finally (let ((res-nf (hnf-meet
                               (loop for j1 from 0 to (1- yj)
                                     when (not (aref rowdone j1))
                                       collect (aref Y j1))
                               X-nf
                               S-nf)))
                  (if res-nf
                    (return (nf-special-join S-nf res-nf))
                    (return S-nf)))))))
```

2.2.6 L'algorithme *GEN-PI*

Nous définissons dans cette section l'opération *GEN-PI* qui intègre l'algorithme de génération d'impliqués premiers que nous allons exposer. Pour un

ensemble de DNF $\Theta = \{\theta_1, \ldots, \theta_k\}$, nous admettons que S_Θ est l'ensemble des symboles de Θ.

Nous rappelons le théorème 2.4 qui dit que, pour une conjonction de DNF $\Theta = \theta_1 \wedge \cdots \wedge \theta_k$, ϑ est un impliqué de Θ si et seulement si $\overline{\vartheta}$ est un élément de $C_\Theta = \Phi_{T_k} \circ \cdots \circ \Phi_{T_1}(C_P)$, avec $T_i = \varepsilon(\theta_i)$ pour $1 \le i \le k$, où C_P a été défini par la fermeture inférieure $\{\{x, \neg x\} \mid x \in S_\Theta\}$. Ce théorème montre qu'il suffit d'appliquer séquentiellement l'opération *F* pour obtenir l'ensemble des impliqués.

Nous supposons que PI_Θ est l'ensemble des impliqués premiers de Θ. Un problème se pose lorsque nous avons une nouvelle DNF θ. Nous devons calculer le nouvel ensemble d'impliqués premiers $PI_{\Theta \cup \{\theta\}}$ à partir de PI_Θ, ce qui peut se faire en appliquant le théorème 2.4. Nous pouvons alors différentier deux cas. Si θ introduit de nouveaux symboles propositionnels ne se trouvant pas encore dans S_Θ, alors $C_{\Theta \cup \{\theta\}} = C_\Theta$, donc $C_{\Theta \cup \{\theta\}} = \Phi_T(C_P)$ avec $T = \varepsilon(\theta)$. Si, par contre, il y a des nouveaux symboles, il faut les introduire dans S_Θ. Mais Φ_{T_i}, avec $1 \le i \le k$, n'a aucun effet sur ces nouveaux symboles; nous avons donc

$$\Phi_{T_k} \circ \cdots \circ \Phi_{T_1}(C_{\Theta \cup \{\theta\}}) = \Phi_{T_k} \circ \cdots \circ \Phi_{T_1}(C_P) \cup X,$$

où X est la fermeture inférieure définie par l'ensemble $\{\{x, \neg x\} \mid x \in S_{\Theta \cup \{\theta\}} - S_\Theta\}$. On voit ainsi qu'on peut facilement calculer $C_{\Theta \cup \{\theta\}}$ progressivement à partir de C_Θ.

Nous obtenons l'algorithme *GEN-PI*, où *S* est l'ensemble croissant des impliqués premiers.

```
FUNCTION GEN-PI(S,T)
for each propositional symbol x in T
  if x is not encountered earlier then
    add {x,¬x} to S;
  end if;
end for;
S := PHI(S,T);
end function GEN-PI.
```

Nous illustrons *GEN-PI*. Considérons les formules suivantes:

$A_3 \rightarrow A_1$;

$A_4 \rightarrow A_2$;

$$(A_1 \wedge A_2) \vee (A_3 \wedge A_4).$$

Nous obtenons, après avoir traité les deux premières clauses, l'ensemble suivant de négations d'impliqués premiers:

$$S_1 = \{A_1, \overline{A}_3\};\ S_2 = \{A_2, \overline{A}_4\};\ S_3 = \{A_1, \overline{A}_1\};$$

$$S_4 = \{A_2, \overline{A}_2\};\ S_5 = \{A_3, \overline{A}_3\};\ S_6 = \{A_4, \overline{A}_4\}.$$

Pour la dernière formule, $t_1 = \{A_1, A_2\}$ et $t_2 = \{A_3, A_4\}$. La dernière opération de F peut alors être illustrée par la table 2.2.

e_j^i	s_1	s_2	s_3	s_4	s_5	s_6	$\vee$
t_1	$\{\overline{A}_3\}$	$\{\overline{A}_4\}$	$\{\overline{A}_1\}$	$\{\overline{A}_2\}$	$\{\ \}$	$\{\ \}$	$\{\{\overline{A}_1\}, \{\overline{A}_2\}, \{\overline{A}_3\}, \{\overline{A}_4\}\}$
t_2	$\{\ \}$	$\{\ \}$	$\{\ \}$	$\{\ \}$	$\{\overline{A}_3\}$	$\{\overline{A}_4\}$	$\{\{\overline{A}_3\}, \{\overline{A}_4\}\}$
New negated prime implicates ($\wedge$)							$\{\{\overline{A}_3\}, \{\overline{A}_4\}\}$

Table 2.2: Une illustration de l'opération *F*.

La nouvelle liste d'impliqués premiers devient ainsi

$$\{A_1 \vee \overline{A}_1,\ A_2 \vee \overline{A}_2,\ A_3,\ A_4\}.$$

Nous présentons une possible implémentation où `hnf-gen-PI` constitue l'algorithme (hnf est l'abréviation de *hyper-normal form* qui est la forme générique des conjonctions de DNF et des disjonctions de CNF; pour plus de précision, voir la section 4.2.4).

```
(defun nf-gen-PI (S-nf T-nf)
  (let* ((S-bs-symbols (ls->bs (ls-symbols S-nf)))
         (T-bs-symbols (ls->bs (ls-symbols T-nf)))
         (tautological-PI-nf
          (nf-tautological-list
           (bs-get-single-elt
            (bs-difference T-bs-symbols S-bs-symbols)))))
    (nf-PHI (append S-nf tautological-PI-nf) T-nf)))

(defun hnf-gen-PI (S-hnf)
  (loop for nf in (cons NIL S-hnf)
        for Prime-Implicates = nf then
        (nf-GEN-PI Prime-Implicates nf)
        finally (return (nf-complement Prime-Implicates))))
```

Il faut noter que `tautological-PI-nf` est la liste des nouveaux symboles que l'on introduit à l'intérieur de l'ensemble des impliqués premiers déjà existants.

2.2.7 Optimisations de *GEN-PI*

Comme nous l'avons déjà vu, la complexité de *GEN-PI* dépend directement de celle de *PHI* qui équivaut à $O\left(c*(n/k)^{2k}\right)$ ou $O\left(c*n^{2k}\right)$.

Les optimisations principales que l'on peut apporter essaient de réduire directement la complexité de $\wedge$ qui est l'opération la plus critique de *PHI*. Considérons tout d'abord deux ensembles maximums $S_1 = \{p_1,\ldots,p_m\}$ et $S_2 = \{q_1,\ldots,q_n\}$. Le procédé le plus évident et le plus simple consiste à calculer l'ensemble $\{p \cup q \mid p \in S_1,\ q \in S_2\}$ en éliminant par la suite tous les éléments non maximaux. La complexité est de $O\left(c*(m*n)^2\right)$.

Une première optimisation peut être apportée en considérant le cas suivant: s'il existe $p \in S_1$ et $q \in S_2$ tel que $p \preceq q$, alors $S_1 \wedge S_2 = \{q\} \vee \left(S_1 \wedge \left(S_2 - \{q\}\right)\right)$. Cela signifie que l'on peut enlever les éléments de S_1 et de S_2 qui sont subsumés par les éléments de l'autre liste. On réalise ensuite l'intersection sur les deux nouvelles listes suivie de l'union avec les éléments que l'on a enlevés. On réussit ainsi à éliminer la production d'un grand nombre d'éléments que l'on ne considérerait de toute façon pas.

Une autre optimisation peut être réalisée si l'on observe la condition suivante: pour $p \in S_1$ et $q \in S_2$, il faut tester la subsumation *(subsumption*) de $p \cap q$ uniquement par rapport à

$$N = \{p \cap q' \mid q' \in S_2,\ q \neq q'\} \cup \{p' \cap q \mid p' \in S_1,\ p \neq p'\}.$$

Cela signifie que, si l'on considère l'intersection comme produit matriciel entre S_1 et S_2 , il faut tester la maximalité de chaque élément de la matrice uniquement par rapport aux éléments de la même colonne et de la même ligne. On réduit ainsi la complexité de *MAX* de $O\left(\left(|S_1|*|S_2|\right)^2\right)$ à $O\left(\left(|S_1|+|S_2|\right)*|S_1|*|S_2|\right)$.

Il vaut la peine, pour l'intéressé, d'analyser les méthodes qui ont été suivies pour l'implémentation de ces différentes optimisations. Les fonctions `nf-intersection+` et `nf-intersection-with-matrix` peuvent être trouvées dans l'annexe.

2.3 Discussion de l'algorithme de Ngair

L'algorithme *GEN-PI* est très flexible quant à son input: une conjonction de DNF, et non pas simplement une conjonction de clauses. Il est clair qu'il est possible de transformer une conjonction de DNF en une formule CNF; mais la complexité d'une telle opération se révèle très importante. Donc, *GEN-PI* est très adapté pour des problèmes se formulant naturellement en une conjonction de DNF. Et si l'entrée est déjà une CNF, l'algorithme calcule l'ensemble des impliqués premiers de manière tout autant efficace que les autres algorithmes existants. Dans ce dernier cas, chaque DNF est considérée comme une simple clause disjonctive.

Deux exemples vont nous permettre de comparer une implémentation particulière du nouvel algorithme avec un autre algorithme qui a été réalisé à partir de la même représentation des formules logiques, ce qui donne une base de comparaison solide. Cet autre algorithme fait partie intégrante du module TEPI; mais pour traiter les formules DNF, il doit tout d'abord les transformer dans les conjonctions correspondantes. Nous verrons que pour des cas extrêmes, la complexité de temps et de mémoire augmente fortement.

2.3.1 Le problème "m(x)k(y)"

Le premier problème que nous voulons aborder est artificiel. Il produit un nombre exponentiel d'impliqués premiers: le problème "m(x)k(y)" (voir [Ngair 1992]). Il a deux paramètres x et y, avec $x*y+1$ clauses en entrée. On peut le résumer par les clauses suivantes:

$$s_1^1 \Rightarrow a_1;\ \ldots;\ s_x^1 \Rightarrow a_1;$$
$$s_1^2 \Rightarrow a_2;\ \ldots;\ s_x^2 \Rightarrow a_2;$$
$$\ldots$$
$$s_1^y \Rightarrow a_y;\ \ldots;\ s_x^y \Rightarrow a_y;$$
$$\neg a_1 \vee \cdots \vee \neg a_n.$$

L'exemple "m(x)k(y)" produit $(x+1)^y + x*y$ impliqués premiers (sans tautologies).

2.3.2 Un circuit digital avec des composantes défectueuses.

Les circuits digitaux se laissent facilement décrire à l'aide de formules logiques. Nous désirons présenter une modélisation particulière de ces circuits: nous attribuons aux composantes de base (les portes *not*, *and*, *nand*, *or*, *nor*, *xor*, etc.) différents modes de comportements. Nous verrons d'ailleurs par après comment le langage DIG reflète parfaitement cette modélisation.

Prenons une porte *XOR* qui possède trois modes de comportement: a) *ok*: *xor* fonctionne correctement; b) *ab1* (pour *abnormal1*): la composante donne toujours 1 en sortie; c) *ab2*: elle se comporte comme une porte *or*.

Une porte possédant plusieurs modes de comportement peut être très facilement décrite à l'aide d'une conjonction dont chaque élément exprime l'un des comportements possibles. Les symboles *a* et *b* sont les entrées; *c* est la sortie.

$$
\begin{aligned}
&\left[ok \rightarrow (c \leftrightarrow a \oplus b)\right] \\
&\wedge\ [ab1 \rightarrow c] \\
&\wedge\ \left[ab2 \rightarrow (c \leftrightarrow a \vee b)\right] \\
&\wedge\ [ok \oplus ab1 \oplus ab2].
\end{aligned}
$$

Cette formule est équivalente à la conjonction de DNF qui suit.

$$
\begin{aligned}
&\left[\left(\overline{a} \wedge \overline{b} \wedge \overline{c}\right) \vee \left(\overline{a} \wedge b \wedge c\right) \vee \left(a \wedge \overline{b} \wedge c\right) \vee \left(a \wedge b \wedge \overline{c}\right) \vee \overline{ok}\right] \\
&\wedge\ \left[c \vee \overline{ab1}\right] \\
&\wedge\ \left[\left(\overline{a} \wedge \overline{b} \wedge \overline{c}\right) \vee (b \wedge c) \vee (a \wedge c) \vee \overline{ab2}\right] \\
&\wedge\ [ok \oplus ab1 \oplus ab2].
\end{aligned}
$$

Il faut remarquer que la dernière partie de la formule, qui une expression avec un OU exclusif (symbole $\oplus$), signifie que la porte ne peut pas être en deux états à la fois (par exemple en même temps *ok* et *ab1*). Les modes sont donc dépendants les uns des autres. Cette formule *xor* peut être exprimée par une formule DNF:

$$
\begin{aligned}
&(ok \oplus ab1 \oplus ab2) \\
&\equiv \left(\overline{ok} \wedge \overline{ab1} \wedge ab2\right) \vee \left(\overline{ok} \wedge ab1 \wedge \overline{ab2}\right) \vee \left(ok \wedge \overline{ab1} \wedge \overline{ab2}\right).
\end{aligned}
$$

Nous verrons à la section 4.3.2 comment implémenter une telle modélisation à travers la réalisation de `defprimitive`.

Pour tester les algorithmes de génération d'impliqués premiers, nous voulons définir au moyen des portes *and*, *or* et *not* un additionneur de 2 bits avec une retenue en entrée. Un tel additionneur est composé de deux demi-additionneurs et d'une porte *or*. Ces circuits sont illustrés aux figures 2.3 et 2.4. On génère pour chaque porte les impliqués premiers correspondants. Plus une porte possède de modes de comportement différents, plus grand sera le nombre d'impliqués premiers engendrés. Nous allons donc tester ce circuit avec différentes déclarations des portes *and* et *or* qui comporteront un à six modes de comportement.

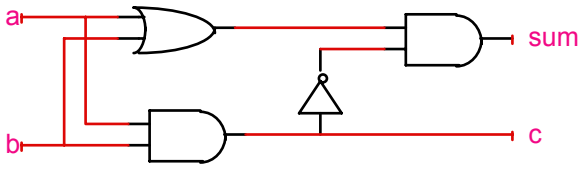


Figure 2.3: Un demi-additionneur.

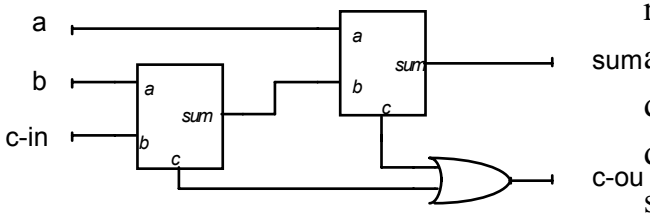


Figure 2.4: Un additionneur.

On se rend compte qu'une description de circuit digital possédant un nombre important de portes peut devenir très élaborée. L'algorithme de génération d'impliqués premiers n'est malgré tout pas trop touché par la grandeur d'un circuit, tout simplement parce que cette génération ne se réalise qu'à niveau local, c'est-à-dire pour chaque porte. La complexité est plutôt dépendante du nombre de modes présents. Dans une première phase de l'implémentation de DIG, nous voulions produire les impliqués premiers à partir de la conjonction de toutes les conjonctions de DNF (qui est égale à une seule conjonction de DNF). Cette idée a été évidemment vite abandonnée en raison de la complexité qui s'en serait suivie.

2.3.3 Analyse des résultats

Les tables 2.5 et 2.6 montrent les résultats obtenus. Les tests ont été réalisés avec un Macintosh Centris 650 (8/240). Nous indiquons le nombre des impliqués premiers (IP) et, pour chaque algorithme, la durée en secondes suivie de la quantité de mémoire utilisée en KBytes.

L'analyse des résultats nous amène à plusieurs conclusions. Le problème "m(x)k(y)" montre clairement que *GEN-PI* déclasse clairement son concurrent,

Exemples.	IP.	GEN - PI		TEPI	
m(2)k(3)	*33*	*0.1 s*	*28 Kb*	*1.7 s*	*575 Kb*
m(3)k(3)	*73*	*0.4 s*	*90 Kb*	*9.6 s*	*3' 335 Kb*
m(2)k(4)	*89*	*0.6 s*	*121 Kb*	*33.8 s*	*11' 994 Kb*
m(2)k(5)	*253*	*3.7 s*	*743 Kb*	*700.0 s*	*268' 077 Kb*
m(3)k(4)	*268*	*4.0 s*	*810 Kb*	*364.0 s*	*139' 425 Kb*

Table 2.5: Comparaison d'algorithmes de génération d'impliqués premiers pour le problème "m(x)k(y)".

Primitives	IP.	GEN - PI		TEPI	
1 mode	*25*	*0.1 s*	*35 Kb*	*0.04 s*	*12 Kb*
2 modes	*63*	*0.4 s*	*94 Kb*	*0.3 s*	*70 Kb*
3 modes	*91*	*1.1 s*	*318 Kb*	*1.2 s*	*386 Kb*
4 modes	*162*	*4.0 s*	*1' 105 Kb*	*6.2 s*	*2' 085 Kb*
5 modes	*234*	*10.3 s*	*2' 907 Kb*	*17.0 s*	*5' 907 Kb*
6 modes	*333*	*22.0 s*	*6' 243 Kb*	*43.6 s*	*15' 821 Kb*

Table 2.6: Comparaison d'algorithmes de génération d'impliqués premiers pour la modélisation d'un additionneur.

tant au niveau du temps que de la mémoire utilisée; pour *m2k5*, l'algorithme de Ngair est même environ deux cents fois plus rapide que celui de TEPI. Cela s'explique aisément par le fait que TEPI doit tout d'abord transformer la conjonction de DNF en une formule CNF et que cette transformation est d'une importante complexité.

Les résultats du deuxième exemple sont également intéressants. On voit tout d'abord que plus il y a de modes de comportements définis pour les primitives, plus nombreux sont les impliqués premiers générés. La complexité totale croît de ce fait également. On constate que l'algorithme de TEPI surpasse *GEN-PI* pour les deux premiers cas, c'est-à-dire pour un nombre restreint d'impliqués premiers. Mais dès le troisième mode, *GEN-PI* devient de plus en plus efficace par rapport à l'autre méthode, en atteignant à six modes un facteur d'environ un pour deux.

On remarque que la complexité de *GEN-PI* n'est pas totalement libérée d'une explosion exponentielle. En traitant chaque formule DNF comme une unique entrée, la complexité dépend directement de la grandeur de la sortie. On peut

montrer pour des exemples avec une entrées très importante et une sortie minime que *GEN-PI* donne des résultat très satisfaisants.

Il faut souligner la flexibilité de *GEN-PI*. En théorie, l'algorithme se montre tout autant efficace qu'un algorithme standard lorsqu'il doit traiter des problèmes formulés en CNF. Pour des problèmes en conjonction de DNF, il déclasse ses concurrents classiques. Mais il est encore souhaitable d'apporter de nouvelles améliorations à l'implémentation de *GEN-PI*; certaines sont mêmes proposées par Ngair. La structure de données, que l'on traitera au quatrième chapitre, pourrait être également perfectionnée.

3 Le langage et l'environnement DIG

Le langage et l'environnement DIG donnent les outils appropriés pour définir et manipuler des descriptions de circuits digitaux possédant des composantes défectueuses. Un langage logique peut décrire directement l'un de ces systèmes en utilisant des composantes de base (*and*, *nand*, *or*, etc.) prédéfinies par le système ou que l'utilisateur lui-même aura déclarées. Les différentes connaissances sur un circuit donné (par exemple le fait que toutes les entrées sont égales à *0* et toutes les sorties à *1*) permettent d'étendre la connaissance globale sur celui-ci et de cibler plus adéquatement le diagnostique ultérieur.

DIG permet de définir les portes de base et de créer des modules qui peuvent être instanciés. Le présent chapitre expose par des illustrations les possibilités du langage DIG. Pour permettre une plus grande convivialité, une interface est mise à disposition de l'utilisateur. Ce dernier peut ainsi aisément manipuler et surtout visualiser les différentes structures engendrées par DIG.

Pour augmenter la puissance de DIG, on y a intégré le système ABL 1.0 (*Assumption-Based Language*) qui lui-même contient MCL 2.0. ABL est un langage qui permet également de modéliser de manière élégante des descriptions de circuits digitaux; cela rend très intéressante la comparaison avec DIG des résultats effectifs et des complexités de temps et de mémoire.

MCL 2.0 (Macintosh Common Lisp 2.0) intègre le standard de Lisp décrit dans [Steele 1992] et [MCL 1992]. DIG devient ainsi une extension de Lisp, et ne reste pas un langage pour soi. En plus de posséder toute la puissance et l'élégance de Lisp, DIG intègre l'environnement MCL permettant une programmation très efficace, notamment pour le déverminage de programmes. Une bonne introduction à Lisp peut être trouvée dans [Norvig 1992].

En annexe de la documentation se trouve les disquettes contenant l'application DIG compactée à l'aide de COMPACT PRO (Version 1.34, de Bill Goodman) et les fichiers de l'implémentation. En présence d'ABL, l'installation de DIG peut se réaliser aisément au moyen du fichier `Makefile` (voir annexe C28) qui permet de compiler et de charger les fichiers.

3.1 Le langage DIG

DIG, comme d'autres langages qui se posent comme but de décrire des circuits digitaux et de permettre le diagnostique [Reiter 87] dans ces descriptions, se divise en trois sous-langages:

• Le sous-langage de définition: pour définir les composantes de base et les modules.

• Le sous-langage de manipulation: pour instancier, effacer, etc.

• Le sous-langage d'interrogation: pour réaliser des tests sur différentes hypothèses.

La présente section montre à l'aide de plusieurs illustrations ce que DIG permet de faire.

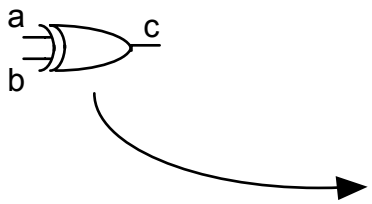
3.1.1 Définir des primitives

Pour décrire un circuit digital, il faut d'abord procéder à la modélisation des éléments sous-jacents, c'est-à-dire des portes *and*, *nand*, *or, nor*, *xor*, etc. Ces composantes de base, que l'on nomme des *primitives*, sont mises à disposition dans le système de manière standard et peuvent être redéfinies par l'utilisateur. Différents comportements peuvent être attribués à chacune d'elles. Un comportement décrit un état, dans lequel la porte de base peut se trouver.

Admettons que nous voulions redéfinir la composante *xor*. Celle-ci se composerait des états suivants: a) l'état *ok*, pour un comportement juste, avec une probabilité de 95%; b) l'état *ab1*, pour lequel on observerait toujours *1* en sortie, avec une probabilité de 1%; c) enfin l'état *ab2* qui réaliserait la fonction *or*, avec une probabilité de 4%. Comme on le voit, à chacun des états est attribué un nombre compris entre *0* et *1* qui indique la probabilité que cette composante se comporte selon le mode respectif; un tel chiffre est intitulé *bpa* (*basic probability assignement*) [Haenni, Monney, Kohlas 1995 et Schumacher 1994].

Une telle primitive peut être déclarée de la manière suivante:

```
? (defprimitive new-xor ((ok a-xor-b 0.95)
                         (ab1 c=1 0.01)
                         (ab2 a-or-b 0.04)))
NEW-XOR
?
```



a	*b*	*c*	*ok*	*ab1*	*ab 2*
0	*0*	*0*	*1*	*0*	*1*
0	*0*	*1*	*0*	*1*	*0*
0	*1*	*0*	*0*	*0*	*0*
0	*1*	*1*	*1*	*1*	*1*
1	*0*	*0*	*0*	*0*	*0*
1	*0*	*1*	*1*	*1*	*1*
1	*1*	*0*	*1*	*0*	*0*
1	*1*	*1*	*0*	*1*	*1*

Figure 3.1: Une description de la primitive `new-xor`.

La figure 3.1 schématise les comportements de la nouvelle primitive. Celle-ci devient en fait une fonction Lisp avec des paramètres formels implicites, par exemple *in1*, *in2*, *out* et *new-xor-mode*. On peut ainsi l'employer dans la définition d'un module en lui passant des paramètres actuels.

```
? (describe 'new-xor)
Symbol: NEW-XOR
Function
INTERNAL in package: #<Package "COMMON-LISP-USER">
Print name: "NEW-XOR"
Value: #<Unbound>
Function: #<Compiled-function NEW-XOR #x49D936>
Arglist: (CCL::ARG-0 CCL::ARG-1 CCL::ARG-2 CCL::ARG-3 &OPTIONAL CCL::OPT-0)
Plist: NIL
?
```

Pour définir une primitive, il faut attribuer à un mode un comportement précis avec son *bpa* correspondant. La somme des *bpa's* doit être égale à *1*. Les différents comportements que l'on peut trouver dans une primitive sont résumés aux tables 3.2 et 3.3. Les comportements peuvent être dissociés en deux grands groupes: pour les portes à deux entrées et une sortie (exemple: *or input1 input2 output*), et pour celles à une seule entrée et une seule sortie (exemple: *not input output*).

Les primitives standards de DIG sont toutes prédifinies avec un mode `ok` d'un *bpa* de 1. Elles se nomment: `not@`, `and@`, `nand@`, `or@`, `nor@`, `xor@`. Le symbole "@" a été choisi pour compléter les noms des fonctions logiques, tout simplement pour ne pas les confondre avec les fonctions de Lisp et d'ABL

a	*b*	*c*	*a−and−b*	*a−nand−b*	*a−or−b*	*a−nor−b*	*a−xor−b*	*c=0*	*c=1*
0	*0*	*0*	*1*	*0*	*1*	*0*	*1*	*1*	*0*
0	*0*	*1*	*0*	*1*	*0*	*1*	*0*	*0*	*1*
0	*1*	*0*	*1*	*0*	*0*	*1*	*0*	*1*	*0*
0	*1*	*1*	*0*	*1*	*1*	*0*	*1*	*0*	*1*
1	*0*	*0*	*1*	*0*	*0*	*1*	*0*	*1*	*0*
1	*0*	*1*	*0*	*1*	*1*	*0*	*1*	*0*	*1*
1	*1*	*0*	*0*	*1*	*0*	*1*	*1*	*1*	*0*
1	*1*	*1*	*1*	*0*	*1*	*0*	*0*	*0*	*1*

a	*b*	*c*	*c=a*	*c=not−a*	*c=b*	*c=not−b*	*tautology−3*
0	*0*	*0*	*1*	*0*	*1*	*0*	*1*
0	*0*	*1*	*0*	*1*	*0*	*1*	*1*
0	*1*	*0*	*1*	*0*	*0*	*1*	*1*
0	*1*	*1*	*0*	*1*	*1*	*0*	*1*
1	*0*	*0*	*0*	*1*	*1*	*0*	*1*
1	*0*	*1*	*1*	*0*	*0*	*1*	*1*
1	*1*	*0*	*0*	*1*	*0*	*1*	*1*
1	*1*	*1*	*1*	*0*	*1*	*0*	*1*

Table 3.2: Les comportements des portes à deux entrées et une sortie.

a	*b*	*b=a*	*b=not−a*	*b=1*	*b=0*	*tautology−2*
0	*0*	*1*	*0*	*0*	*1*	*1*
0	*1*	*0*	*1*	*1*	*0*	*1*
1	*0*	*0*	*1*	*0*	*1*	*1*
1	*1*	*1*	*0*	*1*	*0*	*1*

Table 3.3: Les comportements des portes à une entrée et une sortie.

Nous aimerions insister sur le fait que dans la version actuelle de DIG, on ne peut obtenir que des résultats symboliques et pas numériques, lors d'interrogations du système sur certaines hypothèses que l'on ferait sur des circuits instanciés. Cette difficulté provient du fait que les modes de comportement d'une primitive sont dépendants les uns des autres et que les méthodes adoptées pour cette modélisation ne peuvent donner que des réponses symboliques. Malgré tout, nous avons décidé de ne pas changer le langage par rapport à sa version précédente que intégrait déjà les *bpa's* (voir [Schumacher 1994]), notamment pour permettre une extension ultérieure du système plus aisée.

3.1.2 Définir des modules

Après avoir défini les portes de base avec les différents comportements, nous avons besoin d'un outil pour définir des modules de circuits digitaux. Ces mêmes modules peuvent être réutilisés directement dans la définition d'autres modules.

Considérons l'exemple d'un demi-additionneur de chiffres binaires: `halfadder`. Ce module, illustré à la figure 2.3 du précédent chapitre, est réalisé à l'aide des portes `or@`, `and@` et `not@`. Nous procédons à la redéfinition des primitives désirées, dans notre cas les composantes `or@` et `and@`:

```
(defprimitive and@ ((ok a-and-b 0.98)
                    (ab c=1     0.02)))

(defprimitive or@ ((ok a-or-b  0.94)
                   (ab c=0     0.06)))
```

On peut dès lors définir le module `halfadder`:

```
(defmodule halfadder (a b sum c)
  (int ((wires e d)
        (modes m-or1 m-and1 m-and2 m-not1))
    (inst or1  (or@  a b e m-or1))
    (inst and1 (and@ a b c m-and1))
    (inst not1 (not@ c d m-not1))
    (inst and2 (and@ e d sum m-and2))))
```

La macro `defmodule` est similaire à la forme spéciale `defun` de Lisp. Des paramètres formels sont nécessaires. Ils constituent l'entrée (dans notre cas `a` et `b`) et la sortie (`sum` et `c`). La description d'un circuit requiert fréquemment des variables internes; on peut les déclarer à l'aide du mot-clé `wires`. Le mot-clé `modes` déclare les paramètres formels des modes des différentes primitives. Il est possible de ne rien mettre après ces deux mots-clés. Par la suite, après avoir instancié un module, il sera possible de réaliser différentes demandes sur les modes.

Après la déclaration de tous les paramètres formels nécessaires vient le coeur de la définition du module. On indique au module d'instancier les uns après les autres différents éléments pouvant être soit une primitive (comme c'est le cas dans la définition de `halfadder`), soit un autre module prédéfini. Chacune de ces sous-expressions débute par le mot-clé `inst` suivi du symbole de la sous-instance et de l'appel de primitive ou de module.

Il faut souligner le fait qu'un module, tout comme une primitive, est en fait une fonction Lisp. On le voit très bien avec une description de `halfadder`.

```
? (describe
Symbol: HALFADDER
Function
INTERNAL in package:
USER">
Print name:
Value: #<Unbound>
Function:
HALFADDER #x593346>
Arglist: (CCL::ARG-0
CCL::ARG-3 &OPTIONAL
Plist: NIL
?
```

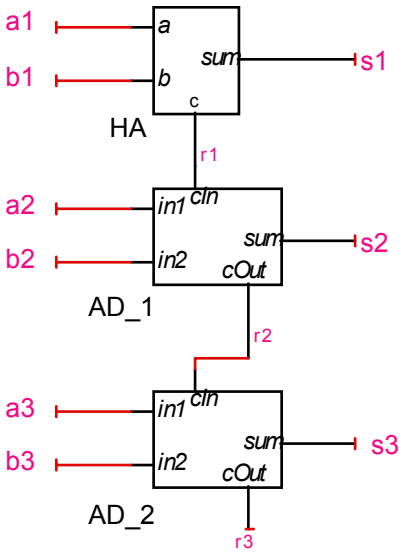


Figure 3.4: Le module `fulladder`.

```
'halfadder)

#<Package "COMMON-LISP-

"HALFADDER"

#<Compiled-function

CCL::ARG-1 CCL::ARG-2
CCL::OPT-0)
```

Pour montrer la modularité de DIG, nous voulons déclarer de nouveaux modules. Nous définissons tout d'abord un additionneur `adder` qui emploie `halfadder`. La figure 2.4 du précédent chapitre présente ce nouveau circuit.

```
(defmodule adder (a
  (int ((wires s c1 c2)
        (modes m-or1))
    (inst ha_1 (halfadder b cIn s c1))
    (inst ha_2 (halfadder a s sum c2))
    (inst or1  (or@ c1 c2 cOut m-or1))))
```

```
b cIn sum cOut)
```

Nous voulons encore définir un additionneur complet à trois bits. Il se compose de deux `adder` et d'un `halfadder` (cf. figure 3.4).

```
(defmodule fulladder (a1 a2 a3 b1 b2 b3 s1 s2 s3 r3)
  (int ((wires r1 r2)
        (modes ))
    (inst ha (halfadder a1 b1 s1 r1))
    (inst ad_1 (adder a2 b2 r1 s2 r2))
    (inst ad_2 (adder a3 b3 r2 s3 r3))))
```

3.1.3 Déclarer des variables globales

`New-var` permet de déclarer des variables globales que l'on pourra employer lorsqu'on instanciera des modules. L'exemple suivant montre comment on peut utiliser cette simple macro:

```
? (new-var input1 input2 output mode)
MODE
? (new-var in1 in2 carry-in sum carry-out)
CARRY-OUT
? (new-var in11 in12 in13 in21 in22 in23 sum1 sum2 sum3 carry)
CARRY
?
```

Les variables déclarées ont dès lors comme valeur de symbole leur propre symbole.

```
? input1
INPUT1
? in12
IN12
?
```

3.1.4 Instancier un module ou une primitive

Pour manipuler un système concret de circuit logique, il faut instancier un module ou une primitive. Cela se fait à l'aide de la macro `inst`. Les exemples suivants nous indiquent comment procéder:

```
? (inst the-and (and@ input1 input2 output mode))
#<GATE #x5B4631>
? (inst AD (adder in1 in2 carry-in sum carry-out))
#<GATE #x5B0669>
? (inst FA (fulladder in11 in12 in13 in21 in22 in23 sum1 sum2 sum3 carry))
#<GATE #x5B0707>
?
```

Cette macro attribue à un symbole le résultat de l'appel de fonction du module ou de la primitive: un objet de la classe `GATE`. Il faut passer des paramètres actuels qui auront été déclarés auparavant à l'aide de la macro `new-var`. Différents messages d'erreur sont redonnés si cela est nécessaire (par exemple: paramètres actuels trop nombreux ou pas déclarés).

A partir de ce stade, il est possible de manipuler les circuits instanciés, par exemple en leur ajoutant des connaissances au moyen de la macro `obs` ou en faisant une quelconque demande pour une hypothèse donnée.

3.1.5 Observer une variable d'un circuit

Après avoir décrit un circuit logique, nous en avons créé un exemplaire en instanciant un module défini. Nous devons encore être en mesure d'ajouter des connaissances au nouveau circuit: des observations que l'on aura faites. Ainsi, si les valeurs des entrées et des sorties ne correspondent pas aux attentes, il faudra rechercher la ou les composantes défectueuses.

Pour ajouter une observation à un circuit, on utilise la macro `obs`. Après avoir indiqué le circuit dans lequel l'on désire introduire une connaissance, on attribue à une variable interne ou externe une valeur possible. Cette valeur peut

être soit un *0* ou un *1*, soit la valeur d'un mode appartenant au domaine de définition de celui-ci. Il est bon de noter encore la manière d'accéder aux variables internes: il faut faire suivre les noms des différents modules concernés, le tout suivi par le nom de la variable elle-même; chaque terme est séparé par un point.

Nous continuons avec l'exemple de l'additionneur `adder` de deux nombres à un bit chacun. Nous observons que `AD` ne se comporte pas comme nous pourrions l'attendre: l'addition de *0* et de *0* avec *1* en retenue d'entrée donne comme résultat *0* avec *0* en retenue de sortie. Le résultat correct serait *1* avec également *0* en retenue de sortie. Il est possible de déclarer tout cela très simplement:

```
(progn
  (obs AD in1 0)
  (obs AD in2 0)
  (obs AD carry-in 1)
  (obs AD sum 0)
  (obs AD carry-out 0))
```

Ces observations se restreignent au comportement externe du circuit. Elles aident à focaliser l'endroit vague de l'erreur. Mais nous n'avons pas encore assez d'informations pour pouvoir apporter un jugement adéquat quand à la ou les composantes faussées.

3.1.6 Questionner le système

Après avoir observé qu'un circuit donné possède de manière évidente un ou plusieurs dysfonctionnements, il faut passer à la phase de recherche d'erreurs. Cela est réalisé par l'utilisateur qui, en constatant de manière ciblée les parties du système se comportant de manière anormale, émet certaines hypothèses au sujet de certaines composantes. Comme nous le verrons plus loin, l'utilisateur peut réaliser quatre sortes de tests: il peut calculer pour une hypothèse donnée le soutien (*support*), le quasi-soutien (*quasi-support*), le doute (*doubt*) et la plausibilité (*plausibility*). Les fonctions DIG sont respectivement `DIG-support` (ou `DIG-sp`), `DIG-quasi-support` (ou `DIG-qs`), `DIG-doubt` (ou `DIG-db`) et `DIG-plausibility` (ou `DIG-pl`). Les réponses lui seront données uniquement en formules logiques, plus précisément en formes normales disjonctives (DNF) composées exclusivement des modes de comportement; il n'y a pas de réponse numérique dans la version actuelle de DIG. Cela est redevable à la modélisation de la base de données du circuit digital (pour plus de détails sur la modélisation, voir la section 2.3.2). En introduisant une dépendance entre les modes des portes

au moyen d'une formule *xor*, le système ne peut répondre que symboliquement avec les méthodes qu'il a adoptées.

Une macro d'interrogation possède comme arguments le circuit instancié que l'on désire tester et l'hypothèse sur les différents modes de comportement. La formulation des hypothèses est repris du langage d'ABL 1.0 [Lehmann 1994]. Une hypothèse peut être soit un littéral, soit une relation. Les littéraux d'une hypothèse ne peuvent être que les symboles des modes. Une relation est constituée des fonctions logiques `and`, `or` et `not` (ou `and*`, `or*` et `not*` respectivement) qui combinent des littéraux. Les fonctions logiques et leurs arguments doivent toujours être compris entre des parenthèses. Les arguments suivent toujours la fonction; ils peuvent être eux-mêmes des fonctions logiques. `Not` est la fonction unitaire qui exprime le complément. Les fonctions `and` et `or`, qui expriment respectivement le ET et le OU logiques, ont au moins deux arguments. Il est donc possible de formuler des formules logiques du genre suivant:

```
(or (and c (not d) (not e)) (not f))
```

Pour rendre plus clair le test d'hypothèses, nous continuons d'analyser le circuit `AD` décrivant un additionneur composé de neuf portes. Les primitives ont été définies à la section 3.1.1. Le comportement de cet additionneur est décrit à la section 3.1.5.

En analysant les sorties de `AD`, on remarque que la variable `sum` devrait être égale à *1* au lieu de sa valeur *0*. Nous analysons la situation en nous référant à la figure 3.5 qui nous réplique l'additionneur instancié `AD`.

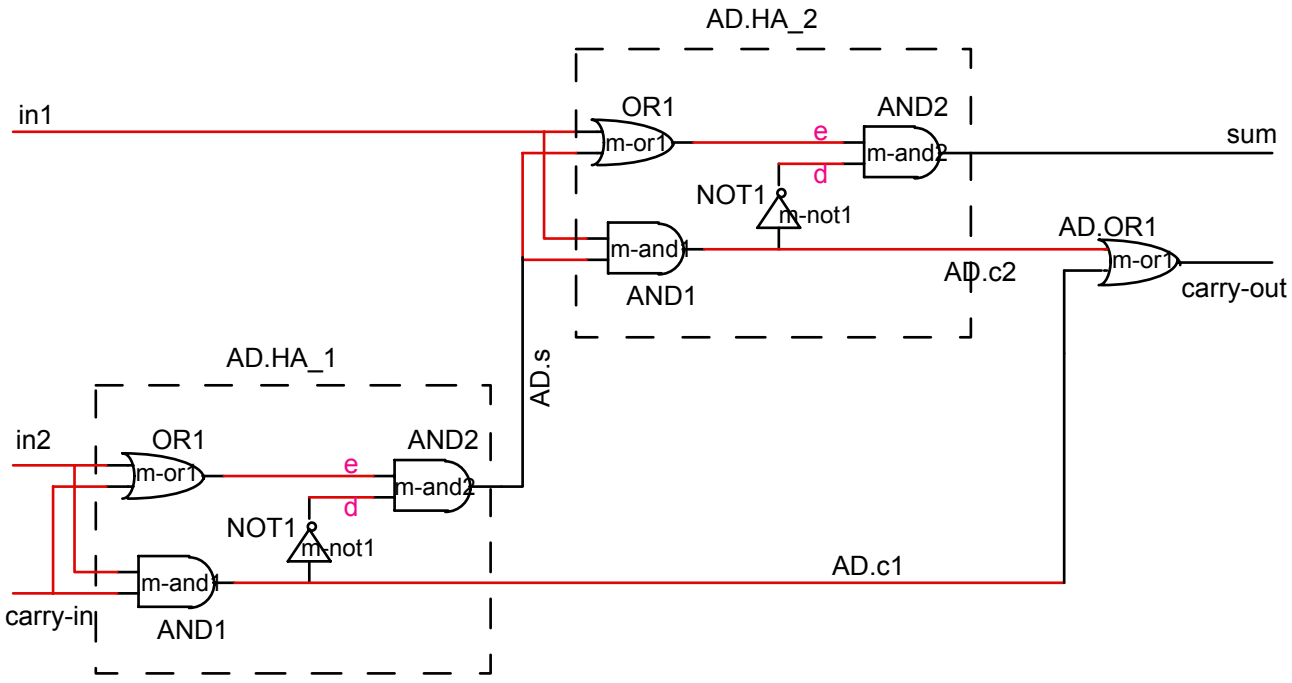


Figure 3.5: Le circuit instancié `AD`.

La variable ayant une valeur fausse est donc `sum`. Elle est une sortie du sous-circuit `AD.HA_2` qui est directement dépendant des entrées `in1` et `AD.s`. L'entrée `AD.s` est une des sorties du sous-circuit `AD.HA_1`. L'erreur peut avoir différentes sources.

• Nous pourrions supposer que le circuit `AD.HA_1` redonne une fausse sortie `AD.s` qui équivaudrait à *0* au lieu de *1*. Il faudrait alors rechercher l'erreur à l'intérieur du demi-additionneur `AD.HA_1`. Nous savons que la composante `NOT1` se comporte toujours correctement. La mode d'erreur de `AND2` est celui de redonner en sortie toujours la valeur *1*; donc un dysfonctionnement de cette composante n'aurait aucun effet. Il reste alors les portes `OR1` et `AND1` qui peuvent toutes deux être sources de l'erreur.

• L'autre possibilité consiste à supposer que l'erreur se situe dans le demi-additionneur `AD.HA_2`. Nous pouvons faire le même raisonnement que pour l'hypothèse précédente.

• Enfin, l'erreur pourrait se situer dans les deux sous-circuits.

Nous supposons que nous disposons de nouvelles connaissances pour cibler le diagnostique. Nous obtenons plusieurs informations grâce à deux observations:

```
(progn
  (obs AD c1 0)
  (obs AD s  1))
```

Nous obtenons un nouveau système plus simple à gérer. Nous savons que les sorties du sous-système `AD.HA_1` sont correctes. Cela nous amène à ne considérer que la deuxième source éventuelle d'erreur.

Pour montrer les possibilités du langage d'interrogation, nous en donnons quelques exemples. Nous testons le soutien (au moyen de la macro `DIG-support`) de l'hypothèse qui suppose la composante `HA_2.m-OR1` en dysfonctionnement.

```
? (DIG-support AD HA_2.m-OR1.ab)
(HA_1.M-AND1.AB AND HA_2.M-OR1.AB AND (NOT HA_1.M-AND1.OK))
?
```

Le système nous répond en nous disant que le soutien de cette hypothèse est simplement que le mode `HA_2.M-OR1` est en dysfonctionnement, la variable `HA_1.M-AND1` en dysfonctionnement et `HA_1.M-AND1.OK` non vrai.

Supposons que nous disposions d'une nouvelle observation:

```
(obs AD HA_2.e 0)
```

Si l'on teste le doute de l'hypothèse que la composante `HA_2.m-OR1` soit dans l'état anormal (`ab`), nous obtenons la réponse suivante:

```
? (DIG-doubt AD HA_2.m-OR1.ab)
((NOT HA_2.M-OR1.AB)) OR
((NOT HA_1.M-AND1.AB)) OR
(HA_1 M_AND1.OK)
?
```

Testons encore le soutien d'une hypothèses combinée:

```
? (DIG-support AD (and HA_2.m-OR1.ab HA_1.m-OR1.ok))
(HA_1.M-OR1.OK AND HA_1.M-AND1.AB AND HA_2.M-OR1.AB AND (NOT HA_1.M-
AND1.OK))
?
```

Les réponses obtenues ne sont pas surprenantes. Nous laissons au soin de l'utilisateur de réaliser d'autres tests avec des circuits pertinents.

3.1.7 Quelques manipulations supplémentaires

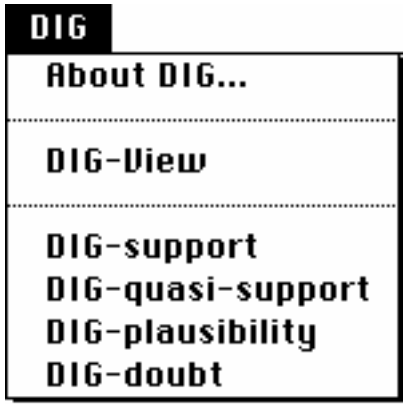


Figure 3.6: Le menu DIG.

Étant donné que les objets créés sont de grandeur importante, il est nécessaire d'offrir des facilités pour effacer ce que l'on a défini. Le ramasse-miettes (*garbage-collector*) intégré à l'environnement de Lisp libérera ensuite la mémoire dès que cela sera nécessaire.

`Delete-module` et `delete-gate` permettent respectivement d'effacer des modules et des circuits. Il faut simplement faire suivre la macro voulue du nom concerné. Comme le dit la section qui suit, il est également possible de réaliser cette fonction à l'aide de l'utilité *DIG-View*.

```
? (delete-module adder)
ADDER
?
```

Enfin, la fonction `install-menu` permet d'installer le menu DIG décrit à la section 3.2, au cas où celui-ci ne serait pas encore présent.

3.2 Le menu DIG

Le système DIG intègre une interface graphique qui permet à l'utilisateur de réaliser avec facilité quelques fonctionnalités. DIG a ainsi beaucoup gagné en efficacité. Pour intégrer DIG dans un environnement cohérent, nous avons voulu lui donner une approche analogue à celle —déjà réalisée— d'ABL. Un menu a donc été ajouté (figure 3.6). Il comporte six éléments. Il y a six boîtes de dialogues correspondant à chaque élément du menu: le dialogue standard "*About Dig...*"; "*DIG-View*", possédant plusieurs fonctionnalités de manipulations des circuits; et quatre autres boîtes pour le langage d'interrogation du système.

L'élément "*About DIG...*" indique tout simplement ce qu'est le système DIG, qui en est l'auteur et qui y a collaboré. Une telle boîte de dialogue (figure 3.7) appartient à toute application standard Macintosh.

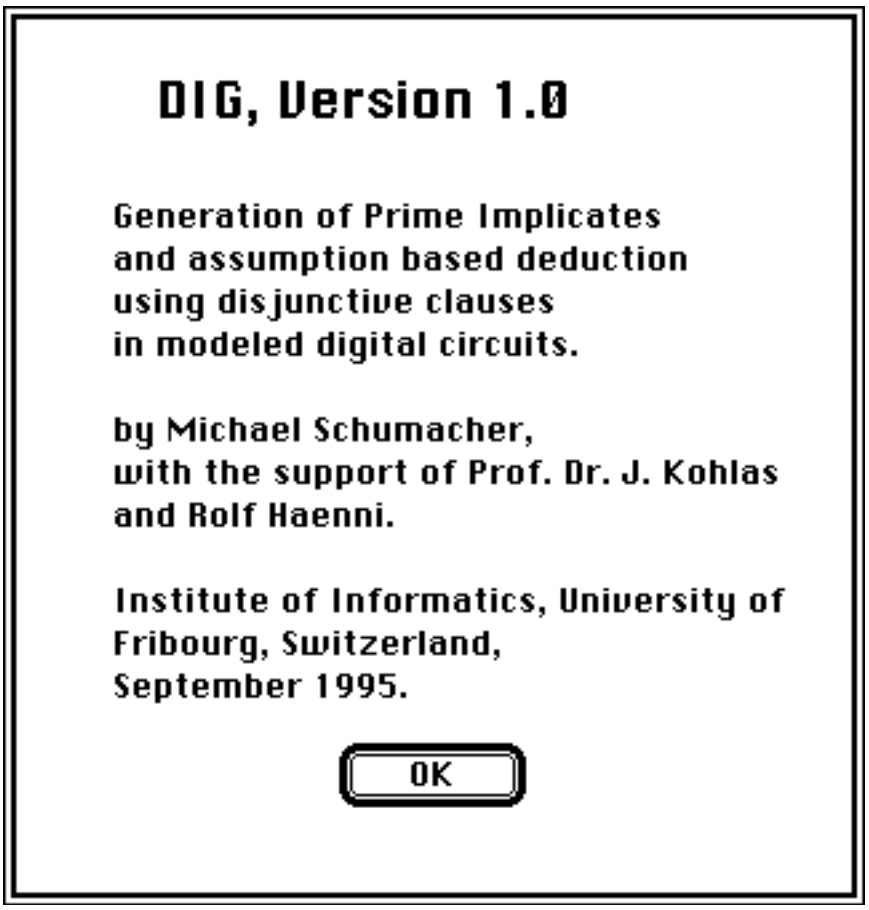


Figure 3.7: Dialogue "About DIG".

Le système possède un dialogue *DIG-View* similaire à *XViewLogic* d'ABL. Différentes actions peuvent y être réalisées sur les modules et les circuits existants. Ceux-ci sont affichés dans deux sous-fenêtres avec ascenseurs. La figure 3.8 montre comment les modules `HALFADDER`, `ADDER` et `FULLADDER` et les circuits `AD`, `FA` et `THE-AND` sont affichés à l'intérieur de *DIG-View*.

Il est possible de choisir directement au moyen de la souris un ou plusieurs éléments; les boutons *All* et *None* permettent de sélectionner tous ou aucun des éléments des modules et respectivement des circuits. On peut ensuite appliquer une action aux éléments choisis.

• *Inspect*: ce bouton permet d'inspecter, à la façon de MCL, les modules et les circuits voulus. La figure 3.9 nous montre la fenêtre *Inspector* de l'additionneur `AD`.

On analyse ainsi la structure de données des modules (qui sont des fonctions Lisp) et des circuits (qui sont des objets CLOS de la classe `gate`).

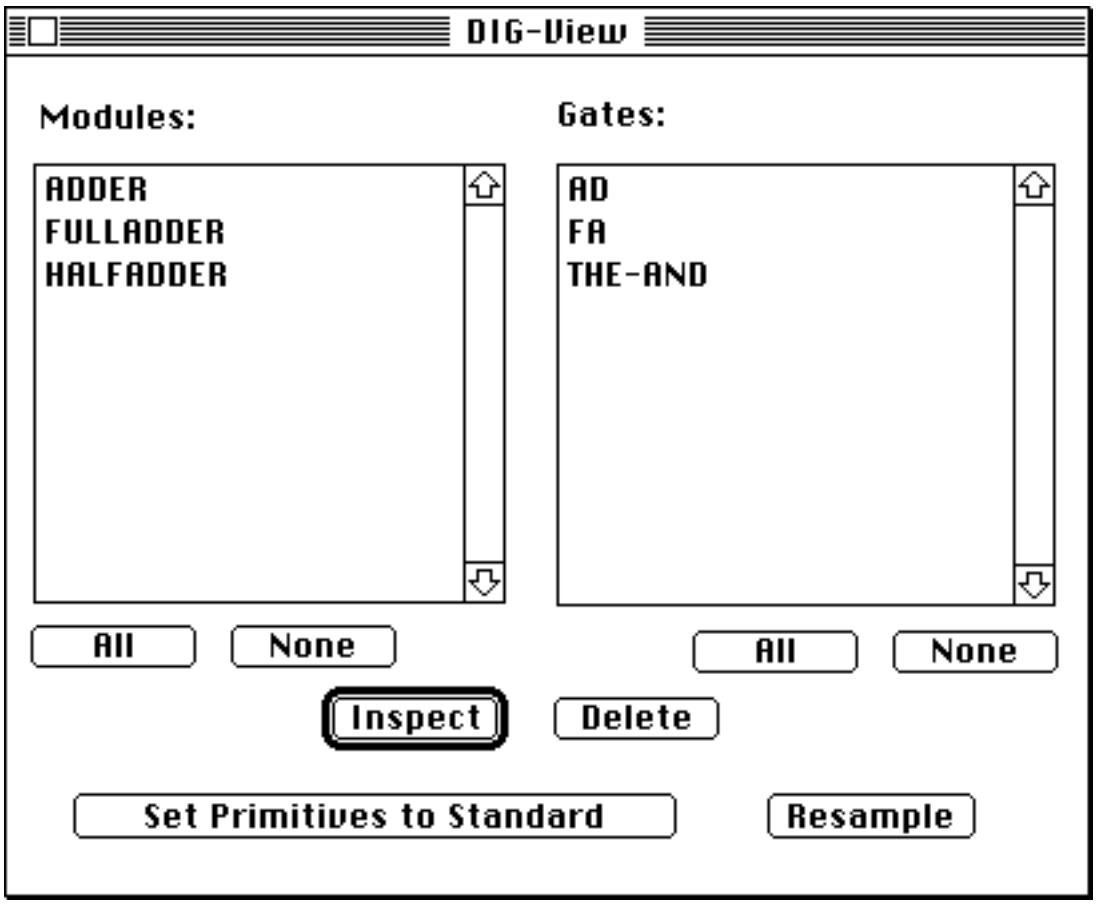


Figure 3.8: *DIG-View*.

Cette fonctionnalité permet de travailler de manière très efficace lorsque l'on recherche à analyser les réseaux de valuations engendrés par le programme. Il faut encore noter qu'un double *click* sur l'un des éléments réagirait exactement comme *inspect*. Pour des renseignements complémentaires sur la facilité de l'*Inspector*, il faut consulter [MCL 1992].

• *Delete* permet d'effacer un module ou un circuit pour libérer de la mémoire ce qui ne serait plus utilisé. On peut par après recourir au ramasse-miettes en tapant dans le *Listener* la commande `(gc)`.

• *Set Primitives to Standard* redéfinit les primitives selon le mode standard:

```
(progn
  (defprimitive and@  ((ok a-and-b  1.0)))
  (defprimitive nand@ ((ok a-nand-b 1.0)))
  (defprimitive or@   ((ok a-or-b   1.0)))
  (defprimitive nor@  ((ok a-nor-b  1.0)))
  (defprimitive xor@  ((ok a-xor-b  1.0)))
  (defprimitive not@  ((ok b=not-a  1.0))))
```

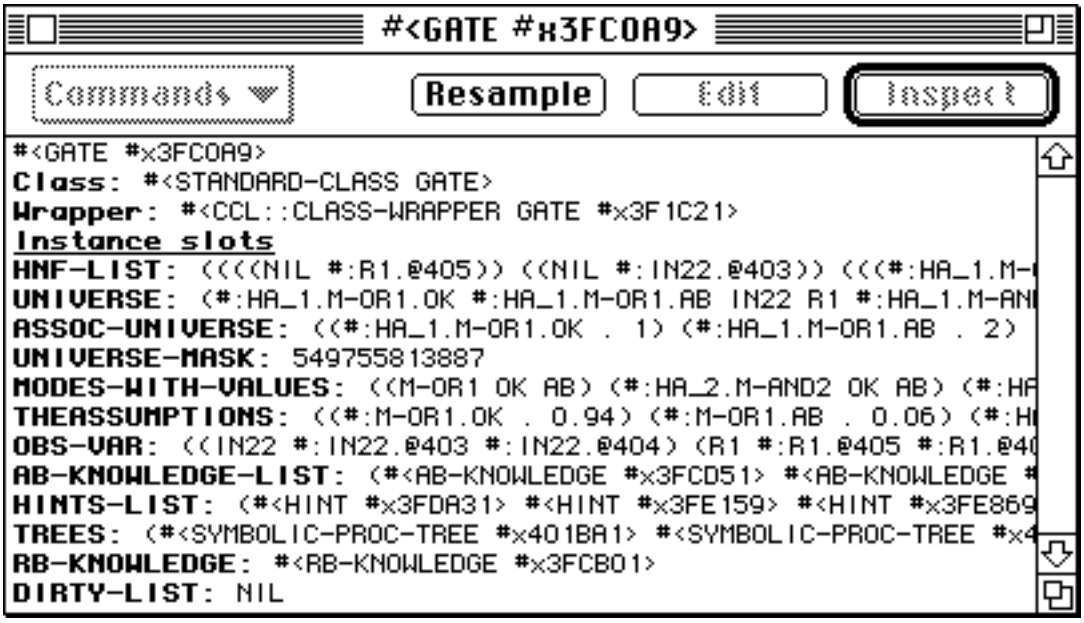


Figure 3.9: Une fenêtre *Inspector*.

• *Resample* permet simplement de remettre à jour *DIG-View*. En effet, si, après avoir ouvert cet utilitaire, on redéfinissait de nouveaux modules ou on en instanciait à nouveaux, ces nouvelles modifications de la base de données n'apparaîtraient pas dans la boîte de dialogue.

Les quatre derniers éléments du menu "DIG" permettent de réaliser à l'aide d'une boîte de dialogue les interrogations de support, quasi-support, plausibilité et doute (cf. figure 3.10).

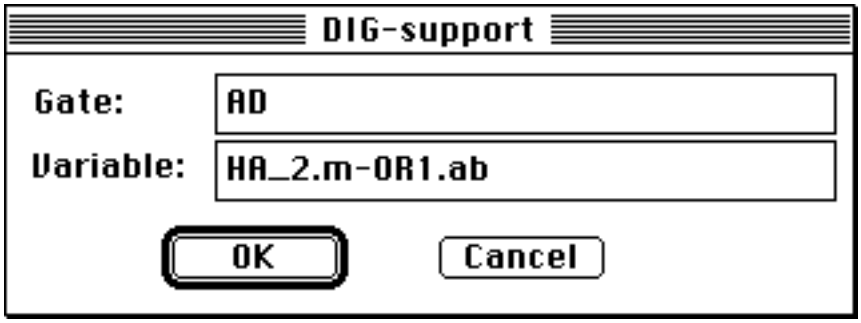


Figure 3.10: Une boîte de dialogue d'interrogation.

4 Architecture et implémentation

L'implémentation d'un langage et d'un système permettant le diagnostique de descriptions de circuits digitaux ayant des erreurs, à partir d'une modélisation réalisée à l'aide de disjonctions, se laisse résoudre de manière élégante en Lisp. Le choix de ce langage a permis de profiter de sa puissante expressivité. Les facilités données par les macros, qui permettent d'étendre Lisp et même de créer un nouveau langage, donne des atouts que l'on peut difficilement trouver, pour le but recherché, dans d'autres langages. L'extension orientée objet CLOS[1] a permis d'atteindre les buts de la réusabilité et de la possibilité d'extension future du système. La macro du `loop`, qui est pratiquement un sous-langage à l'intérieur de Lisp, rend très compacte la réalisation des algorithmes à caractère itératif qui ne sont pas peu nombreux dans l'implémentation de DIG.

Le chapitre sur l'architecture et l'implémentation du système DIG va décrire quels sont les éléments principaux de ce projet, en expliquant plus en détails quelques parties importantes. Pour accompagner la lecture de ce chapitre, il convient de se référer à l'annexe C qui contient le code complet de DIG.

4.1 L'architecture générale

Ce sous-chapitre se propose de montrer quelle est la structure complète du projet DIG. On verra par la suite plus dans le détail comment les noyaux des différents modules ont été implémentés. Une première approche a déjà été réalisée dans le chapitre 2, lors de l'exposé de l'implémentation de l'algorithme de génération d'impliqués premiers.

L'exposé de cette section se réfère directement à la figure 4.1. L'implémentation de DIG hérite de TEPI (*Theory of Evidence Programming Interface*) qui est un module pour le traitement d'informations incertaines, selon l'approche basée sur suppositions de la théorie de l'évidence; ce module est formé d'un ensemble de structures de données et de procédures qui réalisent différentes

[1] Dans [Kenne 1989], on trouvera un exposé complet de CLOS.

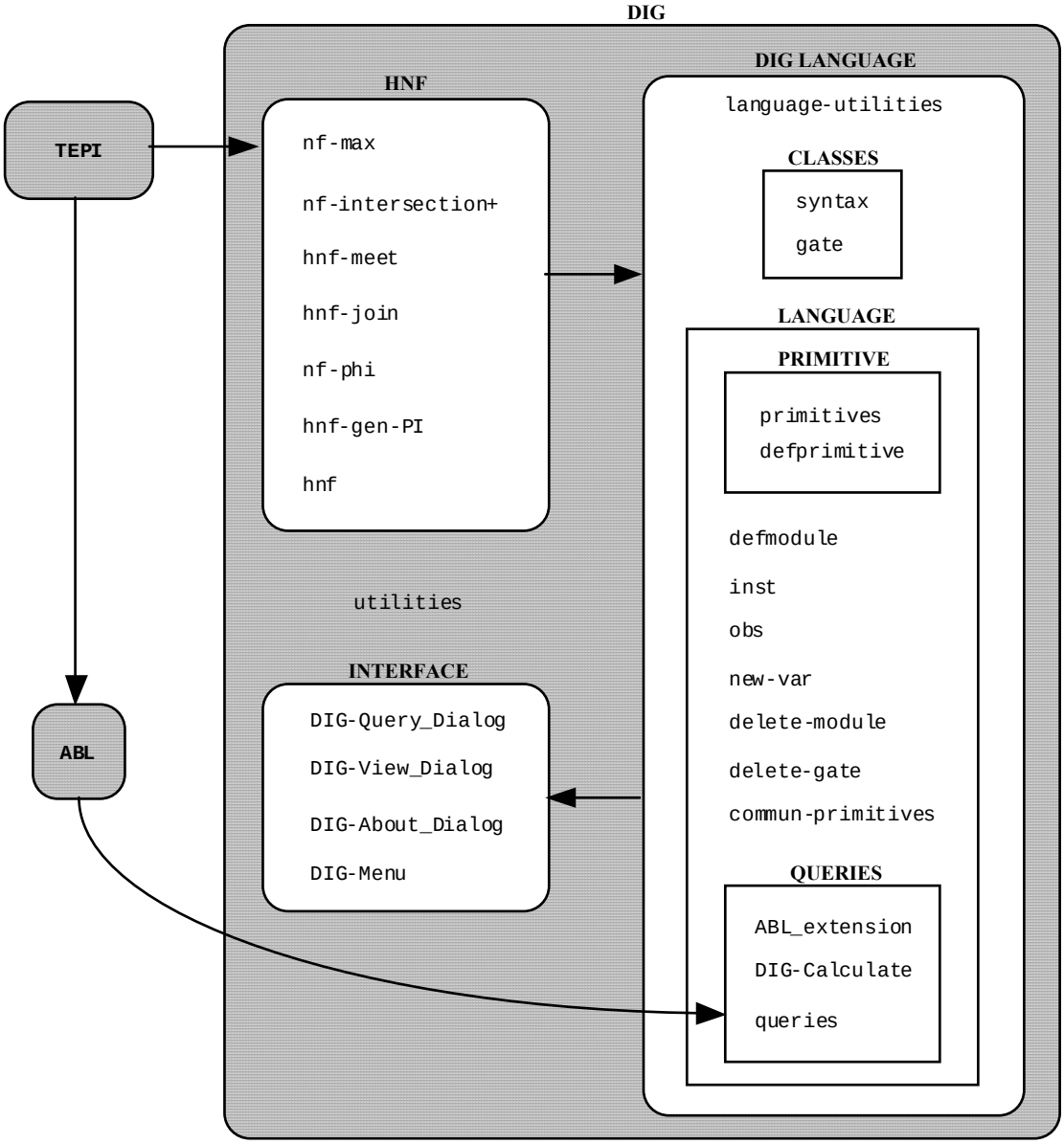


Figure 4.1: Architecture globale du projet DIG.

tâches comme, par exemple, la représentation de formules logiques ou la construction de réseaux de valuations. Nous avons étendu et complété ces structures en y rajoutant également différentes fonctions importantes, comme celle de *GEN-PI*.

Il a fallu d'abord trouver une nouvelle structure de données permettant d'exprimer la modélisation des circuits digitaux en conjonctions de DNF. Nous avons pour ce fait bâti sur les *nf* (*normal forms*) en définissant des *hyper-normal forms* (voir la section 4.2). Le module *HNF* est donc constitué de la définition de ces nouvelles structures de données et des différentes fonctions qui s'y appliquent. Il s'y trouve également de nouvelles fonctions pour les formes

normales. Parmi les plus importantes, on retrouve celles qui réalisent les *max*, *meet*, *join*, *phi*, et *gen-pi* (voir le chapitre 2). Le module *HNF* est ensuite utilisé pour la définition du langage.

Le module *DIG language* présente une hiérarchie plus complexe. Le fichier `language-utilities` inclut avant tout la déclaration des variables globales au système. Ensuite vient la définition des classe syntaxiques (voir la section 4.3): les classes `syntax`, `definition`, `declaration` et `manipulation`.

Pour chaque macro du langage, à part celles d'interrogation et certaines de manipulation, une classe syntaxique a été définie dans le fichier correspondant à la macro: il s'agit des classes `primitive-definition`, `module-definition`, `new-variables`, `instances` et `observation`. Ces classes sont respectivement déclarées dans les fichiers `defprimitive`, `defmodule`, `inst`, `obs` et `new-var`.

Les macros d'interrogation diffèrent radicalement des autres éléments du langage. Pour augmenter la puissance du langage d'interrogation de DIG, il nous a paru opportun d'intégrer la même formulation des hypothèses qu'ABL. Cela permet à l'utilisateur de disposer de l'élégance et de l'expressivité de ce sous-langage. Cette intégration a constitué une étape importante dans la réalisation du présent projet. Elle a également permis de montrer clairement la réusabilité d'un code bien écrit, tel que l'est celui d'ABL.

Ensuite vient un module complètement indépendant de celui de *DIG Language*: le module d'interface. On y a implémenté toutes les facilités de style Macintosh et MCL, permettant une manipulation et des inspections des données très agréables. Le style introduit par ABL a été strictement suivi afin d'offrir un ensemble cohérent et de permettre facilement une extension du système. Il faut souligner le fait que le code de la programmation de l'interface est indépendant de celui de *HNF* et *DIG Language*. Ceci permet de transporter très facilement l'application sur d'autres plates-formes en y apportant les modifications nécessaires.

4.2 Les formes hyper-normales comme extension des formes normales

Pour le genre de problèmes auquel la réalisation du système DIG est confrontée, la structure de données choisie est d'une importance radicale.

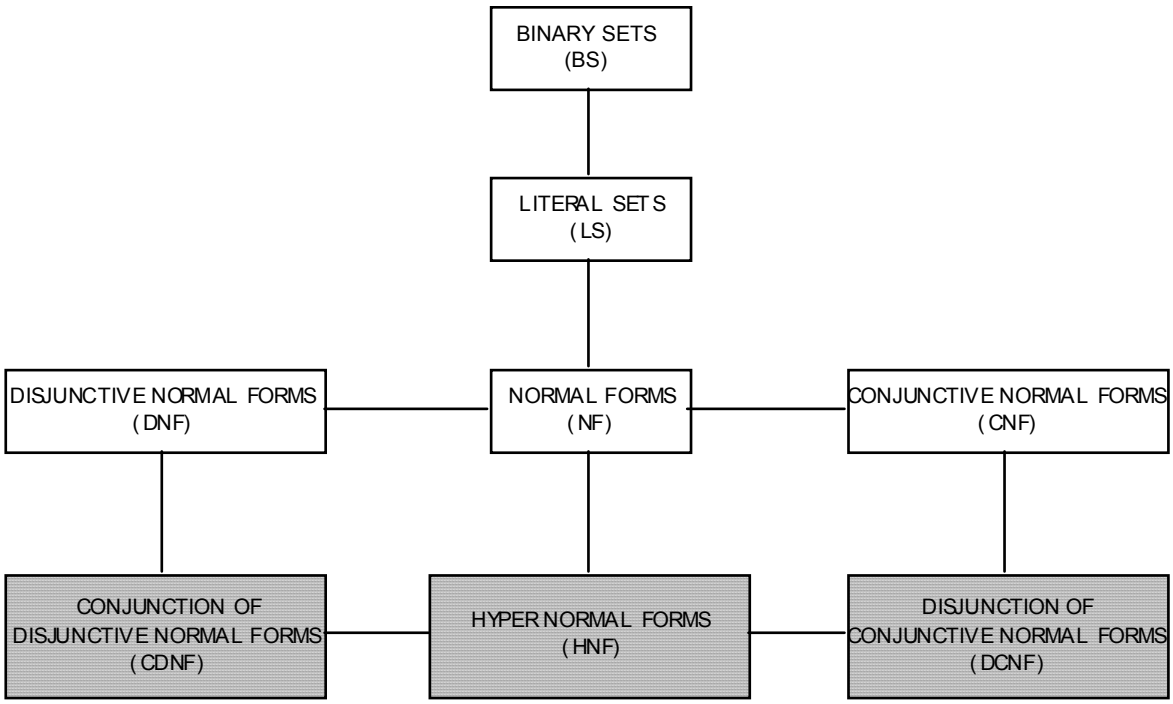


Figure 4.2: L'architecture HNF.

L'implémentation a hérité de la modélisation de TEPI [Haenni 1995] pour ce qui concerne les formules logiques. Cette modélisation a été étendue pour renforcer l'expressivité. La figure 4.2 montre concrètement la construction qui a été choisie. Les zones en gris indiquent ce que DIG apporte de nouveau.

4.2.1 *Binary sets*

Un *binary set* (`bs`) est un ensemble de symboles propositionnels représenté par un chiffre binaire. Il peut être interprété comme une conjonction ou une disjonction de littéraux positifs.

On associe toujours à chaque représentation un univers. Un univers est la liste des symboles propositionnels déclarés. A un bit d'un chiffre binaire *C* se trouvant à la position *i* correspond un symbole dans l'univers se trouvant également à la position *i*. Ainsi, pour l'univers *(a, b, c, d, e)*, la lettre *a* est représentée par *00001*, la lettre *b* par *00010*, etc. L'ensemble $\{a, d\}$, c'est-à-dire la conjonction $a \wedge d$ ou la disjonction $a \vee d$, est représenté par le chiffre binaire *01001* (qui équivaut à *9*).

4.2.2 *Literal sets*

Un *literal set* (`ls`) est un ensemble de littéraux représenté par une paire de *binary sets*.

L'ensemble $\{b, c, \neg a, \neg d\}$, c'est-à-dire la conjonction $b \wedge c \wedge \overline{a} \wedge \overline{d}$, ou la disjonction $b \vee c \vee \overline{a} \vee \overline{d}$, serait codé, pour l'univers *(a b c d e)*, par le *literal set (00110 . 01001)* ou plus simplement *(6 . 9)*. Un *literal set* peut être interprété comme une conjonction ou une disjonction de littéraux positifs ou négatifs.

4.2.3 Les formes normales.

Une forme normale (`nf`) est un ensemble de *literal sets*. Elle peut être interprétée comme une forme normale conjonctive (CNF) ou disjonctive (DNF). Pour le même univers considéré auparavant, l'ensemble *((6 . 9) (2 . 4) ...)* représente *(((b c) . (a d)) ((b) (c)) ...)*, c'est-à-dire la DNF $\left(b \wedge c \wedge \overline{a} \wedge \overline{d}\right) \vee (b \wedge \overline{c}) \vee \ldots$ ou la CNF $\left(b \vee c \vee \overline{a} \vee \overline{d}\right) \wedge (b \vee \overline{c}) \wedge \ldots$.

TEPI définit une spécialisation des formes normales en `dnf` et `cnf`.

4.2.4 Les formes hyper-normales

Les structures de données décrites ci-dessus ont été fournies par TEPI. L'implémentation de la génération d'impliqués premiers et du système DIG a nécessité pourtant une nouvelle abstraction. On a ainsi introduit des formes *hyper-normales* (`hnf`) qui sont des ensembles de *nf*. Une forme *hyper-normale* peut se spécialiser en une conjonction de DNF, abrégée `cdnf`, ou en une disjonction de CNF, abrégée `dcnf`.

Il faut encore exposer pourquoi une telle structure de donnée a dû être introduite. L'algorithme de génération d'impliqués premiers a comme avantage crucial par rapport à ses concurrents d'être très flexible en terme d'entrée: il accepte des conjonctions de DNF. Il faut également noter que, en vertu des lois de de Morgan, le problème traité est dual. Cela permet donc à l'algorithme d'accepter également en entrée des disjonctions de CNF, et, dans ce cas, de donner en sortie un ensemble d'impliquants premiers, et non d'impliqués premiers.

4.2.5 Discussion de la structure de donnée adoptée

Il est évident que la représentation des données en chiffres binaires fournit à l'algorithme une efficacité très grande qu'une représentation purement symbolique ne saurait donner. Cette efficacité provient par exemple du fait que l'union et l'intersection correspondent respectivement aux fonctions `or` et `and`.

La représentation des formules logiques par des ensembles permet d'intégrer la théorie mathématique de Ngair de manière naturelle. La fonction de maximisation d'ensemble est simplifiée. On peut pourtant porter une critique importante. Il serait certainement profitable d'organiser l'information (les formes normales), par exemple à l'aide de listes ordonnées ou de tableaux pouvant changer de taille, pour éviter la recherche d'information que l'on aurait déjà acquise lors de la construction ou de la modification d'une structure; il faudrait évidemment prévoir dans les fonctions constructrices et modificatrices la manière d'introduire les nouveaux éléments au bon endroit. On pourrait ainsi faire baisser passablement la complexité, lors de la recherche d'un élément précis dans un ensemble. Pour ce qui concerne les formes hyper-normales, une modification de la structure de données allant dans ce sens n'apporterait pas forcément d'amélioration. En effet, il est difficile de trouver un critère d'ordination qui permette une complexité de base basse.

4.3 L'analyse syntaxique et sémantique. La génération de code.

L'implémentation d'un langage comporte toujours une analyse syntaxique et sémantique. En Lisp, la facilité des macros permet de résoudre très élégamment la construction d'un nouveau langage. Définir une macro signifie aller à l'encontre des règles normales d'évaluation de Lisp. Cette définition est très similaire à celle d'une fonction, à la différence que la première n'évalue par les paramètres actuels. Il est ainsi possible d'utiliser une syntaxe propre (que l'on essayera de garder toujours dans le style de Lisp), en posant des règles d'évaluation particulières. La sémantique peut également être contrôlée.

4.3.1 Les classes syntaxiques

Les macros de DIG sont quasiment toutes accompagnées par la définition d'une classe syntaxique. Une classe syntaxique est le descendant de la classe

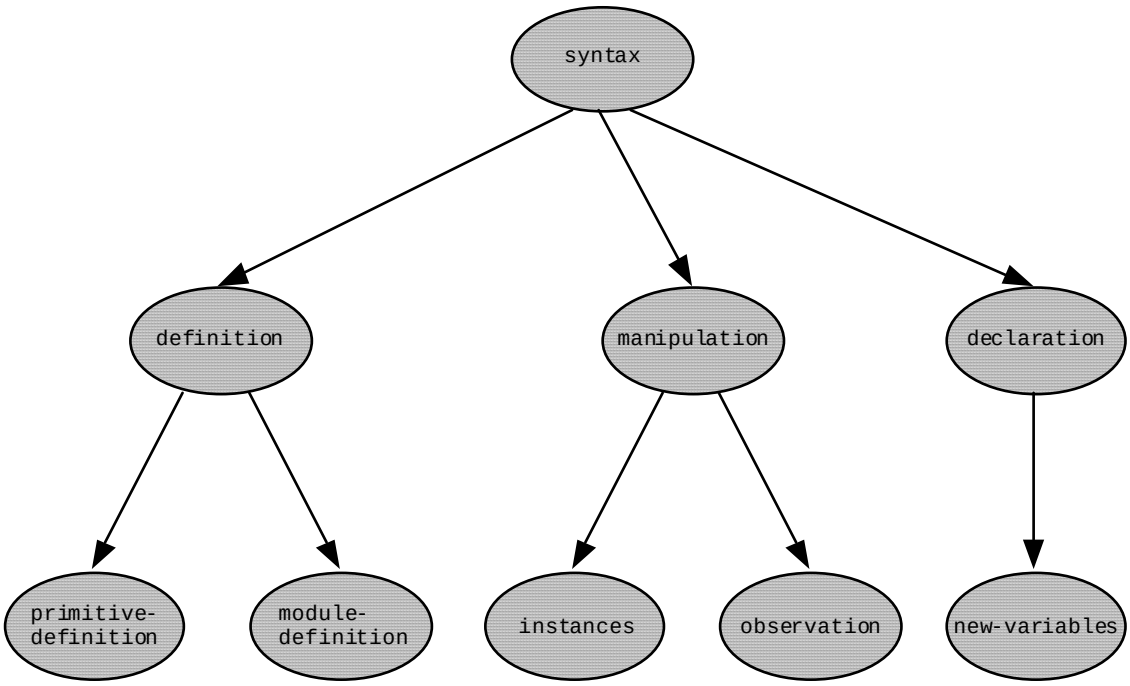


Figure 4.3: Les classes syntaxiques.

`syntax` ou d'une autre classe syntaxique. Pratiquement à chaque macro de DIG correspond l'une de ces classes. On utilise chaque paramètre actuel de la macro pour instancier l'objet syntaxique correspondant. On vérifie la syntaxe et la sémantique des paramètres actuels par la méthode `ok?` appliquée sur l'objet instancié.

Pour permettre de subdiviser les différentes macros, des sous-classes syntaxiques ont été définies. Elles sont les parents de toutes les classes correspondant aux macros (voir la figure 4.3). La classe `definition` est le parent de toutes les classes syntaxiques de définition; la classe `manipulation` des classes de manipulations des données; `declaration` des classes permettant de déclarer par exemple des variables. Une telle modélisation donne un très grand avantage: elle permet d'étendre très facilement la hiérarchie et de compléter les différentes classes par les méthodes désirées. Le code qui suit montre comment les classes parents sont définies.

```
(defclass syntax () ())

(defclass definition (syntax)
  ((name :accessor name :initarg :name)
   (body :accessor body :initarg :body)))

(defclass manipulation (syntax)
  ())
```

```
(defclass declaration (syntax)
  ())
```

Les classes syntaxiques correspondant aux macros héritent des classes mentionnées. Elles sont définies comme suit (nous indiquons également les constructeurs):

```
(defclass primitive-definition (definition)
  ())

(defun make-primitive-definition (name body)
  (make-instance 'primitive-definition :name name :body body))


(defclass module-definition (definition)
  ((lambda-list :accessor lambda-list :initarg :lambda-list)))

(defun make-module-definition (name lambda-list body)
  (make-instance 'module-definition
    :name name
    :lambda-list lambda-list
    :body body))


(defclass instances (manipulation)
  ((name :accessor name :initarg :name)
   (body :accessor body :initarg :body)))

(defun make-instances (name body)
  (make-instance 'instances :name name :body body))


(defclass observation (manipulation)
  ((object :accessor object :initarg :object)
   (variable :accessor variable :initarg :variable)
   (new-value :accessor new-value :initarg :new-value)))

(defun make-observation (object variable new-val)
  (make-instance 'observation
    :object object
    :variable variable
    :new-value new-val))


(defclass new-variables (declaration)
  ((variables :accessor variables :initarg :variables)))

(defun make-new-variables (variables)
  (make-instance 'new-variables :variables variables))
```

4.3.2 Les macros de DIG. `Defprimitive` en exemple.

La plupart des macros de DIG (`defprimitive`, `defmodule`, `new-var`, `inst` et `obs`) ont un schéma d'implémentation similaire. Après avoir créé un objet de la classe voulue, on teste la syntaxe et la sémantique à l'aide de la méthode `ok?`. Ensuite vient la partie spécifique à la macro. Cette section va se

limiter à la discussion de l'implémentation de `defprimitive`. Le code entier de DIG se trouve en annexe.

Tout d'abord, nous pouvons considérer, à titre d'exemple, la méthode `ok?` pour la classe `primitive-definition`. Auparavant, un objet de cette classe a été créé en utilisant les paramètres actuels comme *slots*. Des messages sont donnés selon les différents types d'erreur.

```
(defmethod ok? ((exp primitive-definition))
  (let ((expr (expression exp)))
    (cond
     ((not (symbolp (name exp)))
      (error "Wrong primitive definition:~a~&~a is not a symbol."
             expr
             (name exp)))
     ((not (listp (body exp)))
      (error "Wrong primitive definition: ~a
~&~a is not a list."
             expr
             (body exp)))
     ((not (>= (length (body exp)) 1))
      (error "Wrong primitive definition ~a
~&There is no primitive description."
             expr))
     ((not (= (loop for n in (body exp)
                    sum (caddr n))
              1))
      (error "Wrong primitive definition ~a
The sum of the probabilities is not equal to 1"
             expr))
     ((loop for n in (body exp)
            always
            (cond ((not (listp n))
                   (error "~a is not a list." n))
                  ((not (= (length n) 3))
                   (error "The length of ~a is not three." n))
                  ((not (symbolp (car n)))
                   (error "~a is not a symbol." (car n)))
                  ((not (symbolp (cadr n)))
                   (error "~a is not a symbol." (cadr n)))
                  ((not (numberp (caddr n)))
                   (error "~a is not a number." (caddr n)))
                  ((not (and (>= (caddr n) 0)
                             (<= (caddr n) 1)))
                   (error "~a is not between 0 and 1" (caddr n)))
                  (t nil)))
      (error "Wrong primitive definition:~&~a" expr))
     (t t))))
```

La définition d'une primitive comprend la déclaration des différents comportements possibles. Chaque comportement (`a-xor-b`, `c=1`, `a-or-b`, etc.) est défini par une macro qui forme à partir des paramètres actuels la formule DNF voulue. La macro pour `a-xor-b` est par exemple définie de la manière suivante:

```
(defmacro a-xor-b (in1 in2 out m)
  `(quote ((() . (,in1 ,in2 ,out))
           ((,in2 ,out) . (,in1))
           ((,in1 ,out) . (,in2))
```

```
          ((,in1 ,in2) . (,out))
          (() . (,m)))))
```

Comme on peut le remarquer, en passant à `a-xor-b` des paramètres, on construit la formule DNF qui décrit ce comportement:

```
? (a-xor-b x1 x2 output theMode)
((NIL X1 X2 OUTPUT)
 ((X2 OUTPUT) X1)
 ((X1 OUTPUT) X2)
 ((X1 X2) OUTPUT)
 (NIL THEMODE))
?
```

Cette formule devra être transformée dans la forme normale binaire voulue, selon l'univers prédéfini. Lors de la définition d'une primitive, une fonction nouvelle est construite à l'aide de *backquotes*. Cette phase est celle de la génération de code. On passe à chaque comportement des paramètres actuels implicites permettant la construction d'une formule DNF. Tous les comportements mis ensembles forment alors une conjonction de DNF (`cdnf`), selon la modélisation présentée à la section 2.3.2. Après avoir différentié de quel comportement il s'agit, on décide du nombre de paramètres formels à employer.

```
(defmacro defprimitive (name body)
  (let ((exp (make-primitive-definition name body)))
    (when (ok? exp)
      (case (formal-param-length body)

        (4 `(defun ,name (in1 in2 out m &optional inst-name)
              (declare (ignore inst-name))

              (setf *list-of-modes-values*
                    (cons (cons m (modes-values (quote ,body)))
                          *list-of-modes-values*))

              (loop for descr in (quote ,body)
                    for mode = (concatenate-special-3 m
                                                      '\.
                                                      (get-mode descr))
                    for logprim = (get-logprim descr)
                    for bpa = (get-bpa descr)

                    collect (eval (list logprim in1 in2 out mode))
                    into primitive-description
                    collect mode into modes
                    collect (cons mode bpa) into new-assumptions

                    finally (progn
                              (setf *temp-assumptions*
                                    (append new-assumptions
                                            *temp-assumptions*))
                              (return
                               (let ((xor-dnf (create-xor-dnf modes)))
                                 (if xor-dnf
                                   (list (cons xor-dnf
                                               primitive-description))
                                   (list primitive-description)))))))))

        (3 `(defun ,name (in out m &optional inst-name)
```

```
(declare (ignore inst-name))

(setf *list-of-modes-values*
      (cons (cons m (modes-values (quote ,body)))
            *list-of-modes-values*))

(loop for descr in (quote ,body)
      for mode = (concatenate-special-3 m
                                        '\.
                                        (get-mode descr))
      for logprim = (get-logprim descr)
      for bpa = (get-bpa descr)

      collect (eval (list logprim in out mode))
      into primitive-description
      collect mode into modes
      collect (cons mode bpa) into new-assumptions

      finally (progn
                (setf *temp-assumptions*
                      (append new-assumptions
                              *temp-assumptions*))
                (return
                 (let ((xor-dnf (create-xor-dnf modes)))
                   (if xor-dnf
                     (list (cons xor-dnf
                                 primitive-description))
                     (list primitive-description)))))))))))
```

4.4 Construction et modification de la base de données. Différentes interrogations.

Il est intéressant d'étudier comment DIG construit les réseaux de valuations au moyen des outils mis à disposition par TEPI. L'élaboration des réseaux en DIG est résumée par la figure 4.4; celle-ci n'est évidemment pas complète, mais elle permet de se faire une idée du procédé choisi. Une consultation du code permettra de se faire une idée plus exacte de comment on a procédé.

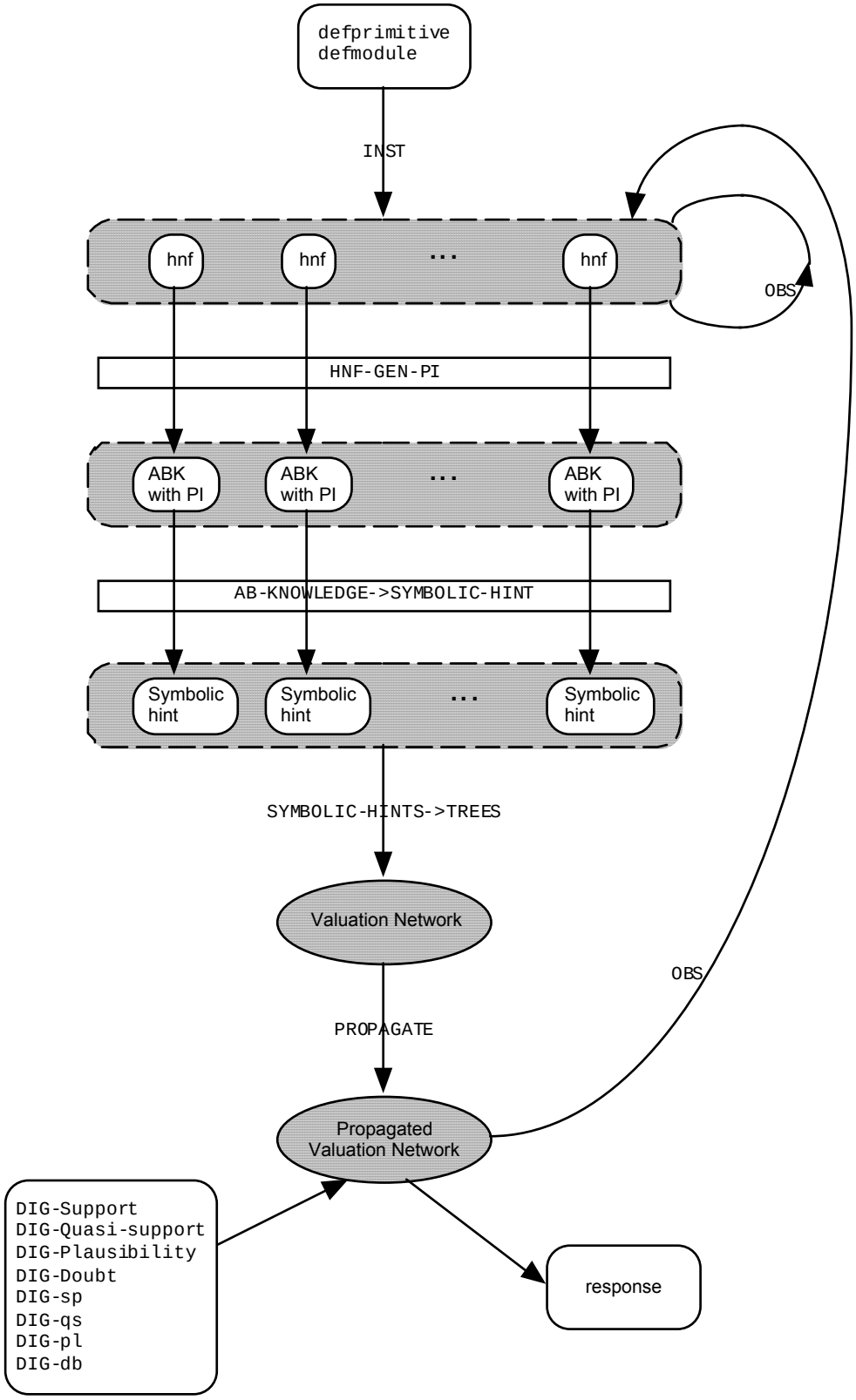


Figure 4.4: La construction du réseau de valuations.

En instanciant un module, on construit un objet du type `gate` contenant toutes les informations nécessaires à la construction du réseau de valuations. Le nouvel objet possède la description complète du circuit. Nous rappelons que cette description est constituée pour chaque primitive instanciée (pour chaque porte) d'une conjonction de DNF.

Si l'on apporte des nouvelles connaissances à ce circuit à l'aide de la macro `obs`, on devra créer une nouvelle formule DNF à rajouter à chaque description de porte où la variable concernée est présente. Pour assurer que le réseau ultérieur construit bien un arbre, on renomme ces variables.

C'est seulement au moment d'une interrogation du système que le réseau de valuations est construit. On génère tout d'abord pour chaque composante de base les impliqués premiers à l'aide de la fonction `HNF-GEN-PI`. Chaque ensemble d'impliqués premiers est utilisé pour la composition d'un objet de la classe `ab-knowledge` (abrégée `ABK`). Il en résulte donc une liste d'objets de cette classe. En appliquant la fonction `AB-KNOWLEDGE->SYMBOLIC-HINT` à chacun de ces objets, on crée une liste de *hints* symboliques [Kohlas, Monney 1995] qui va être utilisée pour la construction du réseau de valuations à l'aide de la fonction `SYMBOLIC-HINTS->TREES`. Le réseau est alors propagé.

Si l'interrogation du système ne redonne aucune réponse, on propage à nouveau le réseau de valuations après y avoir apporté des modification adéquates. Au cas où cette construction du réseau ne permet toujours pas d'obtenir une réponse, on abandonne le processus.

Si l'utilisateur désire faire d'autres interrogations, le système les réalisera sur le réseau déjà propagé, ce qui, normalement, ne représentera pas de grands délais. Si, toutefois, on avait auparavant apporté quelque observation, il faudra reconstruire toute la base de données. En effet, il ne faut pas oublier qu'une observation représente une nouvelle formule logique pouvant influencer le réseau de valuations.

5 Conclusion

La génération des impliqués premiers permet une généralisation du système ATMS proposé par DeKleer. Ngair a présenté une méthode efficace pour obtenir ces impliqués. Son algorithme possède comme avantage important qu'il est plus flexible quant à son entrée: il accepte des conjonctions de DNF. *GEN-PI* atteint ainsi un niveau de généralisation plus élevé que les autres algorithmes de génération d'impliqués premiers. Pour les problèmes exprimés en conjonctions, il est tout autant efficace que ses concurrents.

Le système DIG a été défini pour permettre d'appliquer directement *GEN-PI* à un problème de diagnostique. La modélisation des circuits en conjonctions de DNF a permis facilement cette application. Le langage défini donne les outils adéquats pour la description de ces circuits défectueux.

L'implémentation de DIG résout très élégamment l'algorithme que Ngair a proposé pour la génération d'impliqués premiers. L'environnement de programmation a permis une grande concision. La facilité du `loop` a constitué un outil puissant pour implémenter les parties itératives. Les facilités de `macro` ont permis la définition d'un nouveau langage, en laissant au programmeur la tâche de vérifier la syntaxe et la sémantique des paramètres actuels. La programmation orientée objet donna les instruments les plus adéquats pour réaliser cette vérification. Le système démontre également qu'il est facilement réutilisable et extensible.

DIG a beaucoup gagné par l'intégration du langage d'interrogation d'ABL. Cette intégration a démontré comment un code peut être directement réutilisé et adapté à des besoins spécifiques.

Pour une extension future du système, il serait souhaitable de prendre en considération quelques points. Il est possible d'optimiser encore l'algorithme de Ngair. La syntaxe de DIG, en ce qui concerne la définition de primitives et de modules, pourrait se rapprocher de celle du langage d'interrogation d'ABL. L'interface graphique pourrait être améliorée. On pourrait notamment permettre à l'utilisateur la définition de primitives et la construction de modules à l'aide

d'outils graphiques. Ces quelques propositions ne se veulent pas exhaustives. Elles doivent être le point de départ d'une réflexion pouvant amener à une amélioration du système actuel.

Annexe A Grammaire de DIG

DIG ::=	{primitive-definition | module-definition | new-variables | instance | observation | query-exp | module-deletion | gate-deletion | menu-installation}*
primitive-definition ::=	(**`defprimitive`** name primitive-body)
primitive-body ::=	({primdef-elt}*)
primdef-elt ::=	(mode mode-description bpa)
mode ::=	symbol
mode-description ::=	**`a-and-b`** | **`a-nand-b`** | **`a-or-b`** | **`a-nor-b`** | **`a-xor-b`** | **`c=0`** | **`c=1`** | **`c=a`** | **`c=not-a`** | **`c=b`** | **`c=not-b`** | **`tautology-3`** | **`b=a`** | **`b=not-a`** | **`b=1`** | **`b=0`** | **`tautology-2`**
bpa ::=	[**0** . . **1**]
module-definition ::=	(**`defmodule`** name lambda-list module-body)
lambda-list ::=	({var}*)
module-body ::=	(**`int`** intern-variables core)
intern-variables ::=	((**`wires`** {var}*) (**`modes`** {var}*))
core ::=	{instance-in-module}*
instance-in-module ::=	(**`inst`** var (module-or-primitive {var}*)
module-or-primitive ::=	symbol
new-variables ::=	**`(new-var`** variable-list**`)`**
variable-list ::=	{var}*

instance ::=	(**`inst`** name module-name)
module-name ::=	({primitive | module | **`and@`** | **`nand@`** | **`or@`** | **`nor@`** | **`xor@`** | **`not@`**} {var}*)
primitive ::=	symbol
module ::=	symbol
observation ::=	(**`obs`** gate var value)
gate ::=	symbol
value ::=	symbol | **0** | **1**
query-exp ::=	({**`DIG-support`** | **`DIG-sp`** | **`DIG-quasi-support`** | **`DIG-qs`** | **`DIG-doubt`** | **`DIG-db`** | **`DIG-plausibility`** | **`DIG-pl`**} gate (hypothesis*))
hypothesis ::=	litteral | relation
litteral ::=	var
relation ::=	({unary-logical-function var} | {n-ary-logical-function arguments})
unary-logical-function ::=	{**`not`** | **`not*`**}
n-ary-logical-function::=	{**`and`** | **`and*`** | **`or`** | **`or*`**}
arguments ::=	var {var}$^{+}$
module-deletion ::=	(**`delete-module`** module)
gate-deletion ::=	(**`delete-gate`** gate)
menu-installation ::=	(**`install-menu`**)
var ::=	symbol
name ::=	symbol

Annexe B Bibliographie

[Chang, Lee 1973] : Chang C.E., Lee R.C.T. - Symbolic Logic and Mechanical Theorem Proving. Academic Press.

[DeKleer 1986a] : DeKleer J. - An assumption-based TMS. Artificial Intelligence, 28, 127-162.

[DeKleer 1986b] : DeKleer J. - Extending the ATMS. Artificial Intelligence, 28, 163-196.

[DeKleer 1987] : DeKleer J., Williams B.C. - Diagnosing Multiple Faults. Artificial Intelligence, 32, 97-130.

[Haenni, Monney, Kohlas 1995] : Haenni R., Monney P.A., Kohlas J. - Assumption-Based Reasoning and Model-Based Diagnostics. Institute of Informatics, University of Fribourg.

[Haenni 1995] : Haenni R. - Theory of Evidence Programming Interface. Institute of Informatics, University of Fribourg.

[Kenne 1989] : Keene S. - Object-Oriented Programming in Common Lisp. Addison-Wesley Publishing Company. Reading, Massachusetts.

[Kohlas 1993a] : Kohlas J. - Formale Methoden des Reasoning. Blockseminar Bern-Fribourg. Institut für Informatik, Universität Freiburg, Schweiz.

[Kohlas 1993b] : Kohlas J. - Symbolic Evidence, Arguments, Supports and Valuation Networks. In: "Symbolic and Quantitative Approaches to Reasoning and Uncertainty", pages 186-198, Springer.

[Kohlas, Monney 1995] : Kohlas J., Monney P.A. - A Mathematical Theory of Hints. An Approach to the Dempster Schafer Theory of Evidence. Lectures notes in Economics and Mathematical systems, 425, Springer.

[Lehmann 1994] : Lehmann N. - .Entwurf und Implementation einer annahmenbasierten Sprache. Institute of Informatics, University of Fribourg.

[MCL 1992] : Apple Computer, Inc - Macintosh Common Lisp Reference, Version 2.0. APDA.

[Ngair 1992] : Ngair T.H. - Convex spaces as an Order-theoric Basis for Problem Solving. University of Pennsylvania.

[Norvig 1992] : Norvig P. - Paradigms of Artificial Intelligence Programming. Case Studies in Common Lisp. Morgan Kaufmann Publishers. San Mateo, California.

[Reiter 1987] : Reiter R. - A Theory of Diagnosis From First Principles. Artificial Intelligence, Elsevier Science Publisher B.V. (Amsterdam), 32, 57-95.

[Reiter, DeKleer 1987] : Reiter R., DeKleer J. - Foundations of assumption-based truth maintenance systems: Preliminary report. Proc. of AAAI-87, pages 183-188.

[Schumacher 1994] : Schumacher M. - Model based diagnostic of digital circuits using Dighint. Institute of Informatics, University of Fribourg.

[Shenoy 1993] : Shenoy P.P. - Valuation Networks and Conditional Independence. In: "Uncertainty in Artificial Intelligence, Proceeding of the 9th Conference" (D. Heckermann and E. Mamdani, Eds.), pages 191-199, Morgan Kaufmann Publishers, Inc.

[Steele 1992] : Steele G.L. jr. - Common Lisp, the Language. Digital Press.

[Tannenbaum 1984] : Tannenbaum A. - Structured Computer Organization. Printice-Hall, Inc., Englewood Cliffs.

Annexe C Implémentation de DIG

Annexe C1 `Utilities.lisp`

```
; ——————————————————————————————————————————————————————————————————————————
——
; ———                              FRIBOURG UNIVERSITY                          —
——
; ———                           INSTITUTE OF INFORMATICS                        —
——
; ———                 Rue Faucigny 2, CH-1700 Fribourg, Switzerland             —
——
; ——————————————————————————————————————————————————————————————————————————
——
; TITLE:    utilities.lisp
; SUPPORT:  Michael Schumacher
; PROJECT:  DIG
; DATE:     15.05.95
; VERSION:  1.0
; ——————————————————————————————————————————————————————————————————————————
——
; OVERVIEW: différentes utilités.
; ——————————————————————————————————————————————————————————————————————————
——



; ----------------------------------------------------------------------------
--
; FONCTIONS SUR DES LISTES
; ----------------------------------------------------------------------------
--

; accumulate est défini comme un accumulateur:

(defmacro accumulate (theList funct initValue)
  `(loop for i in (cons ,initValue ,theList)
         for acc = i then (funcall ,funct acc i)
         finally (return acc)))


; Set-difference est identique à set-difference, à la différence prêt qu'elle
; ne réalise pas d'inversion.

(defun set-difference* (s1 s2 &key test)
  (loop for elt1 in s1
        when (not (loop for elt2 in s2
                        thereis (funcall test elt1 elt2)))
        collect elt1))


; cmp compare deux listes de sets:

(defun cmp (s1 s2)
  (let ((res (append
              (list "1st" (loop for n1 in s1
                                when (not (loop for n2 in s2
                                                thereis (ls-equal? n1 n2)))
                                collect n1))
              (list "2nd" (loop for n2 in s2
                                when (not (loop for n1 in s1
```

```
                                                thereis (ls-equal? n1 n2)))
                                  collect n2)))))
    (if res res T)))


; Flatten-list transforme la liste my-list avec structure d'arbre en une liste
; contenant les noeux de cet arbre.

(defun flatten-list (list)
  "Transform the tree list into a list of his nodes."
  (loop for n in list
        append (if (atom n)
                   (list n)
                   (flatten-list n))))


; Filter filtre la liste my-list au moyen du test test.

(defun filter (list &optional test)
  (loop for n in list
        when (funcall test n) collect n))


; Included? teste si l'élément key se trouve dans la liste the-list

(defun included? (key list)
  "Test with equal? if key is in list."
  (loop for s in list
        if (equal? s key)
        return t))


; Alist est une liste de listes. My-assq cherche la liste dont le premier élément
; est identique à key.

(defun my-assq (key alist)
  "In the list alist, my-assq searches the list beginning with key."
  (assoc key alist :test #'eq))



; ------------------------------------------------------------------------------
--
; OPERATIONS SUR DES SYMBOLES
; ------------------------------------------------------------------------------
--

; Concatenate-symbols crée un nouveau symbole à partir des symboles reçus comme
; paramètres.

(defun concatenate-symbols (&rest symbols)
  "Concatenates the symbols list into a new symbol."
  (make-symbol (eval (append '(concatenate 'string)
                             (loop for s in symbols
                                   collect (string s))))))


; Concatenate-special-3 unit trois symboles en un seul.

(defun concatenate-special-3 (first second third)
  (cond ((null first)
         third)
        (t (concatenate-symbols first second third))))



; ------------------------------------------------------------------------------
--
; COMPARAISONS:
; ------------------------------------------------------------------------------
--

(defun sp-equal? (a b)
  (equal (string a) (string b)))
```

Annexe C2 `Nf-max.lisp`

```
; ——————————————————————————————————————————————————————————————————————————
—
; ——                              FRIBOURG UNIVERSITY                          —
—
; ——                            INSTITUTE OF INFORMATICS                       —
—
; ——                  Rue Faucigny 2, CH-1700 Fribourg, Switzerland            —
—
; ——————————————————————————————————————————————————————————————————————————
—
; TITLE:    nf-MAX.lisp
; SUPPORT:  Michael Schumacher
; PROJECT:  DIG
; DATE:     18.2.95
; VERSION:  1.0
; ——————————————————————————————————————————————————————————————————————————
—
; OVERVIEW: implémentation du Maximum.
; ——————————————————————————————————————————————————————————————————————————
—


; C'est la version la plus simple du maximum qui est implémentée ici. Des optimi-
; sations peuvent être apportées. Elles n'ont pas été retenues, parce que la
; complexité de la mémoire devient complètement disproportionnée à celle du temps.


(defun nf-MAX (nf-C)
  (loop for ls1 in nf-C
        when (loop for ls2 in nf-C
                   always (if (ls-subset? ls2 ls1)
                              (equal? ls2 ls1)
                              T))
        collect ls1))
```

Annexe C3 `Nf-intersection+.lisp`

```
; ──────────────────────────────────────────────────────────────────────────────
;                                FRIBOURG UNIVERSITY                           
;                             INSTITUTE OF INFORMATICS                         
;                  Rue Faucigny 2, CH-1700 Fribourg, Switzerland               
; ──────────────────────────────────────────────────────────────────────────────
; TITLE:    nf-intersection+.lisp
; SUPPORT:  Michael Schumacher
; PROJECT:  DIG
; DATE:     15.05.95
; VERSION:  1.0
; ──────────────────────────────────────────────────────────────────────────────
; OVERVIEW: Implémentation de l'algorithme d'intersection proposé par Ngair avec
;           ses optimisations.
; ──────────────────────────────────────────────────────────────────────────────



; Ce fichier comporte l'implémentation l'intersection, telle que l'algorithme est
; proposé par Ngair. On y a apporté encore des optimisations personnelles.
; Cette version de nf-intersection+ exige que nf1 et nf2 soient maximums.


(defun ls-PPI-condition (ppi-ls ¬X-nf S-nf)
  (loop for s-ls in S-nf
        thereis (loop for e-ls in ¬X-nf
                      thereis (ls-subset? s-ls (ls-union ppi-ls e-ls)))))


(defun nf-intersection-with-matrix (nf1
                                    nf2
                                    i_nf1 i_nf2
                                    removed-nf1-elts removed-nf2-elts
                                    X-nf
                                    S-nf)
  (when (and (not (null nf1))
             (not (null nf2)))
    (let* ((x (length nf1))
           (y (length nf2))
           (A (make-array (list x y)))
           (a_i_nf1 (make-array x))
           (a_i_nf2 (make-array y)))

      ; On ignorer S-nf et X-nf qui peuvent être résutilisés pour des
      ; optimisations ultérieures.
      S-nf X-nf
      ; initialisation des tableaux supplémentaires
      (loop for n in i_nf1
            for ls in removed-nf2-elts
            do (setf (aref a_i_nf1 n) (cons ls (aref a_i_nf1 n))))
      (loop for n in i_nf2
            for ls in removed-nf1-elts
            do (setf (aref a_i_nf2 n) (cons ls (aref a_i_nf2 n))))

      ; initialisation du tableau principal
      (loop for ls1 in nf1
            for i from 0 to (1- x)
            do
            (loop for ls2 in nf2
                      for j from 0 to (1- y)
                      do (setf (aref A i j) (ls-union ls1 ls2))))
      (loop for i from 0 to (1- x)
            append
            (loop for j from 0 to (1- y)
                  for elt-ls = (aref A i j)
```

```
                when
                (and  elt-ls
                      (loop for i1 from 0 to (1- x)
                            always (if (ls-subset? (aref A i1 j) elt-ls)
                                       (ls-equal? (aref A i1 j) elt-ls)
                                       T))
                      (loop for j1 from 0 to (1- y)
                            always (if (ls-subset? (aref A i j1) elt-ls)
                                       (ls-equal? (aref A i j1) elt-ls)
                                       T))
                      (if (aref a_i_nf2 j)
                        (loop for theElt-ls in (aref a_i_nf2 j)
                              always (if (ls-subset? theElt-ls elt-ls)
                                         (ls-equal? theElt-ls elt-ls)
                                         T))
                        T)
                      (if (aref a_i_nf1 i)
                        (loop for theElt-ls in (aref a_i_nf1 i)
                              always (if (ls-subset? theElt-ls elt-ls)
                                         (ls-equal? theElt-ls elt-ls)
                                         T))
                        T))
                collect elt-ls)))))


(defun nf-intersection+ (nf1 nf2 X-nf S-nf)
  (labels
    ((give_list_of_new_indexes
         (indexes_of_removed_elts_to_cmp_in_new_array
          indexes_of_eliminated_elements)
       (let ((new_indexes_of_eliminated_elements
              (remove-duplicates indexes_of_eliminated_elements)))
         (loop for n1 in (cons NIL new_indexes_of_eliminated_elements)
               for res = indexes_of_removed_elts_to_cmp_in_new_array
               then (loop for n2 in indexes_of_removed_elts_to_cmp_in_new_array
                          for n3 in res
                          when (< n1 n2)
                          collect (1- n3) into tempRes
                          else collect n3 into tempRes
                          finally (return tempRes))
               finally (return res)))))

    (loop for ls1 in nf1
          for index_i = 0 then (+ index_i 1)
          for rem-elts = (loop for ls2 in nf2
                               for index_j = 0 then (+ index_j 1)

                               when (ls-equal? ls1 ls2)
                               collect ls1 into removed-elt-from-nf1
                               and collect ls2 into removed-elt-from-nf2
                               and collect index_j into index_removed-elt-from-nf2
                               and collect index_i into index_removed-elt-from-nf1

                               else

                               when (ls-subset? ls1 ls2)
                               collect ls2 into removed-elt-from-nf2
                               and collect index_j into index_removed-elt-from-nf2
                               and collect index_i into index_nf1_of_detected_max

                               else

                               when (ls-subset? ls2 ls1)
                               collect ls1 into removed-elt-from-nf1
                               and collect index_i into index_removed-elt-from-nf1
                               and collect index_j into index_nf2_of_detected_max

                               finally (return (list removed-elt-from-nf1
                                                     removed-elt-from-nf2
                                                     index_removed-elt-from-nf1
                                                     index_removed-elt-from-nf2
                                                     index_nf1_of_detected_max
                                                     index_nf2_of_detected_max)))

          append (first  rem-elts) into removed-nf1-elts
          append (second rem-elts) into removed-nf2-elts
          append (third  rem-elts) into index-removed-nf1-elts
```

```
append (fourth rem-elts) into index-removed-nf2-elts
append (fifth  rem-elts) into index_nf1_of_detected_max
append (sixth  rem-elts) into index_nf2_of_detected_max

finally
(let ((new-nf1 (set-difference* nf1 removed-nf1-elts :test #'ls-equal?))
      (new-nf2 (set-difference* nf2 removed-nf2-elts :test #'ls-equal?))
      (i_nf1 (give_list_of_new_indexes index_nf1_of_detected_max
                                       index-removed-nf1-elts))
      (i_nf2 (give_list_of_new_indexes index_nf2_of_detected_max
                                       index-removed-nf2-elts)))
  (return
   (append
    (union removed-nf1-elts removed-nf2-elts :test #'ls-equal?)
    (nf-intersection-with-matrix new-nf1
                                 new-nf2
                                 i_nf1 i_nf2
                                 removed-nf1-elts removed-nf2-elts
                                 X-nf
                                 S-nf)))))))
```

Annexe C4 `Hnf-meet.lisp`

```
; ————————————————————————————————————————————————————————————————————————————
;;;                                FRIBOURG UNIVERSITY                          ;;;
;;;                             INSTITUTE OF INFORMATICS                        ;;;
;;;                  Rue Faucigny 2, CH-1700 Fribourg, Switzerland              ;;;
; ————————————————————————————————————————————————————————————————————————————
; TITLE:    hnf-meet.lisp
; SUPPORT:  Michael Schumacher
; PROJECT:  DIG
; DATE:     21.2.95
; VERSION:  1.0
; ————————————————————————————————————————————————————————————————————————————
; OVERVIEW: Implementation de l'opération meet.
; ————————————————————————————————————————————————————————————————————————————


; Le résultat d'un hnf-extended-intersection est une forme normale.

(defun hnf-extended-intersection (hnf X-nf S-nf)
  "The result of hnf-extended-intersection is a nf"
  (loop
    for nf in hnf
    for new-X-nf on X-nf
    for acc-nf = (remove-duplicates nf :test #'ls-equal?)
    then (nf-intersection+ acc-nf nf new-X-nf S-nf)
    finally (return acc-nf)))


; Le résultat d'un hnf-meet est une forme normale.

(defun hnf-meet (hnf X-nf S-nf)
  (when (not (null hnf))
    (hnf-extended-intersection hnf X-nf S-nf)))
```

Annexe C5 `Hnf-join.lisp`

```
; ——————————————————————————————————————————————————————————————————————————
—
; ——                              FRIBOURG UNIVERSITY                          —
—
; ——                           INSTITUTE OF INFORMATICS                        —
—
; ——                 Rue Faucigny 2, CH-1700 Fribourg, Switzerland             —
—
; ——————————————————————————————————————————————————————————————————————————
—
; TITLE:    hnf-join.lisp
; SUPPORT:  Michael Schumacher
; PROJECT:  DIG
; DATE:     11.4.95
; VERSION:  1.0
; ——————————————————————————————————————————————————————————————————————————
—
; OVERVIEW: Implémentation de l'opération join.
; ——————————————————————————————————————————————————————————————————————————
—



; JOIN:
; -----

(defun hnf-extended-union (hnf)
  (loop for nf in hnf
        for acc-nf = nf then (union nf acc-nf :test #'ls-equal?)
        finally (return acc-nf)))


(defun hnf-join (hnf)
  "The result of hnf-join is a nf"
  (when (not (null hnf))
    (nf-MAX (hnf-extended-union hnf))))


; nf-special-join
; ---------------

; cette version fait un join sur deux paramètres. On part du principe qu'ils sont
; maximums. Il s'agit d'une optimisation importante.

(defun nf-special-join (nf1 nf2)
  (if (nf-empty? nf1)
    nf2
    (if (nf-empty? nf2)
      nf1
      (let*
        ((x (length nf1))
         (y (length nf2))
         (A1 (make-array x))
         (A2 (make-array y))
         (not-considered 1))
        (append
         (loop for ls1 in nf1
               for i1 from 0 to (1- x)
               for res-ls = (loop for ls2 in nf2
                                  for i2 from 0 to (1- y)
                                  when (not (aref A2 i2))
                                  when (ls-subset? ls1 ls2)
                                  do (setf (aref A2 i2) not-considered)
                                  else
                                  when (ls-subset? ls2 ls1)
                                  do (setf (aref A1 i1) not-considered)
                                  and do (return NIL)
                                  else
                                  when (ls-equal? ls1 ls2)
                                  do (setf (aref A2 i2) not-considered)
                                  and do (return ls1)
```

```
                          finally (return ls1))
      when res-ls collect res-ls)
(loop for ls2 in nf2
      for i2 from 0 to (1- y)
      when (not (aref A2 i2))
      collect ls2))))))
```

Annexe C6 `Nf-PHI.lisp`

```
; ——————————————————————————————————————————————————————————————————————
——
; ———                              FRIBOURG UNIVERSITY                              —
——
; ———                            INSTITUTE OF INFORMATICS                           —
——
; ———                 Rue Faucigny 2, CH-1700 Fribourg, Switzerland                 —
——
; ——————————————————————————————————————————————————————————————————————
——
; TITLE:    nf-PHI.lisp
; SUPPORT:  Michael Schumacher
; PROJECT:  DIG
; DATE:     18.5.95
; VERSION:  1.0
; ——————————————————————————————————————————————————————————————————————
——
; OVERVIEW: Implémentation de l'opération P-PHI, comme elle est décrite par Ngair.
; ——————————————————————————————————————————————————————————————————————
——


; Ce fichier implémente l'opération P-PHI, telle qu'elle est décrite dans Ngair.


; ------------------------------------------------------------------------------
--
; CONDITIONS D'OPTIMISATION
; ------------------------------------------------------------------------------
--

(defun ls-row-done? (cell-ls)
  (ls-empty? cell-ls))


(defun ls-empty-cell? (ls cell-ls)
  (ls-equal? ls cell-ls))


(defun nf-disjoint? (T-nf)
  (loop for ls1 in T-nf
        for x = 1 then (1+ x)
        always (loop for ls2 in T-nf
                     for y = 1 then (1+ y)
                     always (if (ls-empty? (ls-intersection ls1 ls2))
                                T
                                (and (= x y)
                                     (ls-equal? ls1 ls2))))))


(defun nf-only-literals? (T-nf)
  (loop for ls in T-nf
        always (= (ls-member ls)
                  1)))


; ------------------------------------------------------------------------------
--
; L'OPERATION P-PHI
; ------------------------------------------------------------------------------
--

(defun  nf-PHI-for-lit-and-dis (S-nf T-nf)
  (let* ((matrixdone T)
         (hnf2 (loop for tj-ls in T-nf
                     for hnf1 =
                     (progn
                       (setf matrixdone T)
                       (loop for si-ls in S-nf
```

```
                              for cell-ls = (ls-difference si-ls tj-ls)
                              when (ls-row-done? cell-ls)
                              do (setf matrixdone NIL)
                              and return '()
                              else
                              when (ls-empty-cell? si-ls cell-ls)
                              do (setf matrixdone (and matrixdone T))
                              else
                              do (setf matrixdone NIL)
                              and when (ls-subset? tj-ls si-ls)
                              collect (list cell-ls)))
                        when matrixdone return matrixdone
                        else
                        when hnf1
                        collect (hnf-join hnf1))))
      (if matrixdone
        S-nf
        (if hnf2
          (nf-special-join S-nf (hnf-meet hnf2 T-nf S-nf))))))


(defun nf-PHI-normal (S-nf T-nf)
  (let* ((matrixdone T)
         (hnf2 (loop for tj-ls in T-nf
                     for hnf1 =
                     (progn
                       (setf matrixdone T)
                       (loop for si-ls in S-nf
                             for cell-ls = (ls-difference si-ls tj-ls)
                             when (ls-row-done? cell-ls)
                             do (setf matrixdone NIL)
                             and return '()
                             else
                             when (not (ls-empty-cell? si-ls cell-ls))
                             do (setf matrixdone NIL)
                             and collect (list cell-ls)
                             else
                             do (setf matrixdone (and matrixdone T))))
                     when matrixdone return matrixdone
                     else
                     when hnf1
                     collect (hnf-join hnf1))))
      (if matrixdone
        S-nf
        (if hnf2
          (nf-special-join S-nf (hnf-meet hnf2 T-nf S-nf))))))


(defun nf-PHI (S-nf T-nf)
  (if (and (nf-only-literals? T-nf)
           (nf-disjoint? T-nf))
    (nf-PHI-for-lit-and-dis S-nf T-nf)
    (nf-PHI-normal S-nf T-nf)))


; Avec une representation en array:
; ---------------------------------

(defun nf-PHI-for-lit-and-dis (S-nf T-nf)
  (let* ((X-nf T-nf)
         (xi (length S-nf))
         (yj (length T-nf))
         (A (make-array (list xi yj)))
         (Y (make-array yj))
         (rowdone (make-array yj))
         (matrixdone T))
    (loop for tj-ls in T-nf
          for j from 0 to (1- yj)
          do (loop for si-ls in S-nf
                   for i from 0 to (1- xi)
                   for cell-ls = (ls-difference si-ls tj-ls)
                   when (ls-row-done? cell-ls)
                   do (progn (setf matrixdone NIL)
                             (setf (aref rowdone j) T))
                   else
                   when (ls-empty-cell? si-ls cell-ls)
                   do (setf matrixdone (and matrixdone T))
```

```
                else
                do (setf matrixdone NIL)
                and when (ls-subset? tj-ls si-ls)
                do (setf (aref A i j) (list cell-ls))
                until (aref rowdone j))
        until matrixdone)
  (if matrixdone
    S-nf
    (loop
      for j from 0 to (1- yj)
      when (not (aref rowdone j))
      do (setf (aref Y j)
               (hnf-join (loop for i from 0 to (1- xi)
                               for elt = (aref A i j)
                               when elt collect elt)))

      finally (let ((res-nf (hnf-meet
                             (loop for j1 from 0 to (1- yj)
                                   when (not (aref rowdone j1))
                                   collect (aref Y j1))
                             X-nf
                             S-nf)))
                (if res-nf
                  (return (nf-special-join S-nf res-nf))
                  (return S-nf)))))))
```

Annexe C7 `Hnf-gen-PI.lisp`

```
; ------------------------------------------------------------------------------
--
; ---                          FRIBOURG UNIVERSITY                             -
--
; ---                        INSTITUTE OF INFORMATICS                          -
--
; ---              Rue Faucigny 2, CH-1700 Fribourg, Switzerland               -
--
; ------------------------------------------------------------------------------
--
; TITLE:    hnf-gen-PI.lisp
; SUPPORT:  Michael Schumacher
; PROJECT:  DIG
; DATE:     15.05.95
; VERSION:  1.0
; ------------------------------------------------------------------------------
--
; OVERVIEW: Algorithme de la génération de prime implicates ou prime implicants,
;           proposé par Ngair.
; ------------------------------------------------------------------------------
--



; ------------------------------------------------------------------------------
--
; EXTENSION DES BS
; ------------------------------------------------------------------------------
--

(defsynonym bs-index-elt? logbitp)


(defun bs-get-single-elt (bs)
  (loop with max = (integer-length bs)
        for index = 0 then (1+ index)
        while (< index max)
        when (bs-index-elt? index bs)
        collect (ash 1 index)))



; ------------------------------------------------------------------------------
--
; EXTENSION DES LS
; ------------------------------------------------------------------------------
--

(defun ls-symbols (nf)
  (accumulate nf #'ls-union '(0 . 0)))



; ------------------------------------------------------------------------------
--
; EXTENSION DES NF
; ------------------------------------------------------------------------------
--

(defun nf-tautological-list (bs-list)
  (loop for bs in bs-list
        collect (make-ls bs bs)))



; ------------------------------------------------------------------------------
--
; GENERATION DE PRIME IMPLICATES OU DE PRIME IMPLICANTS
; ------------------------------------------------------------------------------
--
```

Génération d'impliqués premiers et déduction basée sur suppositions au moyen de clauses disjonctives.

```
; nf-gen-PI génère des pi à partir d'un ensemble de prime implicates (ou respecti-
; vement de prime implicants), c-à-d S-nf, et d'un DNF (ou respectivement d'un
CNF),
; c-à-d T-nf.

(defun nf-gen-PI (S-nf T-nf)
  (let* ((S-bs-symbols (ls->bs (ls-symbols S-nf)))
         (T-bs-symbols (ls->bs (ls-symbols T-nf)))
         (tautological-PI-nf (nf-tautological-list
                              (bs-get-single-elt
                               (bs-difference T-bs-symbols S-bs-symbols)))))
    (nf-PHI (append S-nf tautological-PI-nf) T-nf)))


; hnf-gen-PI génère des primes implicates si S-hnf est une cdnf,
; sinon des prime implicants si S-hnf est une dcnf.

(defun hnf-gen-PI (S-hnf)
  (loop for nf in (cons NIL S-hnf)
        for Prime-Implicates = nf then
        (nf-GEN-PI Prime-Implicates nf)
        finally (return (nf-complement Prime-Implicates))))
```

Annexe C8 `Hnf.lisp`

```
; ————————————————————————————————————————————————————————————————————————————
—
; ——                              FRIBOURG UNIVERSITY                           —
—
; ——                            INSTITUTE OF INFORMATICS                        —
—
; ——                 Rue Faucigny 2, CH-1700 Fribourg, Switzerland              —
—
; ————————————————————————————————————————————————————————————————————————————
—
; TITLE:    hnf.lisp
; SUPPORT:  Michael Schumacher
; PROJECT:  DIG
; DATE:     29.06.95
; VERSION:  1.0
; ————————————————————————————————————————————————————————————————————————————
—
; OVERVIEW: hnf, cdnf et dcnf
; ————————————————————————————————————————————————————————————————————————————
—


; Ce fichier contient les types de données hnf (hyper normal forms). Il s'agit
; d'une généralisation de cdnf (conjunction of disjunctive normal forms) et
; de dcnf (disjunction of conjunctive normal forms).
; Les fonctions suivantes ont été definis dans d'autres fichiés, pour favoriser
; la modularité. Elles font donc implicitement partie de ce module.
;   - hnf-gen-PI
;   - hnf-extended-union
;   - hnf-join
;   - hnf-extended-intersection
;   - hnf-meet


(defsynonym make-hnf list)

(defun hnf-build (new-nf hnf)
  (cons new-nf hnf))

(defsynonym hnf-first first)

(defsynonym hnf-second second)

(defsynonym hnf-rest rest)

(defsynonym hnf-third->rest cddr)

(defsynonym hnf-length length)

(defun hnf-get-universe (hnf)
  (loop for nf in hnf
        append (remove-duplicates (loop for ls in nf
                                        when (not (null ls))
                                        append (append (+L ls) (-L ls)))
                                  :test #'sp-equal?)))

(defun get-universe (hnf-list)
  (remove-duplicates (loop for hnf in hnf-list
                           append (hnf-get-universe hnf))
                     :test #'sp-equal?))



; ----------------------------------------------------------------------------
--
; cdnf: CONJUNCTION OF DNF
; ----------------------------------------------------------------------------
--

(defun cdnf-contradiction? (hnf)
```

```
  (loop for nf in hnf
        thereis (dnf-contradiction? nf)))

(defun cdnf-tautology? (hnf)
  (loop for nf in hnf
        always (dnf-tautology? nf)))


(defun cdnf->dnf (cdnf)
  (loop for i in cdnf
        for acc = i then (dnf-and acc i)
        finally (return acc)))



; ------------------------------------------------------------------------------
--
; dcnf: DISJUNCTION OF CNF
; ------------------------------------------------------------------------------
--

(defun dcnf-contradiction? (hnf)
  (loop for nf in hnf
        always (cnf-contradiction? nf)))

(defun dcnf-tautology? (hnf)
  (loop for nf in hnf
        thereis (cnf-tautology? nf)))


(defun dcnf->cnf (dcnf)
  (loop for i in dcnf
        for acc = i then (cnf-or acc i)
        finally (return acc)))



; ------------------------------------------------------------------------------
--
; SYMBOLIC <-> BINARY CONVERSIONS
; ------------------------------------------------------------------------------
--

; symbolic -> binary:
; -------------------

(defun hnf-symbolic->binary (hnf assoc-universe)
  (loop for nf in hnf
        collect (nf-symbolic->binary nf assoc-universe)))


; binary -> symbolic:
; -------------------

(defun hnf-binary->symbolic (hnf assoc-universe)
  (loop for nf in hnf
        collect (nf-binary->symbolic nf assoc-universe)))


; binary -> symbolic (pretty):
; ----------------------------

(defun dcnf-binary->symbolic-pretty (dcnf assoc-universe)
  (loop for cnf in dcnf
        for x = NIL then T
        when x
        do (progn (terpri)
                  (write-string " OR ")
                  (terpri)
                  (princ '\()
                  (cnf-binary->symbolic-pretty cnf assoc-universe)
                  (princ '\)))
        else
        do (progn (terpri)
                  (princ '\()
                  (cnf-binary->symbolic-pretty cnf assoc-universe)
                  (princ '\)))))
```

```
(defun cdnf-binary->symbolic-pretty (cdnf assoc-universe)
  (loop for dnf in cdnf
        for x = NIL then T
        when x
        do (progn (terpri)
                  (write-string " AND ")
                  (terpri)
                  (princ '\()
                  (dnf-binary->symbolic-pretty dnf assoc-universe)
                  (princ '\)))
        else
        do (progn (terpri)
                  (princ '\()
                  (dnf-binary->symbolic-pretty dnf assoc-universe)
                  (princ '\))))))
```

Annexe C9 Language-utilities.lisp

```
; ———————————————————————————————————————————————————————————————————————
——
; ———                              FRIBOURG UNIVERSITY                              —
——
; ———                            INSTITUTE OF INFORMATICS                           —
——
; ———                  Rue Faucigny 2, CH-1700 Fribourg, Switzerland                —
——
; ———————————————————————————————————————————————————————————————————————
——
; TITLE:    language-utilities.lisp
; SUPPORT:  Michael Schumacher
; PROJECT:  DIG
; DATE:     29.05.95
; VERSION:  1.0
; ———————————————————————————————————————————————————————————————————————
——
; OVERVIEW: Constantes et variables globales du module de langage de DIG.
;           Quelques fonctions utilisée pour l'implémentation du langage DIG.
; ———————————————————————————————————————————————————————————————————————
——



; ---------------------------------------------------------------------------
--
; CONSTANTES
; ---------------------------------------------------------------------------
--

(defconstant no-support 'no-support)
(defconstant SECOND-PATH T)
(defconstant NOT-SECOND-PATH NIL)



; ---------------------------------------------------------------------------
--
; VARIABLES GLOBALES
; ---------------------------------------------------------------------------
--

; *list-of-modes-values* est une variable globale utilisée lorsqu'on instancie
; un circuit logique.

(defvar *list-of-modes-values* NIL)


; *temp-assumptions* est une variable globale utilisée lorsqu'on instancie
; un circuit logique. Elle permet d'établir les assumptions.

(defvar *temp-assumptions* NIL)


; *last-inst-name* est une variable globale utilisée pour la renomination
; de variables internes d'un circuit.

(defvar *last-inst-name* NIL)


; *gates* est la liste de tous les circuits instanciés.

(defvar *gates* NIL)


; *DIG-modules* est la list de tous les modules définis.

(defvar *DIG-modules* NIL)


; Indent:
```

```
; -------

(setf *fred-special-indent-alist*
      (cons (cons 'int 1) *fred-special-indent-alist*))


; Utilisé pour les queries:
; -------------------------

; *temp-trees*, *temp-RB-Knowledge*, *second-path*, *temp-CalculateNeeded* sont
; mémorisent le contexte ABL actuel lors d'un query. Dès que le query a été
; réalisé, le context est rendu à nouveau cohérent. Voir queries.lisp.

(defvar *temp-trees* NIL)

(defvar *temp-RB-Knowledge* NIL)

(defvar *second-path* NIL)

(defvar *temp-CalculateNeeded* NIL)



; ------------------------------------------------------------------------------
--
; FUNCTIONS
; ------------------------------------------------------------------------------
--

; Sorting functions:
; ------------------

(defun sort-alphabetically (s)
  (sort s #'string-lessp))

(defun sort-numerically (s)
  (sort s #'<))
```

Annexe C10 `Syntax.lisp`

```
; ——————————————————————————————————————————————————————————————————————————
—
; ——                              FRIBOURG UNIVERSITY                          —
—
; ——                            INSTITUTE OF INFORMATICS                       —
—
; ——                 Rue Faucigny 2, CH-1700 Fribourg, Switzerland             —
—
; ——————————————————————————————————————————————————————————————————————————
—
; TITLE:    syntax.lisp
; SUPPORT:  Michael Schumacher
; PROJECT:  DIG
; DATE:     15.05.95
; VERSION:  1.0
; ——————————————————————————————————————————————————————————————————————————
—
; OVERVIEW: La classe SYNTAX.
; ——————————————————————————————————————————————————————————————————————————
—



; Cette classe est le parent de toutes les classes syntaxiques. Les classes
; syntaxiques permettent d'analyser le contenu syntaxique et sémantique des
; paramètres des différentes macros.

(defclass syntax () ())

; Le programme implémente entre autres cinq macros: defprimitive, defmodule,
; new-var, inst et obs. A chaque macro correspond une classe syntaxique.
; Les macros de DIG possèdent deux parties. Tout d'abord, elles analysent le
; contenu syntaxique et sémantique des paramètres actuels en créant un objet
; syntaxique dont les slots sont constitués de ces paramètres. Pour réaliser cela,
; chaque classe syntaxique possède entre autres deux méthodes: ok? et expression.
; Ok? réalise le test grammatical des paramètres actuels de la macro. Expression
est
; utilisée pour indiquer de quelle expression il s'agit lors de messages d'erreurs.
; La deuxième partie de la macro diffère de cas en cas.


; Les classes syntaxiques peuvent se spécialiser en plusieurs sous-classes.
; Un premier sous-ensemble de classes syntaxiques est celui des définitions.

(defclass definition (syntax)
  ((name :accessor name :initarg :name)
   (body :accessor body :initarg :body)))

; Nous avons ensuite les classes concernant les manipulations de données.

(defclass manipulation (syntax)
  ())

; Enfin, les classes de déclaration.

(defclass declaration (syntax)
  ())
```

Annexe C11 `Gate.lisp`

```
; ——————————————————————————————————————————————————————————————————————————
;——                                FRIBOURG UNIVERSITY                                —
;——                              INSTITUTE OF INFORMATICS                             —
;——                   Rue Faucigny 2, CH-1700 Fribourg, Switzerland                   —
; ——————————————————————————————————————————————————————————————————————————
; TITLE:    gate.lisp
; SUPPORT:  Michael Schumacher
; PROJECT:  DIG
; DATE:     29.05.95
; VERSION:  1.0
; ——————————————————————————————————————————————————————————————————————————
; OVERVIEW: La classe GATE.
; ——————————————————————————————————————————————————————————————————————————


; La classe gate est employée pour instancier des modules ou des primitives au
; moyen de la macro inst. Elle possède les slots suivants:
;    - HNF-LIST: liste des hnf d'un circuit.
;    - UNIVERSE: ensemble de toutes les variables contenues dans un circuit.
;    - ASSOC-UNIVERSE: assoc-universe associé à UNIVERSE.
;    - UNIVERSE-MASK: mask correspondant à UNIVERSE.
;    - MODES-WITH-VALUES: liste des variables représentant les modes
;      d'un circuit avec les valeurs possibles correspondantes.
;    - THEASSUMPTIONS: les assumptions du circuit (c-à-d les modes).
;    - OBS-VAR: les variables ayant été observées.
;    - AB-KNOWLEDGE-LIST: liste d'objets ab-knowledge utilisés pour la création
;      d'une liste de proc-tree.
;    - HINTS-LIST: liste de hints utilisés pour la création d'une liste de
;      proc-tree.
;    - TREES: liste d'objets proc-tree.
;    - RB-KNOWLEDGE: objet de la classe RB-KNOWLEDGE nécessaire pour les
;      différents queries.
;    - DIRTY-LIST: liste des indices des éléments de AB-KNOWLEDGE-LIST qui
;      comportent des variables ayant été observé; cette liste est nécessaire
;      pour remettre à jour la liste des arbres TREES.
; Pour plus de précision, voir l'implémentation de la macro inst.


(defclass gate ()
  ((hnf-list            :accessor hnf-list
                        :initform nil
                        :initarg :hnf-list)
   (universe            :accessor universe
                        :initform nil
                        :initarg :universe)
   (assoc-universe      :accessor assoc-universe
                        :initform nil
                        :initarg :assoc-universe)
   (universe-mask       :accessor universe-mask
                        :initform nil
                        :initarg :universe-mask)
   (modes-with-values   :accessor modes-with-values
                        :initform nil
                        :initarg :modes-with-values)
   (theAssumptions      :accessor theAssumptions
                        :initform nil
                        :initarg :theAssumptions)
   (obs-var             :accessor obs-var
                        :initform nil
                        :initarg :obs-var)
   (AB-knowledge-list   :accessor AB-knowledge-list
                        :initform nil
                        :initarg :AB-knowledge-list)
```

Génération d'impliqués premiers et déduction basée sur suppositions au moyen de clauses disjonctives.

```
   (hints-list          :accessor hints-list
                        :initform nil
                        :initarg :hints-list)
   (trees               :accessor trees
                        :initform nil
                        :initarg :trees)
   (RB-knowledge        :accessor RB-knowledge
                        :initform nil
                        :initarg :RB-knowledge)
   (dirty-list          :accessor dirty-list
                        :initform nil
                        :initarg :dirty-list)))

(defun make-gate (&key
                  hnf-list
                  universe
                  assoc-universe
                  universe-mask
                  modes-with-values
                  theAssumptions
                  obs-var
                  AB-knowledge-list
                  hints-list
                  trees
                  RB-knowledge
                  dirty-list)
  (make-instance
    'gate
    :hnf-list          hnf-list
    :universe          universe
    :assoc-universe    assoc-universe
    :universe-mask     universe-mask
    :modes-with-values modes-with-values
    :theAssumptions    theAssumptions
    :obs-var           obs-var
    :AB-knowledge-list AB-knowledge-list
    :hints-list        hints-list
    :trees             trees
    :RB-knowledge      RB-knowledge
    :dirty-list        dirty-list))
```

Annexe C12 `Primitives.lisp`

```
; ------------------------------------------------------------------------------
--
; ---                          FRIBOURG UNIVERSITY                             -
--
; ---                        INSTITUTE OF INFORMATICS                          -
--
; ---              Rue Faucigny 2, CH-1700 Fribourg, Switzerland               -
--
; ------------------------------------------------------------------------------
--
; TITLE:    primitives.lisp
; SUPPORT:  Michael Schumacher
; PROJECT:  DIG
; DATE:     15.05.95
; VERSION:  1.0
; ------------------------------------------------------------------------------
--
; OVERVIEW: Differentes descriptions de primitives.
; ------------------------------------------------------------------------------
--



; ------------------------------------------------------------------------------
--
; LES PRIMITIVES
; ------------------------------------------------------------------------------
--

; Les primitives à quatre paramètres
; ----------------------------------

; Le tableau suivant présente les différentes primitives à quatre paramètres:

; +---+---+---+---------+----------+--------+---------+---------+-----+-----+
; | a | b | c | a-and-b | a-nand-b | a-or-b | a-nor-b | a-xor-b | c=0 | c=1 |
; +---+---+---+---------+----------------------------------------------------
; | 0 | 0 | 0 |    1    |    0     |   1    |    0    |    1    |  1  |  0  |
; | 0 | 0 | 1 |    0    |    1     |   0    |    1    |    0    |  0  |  1  |
; | 0 | 1 | 0 |    1    |    0     |   0    |    1    |    0    |  1  |  0  |
; | 0 | 1 | 1 |    0    |    1     |   1    |    0    |    1    |  0  |  1  |
; | 1 | 0 | 0 |    1    |    0     |   0    |    1    |    0    |  1  |  0  |
; | 1 | 0 | 1 |    0    |    1     |   1    |    0    |    1    |  0  |  1  |
; | 1 | 1 | 0 |    0    |    1     |   0    |    1    |    1    |  1  |  0  |
; | 1 | 1 | 1 |    1    |    0     |   1    |    0    |    0    |  0  |  1  |
; +---+---+---+---------+----------+--------+---------+---------+-----+-----+

; +---+---+---+-----+---------+-----+---------+-------------+
; | a | b | c | c=a | c=not-a | c=b | c=not-b | tautology-3 |
; +---+---+---+-----+---------+-----+---------+-------------+
; | 0 | 0 | 0 |  1  |    0    |  1  |    0    |      1      |
; | 0 | 0 | 1 |  0  |    1    |  0  |    1    |      1      |
; | 0 | 1 | 0 |  1  |    0    |  0  |    1    |      1      |
; | 0 | 1 | 1 |  0  |    1    |  1  |    0    |      1      |
; | 1 | 0 | 0 |  0  |    1    |  1  |    0    |      1      |
; | 1 | 0 | 1 |  1  |    0    |  0  |    1    |      1      |
; | 1 | 1 | 0 |  0  |    1    |  0  |    1    |      1      |
; | 1 | 1 | 1 |  1  |    0    |  1  |    0    |      1      |
; +---+---+---+-----+---------+-----+---------+-------------+


(defmacro a-and-b (in1 in2 out m)
  `(quote (((,in1 ,out) . ())
           ((,in2 ,out) . ())
           (() . (,in1 ,in2 ,out))
           (() . (,m)))))

(defmacro a-nand-b (in1 in2 out m)
  `(quote (((,out) . (,in1))
           ((,out) . (,in2))
           ((,in1 ,in2) . (,out))
```

```
           (() . (,m)))))

(defmacro a-or-b (in1 in2 out m)
  `(quote (((,in1 ,out) . ())
           ((,in2 ,out) . ())
           (() . (,in1 ,in2 ,out))
           (() . (,m)))))

(defmacro a-nor-b (in1 in2 out m)
  `(quote (((,in2) . (,out))
           ((,in1) . (,out))
           ((,out) . (,in1 ,in2))
           (() . (,m)))))

(defmacro a-xor-b (in1 in2 out m)
  `(quote ((() . (,in1 ,in2 ,out))
           ((,in2 ,out) . (,in1))
           ((,in1 ,out) . (,in2))
           ((,in1 ,in2) . (,out))
           (() . (,m)))))

(defmacro c=0 (in1 in2 out m)
  (declare (ignore in1))
  (declare (ignore in2))
  `(quote ((() . (,out))
           (() . (,m)))))

(defmacro c=1 (in1 in2 out m)
  (declare (ignore in1))
  (declare (ignore in2))
  `(quote (((,out) . ())
           (() . (,m)))))

(defmacro c=a (in1 in2 out m)
  (declare (ignore in2))
  `(quote ((() . (,in1 ,out))
           ((,in1 ,out) . ())
           (() . (,m)))))

(defmacro c=not-a (in1 in2 out m)
  (declare (ignore in2))
  `(quote (((,out) . (,in1))
           ((,in1) . (,out))
           (() . (,m)))))

(defmacro c=b (in1 in2 out m)
  (declare (ignore in1))
  `(quote ((() . (,in2 ,out))
           ((,in2 ,out) . ())
           (() . (,m)))))

(defmacro c=not-b (in1 in2 out m)
  (declare (ignore in1))
  `(quote (((,out) . (,in2))
           ((,in2) . (,out))
           (() . (,m)))))

(defmacro tautology-3 (in1 in2 out m)
  `(quote ((() . (,in1 ,in2 ,out))
           ((,out) . (,in1 ,in2))
           ((,in2) . (,in1 ,out))
           ((,in2 ,out) . (,in1))
           ((,in1) . (,in2 ,out))
           ((,in1 ,out) . (,in2))
           ((,in1 ,in2) . (,out))
           ((,in1 ,in2 ,out) . ())
           (() . (,m)))))


; Les primitives à trois paramètres
; ---------------------------------

; Le tableau suivant présente les différentes primitives à trois paramètres:

; +---+---+-----+---------+-----+-----+-------------+
; | a | b | b=a | b=not-a | b=1 | b=0 | tautology-2 |
; +---+---+-----+---------+-----+-----+-------------+
```

```
; | 0 | 0 |  1  |    0    |  0  |  1  |      1      |
; | 0 | 1 |  0  |    1    |  1  |  0  |      1      |
; | 1 | 0 |  0  |    1    |  0  |  1  |      1      |
; | 1 | 1 |  1  |    0    |  1  |  0  |      1      |
; +---+---+-----+---------+-----+-----+-------------+


(defmacro b=a (in out m)
  `(quote ((() . (,in ,out))
           ((,in ,out) . ())
           (() . (,m)))))

(defmacro b=not-a (in out m)
  `(quote (((,out) . (,in))
           ((,in) . (,out))
           (() . (,m)))))

(defmacro b=1 (in out m)
  (declare (ignore in))
  `(quote (((,out) . ())
           (() . (,m)))))

(defmacro b=0 (in out m)
  (declare (ignore in))
  `(quote ((() . (,out))
           (() . (,m)))))

(defmacro tautology-2 (in out m)
  `(quote ((() . (,in ,out))
           ((,out) . (,in))
           ((,in) . (,out))
           ((,in ,out) . ())
           (() . (,m)))))



; ------------------------------------------------------------------------------
--
; LA TABLE DE HASH LOG-PRIMITIVES
; ------------------------------------------------------------------------------
--

; Une table de hash est définie pour connaître le nombre des paramètres d'une
; primitive; c-à-d soit trois ou quatre.

(defvar log-primitives (make-hash-table))

(defun get-parameter-length (logprim)
  (gethash logprim log-primitives))

(defun formal-param-length (body)
  (let ((logprim (second (first body))))
    (get-parameter-length logprim)))


(setf (gethash 'a-and-b      log-primitives) 4)
(setf (gethash 'a-nand-b     log-primitives) 4)
(setf (gethash 'a-or-b       log-primitives) 4)
(setf (gethash 'a-nor-b      log-primitives) 4)
(setf (gethash 'a-xor-b      log-primitives) 4)
(setf (gethash 'c=0          log-primitives) 4)
(setf (gethash 'c=1          log-primitives) 4)
(setf (gethash 'c=a          log-primitives) 4)
(setf (gethash 'c=not-a      log-primitives) 4)
(setf (gethash 'c=b          log-primitives) 4)
(setf (gethash 'c=not-b      log-primitives) 4)
(setf (gethash 'tautology-3  log-primitives) 4)

(setf (gethash 'b=a          log-primitives) 3)
(setf (gethash 'b=not-a      log-primitives) 3)
(setf (gethash 'b=1          log-primitives) 3)
(setf (gethash 'b=0          log-primitives) 3)
(setf (gethash 'tautology-2  log-primitives) 3)
```

Annexe C13 `Defprimitive.lisp`

```
; ------------------------------------------------------------------------------
--
; ---                            FRIBOURG UNIVERSITY                           -
--
; ---                          INSTITUTE OF INFORMATICS                        -
--
; ---               Rue Faucigny 2, CH-1700 Fribourg, Switzerland              -
--
; ------------------------------------------------------------------------------
--
; TITLE:    defprimitive.lisp
; SUPPORT:  Michael Schumacher
; PROJECT:  DIG
; DATE:     23.05.95
; VERSION:  1.0
; ------------------------------------------------------------------------------
--
; OVERVIEW: La définition de primitives.
; ------------------------------------------------------------------------------
--


; La macro defprimitive permet de définir de nouvelles primitives.

; Exemple:
; (defprimitive my-and ((ok  a-and-b     0.93)
;                       (ab1 tautology-3 0.03)
;                       (ab2 c=0         0.04)))

; Formellement, l'expression est la suivante: (defprimitive <name> <body>)
; La macro crée un objet de la classe primitive-definition au moyen de name
; et de body. Cet objet est analysé par la méthode ok?.



; ------------------------------------------------------------------------------
--
; LA CLASSE PRIMITIVE-DEFINITION
; ------------------------------------------------------------------------------
--

; La classe primitive-definition est la classe syntaxique de la macro defprimitive.
; Elle est hérite de definition (défini dans syntax.lisp).

(defclass primitive-definition (definition)
  ())

(defun make-primitive-definition (name body)
  (make-instance 'primitive-definition :name name :body body))


; expression
; ----------

(defmethod expression ((exp primitive-definition))
  (list 'defprimitive (name exp) (body exp)))


; ok?
; ---

(defmethod ok? ((exp primitive-definition))
  (let ((expr (expression exp)))
    (cond
     ((not (symbolp (name exp)))
      (error "Wrong primitive definition:~a~&~a is not a symbol."
             expr
             (name exp)))
     ((not (listp (body exp)))
      (error "Wrong primitive definition: ~a~&~a is not a list."
             expr
```

```
             (body exp)))
     ((not (>= (length (body exp)) 1))
      (error "Wrong primitive definition ~a~&There is no primitive description."
             expr))
     ((not (= (loop for n in (body exp)
                    sum (caddr n))
              1))
      (error "Wrong primitive definition ~a
The sum of the probabilities is not equal to 1"
             expr))
     ((loop for n in (body exp)
            always
            (cond ((not (listp n))
                   (error "~a is not a list." n))
                  ((not (= (length n) 3))
                   (error "The length of ~a is not three." n))
                  ((not (symbolp (car n)))
                   (error "~a is not a symbol." (car n)))
                  ((not (symbolp (cadr n)))
                   (error "~a is not a symbol." (cadr n)))
                  ((not (numberp (caddr n)))
                   (error "~a is not a number." (caddr n)))
                  ((not (and (>= (caddr n) 0)
                             (<= (caddr n) 1)))
                   (error "~a is not between 0 and 1" (caddr n)))
                  (t nil)))
      (error "Wrong primitive definition:~&~a" expr))
     (t t))))



; ----------------------------------------------------------------------------
--
; LA MACRO DEFPRIMITIVE
; ----------------------------------------------------------------------------
--

; Les paramètres d'une primitive sont implicites: deux ou trois pour les entrées
; et sorties; un dernier pour le mode dans lequel se trouve la composante.

; La macro defprimitive est une définition de fonction. Name devient une fonction
; avec des paramètres formels implicites.

(defun get-mode (descr)
  (first descr))

(defun get-logprim (descr)
  (second descr))

(defun get-bpa (descr)
  (third descr))

(defun create-xor-dnf (vars)
  (when (> (length vars) 1)
    (loop for v in vars
          collect (cons (list v) (remove v vars)))))

(defun modes-values (body)
  (loop for descr in body
        collect (first descr)))


(defmacro defprimitive (name body)
  (let ((exp (make-primitive-definition name body)))
    (when (ok? exp)
      (case (formal-param-length body)

        (4 `(defun ,name (in1 in2 out m &optional inst-name)
              (declare (ignore inst-name))

              (setf *list-of-modes-values*
                    (cons (cons m (modes-values (quote ,body)))
                          *list-of-modes-values*))

              (loop for descr in (quote ,body)
                    for mode = (concatenate-special-3 m
                                                      '\.
```

```
                                              (get-mode descr))
                for logprim = (get-logprim descr)
                for bpa = (get-bpa descr)

                collect (eval (list logprim in1 in2 out mode))
                into primitive-description
                collect mode into modes
                collect (cons mode bpa) into new-assumptions

                finally (progn
                          (setf *temp-assumptions*
                                (append new-assumptions
                                        *temp-assumptions*))
                          (return
                           (let ((xor-dnf (create-xor-dnf modes)))
                             (if xor-dnf
                               (list (cons xor-dnf
                                           primitive-description))
                               (list primitive-description)))))))))

      (3 `(defun ,name (in out m &optional inst-name)
            (declare (ignore inst-name))

            (setf *list-of-modes-values*
                  (cons (cons m (modes-values (quote ,body)))
                        *list-of-modes-values*))

            (loop for descr in (quote ,body)
                  for mode = (concatenate-special-3 m
                                                    '\.
                                                    (get-mode descr))
                  for logprim = (get-logprim descr)
                  for bpa = (get-bpa descr)

                  collect (eval (list logprim in out mode))
                  into primitive-description
                  collect mode into modes
                  collect (cons mode bpa) into new-assumptions

                  finally (progn
                            (setf *temp-assumptions*
                                  (append new-assumptions
                                          *temp-assumptions*))
                            (return
                             (let ((xor-dnf (create-xor-dnf modes)))
                               (if xor-dnf
                                 (list (cons xor-dnf
                                             primitive-description))
                                 (list primitive-description)))))))))))
```

Annexe C14 `Defmodule.lisp`

```
; ————————————————————————————————————————————————————————————————————————————
——
; ——                              FRIBOURG UNIVERSITY                           —
——
; ——                           INSTITUTE OF INFORMATICS                         —
——
; ——                 Rue Faucigny 2, CH-1700 Fribourg, Switzerland              —
——
; ————————————————————————————————————————————————————————————————————————————
——
; TITLE:    defmodule.lisp
; SUPPORT:  Michael Schumacher
; PROJECT:  DIG
; DATE:     15.05.95
; VERSION:  1.0
; ————————————————————————————————————————————————————————————————————————————
——
; OVERVIEW: Definition de modules.
; ————————————————————————————————————————————————————————————————————————————
——



; La macro defmodule permet de définir des modules à partir de primitives
; ou d'autres macros.

; Exemples:

; (defmodule halfadder (in1 in2 sum carry)
;   (int ((wires e d)
;         (modes m_or1 m_and1 m_not1 m_and2))
;     (inst or1 (or@ in1 in2 e m_or1))
;     (inst and1 (and@ in1 in2 carry m_and1))
;     (inst not1 (not@ carry in2 m_not1))
;     (inst and2 (and@ e d sum m_and2))))

; (defmodule adder (in1 in2 c-in sum c-out)
;   (int ((wires s c1 c2)
;         (modes m_or1))
;     (inst halfadder_1 (halfadder in2 c-in s c1))
;     (inst halfadder_2 (halfadder in1 s sum c2))
;     (inst or1 (or@ c1 c2 c-out m_or1))))
;
; Formellement: (defmodule <name> <lambda-list> <body>)
;
; La classe syntaxique est implémentée de manière analogue à la classe
; primitive-definition.



; ----------------------------------------------------------------------------
--
; LA CLASSE MODULE-DEFINITION
; ----------------------------------------------------------------------------
--

(defclass module-definition (definition)
  ((lambda-list :accessor lambda-list :initarg :lambda-list)))

(defun make-module-definition (name lambda-list body)
  (make-instance 'module-definition
    :name name
    :lambda-list lambda-list
    :body body))


; expression
; ----------

(defmethod expression ((exp module-definition))
  (list 'defmodule (name exp) (lambda-list exp) (body exp)))
```

```
; selectors
; ---------

(defmethod intern-variables ((exp module-definition))
  (cadr (body exp)))

(defmethod intern-wires ((exp module-definition))
  (cdar (intern-variables exp)))

(defmethod intern-modes ((exp module-definition))
  (cdadr (intern-variables exp)))

(defmethod only-intern-variables ((exp module-definition))
  (append (intern-wires exp)
          (intern-modes exp)))

(defmethod core ((exp module-definition))
  (cddr (body exp)))


; selectors of one core element
; -----------------------------

(defun inst-expr (core-elt)
  (caddr core-elt))

(defun inst-name (core-elt)
  (cadr core-elt))


; ok?
; ---

(defun test-non-atom (exp key)
  (labels ((key-word (exp)
             (car exp)))
    (if (atom exp)
      nil
      (eq (key-word exp) key))))

(defun instance? (exp)
  (test-non-atom exp 'inst))

(defun module-definition? (exp)
  (test-non-atom exp 'defmodule))

(defun primitive-definition? (exp)
  (test-non-atom exp 'defprimitive))

(defmethod ok? ((exp module-definition))
  (let ((expr (expression exp)))
    (cond
     ((not (symbolp (name exp)))
      (error "Wrong module definition ~a~&~a is not a symbol."
             expr
             (name exp)))
     ((not (or (eq (car (body exp)) 'int)
               (eq (caadr (intern-variables exp)) 'wires)
               (eq (cadadr (intern-variables exp)) 'modes)))
      (error "Wrong module definition ~a~&You have forgotten one or more keywords."
             expr))
     ((not
       (if (listp (core exp))
         (loop for n in (core exp)
               always (cond
                       ((not (listp n))
                        (error "~a is not a list." n))
                       ((not (or (instance? n)
                                 (module-definition? n)
                                 (primitive-definition? n)))
                        (error "Wrong module definition.
~a must be an instance, a module definition or a primitive definition." n))
                       (t t)))
         (error "Wrong module definition.~&~a is not a list." (core exp)))))
     (t t))))
```

Génération d'impliqués premiers et déduction basée sur suppositions au moyen de clauses disjonctives.

```
; ------------------------------------------------------------------------------
--
; LA MACRO DEFMODULE
; ------------------------------------------------------------------------------
--

; La macro defmodule est une définition de fonction. Le corps de cette
; fonction comprend la création d'un nouvel environnement au moyen la forme
; spéciale let*. L'expression est construite par un loop en utilisant des
; backquotes.

(defmacro defmodule (name lambda-list body)
  (let ((exp (make-module-definition name lambda-list body)))
    (when (ok? exp)
      (progn
        (when (not (find (name exp) *DIG-modules*))
          (setf *DIG-modules* (sort-alphabetically (cons (name exp) *DIG-
modules*)))
          name)
        `(defun ,name
                ,(append lambda-list '(&optional *last-inst-name*))
           (let* ,(append (loop for r in (only-intern-variables exp)
                                collect  `(,r (concatenate-special-3
                                               *last-inst-name*
                                               '\.
                                               (quote ,r))))
                          (loop for n in (core exp)
                                collect (list (inst-name n)
                                              (append
                                               (inst-expr n)
                                               `((concatenate-special-3
                                                  *last-inst-name*
                                                  '\.
                                                  ',(inst-name n)))))))
             (loop for descr in ,(cons 'list (loop for n in (core exp)
                                                   collect (second n)))
                   append descr)))))))
```

Annexe C15 `Inst.lisp`

```
; ——————————————————————————————————————————————————————————————————————————
——
; ———                              FRIBOURG UNIVERSITY                          —
——
; ———                            INSTITUTE OF INFORMATICS                       —
——
; ———                 Rue Faucigny 2, CH-1700 Fribourg, Switzerland             —
——
; ——————————————————————————————————————————————————————————————————————————
——
; TITLE:    inst.lisp
; SUPPORT:  Michael Schumacher
; PROJECT:  DIG
; DATE:     15.05.95
; VERSION:  1.0
; ——————————————————————————————————————————————————————————————————————————
——
; OVERVIEW: Instancier un module ou une primitive.
; ——————————————————————————————————————————————————————————————————————————
——



; Inst attribue à une variable la création d'un ensemple de Prime Implicates.
; Exemple:
; (inst my-adder (adder new-var in1 in2 carry-in the-sum carry-out))
; Formellement: (inst <name> <body>)



; ------------------------------------------------------------------------------
--
; LA CLASSE INSTANCES
; ------------------------------------------------------------------------------
--

(defclass instances (manipulation)
  ((name :accessor name :initarg :name)
   (body :accessor body :initarg :body)))

(defun make-instances (name body)
  (make-instance 'instances :name name :body body))


; expression
; ----------

(defmethod expression ((exp instances))
  (list 'inst (name exp) (body exp)))


; selectors
; ---------

(defmethod module-name ((exp instances))
  (car (body exp)))

(defmethod actual-parameters ((exp instances))
  (cdr (body exp)))


; ok?
; ---

(defun list-of-symbol? (my-list)
  (loop for elt in my-list
        always (symbolp elt)))

(defmethod ok? ((exp instances))
  (let ((expr (expression exp)))
    (cond
```

```
     ((not (symbolp (name exp)))
      (error "Wrong instance: ~a~&~a is not a symbol."
             expr
             (name exp)))
     ((not (listp (body exp)))
      (error "Wrong instance: ~a~&~a is not a list."
             expr
             (body exp)))
     ((not (symbolp (module-name exp)))
      (error "Wrong instance: ~a~&~a is not a symbol."
             expr
             (module-name exp)))
     ((not (list-of-symbol? (actual-parameters exp)))
      (error "Wrong instance: ~a~&~a is not a list of symbols."
             expr
             (actual-parameters exp)))
     (t t))))


; ------------------------------------------------------------------------------
--
; LA MACRO INST
; ------------------------------------------------------------------------------
--

; Inst évalue body qui est un appel de primitive ou de module.

(defmacro inst (name body)
  (let ((exp (make-instances name body)))
    (when (ok? exp)
      (setf *list-of-modes-values* nil)
      (setf *temp-assumptions* nil)
      (let* ((hnf-list (eval body))
             (universe (get-universe hnf-list))
             (assoc-universe (make-assoc-universe universe)))
        (set name
             (make-gate :hnf-list         hnf-list
                        :universe         universe
                        :assoc-universe   assoc-universe
                        :universe-mask    NIL
                        :modes-with-values *list-of-modes-values*
                        :theAssumptions   *temp-assumptions*
                        :obs-var    NIL
                        :AB-knowledge-list NIL
                        :hints-list       NIL
                        :trees            NIL
                        :RB-knowledge     NIL
                        :dirty-list       NIL))
        (when (not (find (name exp) *gates*))
          (setf *gates* (sort-alphabetically (cons (name exp) *gates*))))
        name))))
```

Annexe C16 `Obs.lisp`

```
; ————————————————————————————————————————————————————————————————————————————
——
; ——                              FRIBOURG UNIVERSITY                           —
——
; ——                            INSTITUTE OF INFORMATICS                        —
——
; ——                 Rue Faucigny 2, CH-1700 Fribourg, Switzerland              —
——
; ————————————————————————————————————————————————————————————————————————————
——
; TITLE:    obs.lisp
; SUPPORT:  Michael Schumacher
; PROJECT:  DIG
; DATE:     22.05.95
; VERSION:  1.0
; ————————————————————————————————————————————————————————————————————————————
——
; OVERVIEW: Utilité d'observation de variable pour un objet gate.
; ————————————————————————————————————————————————————————————————————————————
——



; Obs attribue à la variable d'un circuit logique préexistant (objet de la classe
; gate) une valeur donnée.
; Exemple:
; (obs my-adder halfadder_1.m4 ok)
; Formellement: (obs <object> <variable> <new-value>)



; ----------------------------------------------------------------------------
--
; LA CLASSE OBSERVATION
; ----------------------------------------------------------------------------
--

(defclass observation (manipulation)
  ((object :accessor object :initarg :object)
   (variable :accessor variable :initarg :variable)
   (new-value :accessor new-value :initarg :new-value)))

(defun make-observation (object variable new-val)
  (make-instance 'observation
    :object object
    :variable variable
    :new-value new-val))


; expression
; ----------

(defmethod expression ((exp observation))
  (list 'obs (object exp) (variable exp) (new-value exp)))


; ok?
; ---

(defmethod ok? ((exp observation))
  (let ((expr (expression exp)))
    (cond
     ((not (symbolp (object exp)))
      (error "Wrong observation expression ~a~&~a is not a symbol."
             expr
             (object exp)))
     ((not (symbolp (variable exp)))
      (error "Wrong observation expression ~a~&~a is not a symbol."
             expr
             (variable exp)))
     ((not (atom (new-value exp)))
```

```
      (error "Wrong observation expression ~a~&~a must be an atomic expression."
             expr
             (new-value exp)))
     (t t))))


; ----------------------------------------------------------------------------
--
; LA MACRO OBS
; ----------------------------------------------------------------------------
--

; symbol-present? vérifie si une variable se trouve dans un hnf qui
; n'est pas encore numérique.

(defun symbol-present? (var hnf)
  (find var (flatten-list hnf) :test #'sp-equal?))


; Good-value-for-variable? est une méthode qui controle si la variable fait partie
; ou non de the-gate et si la valeur que l'utilisateur désire attribuer à cette
; variable est acceptable ou non.

(defmethod good-value-for-variable? (value variable (the-gate gate))
  (let ((mode-with-values (my-assq  variable
                                    (modes-with-values the-gate))))
    (or (and (null mode-with-values)
             (or (eq value 1)
                 (eq value 0)))
        (included? value mode-with-values))))


; new-symbol

(defun new-symbol (symb)
  (concatenate-symbols symb '\. (gensym "@")))


; Create-new-mode-observation-nf crée une forme normal pour un mode avec
; une valeur donnée.

(defun create-new-mode-observation-nf (ext-mode)
  (list (cons (list ext-mode) '())))


; Create-new-var-observation-nf crée une forme normal pour une variable
; avec une valeur donnée.

(defun create-new-var-observation-nf (var val)
  (case val
    (1 (list (cons (list var) '())))
    (0 (list (cons '() (list var) )))))


; Translate-in-good-symbol réalise une légère transformation de la variable
indiquée
; par l'utilisateur.

(defun translate-in-good-symbol-1 (the-symbol the-list)
  (loop for s in the-list
        if (equal? s the-symbol)
        return s))

(defun translate-in-good-symbol-2 (the-symbol the-list)
  (loop for s in the-list
        if (sp-equal? (first s) the-symbol)
        return (first s)))


; Introduce crée une forme normale correspondant à l'observation faite par
; l'utilisateur et l'introduit dans le circuit objet.

(defun introduce (object variable value)
  (let ((the-gate (eval object)))
    (when (not (null the-gate))
      (let* ((hnf-list (hnf-list the-gate))
```

```
      (the-variable (translate-in-good-symbol-1 variable
                                                (universe the-gate)))
      (mode (translate-in-good-symbol-2 variable
                                        (modes-with-values the-gate)))
      (ext-mode (when mode
                  (translate-in-good-symbol-1
                   (concatenate-special-3 variable '\. value)
                   (universe the-gate)))))
; Nous sommes en présence de deux cas:
; a) mode = T, alors il s'agit d'un mode
; ext-mode est le symbole du mode uni a sa valeur
; exemple: si mode = "m" et sa valeur = "ok", alors ext-mode = "m.ok"
; b) the-variable = T, alors il s'agit d'un "wire"
(cond
 (mode
  (if (not (good-value-for-variable? value mode the-gate))
    (error "The mode ~a cannot take the value ~a"
           mode
           value)
    (let*
      ((new-nf (create-new-mode-observation-nf ext-mode))
       (new-hnf-list
        (loop for hnf in hnf-list
              for index from 1
              when (symbol-present? ext-mode hnf)
              collect (progn
                        (setf (dirty-list the-gate)
                              (sort-numerically
                               (remove-duplicates
                                (append (dirty-list the-gate)
                                        (list index)))))
                        (hnf-build new-nf hnf))
              else
              collect hnf)))
      (setf (hnf-list the-gate) new-hnf-list)
      the-gate)))

 (the-variable
  ; Il faut renommer les variables. On utilise un compteur.
  (if (not (good-value-for-variable? value the-variable the-gate))
    (error "The variable ~a cannot take the value ~a"
           mode
           value)
    (let*
      ((new-hnf-list
        (loop for hnf in hnf-list
              for index from 1
              collect
              (if (symbol-present? the-variable hnf)
                (let ((new-nf (create-new-var-observation-nf
                               the-variable
                               value))
                      (new-var (new-symbol the-variable))
                      (already-obs-vars (obs-var the-gate)))
                  (setf (dirty-list the-gate)
                        (sort-numerically
                         (remove-duplicates (append (dirty-list the-gate)
                                                    (list index)))))
                  (setf (Universe the-gate)
                        (append (Universe the-gate) (list new-var)))
                  (setf (assoc-universe the-gate)
                        (make-assoc-universe (Universe the-gate)))
                  (setf (obs-var the-gate)
                        (if already-obs-vars
                          (let ((yet-in-obs-vars NIL))
                            (loop for s in already-obs-vars
                                  collect
                                  (if (sp-equal? the-variable (first s))
                                    (progn (setf yet-in-obs-vars T)
                                           (append s (list new-var)))
                                    s)
                                  into temp-obs-vars
                                  finally
                                  (if yet-in-obs-vars
                                    (return temp-obs-vars)
                                    (return
                                     (append temp-obs-vars
```

```
                                                    (list (list the-variable
                                                                new-var)))))))
                                    (list (list the-variable new-var))))

                              (nsubst new-var
                                      the-variable
                                      (hnf-build new-nf hnf)))
                            hnf))))
                (setf (hnf-list the-gate) new-hnf-list)
                the-gate)))

          (T (progn
                (format t
                        "The variable ~a doesn't exist in the following gate: ~a."
                        variable
                        object)
                (error "Please retry with another variable.")))))))


; obs
; ---

; La macro obs introduit la nouvelle observation au moyen de la fonction
; introduce.

(defmacro obs (object variable value)
  (let ((exp (make-observation object variable value)))
    (when (ok? exp)
      (let ((new-object (introduce object variable value)))
        (set object new-object)))))
```

Annexe C17 New-var.lisp

```
; ————————————————————————————————————————————————————————————————————————
——
; ——                              FRIBOURG UNIVERSITY                          —
——
; ——                            INSTITUTE OF INFORMATICS                       —
——
; ——                Rue Faucigny 2, CH-1700 Fribourg, Switzerland              —
——
; ————————————————————————————————————————————————————————————————————————
——
; TITLE:    new-var.lisp
; SUPPORT:  Michael Schumacher
; PROJECT:  DIG
; DATE:     15.05.95
; VERSION:  1.0
; ————————————————————————————————————————————————————————————————————————
——
; OVERVIEW: Definition de variables globales.
; ————————————————————————————————————————————————————————————————————————
——



; Il est possible de définir des variables globales au moyen de la macro new-var.
; La variable est liée à son symbole lui-même.
; Exemple:
; (new-var in1 in2 carry-in the-sum carry-out)
; Formellement: (new-var <variables>)



; ----------------------------------------------------------------------------
--
; LA CLASSE NEW-VARIABLES
; ----------------------------------------------------------------------------
--

; New-variables est la classe syntaxique de la macro new-var.

(defclass new-variables (declaration)
  ((variables :accessor variables :initarg :variables)))

(defun make-new-variables (variables)
  (make-instance 'new-variables :variables variables))


; ok?
; ---

(defmethod ok? ((exp new-variables))
  (loop for n in (variables exp)
        until (not (symbolp n))
        finally (if (symbolp n)
                    (return t)
                    (error "~a is not a symbol!
Please, redefine the other variables." n))))



; ----------------------------------------------------------------------------
--
; LA MACRO NEW-VAR
; ----------------------------------------------------------------------------
--

; La macro new-var utilise la fonction declare-variables pour définir une liste de
; variables.

(defun declare-variables (the-list)
  (eval (cons 'setf (loop for n in the-list
                          append (list n  (list 'quote n))))))
```

Génération d'impliqués premiers et déduction basée sur suppositions au moyen de clauses disjonctives.

```
(defmacro new-var (&rest variables)
  (let ((exp (make-new-variables variables)))
    (when (ok? exp)
      (declare-variables variables))))
```

Annexe C18 `Delete-module.lisp`

```
; ————————————————————————————————————————————————————————————————————————————
;——                                FRIBOURG UNIVERSITY                          —
;——                             INSTITUTE OF INFORMATICS                        —
;——                  Rue Faucigny 2, CH-1700 Fribourg, Switzerland              —
; ————————————————————————————————————————————————————————————————————————————
; TITLE:    delete-module.lisp
; SUPPORT:  Michael Schumacher
; PROJECT:  DIG
; DATE:     3.07.95
; VERSION:  1.0
; ————————————————————————————————————————————————————————————————————————————
; OVERVIEW: L'utilité delete-modules.
; ————————————————————————————————————————————————————————————————————————————



; Delete-module permet d'éliminer les modules.

; Exemple:
; (defmodule halfadder (in1 in2 sum carry)
;   (int ((wires e d)
;         (modes m_or1 m_and1 m_not1 m_and2))
;     (inst or1 (or@ in1 in2 e m_or1))
;     (inst and1 (and@ in1 in2 carry m_and1))
;     (inst not1 (not@ carry in2 m_not1))
;     (inst and2 (and@ e d sum m_and2))))
; (delete-module halfadder)

(defmacro delete-module (m)
  (setf *DIG-modules* (remove m *DIG-modules*))
  `(fmakunbound (quote ,m)))
```

Annexe C19 Delete-gate.lisp

```
; ------------------------------------------------------------------------------
; ---                        FRIBOURG UNIVERSITY                                -
; ---                    INSTITUTE OF INFORMATICS                               -
; ---           Rue Faucigny 2, CH-1700 Fribourg, Switzerland                   -
; ------------------------------------------------------------------------------
; TITLE:    delete-gate.lisp
; SUPPORT:  Michael Schumacher
; PROJECT:  DIG
; DATE:     3.07.95
; VERSION:  1.0
; ------------------------------------------------------------------------------
; OVERVIEW: L'utilité delete-gate.
; ------------------------------------------------------------------------------


; Delete-gate permet d'éliminer des circuits instanciés.

; Exemple:
; (inst my-adder (adder new-var in1 in2 carry-in the-sum carry-out))
; (delete-gate my-adder)

(defmacro delete-gate (g)
  (setf *gates* (remove g *gates*))
  `(makunbound (quote ,g)))
```

Annexe C20 Commun-primitives.lisp

```
; ————————————————————————————————————————————————————————————————————————————
—
; ——                              FRIBOURG UNIVERSITY                           —
—
; ——                           INSTITUTE OF INFORMATICS                         —
—
; ——                 Rue Faucigny 2, CH-1700 Fribourg, Switzerland              —
—
; ————————————————————————————————————————————————————————————————————————————
—
; TITLE:    commun-primitives.lisp
; SUPPORT:  Michael Schumacher
; PROJECT:  DIG
; DATE:     03.07.95
; VERSION:  1.0
; ————————————————————————————————————————————————————————————————————————————
—
; OVERVIEW: Definition des primitives classiques.
; ————————————————————————————————————————————————————————————————————————————
—



; ----------------------------------------------------------------------------
--
; LES PRIMITIVES CLASSIQUES
; ----------------------------------------------------------------------------
--

; Les primitives classiques sont définies avec un seul mode de comportement:
; le mode ok.

(defun set-to-standard ()
  (defprimitive and@  ((ok a-and-b  1.0)))
  (defprimitive nand@ ((ok a-nand-b 1.0)))
  (defprimitive or@   ((ok a-or-b   1.0)))
  (defprimitive nor@  ((ok a-nor-b  1.0)))
  (defprimitive xor@  ((ok a-xor-b  1.0)))
  (defprimitive not@  ((ok b=not-a  1.0))))

(set-to-standard)
```

Annexe C21 ABL_extension.lisp

```
; ------------------------------------------------------------------------------
--
; ---                            FRIBOURG UNIVERSITY                           -
--
; ---                          INSTITUTE OF INFORMATICS                        -
--
; ---                Rue Faucigny 2, CH-1700 Fribourg, Switzerland             -
--
; ------------------------------------------------------------------------------
--
; TITLE:    ABL_extension.lisp
; SUPPORT:  Michael Schumacher
; PROJECT:  DIG
; DATE:     29.05.95
; VERSION:  1.0
; ------------------------------------------------------------------------------
--
; OVERVIEW: Quelques transformation d'ABL.
; ------------------------------------------------------------------------------
--



; Les fonctions suivantes sont reprises pratiquement intégralement de l'implémen-
; tation d'ABL de Norbert Lehmann. Elles ont été légèrement transformées pour
; nos besoins, sans pour autant porter atteinte à l'implémentation d'ABL.


; ------------------------------------------------------------------------------
--
; IN LANGUAGE_DML.LISP
; ------------------------------------------------------------------------------
--

(defmacro cluster (&whole TheText &rest TheSymbols)
  (extend-cluster *RB-Knowledge* TheSymbols)
  (CalculateNeeded)
  ''done)


; ------------------------------------------------------------------------------
--
; IN LANGUAGE_QL.LISP
; ------------------------------------------------------------------------------
--

; Introduction d'error (élimination de abort)

(defun SymbolicQuery_Handler (TheArg &key proc)
  (let ((EndResult (or (catch 'Invalid_hypothesis (funcall proc TheArg))
                       (progn (Handle_Clustering TheArg)
                              (catch 'Invalid_hypothesis (funcall proc TheArg))))))
    (if EndResult
      EndResult
        (error "Invalid hypothesis"))))



; Introduction d'error (élimination de abort)

(defun NumericQuery_Handler (TheArg &optional AssProbability &key proc)
  (let ((TheResult (or (catch 'Invalid_hypothesis (funcall proc TheArg
AssProbability))
                       (progn
                         (Handle_Clustering TheArg)
                         (catch 'Invalid_hypothesis (funcall proc TheArg
AssProbability))))))
    (if TheResult
      TheResult
      (progn
        (error "Invalid hypothesis")))))
```

Génération d'impliqués premiers et déduction basée sur suppositions au moyen de clauses disjonctives.

```
; Introduction d'error (élimination de abort)

(defun Handle_Clustering (TheArg)
  (let* ((TheSymbols (remove-duplicates
                      (set-difference (Tree->List (list TheArg)) '(and or not not*
or* and*))))
         (FalseSymbols (set-difference TheSymbols (universe *RB-Knowledge*))))
    (cond ((null FalseSymbols) (eval `(cluster ,@TheSymbols)))
          (T (error "Unknown Symbols ~S" FalseSymbols)))))
```

Annexe C22 Dig-Calculate.lisp

```
; ____________________________________________________________________________
__
; ___                          FRIBOURG UNIVERSITY                           _
__
; ___                        INSTITUTE OF INFORMATICS                        _
__
; ___              Rue Faucigny 2, CH-1700 Fribourg, Switzerland             _
__
; ____________________________________________________________________________
__
; TITLE:    dig-Calculate.lisp
; SUPPORT:  Michael Schumacher
; PROJECT:  DIG
; DATE:     02.06.95
; VERSION:  1.0
; ____________________________________________________________________________
__
; OVERVIEW: Changements de contextes d'ABL à DIG et mise en état cohérent des
;           objets gate.
; ____________________________________________________________________________
__



; En plus de la remise en état des objet gate, ce fichier traite des changements
; de contexte nécessaires aux différents queries.
; L'implémentation est similaire en certains points à Calculate and
Handle_Calculate
; d'ABL.



;----------------------------------------------------------------------------
--
; FONCTIONS DIVERSES.
;----------------------------------------------------------------------------
--

; LS-NOT-SPECIAL-TAUT? est prédicat qui indique si un bs est une tautologie.
; exemple: (1024 . 1024)

(defun ls-not-special-taut? (ls)
  (not (and (= (bs-member (+L ls)) 1)
            (= (bs-member (-L ls)) 1)
            (bs-equal? (logand (+L ls) (-L ls))
                       (+L ls))
            (not (bs-equal? (+L ls) -1)))))


; NF-FILTER-TAUT permet d'éliminer les tautologies dans une forme normale.

(defun nf-filter-taut (nf)
  (filter nf #'ls-not-special-taut?))



;----------------------------------------------------------------------------
--
; PREDICATS
;----------------------------------------------------------------------------
--

; Prédicats testants les différents états dans lequel peut se trouver un objet
; de la classe gate.

(defmethod not-extended? ((gate gate))
  (null (AB-knowledge-list gate)))

(defmethod extended? ((gate gate))
  (not (not-extended? gate)))
```

```
(defmethod dirty? ((gate gate))
  (not (null (dirty-list gate))))



;-----------------------------------------------------------------------------
--
; UPDATE-NOT-EXTENDED
;-----------------------------------------------------------------------------
--

(defmethod update-universe-mask ((gate gate))
  (let ((res (loop for elt in (assoc-universe gate)
                   for acc = (cdr elt) then (bs-union acc (cdr elt))
                   finally (return acc))))
    (setf (universe-mask gate) res)))

(defun get-modes (modes-with-values)
  (loop for elt in modes-with-values
        collect (car elt)))

(defun symbols->bs (symbols assoc-universe)
  (loop for s in symbols
        for acc = (proposition->bs s assoc-universe)
        then (bs-union acc (proposition->bs s assoc-universe))
        finally (return acc)))

(defmethod produce-rb-knowledge ((gate gate))
  (let* ((assoc-universe (assoc-universe gate))
         (ass-mask (symbols->bs (get-modes (theAssumptions gate))
                                assoc-universe))
         (prop-mask (bs-difference (universe-mask gate) ass-mask))
         (rb-knowledge (make-instance 'rb-knowledge
                         :prop-mask      prop-mask
                         :ass-mask       ass-mask
                         :universe       (universe gate)
                         :assoc-universe assoc-universe)))
    (setf (TL_AssProbList rb-knowledge) (theAssumptions  gate))
    (setf (rb-knowledge gate) rb-knowledge)))

(defmethod produce-ab-knowledge (hnf-symb (gate gate))
  (let* ((assoc-universe (assoc-universe gate))
         (pImp (nf-filter-taut
                (hnf-gen-PI
                 (hnf-symbolic->binary hnf-symb (assoc-universe gate)))))
         (ls-schema (accumulate pImp #'ls-union '(0 . 0)))
         (ls-symb-schema (ls-binary->symbolic ls-schema assoc-universe))
         (schema (union (car ls-symb-schema)
                        (cdr ls-symb-schema)
                        :test #'sp-equal?))
         (schema-mask (ls->bs ls-schema)))
    (make-ab-knowledge :class          'ab-knowledge
                       :universe       (universe gate)
                       :assoc-universe assoc-universe
                       :schema         schema
                       :schema-mask    schema-mask
                       :prop-mask      (prop-mask (RB-knowledge gate))
                       :ass-mask       (ass-mask  (RB-knowledge gate))
                       :binary-nf      pImp)))

(defmethod produce-ab-knowledge-list ((gate gate))
  (setf (AB-knowledge-list gate)
        (loop for hnf in (hnf-list gate)
              collect (produce-ab-knowledge hnf gate))))

(defmethod update-hints ((gate gate))
  (setf (hints-list gate)
        (loop for abk in (ab-knowledge-list gate)
              collect (ab-knowledge->symbolic-hint abk))))


(defmethod update-not-extended ((gate gate))
  (update-universe-mask gate)
  (produce-rb-knowledge gate)
  (produce-ab-knowledge-list gate)
  (update-hints gate)
  (setf (dirty-list gate) nil))
```

```
;-------------------------------------------------------------------------------
--
; UPDATE-EXTENDED
;-------------------------------------------------------------------------------
--

(defmethod update-rb-knowledge ((gate gate))
  (let ((rbk (rb-knowledge gate)))
    (setf (universe rbk) (universe gate))
    (setf (assoc-universe rbk) (assoc-universe gate))
    (setf (prop-mask rbk) (bs-difference (universe-mask gate) (ass-mask rbk)))
    (setf (rb-knowledge gate) rbk)
    gate))

(defun update-binary-nf (b old-assoc-universe new-assoc-universe)
  (nf-symbolic->binary (nf-binary->symbolic b old-assoc-universe)
                       new-assoc-universe))

(defmethod update-ab-knowledge-list ((gate gate) dirty-list)
  (let ((old-assoc-universe (assoc-universe (first (AB-knowledge-list gate)))))
    (loop for abk in (AB-knowledge-list gate)
          for hnf-symb in (hnf-list gate)
          for counter = 1
          do (progn
               (setf (universe abk) (universe gate))
               (setf (assoc-universe abk) (assoc-universe gate))
               (setf (ass-mask abk) (ass-mask  (RB-knowledge gate)))
               (setf (prop-mask abk) (prop-mask (RB-knowledge gate))))

          when (member counter dirty-list)
          do (setf (binary-nf abk)
                   (nf-filter-taut
                    (hnf-gen-PI
                     (hnf-symbolic->binary hnf-symb (assoc-universe gate)))))
          else
          do (update-binary-nf (binary-nf abk)
                               old-assoc-universe
                               (assoc-universe abk))

          do (let* ((ls-schema (accumulate (binary-nf abk) #'ls-union '(0 . 0)))
                    (ls-symb-schema (ls-binary->symbolic ls-schema
                                                         (assoc-universe abk)))
                    (schema (union (car ls-symb-schema)
                                   (cdr ls-symb-schema)
                                   :test #'sp-equal?))
                    (schema-mask (ls->bs ls-schema)))
               (setf (schema abk) schema)
               (setf (schema-mask abk) schema-mask)))
    (setf (dirty-list gate) nil)
    gate))


(defmethod update-extended ((gate gate))
  (update-universe-mask gate)
  (update-rb-knowledge gate)
  (update-ab-knowledge-list gate (dirty-list gate))
  (update-hints gate))



;-------------------------------------------------------------------------------
--
; CHANGEMENT DE CONTEXTE
;-------------------------------------------------------------------------------
--

; Pour pouvoir utiliser ABL pour les différents queries, il faut mettre à jour
; les différentes variables globales d'ABL.

(defmethod set-ABL-environment ((gate gate))
  (setf *temp-RB-Knowledge* *RB-Knowledge*)
  (setf *RB-Knowledge* (RB-Knowledge gate))
  (setf *temp-trees* (if (boundp '*trees*)
                         *trees*
```

```
                      (make-instance 'proc-tree)))
  (setf *trees* (trees gate)))

(defmethod reset-gate ((gate gate))
  (setf (RB-Knowledge gate) *RB-Knowledge*)
  (setf (trees gate) *trees*))

(defun reset-ABL-environment ()
  (setf *RB-Knowledge* *temp-RB-Knowledge*)
  (setf *trees* *temp-trees*)
  (setf *CalculateNeeded* *temp-CalculateNeeded*))



;-----------------------------------------------------------------------------
--
; DIG-HANDLE_CALCULATE
;-----------------------------------------------------------------------------
--

; Dig-calculate construit un arbre à partir d'une liste de hints mis à jour et
; le propage ensuite.

(defmethod dig-calculate ((gate gate) &optional queries)
  (setf *trees* (symbolic-hints->trees (hints-list gate) queries))
  (propagate *trees*)
  *trees*)


; Dig-Handle_Calculate remet si nécessaire dans un état cohérent un objet du type
; gate qui n'a pas encore été propagé (not-extended), qui a reçue de nouvelles
; informations par des observations (dirty), ou qui a été transformé par un second
; passage pour un query (*second-path*). Si l'état est cohérent, il faut juste
; réaliser un changement de contexte.

(defmethod dig-Handle_Calculate ((TheGate gate))
  (cond (*second-path*
         (setf *temp-CalculateNeeded* *CalculateNeeded*)
         (CalculateNeeded)
         (dig-calculate TheGate
                        (MakeClusterQuery (TL_ClusterList *RB-Knowledge*)))
         (CalculateNotNeeded)
         (reset-gate TheGate))

        ((not-extended? TheGate)
         (setf *temp-CalculateNeeded* *CalculateNeeded*)
         (update-not-extended TheGate)
         (set-ABL-environment TheGate)
         (CalculateNeeded)
         (dig-calculate TheGate
                        (MakeClusterQuery (TL_ClusterList *RB-Knowledge*)))
         (CalculateNotNeeded)
         (reset-gate TheGate))

        ((dirty? TheGate)
         (setf *temp-CalculateNeeded* *CalculateNeeded*)
         (update-extended TheGate)
         (set-ABL-environment TheGate)
         (CalculateNeeded)
         (dig-calculate TheGate
                        (MakeClusterQuery (TL_ClusterList *RB-Knowledge*)))
         (CalculateNotNeeded)
         (reset-gate TheGate))

        (T
         (set-ABL-environment TheGate))))
```

Annexe C23 `Queries.lisp`

```
; ——————————————————————————————————————————————————————————————————————
——
; ———                              FRIBOURG UNIVERSITY                              —
——
; ———                            INSTITUTE OF INFORMATICS                           —
——
; ———                 Rue Faucigny 2, CH-1700 Fribourg, Switzerland                 —
——
; ——————————————————————————————————————————————————————————————————————
——
; TITLE:    queries.lisp
; SUPPORT:  Michael Schumacher
; PROJECT:  DIG
; DATE:     25.06.95
; VERSION:  1.0
; ——————————————————————————————————————————————————————————————————————
——
; OVERVIEW: DIG querry langage.
; ——————————————————————————————————————————————————————————————————————
——



; Ce fichier implémente le Query Language. On peut ainsi tester le support,
; quasi-support, plausibility et doubt de certaines hypothèses.
; L'implémentation est similaire à celle d'ABL, c'est pourquoi on emploie
; directement Language_QL.lisp de Norbert Lehmann. Les contenus de certaines
; fonctions définies par Norbert Lehmann pour ABL sont copiés partiellement .



; ------------------------------------------------------------------------------
--
; FONCTIONS DIVERSES:
; ------------------------------------------------------------------------------
--

; GET-TRANSFORMED-OBS-VAR-IF-OBSERVATED redonne le nouveau symbole d'une variable
; transformée, au cas où cette variable a été auparavant observée; sinon cette
; fonction redonne la variable elle-même.

(defmethod get-transformed-obs-var-if-observated (var (TheGate gate))
  (let ((obs-var (obs-var TheGate)))
    (loop for oVar in obs-var
          when (sp-equal? (first oVar) var)
          do (return (second oVar))
          finally (return var))))


; SET-RIGHT-SYMBS transforme une hypothèse d'un query en y remplaçant les
; symboles nécessaires.

(defmethod set-right-symbs (TheArg (TheGate gate))
  (if (atom TheArg)
    (let ((NewSymb (translate-in-good-symbol-1 TheArg (universe TheGate))))
      (if NewSymb
        (get-transformed-obs-var-if-observated NewSymb TheGate)
        (get-transformed-obs-var-if-observated TheArg  TheGate)))
    (cons (set-right-symbs (first TheArg) TheGate)
          (set-right-symbs (rest  TheArg) TheGate))))



; ------------------------------------------------------------------------------
--
; DIFFERENTS QUERIES:
; ------------------------------------------------------------------------------
--

(defmacro DIG-support (gate theArg)
  (let ((theResult (DIG-SymbolicQuery_Handler gate
```

```
                                                TheArg
                                                :proc #'DIG-support_iter)))
    (reset-ABL-environment)
    `(print* (quote ,theResult))))


(defmacro DIG-quasi-support (gate theArg)
  (let ((theResult (DIG-SymbolicQuery_Handler gate
                                                TheArg
                                                :proc #'DIG-quasi-support_iter)))
    (reset-ABL-environment)
    `(print* (quote ,theResult))))


(defmacro DIG-plausibility (gate theArg)
  (let ((theResult (DIG-SymbolicQuery_Handler gate
                                                TheArg
                                                :proc #'DIG-plausibility_iter)))
    (reset-ABL-environment)
    `(print* (quote ,theResult))))


(defmacro DIG-doubt (gate theArg)
  (let ((TheResult (DIG-SymbolicQuery_Handler gate
                                                TheArg
                                                :proc #'DIG-doubt_iter)))
    (reset-ABL-environment)
    `(print* (quote ,TheResult))))


; Les fonction itératives sont appelées une deuxième fois par
; DIG-SymbolicQuery_Handler, si le premier essai n'a pas obtenu de réponse.

(defun DIG-support_iter (gate TheArg)
  (DIG-Handle_Calculate (eval gate))
  (let ((NewArg (MakeClusterQuery TheArg)))
    (Handle_SymbolicCalc_Conj NewArg :proc #'support*)))


(defun DIG-quasi-support_iter (gate TheArg)
  (DIG-Handle_Calculate (eval gate))
  (let ((NewArg (MakeClusterQuery TheArg)))
    (Handle_SymbolicCalc_Conj NewArg :proc #'quasi-support*)))


(defun DIG-plausibility_iter (gate TheArg)
  (DIG-Handle_Calculate (eval gate))
  (let ((NewArg (MakeClusterQuery TheArg)))
    (Handle_SymbolicCalc_Disj NewArg :proc #'plausibility*)))


(defun DIG-doubt_iter (gate TheArg)
  (DIG-Handle_Calculate (eval gate))
  (let ((NewArg (MakeClusterQuery TheArg)))
    (Handle_SymbolicCalc_Disj NewArg :proc #'doubt*)))


(defun DIG-SymbolicQuery_Handler (gate TheArg &key proc)
  (let* ((theGate (eval gate))
         (newArg (set-right-symbs TheArg theGate)))
    (if (not (typep theGate 'gate))
      (error "~a must be from the class gate." gate)
      (handler-case
        (progn (setf *second-path* NOT-SECOND-PATH)
               (let ((EndResult
                      (or (catch 'Invalid_hypothesis
                            (funcall proc gate newArg))
                          (progn (setf *second-path* SECOND-PATH)
                                 (Handle_Clustering newArg)
                                 (catch 'Invalid_hypothesis
                                   (funcall proc gate newArg))))))
                 (if EndResult
                   EndResult
                   (progn
                     (reset-ABL-environment)
                     (error "Invalid hypothesis")))))
        (error (condition)
```

```
               (progn (reset-ABL-environment)
                      (error condition)))))))


; Mêmes queries, mais en abréviation:
; -----------------------------------

(defmacro DIG-sp (gate TheHypothesis)
  (let ((TheArg (cond ((eq TheHypothesis 'T) 'tautology)
                      ((eq TheHypothesis 'C) 'contradiction)
                      (T TheHypothesis))))
    `(dig-support ,gate ,TheArg)))


(defmacro DIG-pl (gate TheHypothesis)
  (let ((TheArg (cond ((eq TheHypothesis 'T) 'tautology)
                      ((eq TheHypothesis 'C) 'contradiction)
                      (T TheHypothesis))))
    `(dig-plausibility ,gate ,TheArg)))


(defmacro DIG-db (gate TheHypothesis)
  (let ((TheArg (cond ((eq TheHypothesis 'T) 'tautology)
                      ((eq TheHypothesis 'C) 'contradiction)
                      (T TheHypothesis))))
    `(dig-doubt ,gate ,TheArg)))


(defmacro DIG-qs (gate TheHypothesis)
  (let ((TheArg (cond ((eq TheHypothesis 'T) 'tautology)
                      ((eq TheHypothesis 'C) 'contradiction)
                      (T TheHypothesis))))
    `(dig-quasi-support ,gate ,TheArg)))
```

Annexe C24 DIG-Query_Dialog.lisp

```
; ——————————————————————————————————————————————————————————————————————————
—
; ———                              FRIBOURG UNIVERSITY                          —
—
; ———                            INSTITUTE OF INFORMATICS                       —
—
; ———                 Rue Faucigny 2, CH-1700 Fribourg, Switzerland             —
—
; ——————————————————————————————————————————————————————————————————————————
—
; TITLE:    DIG-Query_Dialog.lisp
; SUPPORT:  Michael Schumacher
; PROJECT:  DIG
; DATE:     06.07.95
; VERSION:  1.0
; ——————————————————————————————————————————————————————————————————————————
—
; OVERVIEW: Interface utility for DIG queries.
; ——————————————————————————————————————————————————————————————————————————
—



; Abréviations:
; -------------
; Btn  = Button
; Sqn  = Sequence
; STxt = Simple Text
; Dlg  = Dialog
; Win  = Window
; ET   = Editable Text


(defun translate-in-good-symbol-3 (the-symbol the-list)
  (loop for s in the-list
        if (equal? (string-upcase (string s))
                   (string-upcase (string the-symbol)))
        return s))


(defun DIG-Dlg-query-action (Title GateET VarET)
  (let* ((GateSymb (translate-in-good-symbol-3
                    (make-symbol (dialog-item-text GateET))
                    *gates*))
         (VarSymb (if (not GateSymb)
                    (error "~a must be a gate." (dialog-item-text GateET))
                    (translate-in-good-symbol-3
                     (make-symbol (dialog-item-text VarET))
                     (universe (eval GateSymb))))))
    (cond ((not VarSymb)
           (error "~a does not exist in the gate ~a."
                  (dialog-item-text VarET)
                  GateSymb))
          ((equal Title "DIG-support")
           (eval `(DIG-support ,GateSymb ,VarSymb)))
          ((equal Title "DIG-quasi-support")
           (eval `(DIG-quasi-support ,GateSymb ,VarSymb)))
          ((equal Title "plausibility")
           (eval `(plausibility ,GateSymb ,VarSymb)))
          ((equal Title "doubt")
           (eval `(doubt ,GateSymb ,VarSymb)))
          ((equal Title "sp")
           (eval `(sp ,GateSymb ,VarSymb)))
          ((equal Title "pl")
           (eval `(pl ,GateSymb ,VarSymb)))
          ((equal Title "db")
           (eval `(db ,GateSymb ,VarSymb)))
          ((equal Title "qs")
           (eval `(qs ,GateSymb ,VarSymb))))))
```

Génération d'impliqués premiers et déduction basée sur suppositions au moyen de clauses disjonctives.

```
(defun DIG-Query_Dialog (Title)
  (let* ((Canceled NIL)
         (GateET (MAKE-DIALOG-ITEM 'EDITABLE-TEXT-DIALOG-ITEM
                                   #@(81 10) #@(227 17)
                                   ""
                                   'NIL
                                   :ALLOW-RETURNS NIL))
         (VarET (MAKE-DIALOG-ITEM 'EDITABLE-TEXT-DIALOG-ITEM
                                  #@(81 35) #@(227 17)
                                  ""
                                  'NIL
                                  :ALLOW-RETURNS NIL))
         (OKBtn (MAKE-DIALOG-ITEM 'BUTTON-DIALOG-ITEM
                                  #@(62 68) #@(62 16)
                                  "OK"
                                  #'(LAMBDA (ITEM)
                                      (DECLARE (IGNORE ITEM))
                                      (RETURN-FROM-MODAL-DIALOG T))
                                  :DEFAULT-BUTTON T))
         (CancelBtn (MAKE-DIALOG-ITEM 'BUTTON-DIALOG-ITEM
                                      #@(162 68) #@(62 16)
                                      "Cancel"
                                      #'(LAMBDA (ITEM)
                                          (DECLARE (IGNORE ITEM))
                                          (setf Canceled T)
                                          (RETURN-FROM-MODAL-DIALOG T))
                                      :DEFAULT-BUTTON NIL))
         (GateSTxt (MAKE-DIALOG-ITEM 'STATIC-TEXT-DIALOG-ITEM
                                     #@(9 10) #@(56 16)
                                     "Gate:"
                                     'NIL))
         (VarSTxt (MAKE-DIALOG-ITEM 'STATIC-TEXT-DIALOG-ITEM
                                    #@(8 34) #@(62 17)
                                    "Variable:"
                                    'NIL))
         (QueryDlg (MAKE-INSTANCE 'DIALOG
                     :WINDOW-TYPE
                     :DOCUMENT
                     :WINDOW-TITLE Title
                     :VIEW-POSITION #@(86 88)
                     :VIEW-SIZE #@(317 97)
                     :CLOSE-BOX-P NIL
                     :VIEW-FONT '("Chicago" 12 :SRCOR :PLAIN)
                     :VIEW-SUBVIEWS (LIST OKBtn
                                          CancelBtn
                                          GateET
                                          VarET
                                          GateSTxt
                                          VarSTxt))))
    (modal-dialog QueryDlg)
    (when (not Canceled)
      (DIG-Dlg-query-action Title GateET VarET))))
```

Annexe C25 DIG-View_Dialog.lisp

```
; ————————————————————————————————————————————————————————————————————————————
—
; ———                              FRIBOURG UNIVERSITY                           —
—
; ———                            INSTITUTE OF INFORMATICS                        —
—
; ———                 Rue Faucigny 2, CH-1700 Fribourg, Switzerland              —
—
; ————————————————————————————————————————————————————————————————————————————
—
; TITLE:    DIG-View_Dialog.lisp
; PROJECT:  DIG
; SUPPORT:  Michael Schumacher
; DATE:     03.07.95
; VERSION:  1.0
; ————————————————————————————————————————————————————————————————————————————
—
; OVERVIEW: Dig-View interface utility.
; ————————————————————————————————————————————————————————————————————————————
—



; Abréviations:
; -------------
; Btn  = Button
; Sqn  = Sequence
; STxt = Simple Text
; Dlg  = Dialog
; Win  = Window
; ET   = Editable Text



; ----------------------------------------------------------------------------
--
; Classes:
; ----------------------------------------------------------------------------
--

(defclass DIGVIEWDIALOG (DIALOG)
  ((modules-Sqn :accessor modules-Sqn
                :initarg :modules-Sqn
                :initform '())
   (Gates-Sqn :accessor Gates-Sqn
                      :initarg :Gates-Sqn
                      :initform '())))

(defclass DIGVIEWTABLE (sequence-dialog-item)
  ())

(defclass ModulesDigViewTable (DIGVIEWTABLE)
  ())

(defclass GatesDigViewTable (DIGVIEWTABLE)
  ())



; ----------------------------------------------------------------------------
--
; DIFFERENTES METHODES:
; ----------------------------------------------------------------------------
--

; Get_Selected_ExprList:
; ----------------------

(defmethod Get_Selected_ExprList ((TheTable DIGVIEWTABLE))
  (let ((Sqn (table-sequence TheTable)))
    (loop for i from 0 to (point-v (table-dimensions TheTable))
```

```
          for item in Sqn
          when (cell-selected-p TheTable 0 i)
          collect item)))


; View-click-event-handler:
; -------------------------

(defmethod view-click-event-handler ((TheTable DigViewTable) where)
  (call-next-method TheTable where))

(defmethod view-click-event-handler ((TheTable ModulesDigViewTable) where)
  (let ((TheCellPoint (point-to-cell TheTable where)))
    (when (and TheCellPoint
               (double-click-p))
      (inspect (eval `(function ,(nth (point-v TheCellPoint)
                                      (table-sequence TheTable))))))
    (call-next-method TheTable where)))

(defmethod view-click-event-handler ((TheTable GatesDigViewTable) where)
  (let ((TheCellPoint (point-to-cell TheTable where)))
    (when (and TheCellPoint
               (double-click-p))
      (inspect (eval (nth (point-v TheCellPoint)
                          (table-sequence TheTable)))))
    (call-next-method TheTable where)))



; ------------------------------------------------------------------------------
--
; BUTTONS:
; ------------------------------------------------------------------------------
--

; Set Primitives to Standard:
; ---------------------------

(defmethod digV-Set-primitives-to-standard_Action ((TheButton button-dialog-item))
  (set-to-standard))

(defun Set-primitives-to-standard_Btn ()
  (MAKE-DIALOG-ITEM
   'BUTTON-DIALOG-ITEM
   #@(25 276)
   #@(226 17)
   "Set Primitives to Standard"
   #'digV-Set-primitives-to-standard_Action
   :DEFAULT-BUTTON
   NIL))


; Resample:
; ---------

(defmethod digV-Resample_Action ((TheButton button-dialog-item))
  (let* ((ViewDialog (view-container TheButton))
         (modules-sequence (modules-sqn ViewDialog))
         (Gates-sequence (Gates-sqn ViewDialog)))
    (set-table-sequence modules-sequence *dig-modules*)
    (set-table-sequence Gates-sequence *gates*)))

(defun Resample_Btn ()
  (MAKE-DIALOG-ITEM
   'BUTTON-DIALOG-ITEM
   #@(285 276)
   #@(78 17)
   "Resample"
   #'digV-Resample_Action
   :DEFAULT-BUTTON
   NIL))


; Inspect:
; ---------

(defmethod digV-Inspect_Action ((TheButton button-dialog-item))
```

```
  (let* ((ViewDialog (view-container TheButton))
         (modules-sequence (modules-sqn ViewDialog))
         (Gates-sequence (Gates-sqn ViewDialog))
         (modules (Get_Selected_ExprList modules-sequence))
         (Gates (Get_Selected_ExprList Gates-sequence)))
    (loop for item in modules
          do (inspect (eval `(function ,item))))
    (loop for item in Gates
          do (inspect (eval item)))))


(defun Inspect_Btn ()
  (MAKE-DIALOG-ITEM
   'BUTTON-DIALOG-ITEM
   #@(122 240)
   #@(62 16)
   "Inspect"
   #'digV-Inspect_Action
   :DEFAULT-BUTTON
   T))


; Delete:
; -------

(defmethod digV-Delete_Action ((TheButton button-dialog-item))
  (let* ((ViewDialog (view-container TheButton))
         (modules-sequence (modules-sqn ViewDialog))
         (Gates-sequence (Gates-sqn ViewDialog))
         (selected-modules (Get_Selected_ExprList modules-sequence))
         (selected-gates (Get_Selected_ExprList Gates-sequence)))
    (loop for item in selected-modules
          do (progn (fmakunbound item)
                    (setf *DIG-modules* (remove item *DIG-modules*))))
    (loop for item in selected-gates
          do (progn (makunbound item)
                    (setf *gates* (remove item *gates*))))
    (set-table-sequence modules-sequence *dig-modules*)
    (set-table-sequence Gates-sequence *gates*)))

(defun Delete_Btn ()
  (MAKE-DIALOG-ITEM
   'BUTTON-DIALOG-ITEM
   #@(205 240)
   #@(62 16)
   "Delete"
   #'digV-Delete_Action
   :DEFAULT-BUTTON
   NIL))


; Modules-All:
; ------------

(defmethod digV-Modules-All_Action ((TheButton button-dialog-item))
  (let* ((ViewDialog (view-container TheButton))
         (modules-sequence (modules-sqn ViewDialog)))
    (loop for i from 0 to (1- (point-v (table-dimensions modules-sequence)))
          do (cell-select modules-sequence 0 i))))

(defun Modules-All_Btn ()
  (MAKE-DIALOG-ITEM
   'BUTTON-DIALOG-ITEM
   #@(9 213)
   #@(62 16)
   "All"
   #'digV-Modules-All_Action
   :DEFAULT-BUTTON
   NIL))


; Modules-None:
; -------------

(defmethod digV-Modules-None_Action ((TheButton button-dialog-item))
  (let* ((ViewDialog (view-container TheButton))
         (modules-sequence (modules-sqn ViewDialog)))
```

```
    (loop for i from 0 to (1- (point-v (table-dimensions modules-sequence)))
          do (cell-deselect modules-sequence 0 i))))

(defun Modules-None_Btn ()
  (MAKE-DIALOG-ITEM
   'BUTTON-DIALOG-ITEM
   #@(84 213)
   #@(62 16)
   "None"
   #'digV-Modules-None_Action
   :DEFAULT-BUTTON
   NIL))


; Gates-All:
; ----------

(defmethod digV-Gates-All_Action ((TheButton button-dialog-item))
  (let* ((ViewDialog (view-container TheButton))
         (Gates-sequence (Gates-sqn ViewDialog)))
    (loop for i from 0 to (1- (point-v (table-dimensions Gates-sequence)))
          do (cell-select Gates-sequence 0 i))))

(defun Gates-All_Btn ()
  (MAKE-DIALOG-ITEM
   'BUTTON-DIALOG-ITEM
   #@(257 215)
   #@(62 16)
   "All"
   #'digV-Gates-All_Action
   :DEFAULT-BUTTON
   NIL))


; Gates-None:
; -----------

(defmethod digV-Gates-None_Action ((TheButton button-dialog-item))
  (let* ((ViewDialog (view-container TheButton))
         (Gates-sequence (Gates-sqn ViewDialog)))
    (loop for i from 0 to (1- (point-v (table-dimensions Gates-sequence)))
          do (cell-deselect Gates-sequence 0 i))))

(defun Gates-None_Btn ()
  (MAKE-DIALOG-ITEM
   'BUTTON-DIALOG-ITEM
   #@(332 215)
   #@(62 16)
   "None"
   #'digV-Gates-None_Action
   :DEFAULT-BUTTON
   NIL))



; ------------------------------------------------------------------------------
--
; SEQUENCE DIALOGS ITEMS:
; ------------------------------------------------------------------------------
--

(defun Modules_Sqn ()
  (MAKE-DIALOG-ITEM
   'ModulesDigViewTable
   #@(11 41)
   #@(175 163)
   "MODULES"
   'NIL
   :CELL-SIZE
   #@(159 16)
   :SELECTION-TYPE
   :DISJOINT
   :TABLE-HSCROLLP
   NIL
   :TABLE-VSCROLLP
   T
   :TABLE-SEQUENCE
```

```
   *DIG-modules*))

(defun Gates_Sqn ()
  (MAKE-DIALOG-ITEM
   'GatesDigViewTable
   #@(207 41)
   #@(185 164)
   "MODULES"
   NIL
   :CELL-SIZE
   #@(169 16)
   :SELECTION-TYPE
   :DISJOINT
   :TABLE-HSCROLLP
   NIL
   :TABLE-VSCROLLP
   T
   :TABLE-SEQUENCE
   *gates*))



; ------------------------------------------------------------------------------
--
; SIMPLE TEXTS:
; ------------------------------------------------------------------------------
--


(defun Gates_STxt ()
  (MAKE-DIALOG-ITEM
   'STATIC-TEXT-DIALOG-ITEM
   #@(205 14)
   #@(116 17)
   "Gates:"
   'NIL))

(defun Modules_STxt ()
  (MAKE-DIALOG-ITEM
   'STATIC-TEXT-DIALOG-ITEM
   #@(11 15)
   #@(147 16)
   "Modules:"
   'NIL))



; ------------------------------------------------------------------------------
--
; DIG-VIEW DIALOG:
; ------------------------------------------------------------------------------
--

(defun Dig-View_Dlg ()
  (let* ((modules-sequence (Modules_Sqn))
         (Gates-sequence (Gates_Sqn))
         (dig-view_Win
          (MAKE-INSTANCE 'DIGVIEWDIALOG
            :WINDOW-TYPE
            :DOCUMENT
            :WINDOW-TITLE "DIG-View"
            :VIEW-POSITION #@(38 62)
            :VIEW-SIZE #@(405 314)
            :VIEW-FONT '("Chicago" 12 :SRCOR :PLAIN)
            :VIEW-SUBVIEWS (LIST (set-primitives-to-standard_Btn)
                                 (Resample_Btn)
                                 (Inspect_Btn)
                                 (Gates_STxt)
                                 (Modules_STxt)
                                 (Delete_Btn)
                                 (Modules-All_Btn)
                                 (Modules-None_Btn)
                                 (Gates-All_Btn)
                                 (Gates-None_Btn)
                                 modules-sequence
                                 Gates-sequence)
            :MODULES-SQN modules-sequence
```

```
            :GATES-SQN Gates-sequence)))
    dig-view_Win))


; (Dig-View_Dlg)
```

Annexe C26 DIG-About_Dialog.lisp

```
; ——————————————————————————————————————————————————————————————————————————
—
; ——                              FRIBOURG UNIVERSITY                          —
—
; ——                           INSTITUTE OF INFORMATICS                        —
—
; ——                Rue Faucigny 2, CH-1700 Fribourg, Switzerland              —
—
; ——————————————————————————————————————————————————————————————————————————
—
; TITLE:    DIG-About_Dialog.lisp
; SUPPORT:  Michael Schumacher
; PROJECT:  DIG
; DATE:     3.07.95
; VERSION:  1.0
; ——————————————————————————————————————————————————————————————————————————
—
; OVERVIEW: About dialog of DIG.
; ——————————————————————————————————————————————————————————————————————————
—


(defun DIG-About-Dialog ()
  (let ((AboutDialog
         (MAKE-INSTANCE 'DIALOG
           :WINDOW-TYPE
           :DOUBLE-EDGE-BOX
           :WINDOW-TITLE
           ""
           :VIEW-POSITION
           #@(139 65)
           :VIEW-SIZE
           #@(307 321)
           :CLOSE-BOX-P
           NIL
           :VIEW-FONT
           '("Chicago" 12 :SRCOR :PLAIN)
           :VIEW-SUBVIEWS
           (LIST (MAKE-DIALOG-ITEM
                  'STATIC-TEXT-DIALOG-ITEM
                  #@(32 63)
                  #@(260 220)
                  "Generation of Prime Implicates
and assumption-based deduction
using disjunctive clauses
in modeled digital circuits.

by Michael Schumacher,
with the support of Prof. Dr. J. Kohlas
and Rolf Haenni.

Institute of Informatics, University of Fribourg, Switzerland,
September 1995."
                  'NIL)
                 (MAKE-DIALOG-ITEM
                  'BUTTON-DIALOG-ITEM
                  #@(122 272)
                  #@(62 16)
                  "OK"
                  #'(LAMBDA (ITEM)
                      (DECLARE (IGNORE ITEM))
                      (RETURN-FROM-MODAL-DIALOG T))
                  :VIEW-NICK-NAME
                  'DIG-ABOUT-DIALOG
                  :DEFAULT-BUTTON
                  T)
                 (MAKE-DIALOG-ITEM
                  'STATIC-TEXT-DIALOG-ITEM
                  #@(48 16)
                  #@(181 27)
```

```
                "DIG, Version 1.0"
                'NIL
                :VIEW-FONT
                '("Chicago" 18 :SRCOR))))))
    (modal-dialog AboutDialog)))
```

Annexe C27 DIG-Menu.lisp

```
; ————————————————————————————————————————————————————————————————————————————
—
; ——                              FRIBOURG UNIVERSITY                           —
—
; ——                            INSTITUTE OF INFORMATICS                        —
—
; ——                 Rue Faucigny 2, CH-1700 Fribourg, Switzerland              —
—
; ————————————————————————————————————————————————————————————————————————————
—
; TITLE:    DIG-Menu.lisp
; SUPPORT:  Michael Schumacher
; PROJECT:  DIG
; DATE:     06.07.95
; VERSION:  1.0
; ————————————————————————————————————————————————————————————————————————————
—
; OVERVIEW: DIG Menu utility.
; ————————————————————————————————————————————————————————————————————————————
—



; Abréviations:
; -------------
; Btn  = Button
; Sqn  = Sequence
; STxt = Simple Text
; Dlg  = Dialog
; Win  = Window
; ET   = Editable Text


(defun DIG-install-menu ()
  (let ((DIG-menu
         (MAKE-INSTANCE 'MENU
           :MENU-TITLE
           "DIG"
           :MENU-ITEMS
           (LIST (MAKE-INSTANCE 'MENU-ITEM
                   :MENU-ITEM-TITLE
                   "About DIG..."
                   :MENU-ITEM-ACTION
                   #'(LAMBDA NIL (DIG-ABOUT-DIALOG)))
                 (MAKE-INSTANCE 'MENU-ITEM :MENU-ITEM-TITLE "-" :DISABLED T)
                 (MAKE-INSTANCE 'MENU-ITEM
                   :MENU-ITEM-TITLE
                   "DIG-View"
                   :MENU-ITEM-ACTION
                   #'(LAMBDA NIL (DIG-VIEW_DLG)))
                 (MAKE-INSTANCE 'MENU-ITEM :MENU-ITEM-TITLE "-" :DISABLED T)
                 (MAKE-INSTANCE 'MENU-ITEM
                   :MENU-ITEM-TITLE
                   "DIG-support"
                   :MENU-ITEM-ACTION
                   #'(LAMBDA NIL (DIG-Query_Dialog "DIG-support")))
                 (MAKE-INSTANCE 'MENU-ITEM
                   :MENU-ITEM-TITLE
                   "DIG-quasi-support"
                   :MENU-ITEM-ACTION
                   #'(LAMBDA NIL (DIG-Query_Dialog "DIG-quasi-support")))
                 (MAKE-INSTANCE 'MENU-ITEM
                   :MENU-ITEM-TITLE
                   "DIG-plausibility"
                   :MENU-ITEM-ACTION
                   #'(LAMBDA NIL (DIG-Query_Dialog "DIG-plausibility")))
                 (MAKE-INSTANCE 'MENU-ITEM
                   :MENU-ITEM-TITLE
                   "DIG-doubt"
                   :MENU-ITEM-ACTION
                   #'(LAMBDA NIL (DIG-Query_Dialog "DIG-doubt")))))))
```

```
    (menu-install DIG-Menu)
    (Init_Fred-special-indent-alist)
    'done))


(DIG-install-menu)
```

Annexe C28 `Makefile`

```
; ------------------------------------------------------------------------------
--
; ---                            FRIBOURG UNIVERSITY                           -
--
; ---                          INSTITUTE OF INFORMATICS                        -
--
; ---               Rue Faucigny 2, CH-1700 Fribourg, Switzerland              -
--
; ------------------------------------------------------------------------------
--
; TITLE:    Makefile
; SUPPORT:  Michael Schumacher
; PROJECT:  DIG
; DATE:     15.06.95
; VERSION:  1.0
; ------------------------------------------------------------------------------
--
; OVERVIEW: Compiling and loading DIG.
; ------------------------------------------------------------------------------
--



; ------------------------------------------------------------------------------
--
; FILENAMES:
; ------------------------------------------------------------------------------
--

(defparameter HNF-Filename_list
  '("utilities"
    "nf-MAX"
    "nf-intersection+"
    "hnf-meet"
    "hnf-join"
    "nf-phi"
    "hnf-gen_pi"
    "hnf"))


(defparameter DIG-Language-Filename_list
  '("ABL_extension"
    "language-utilities"
    "primitives"
    "syntax"
    "gate"
    "defprimitive"
    "defmodule"
    "inst"
    "obs"
    "new-var"
    "delete-module"
    "delete-gate"
    "commun-primitives"
    "dig-Calculate"
    "queries"))


(defparameter DIG-Interface-Filename_list
  '("DIG-Query_Dialog"
    "DIG-View_Dialog"
    "DIG-About_Dialog"
    "DIG-menu"))


; ------------------------------------------------------------------------------
--
; PATHNAMES:
; ------------------------------------------------------------------------------
--
```

```
; !!!!!!!!!!!!!!!!!!!!!!!!!!!!
; PLEASE REPLACE YOUR-PATHNAME
; !!!!!!!!!!!!!!!!!!!!!!!!!!!!

(defparameter fasl_PN #P"YOUR-PATHNAME")

(defun default_PN ()
  (setf *default-pathname-defaults* #P"ccl;"))


#|
(defun fasl_PN ()
  (setf *default-pathname-defaults* #P"YOUR-PATHNAME:FASL_Files:"))

(defun hnf_PN ()
  (setf *default-pathname-defaults* #P"YOUR-PATHNAME:HNF:"))

(defun DIG-Language_PN ()
  (setf *default-pathname-defaults* #P"YOUR-PATHNAME:DIG-Language:"))

(defun Interface_PN ()
  (setf *default-pathname-defaults* #P"YOUR-PATHNAME:Interface:"))


; ------------------------------------------------------------------------------
--
; COMPILING:
; ------------------------------------------------------------------------------
--

(defun DIG_CompileFiles (Files &key proc-set-pn)
  (funcall proc-set-pn)
  (loop for FileName in Files
        do (progn (format t "~&Compiling: ~a" Filename)
                  (compile-file
                   FileName
                   :output-file (merge-pathnames fasl_PN FileName)))))


(defun DIG_Compile ()
  (DIG_CompileFiles HNF-Filename_list :proc-set-pn #'hnf_PN)
  (DIG_CompileFiles DIG-Language-Filename_list :proc-set-pn #'DIG-Language_PN)
  (DIG_CompileFiles DIG-Interface-Filename_list :proc-set-pn #'Interface_PN)
  (default_PN))



; ------------------------------------------------------------------------------
--
; LOADING:
; ------------------------------------------------------------------------------
--

(defun DIG_LoadFiles (Files &key proc-set-pn)
  (funcall proc-set-pn)
  (loop for FileName in Files
        do (progn (format t "~&Loading: ~a" Filename)
                  (load FileName))))


(defun DIG_Load ()
  (DIG_LoadFiles HNF-Filename_list :proc-set-pn #'fasl_PN)
  (DIG_LoadFiles DIG-Language-Filename_list :proc-set-pn #'fasl_PN)
  (DIG_LoadFiles DIG-Interface-Filename_list :proc-set-pn #'fasl_PN)
  (default_PN))


; ------------------------------------------------------------------------------
--
; COMPILING AND LOADING:
; ------------------------------------------------------------------------------
--

(defun DIG_Compile_and_Load_Files (Files &key proc-set-pn)
  (loop for FileName in Files
        do (progn (funcall proc-set-pn)
```

```
                  (format t "~&Compiling: ~a" Filename)
                  (compile-file
                   FileName
                   :output-file (merge-pathnames fasl_PN FileName))
                  (fasl_PN)
                  (format t "~&Loading:   ~a" Filename)
                  (load FileName))))


(defun DIG_Compile_and_Load ()
  (DIG_Compile_and_Load_Files HNF-Filename_list :proc-set-pn #'hnf_PN)
  (DIG_Compile_and_Load_Files DIG-Language-Filename_list :proc-set-pn #'DIG-
Language_PN)
  (DIG_Compile_and_Load_Files DIG-Interface-Filename_list :proc-set-pn
#'Interface_PN)
  (default_PN))



;
================================================================================
; (DIG_Compile_and_Load)
; (DIG_Compile)
; (DIG_load)
```