Partie 1

Les bases de la programmation avec LabVIEW

Après la présentation des principes généraux de la programmation graphique, cette partie va introduire l'environnement de programmation de ce langage graphique dans toutes ses particularités.

Le premier chapitre est consacré à la programmation flot de données utilisée dans l'environnement LabVIEW en montrant ses spécificités en termes de contrôle d'exécution et de propagation des données. Il met en exergue la limitation de cette programmation qui impose l'ajout de structures de contrôle et d'éléments permettant la mémorisation des données.

Le deuxième chapitre présente l'environnement de développement très particulier avec ses deux facettes : l'interface utilisateur et le programme flot de données lui-même. Cela permet de découvrir d'une part les deux palettes différentes des objets qui peuvent être utilisées dans le développement de l'interface utilisateur et du programme et d'autre part la palette des outils qui permettent les actions sur ce développement et sur les objets manipulés.

Le troisième chapitre est consacré à la réalisation d'un programme au sens propre de la construction de ce flot de données. Le câblage associé à cette réalisation du programme est présenté de façon détaillée afin d'offrir des bases très solides d'« écriture » d'un programme LabVIEW à un nouvel utilisateur.

Le quatrième chapitre qui termine cette prise en main de l'environnement de développement LabVIEW traite de l'exécution d'un programme réalisé, mais surtout de la mise au point de ce programme. La richesse de ces outils de mise au point répond ainsi à la complexité de cet environnement. Ce chapitre se termine par la création d'un sous-programme, c'est-à-dire l'encapsulation d'un programme qui est la base de la réalisation de programmes simples.

Chapitre 1

Les grands paradigmes de la programmation graphique

1 La naissance des langages graphiques

Un langage informatique est une notation textuelle ou graphique qui permet d'exprimer un ensemble d'actions ou fonctions, appelé algorithme, traduisant l'application souhaitée par l'utilisateur. Ces fonctions agissent sur des données (variables ou constantes) spécifiques de l'application qui est désignée par « programme informatique ». Ainsi, les applications ou programmes sont d'une très grande variété, allant du programme de gestion (*i.e.* gestion financière) au programme de calcul scientifique (*i.e.* programme de simulation météorologique) en passant par des programmes de contrôle/commande (*i.e.* pilotage d'expérimentations). Dans cet ouvrage, nous nous intéresserons plus particulièrement à ces derniers étant donné l'orientation du langage de programmation LabVIEW.

Afin de mieux comprendre ce mode spécifique de programmation, il est intéressant de situer ce nouveau paradigme du langage graphique flot de données, sur lequel LabVIEW est basé, dans l'ensemble des modes de programmation (Fig. 1.1). Les langages de programmation sont découpés en deux familles : langages impératifs (le programme consiste à exécuter les instructions ou les fonctions les unes après les autres) et langages déclaratifs (le programme, composé d'un ensemble de fonctions ou règles, consiste à les évaluer en fonction des données disponibles).

La première classe est la plus nombreuse et se compose des trois sous-familles suivantes :

- des langages à programmation procédurale comme Ada, Pascal ou C (l'exécution du programme suit « séquentiellement » la suite des ordres ou instructions) ;
- des langages à programmation orientée objet comme C++ ou Java (découpage du programme en entités « quasi indépendantes » et interconnectées en exécution avec des échanges de données) ;
- des langages à programmation concurrente comme Ada (exécution de parties du programme en parallèle).

La première catégorie de cette classe, qui est la plus fournie, la plus ancienne (*i.e.* langages assembleurs) et la plus classique, a vu aussi des développements de quelques langages graphiques spécifiques comme LUSTRE et Esterel entre autres.

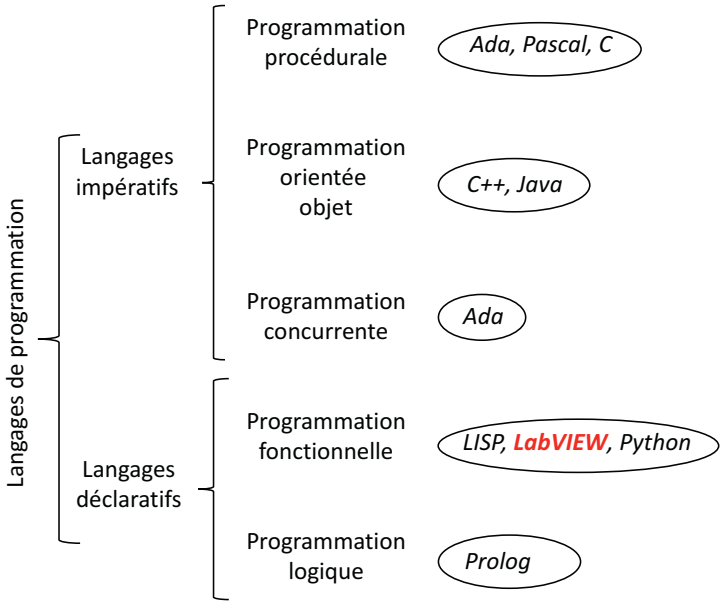

Figure 1.1 – Classification des langages de programmation.

La deuxième grande famille de langages de programmation « langages déclaratifs » regroupe deux entités : les langages basés sur une programmation fonctionnelle, dont fait partie le langage graphique LabVIEW, et les langages basés sur une programmation logique (langages sources de l'intelligence artificielle).

Même si souvent des aspects graphiques ont été ajoutés à l'environnement de programmation comme une aide au développement afin de rendre un programme plus rapidement compréhensible avec par exemple pour un texte des moyens graphiques, tels que le surlignage, les italiques, les caractères gras, la couleur ou l'indentation d'un texte. Pourtant, ce qui a souvent rendu complexe la compréhension, la mise au point et la correction (debuggage) des programmes est le fait qu'ils soient ensuite généralement compilés donnant des programmes exécutables par la machine (code du processeur) qui peut s'éloigner fortement de l'idée initiale du concepteur.

Cela a conduit au développement de la programmation basée sur un environnement graphique qui date des années 1980 et qui a vu le jour grâce à l'apparition d'ordinateur proposant cette capacité liée à une interface graphique (Macintosh d'Apple™ en 1984). La possibilité de dessiner son programme (la liste des actions souhaitées) permet au concepteur d'exprimer ses idées d'une manière plus intuitive et plus naturelle. L'esprit humain perçoit et comprend les concepts compliqués plus rapidement lorsque ceux-ci sont représentés graphiquement. Lorsque des objets peuvent être manipulés graphiquement, la communication homme-machine est grandement facilitée. Ce principe a

conduit au développement des systèmes d'exploitation graphiques de tous les ordinateurs actuels.

Le but de la programmation graphique est donc de faciliter la mise en œuvre d'applications informatiques. On peut dégager les avantages suivants :

- la facilité d'utilisation par les non-programmeurs : les images sont en général préférées aux mots, la programmation devient alors plus intuitive ;
- la sémantique des images est plus puissante que celle des mots (davantage de signification dans une unité d'expression plus concise) ;
- les images ne sont pas sujettes aux barrières des langues.

Il est évident que toute méthode à ses inconvénients. Ainsi les principaux problèmes liés à la programmation graphique sont les suivants :

- la difficulté de visualisation pour les programmes de taille importante nécessitant une architecture modulaire et hiérarchique ;
- l'ambiguïté dans l'interprétation des graphismes ;
- la nécessité de disposer d'un environnement de développement efficace (éditeurs, outil de mise au point, outil de test…).

Lorsque des personnes imaginent ou conçoivent une application, ils ont tendance à dessiner des schémas pour expliquer l'enchaînement et le fonctionnement de l'application qu'ils veulent réaliser. Nous retrouvons alors la représentation naturelle de conception sous forme de « blocs fonctionnels » d'une application. Chaque bloc fonctionnel décrit l'action à réaliser qui peut être plus ou moins complexe.

Ce mode de programmation s'applique parfaitement à la description du contrôle/commande des systèmes industriels (machine outils, véhicules de transport, appareils électroménagers, etc.). Prenons un exemple très simple d'un portail télécommandé (Fig. 1.2). Les trois blocs fonctionnels sont successivement : « Attendre », « Ouvrir portail », « Fermer portail ». Le schéma fonctionnel de cette application ouverture/fermeture automatique d'un portail amène à deux remarques :

- chacun des blocs fonctionnels intègre des actions complexes : commande du moteur électrique du portail, temporisation éventuelle, test de fin d'ouverture ou de fermeture, arrêt de l'action si réception d'un contre-ordre (cas réel de fonctionnement de ce type d'automatisme), etc. ;
- l'enchaînement de ces actions nécessite des évènements ou des données externes pour s'exécuter qui sont ici symbolisés par des « arrivées » entre les blocs fonctionnels. Le passage du dernier bloc fonctionnel à l'état « Attendre » est déclenché par la fin du bloc fonctionnel précédent « Fermer portail ».

La deuxième remarque, effectuée précédemment, conduit à deux formalismes déterminant le mode d'exécution du programme : soit orienté flux de données, soit orienté flux de contrôle. Les représentations orientées flux de contrôle ont longtemps été utilisées pour décrire des algorithmes (avant même l'apparition des premiers ordinateurs). Ces représentations décrivent les programmes comme étant des nœuds de

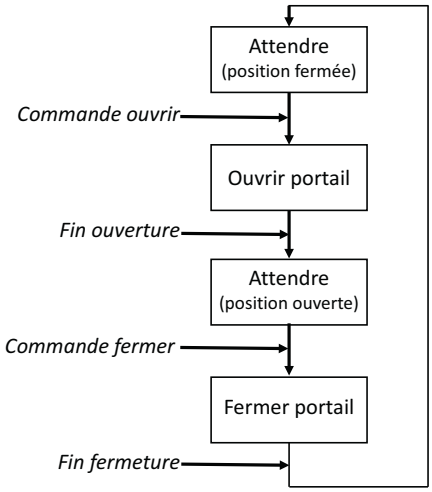

Figure 1.2 – Exemple de la description fonctionnelle graphique d'une application « portail télécommandé ».

calcul connectés par des arcs spécifiant quel est l'ordre de calcul : exemple du modèle GRAFCET très utilisé dans les automatismes industriels réalisés souvent à base de relais électromécaniques (Fig. 1.3 et Encart 1.1).

Au contraire, dans les graphes pilotés par le flux de données, ce sont les données qui dirigent l'exécution du programme, analogie avec un circuit électronique et la propagation du signal à travers ce circuit. Ce type de fonctionnement est celui adopté par le langage de programmation graphique LabVIEW : la **programmation graphique flot de données**.

Encart 1.1 Le GRAFCET, un modèle adapté aux applications de contrôle commande

Le GRAFCET, graphe fonctionnel de commande étape transition ou diagramme fonctionnel en séquence, a été créé et normalisé dans les années 1990. Il est destiné principalement à représenter des automatismes (machines outils, appareils électroménagers, etc.), c'est un moyen de communication adapté entre l'automaticien et le client pour la rédaction du cahier des charges de son application. Le graphe fonctionnel ainsi élaboré est appelé « grafcet » qui est graphe bipartite alterné, composé alternativement de transitions et d'étapes. L'étape représente un état du système qui est en général associé à une action ; elle est dite valide si le système est dans l'état représenté par cette étape. Les transitions sont associées à une condition de franchissement qui peut-être un événement ou une condition, appelée réceptivité.

Les règles d'évolution sont assez simples :

- une transition est validée si toutes les étapes amont sont actives ;
- une transition est franchissable si elle est validée (voir ci-avant) et si sa condition de franchissement est remplie ;

- toutes les transitions franchissables sont franchies en même temps ;
- après ce franchissement, les étapes amont sont invalidées et les étapes aval sont validées.

Ce modèle a été enrichi pour répondre aux besoins en ajoutant en particulier une possibilité d'exprimer les conditions de franchissement des transitions avec une expression en logique booléenne.

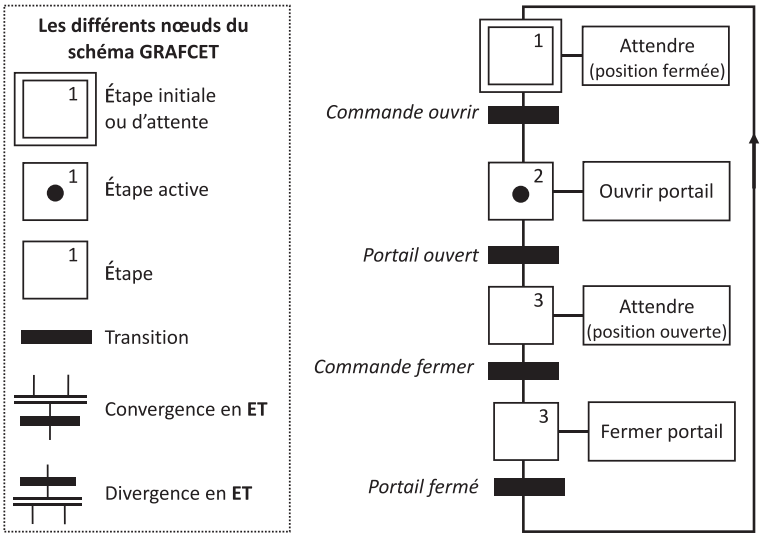

Figure 1.3 – Description de l'application « portail télécommandé » en modèle GRAFCET avec la légende des différents nœuds.

Remarque Dans une programmation graphique flot de données, une action ou une fonction ne s'exécute que si toutes les données nécessaires à son exécution sont présentes (« nouvelles valeurs »), contrairement à un langage impératif (Ada, C…) où une fonction peut s'exécuter avec des données quelles qu'elles soient.

2 La programmation graphique flot de données

La programmation graphique orientée flot de données va se construire sur les bases d'un diagramme flux de données. Ce diagramme flux de données permet de décrire un programme de manière graphique, constitué de nœuds de traitements ou calculs interconnectés par des arcs orientés qui spécifient le flux des données transitant entre les nœuds producteurs de données et les nœuds consommateurs de données. La principale différence par rapport aux langages de type impératif est liée au fait que l'exécution du programme est dictée implicitement par les flots de données. Effectivement dans un

langage impératif comme le langage C, l'exécution va suivre le séquencement des lignes du code écrit sans s'occuper des données traitées ou à traiter.

En résumé, le diagramme flux de données est un graphe acyclique qui peut être composé des trois éléments différents suivants (Fig. 1.4) :

- **terminaux** : ce sont les liens avec l'extérieur qui représentent la production ou la consommation de données. Cette notion de terminaux prend tout son sens lorsque nous étudierons des applications de type industriel, comme le contrôle/commande d'une enceinte thermostatée (lecture d'une température en entrée, commande du chauffage en sortie) ;
- **nœuds** : ce sont les traitements à effectuer qui sont représentés par une figure géométrique pouvant contenir une image illustrant leur fonctionnalité. Ils possèdent des connexions d'entrée et des connexions de sortie. Nous dirons qu'ils consomment toutes leurs données d'entrée lorsqu'ils s'exécutent et qu'ils produisent des données sur toutes leurs sorties à la fin de leur exécution ;
- **arcs orientés** : ils relient nœuds et terminaux ou nœuds et nœuds. Ils permettent d'indiquer le passage de données d'un nœud vers un autre. Un arc orienté peut se séparer en plusieurs arcs : il y a alors duplication des données. En revanche les arcs orientés ne peuvent pas se regrouper, la convergence est interdite.

De l'exemple simple de la figure 1.4, nous pouvons faire trois remarques qui seront détaillées dans la suite de ce chapitre :

En se basant sur les règles d'exécution, plusieurs nœuds peuvent être exécutés simultanément lorsque ces différents nœuds possèdent des données sur toutes leurs entrées. Ainsi dans le diagramme de la figure 1.4, les deux nœuds nommés « – » et « + » peuvent être exécutés en même temps en consommant les données *a* et *b*. La programmation flux de données intègre donc implicitement la notion de **parallélisme**.

- Les données à l'entrée d'un nœud doivent être conformes à l'attente de la fonction à exécuter. Par exemple, les fonctions mathématiques réalisent des opérations entre deux numériques de type flottant ou entre deux numériques entiers, le mélange de

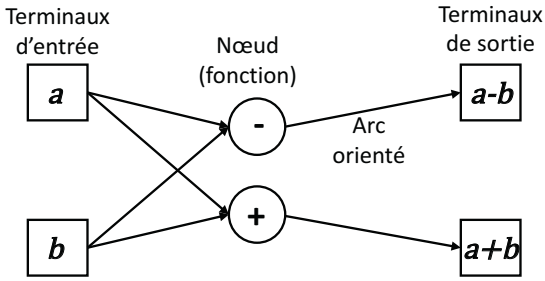

Figure 1.4 – Diagramme flux de données représentant le calcul de la somme et la différence de deux nombres *a* et *b*.

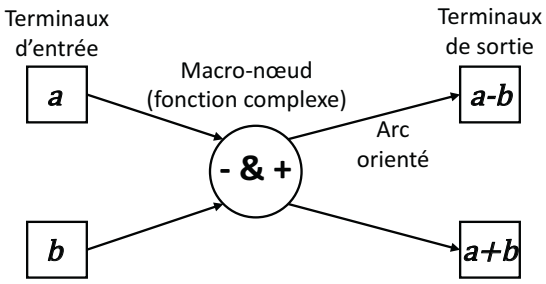

Figure 1.5 – Encapsulation du diagramme flux de données représentant le calcul de la somme et la différence de deux nombres *a* et *b* de la figure 1.4 avec une représentation du macro nœud par « – & + ».

ces deux types doit être voulu, analysé et validé… Nous étudierons cet aspect du **typage des données**.

- Un diagramme graphique flot de données peut rapidement conduire à des difficultés de lisibilité. Ainsi, avec l'exemple présenté, le croisement des arcs orientés sera très fortement déconseillé dans les règles d'édition proposées avant. Il sera rapidement conseillé de travailler avec l'encapsulation ou la création de macronœud correspondant à la notion de sous-programme (Fig. 1.5). Il est important de noter qu'il sera indispensable de créer une documentation précise de la fonction réalisée par ce macronœud.

Une programmation flot de données peut être analysée comme la propagation ou l'écoulement des données le long des arcs orientés entre des nœuds ou des terminaux et des nœuds. Aussi, afin d'améliorer la compréhension de l'exécution de ces diagrammes flots de données, il est possible de représenter les données transitant sur les arcs par des jetons ou marqueurs. Ce symbolisme de propagation de données par mouvement d'un jeton est utilisé dans un outil de mise au point de l'environnement LabVIEW, appelé mode animation que nous étudierons dans la suite de cet ouvrage. La production et la consommation des jetons dans le diagramme flux de données sont en général régies par les règles d'évolution suivantes :

- lors de l'initialisation, chaque terminal d'entrée produit un jeton sur son arc de sortie ;
- lorsqu'un nœud possède un jeton sur chacun de ses arcs d'entrée, alors le nœud peut être exécuté immédiatement : chaque jeton d'entrée est alors consommé et un jeton est produit sur chacun des arcs de sortie du nœud ;
- en corollaire de la règle précédente, il est important de souligner qu'un nœud qui n'a pas sur toutes ses entrées une donnée disponible (un jeton non consommé) ne peut pas s'exécuter. Ce cas peut se produire par exemple si une des entrées d'un nœud n'est pas connectée (édition du programme en cours ou oubli du programmeur). En effet un nœud ne peut pas faire une hypothèse de valeur par défaut de cette entrée non reliée à un arc.

À partir de l'exemple du diagramme de la figure 1.4, l'exécution d'un diagramme flot de données par la propagation de jetons est présentée sur la figure 1.6 en six étapes correspondant au passage des jetons présents dans les terminaux d'entrée (étape initiale) jusqu'à l'arrivée des jetons produits présents dans les terminaux de sortie. Trois remarques importantes peuvent être faites sur cet exemple :

- la duplication des jetons sur la base des arcs orientés qui fournissent une donnée aux deux nœuds de calcul ;
- les étapes 2 et 3 sont en réalité confondues dans le sens où la duplication des données est un effet immédiat ;
- l'exécution en parallèle des deux nœuds de calcul.

3 Le parallélisme

Pour bien comprendre le principe analysé, il est nécessaire de distinguer le parallélisme de conception et le parallélisme d'exécution à propos d'un programme :

- le parallélisme de conception est l'écriture ou le dessin d'un programme permettant à des fonctions de s'exécuter en même temps. Pour la programmation textuelle, elle fait appel à des commandes de contrôle spécifique, mais pour la programmation graphique, comme nous l'avons vu, cela est inhérent au mode d'édition du programme, sachant que lorsque deux nœuds possèdent leurs entrées valides (données nouvelles disponibles), ils peuvent s'exécuter en « même temps » au sens de la conception ;
- le parallélisme d'exécution réel est la capacité d'exécuter en même temps des fonctions différentes nécessitant d'avoir un ordinateur multiprocesseur ou multi-cœur. Une fonction de répartition des fonctions à exécuter est chargée de répartir à un instant donné ces fonctions sur les différents processeurs avec leur contexte. Si l'ordinateur possède un seul processeur, il procède à l'exécution des tâches ou de parties de fonctions les unes après les autres. Ainsi, nous avons l'entrelacement temporel des exécutions des fonctions.

Nous nous intéresserons ici uniquement au parallélisme de conception indépendant de la machine d'exécution de type monoprocesseur ou multiprocesseur.

Nous avons déjà mis en évidence cette exécution en parallèle de deux nœuds dans un même diagramme comme l'étape 4 visualisée sur la figure 1.6 de l'exemple de la figure 1.4.

Mais d'une manière générale, il faudra privilégier une programmation de conception de type parallèle qui peut apporter une efficacité d'exécution. Ainsi l'exemple de la figure 1.7 montre un programme qui peut intégrer quatre ou cinq étapes selon la programmation réalisée, même si le nombre de nœuds (ou fonctions) utilisés est le même (trois fonctions « multiplier »). Cet exemple très simple permet de mettre en évidence ce parallélisme inhérent à la programmation graphique (mode d'écriture à

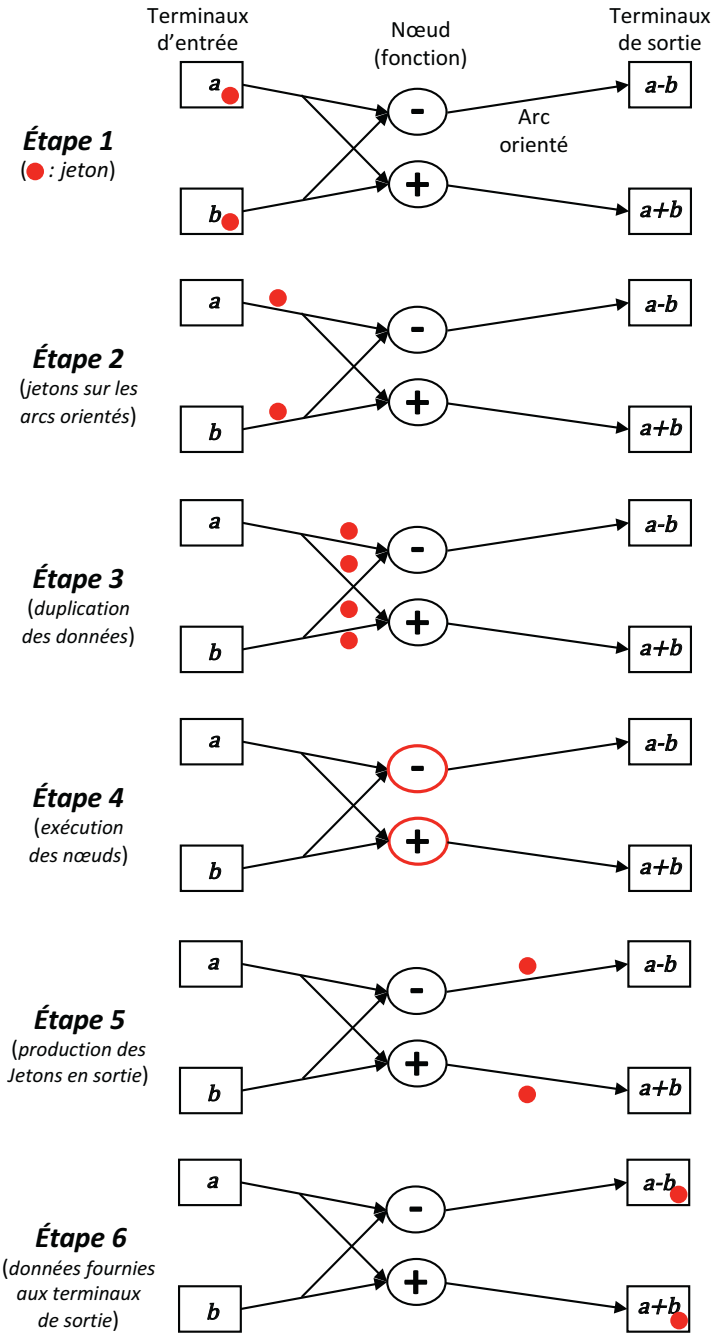

Figure 1.6 – Visualisation de l'exécution d'un diagramme flot de données par la propagation de marqueurs ou jetons : exemple de l'exécution du diagramme de la figure 1.4.

deux dimensions). La durée réelle d'exécution de ces deux programmes sera identique sur une architecture monoprocesseur quoique sa forme symétrique pourrait conduire à une plus grande efficacité en termes de mise en mémoire cache des données (mémoire préparant les données pour le calcul) et donc de temps de calcul. Bien entendu, dans une architecture multiprocesseur, l'exécution sera de fait plus rapide.

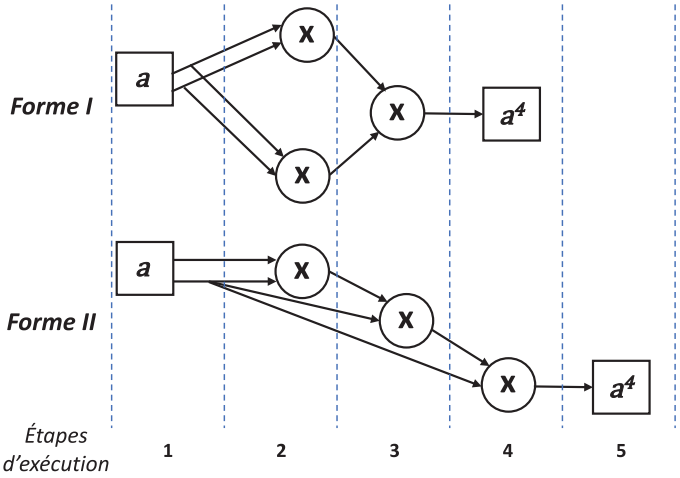

Figure 1.7 – Programmation de l'expression mathématique a^4 selon deux conceptions : la forme **I** est plus parallèle que la forme **II** puisqu'elle requière seulement quatre étapes d'exécution au lieu de cinq étapes pour la seconde forme avec le même nombre de fonction « **x** ».

Sur la base de cet exemple très simple, une autre remarque importante peut être faite. Si nous souhaitons encapsuler cette fonction mathématique « élévation d'un nombre à la puissance 4 », l'un ou l'autre des diagrammes peut être mis dans ce macro-nœud. L'utilisateur final, qui prendra ce sous-programme sans en connaître l'implémentation interne, peut élaborer un programme dont l'efficacité en termes d'exécution est différente selon les deux formes.

4 Le typage des données

Dans un langage graphique flot de données, le typage des données semble essentiel puisque la propagation de ces données pilote l'exécution du programme. Aussi cet aspect méritera toute notre attention dans la programmation avec l'environnement LabVIEW.

Tous les langages de programmation proposent plus ou moins des types de données qui peuvent être utilisées par les fonctions proposées par ce langage. Les types les plus standards sont les suivants :

- type booléen : valeur « vrai » (« true » ou « 1 ») ou faux (« false » ou « 0 ») ;

- type numérique entier, réel ou flottant, signé ou non signé avec une précision liée au codage utilisé allant de 8 bits à 64 bits voire plus pour les types numériques flottants (aussi dépendant de l'ordinateur) ;
- type chaîne de caractères qui intègre un ensemble d'éléments alphanumériques et de ponctuation :
  - exemples : {azerty} ou {nom@siteweb.com} ;
- type énuméré qui correspond à un ensemble fini de valeurs pour une donnée :
  - exemples : type « unité_électrique » {volt, ampère, ohm, watt, joule, henry, siemens, coulomb, farad} ou type 6_entier_premier {1,3,5,7,11,13} ;
- type composé qui est un ensemble d'éléments de même type, appelé tableau (possédant une ou plusieurs dimensions) sur lequel des opérations mathématiques peuvent être opérées, ou d'éléments de type différent, appelé ensemble ou cluster ou classe (chaque élément défini un champ) :
  - exemple : type tableau_entier_1D (w,x,y,z) :
    - « w,x,y,z » sont des données de type entier.
  - exemple : type cluster_inventaire_article (désignation, numéro, disponible, prix) :
    - « désignation » : donnée de type chaîne de caractères ;
    - « numéro » : donnée de type entier ;
    - « disponible » : donnée de type booléen ;
    - « prix » : donnée de type numérique flottant.

La déclaration du type d'une donnée variable ou constante peut être faite de façon implicite (au moment de la compilation, le type de la donnée est fixé étant donné son usage ou sa valeur initiale) ou de façon explicite (l'utilisateur définit pour l'ensemble des données le type associé à celle-ci).

Il est aussi possible de qualifier un langage de typage fort ou faible. Même si cette notion est assez floue, la distinction est liée soit à la richesse de ce typage de données d'un langage avec en particulier soit la possibilité de créer des types spécifiques (*i.e.* grandeurs électriques), soit à la capacité de vérification du typage lors par exemple de la compilation en exprimant les erreurs de type. Comme nous le verrons, le langage LabVIEW est un langage à typage fort pour les deux raisons énoncées ci-avant.

Concernant le typage de données, un dernier sujet très important est la conversion de type. Effectivement dans un programme, il est parfois nécessaire de modifier le type lors de l'exécution de certaines fonctions. L'exemple le plus simple à comprendre est celui d'une opération mathématique simple entre deux nombres de deux types différents : un nombre de type entier à additionner à un nombre de type numérique flottant. Dans ce cas, l'entier devra d'abord être converti en type numérique flottant avant de procéder à l'addition. Cette conversion n'est pas une opération complexe, mais il est très important d'avoir la connaissance de cette transformation qui peut dans certains cas être due à une erreur de conception de programme. Ainsi dans un programme il y aura un module de conversion

soit explicite (à mettre en place), soit implicite (une simple indication révèle cette transformation pour que l'exécution du programme se déroule correctement) (Fig. 1.8).

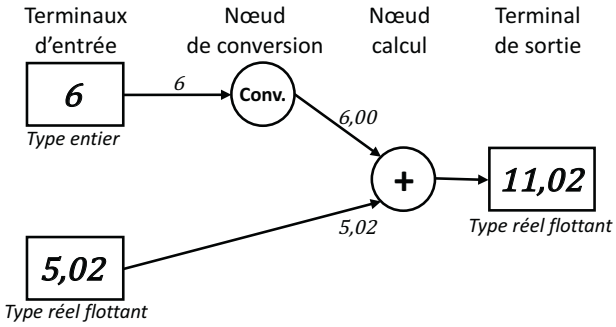

Figure 1.8 – Exemple visualisant la conversion de type de données avant l'exécution d'un nœud de calcul (addition).

5 Les extensions des diagrammes flot de données

Le modèle flux de données, présenté dans les paragraphes précédents, est trop simple pour être utilisé en tant que langage de programmation. En effet, cette programmation flux de données pure souffre de deux manques qui interdisent d'avoir une puissance d'expression suffisante pour traduire les algorithmes classiques :

- structures de programmation pour exprimer des séquencements, des choix ou des itérations ;
- mémorisation momentanée de données afin de pouvoir réutiliser ces données déjà calculées.

5.1 Les structures de programmation

Afin d'exprimer tous les algorithmes et de pouvoir respecter les règles d'un développement de qualité d'un programme, il est nécessaire de disposer d'éléments permettant de structurer la programmation. Soient les structures suivantes :

- structure d'une suite ou d'une séquence de programmes ;
- structure de répétition ;
- structure alternative ou de choix ou encore conditionnelle.

5.1.1 Structure de séquencement

Cela peut paraître paradoxal, mais étant donné le parallélisme implicite de la programmation graphique flux de données, une structure permettant d'imposer une séquentialité

dans les traitements semble indispensable. Cette structuration du programme doit permettre d'exécuter une suite d'actions sans utiliser le flot de données pour assurer de façon implicite la suite de ces actions. En effet si une donnée est nécessaire à l'exécution de ces différentes actions, la structure de séquencement n'est plus nécessaire (Fig. 1.9). Ainsi, dans cet exemple de l'enchaînement de trois actions (actions numérotées de 1 à 3), l'action 2 ne s'exécutera que lorsque l'action 1 aura été exécutée, et l'action 3 ne s'exécutera que lorsque l'action 2 aura été exécutée ; le séquencement a bien été respecté grâce au flot de données.

Une expression graphique complémentaire doit traduire la séquentialité obligatoire quel que soit le flot de données (Fig. 1.10). Dans le cas du programme A et contrairement à celui de la figure 1.9, les trois actions peuvent s'exécuter de manière parallèle et donc dans un ordre quelconque lié soit à la disponibilité des données en entrée, soit à la décision du répartiteur d'exécution de l'ordinateur. Aussi, le forçage de l'exécution séquentielle des trois actions du programme B n'est possible qu'en ajoutant une structure de contrôle « structure de séquencement ». Les numéros de cette structure imposent alors l'exécution complète de l'action de la structure « n » avant l'exécution de l'action de la structure « n + 1 ».

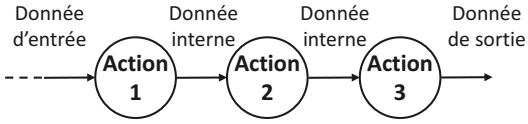

Figure 1.9 – Exemple montrant l'exécution séquentielle de trois actions simplement par le lien du flot de données.

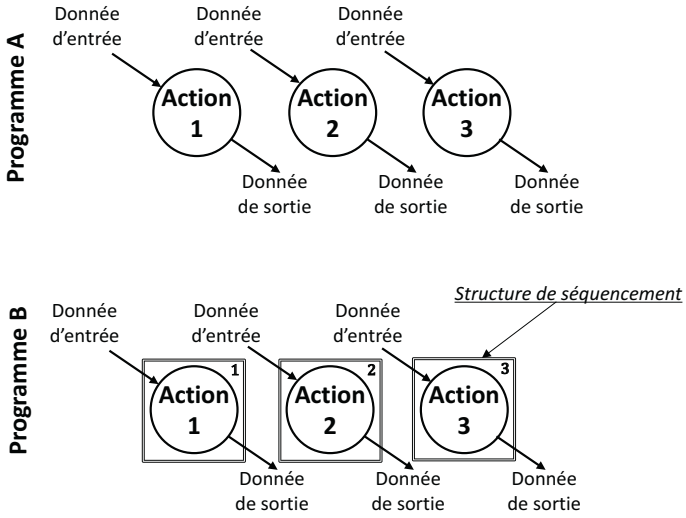

Figure 1.10 – Exemple montrant l'exécution séquentielle de trois actions grâce à l'ajout d'une structure de séquencement.

Il est important de souligner que le résultat de l'exécution des programmes A et B peut être très différent et ne pas répondre au cahier des charges dans le cas du programme A. Considérons l'application simplifiée concernant le contrôle du port de la ceinture de sécurité lorsque le conducteur est présent et la voiture est en marche. Il est nécessaire d'indiquer au conducteur par un voyant rouge sur le tableau de bord du véhicule avant que toute sonnerie intervienne, même si le véhicule est en marche afin que celui-ci puisse clairement comprendre l'origine de l'alarme sonore. Ainsi la figure 1.11 montre le programme constitué de deux séquences :

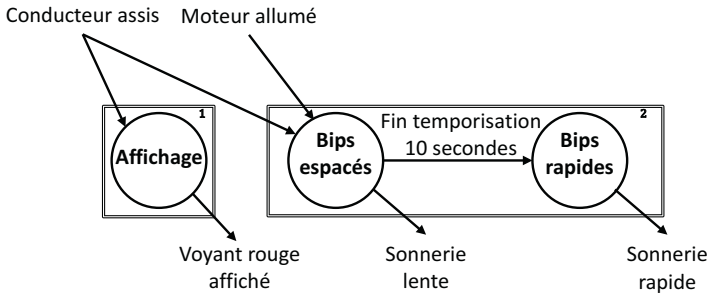

Figure 1.11 – Exemple d'application obligeant l'exécution séquentielle de différentes actions grâce à l'ajout d'une structure de séquencement.

- une correspondant à l'allumage du voyant indiquant la présence du conducteur et le fait que la ceinture n'est pas attachée ;
- l'autre indiquant cet oubli par une alarme sonore qui va augmenter de rythme après une temporisation de 10 secondes.

Si cette structuration du programme par la mise en place du séquencement n'était pas effectuée, il serait alors possible que la sonnerie se mette en marche avant l'allumage du voyant étant donné l'exécution en parallèle : cela est non conforme au cahier des charges.

5.1.2 Structure alternative

Cette structure de programmation qui s'ajoute à la programmation flot de données classiques peut être considérée comme la mise en place d'un aiguillage dans ce flot de données à deux ou plusieurs voies. Ainsi selon l'état d'un booléen (aiguillage à deux voies) ou la valeur d'une donnée quelconque (aiguillage à plusieurs voies), une branche du flot de données, qui suit l'analyse ou test de cette alternative, sera la seule exécutée (Fig. 1.12). Dans les langages textuels, ce type de structure est « Si_Alors_Sinon (*If_Then_Else*) ».

Il est important de souligner que, dans le cas d'une structure alternative à deux choix, c'est-à-dire pilotée par une donnée de type booléen, l'exécution sera donnée à l'un des deux flots de données qui suit.

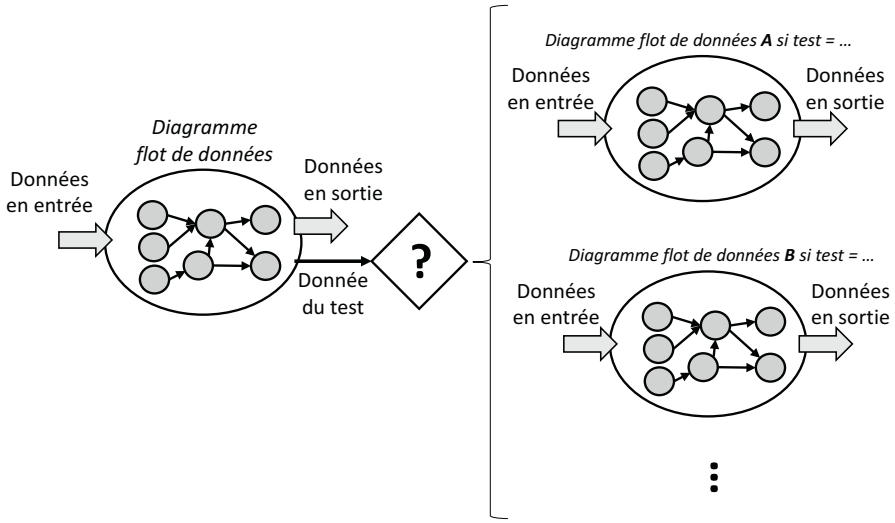

Figure 1.12 – La structure alternative symbolisée par le losange « **?** » permet d'aiguiller selon la réponse du test vers un des flots de données potentiels qui suit.

Dans le cas d'une structure alternative multi-choix, pilotée par une donnée de type entier ou par une liste énumérée de données (entier, liste de données…), il est nécessaire de prévoir un flot de données qui s'exécutera « par défaut » lorsque le résultat du test de la donnée n'a pas de flot de données attaché.

Considérons un exemple simple pour illustrer l'utilisation de la structure alternative à deux choix. L'application concerne la commande d'un éclairage à base de LED lorsque la luminosité extérieure passe en dessous d'un seuil fixé. La figure 1.13 présente cette application avec en entrée un capteur de cellule photovoltaïque fournissant une tension et un bouton permettant de fixer une valeur seuil de tension. La fonction centrale fait

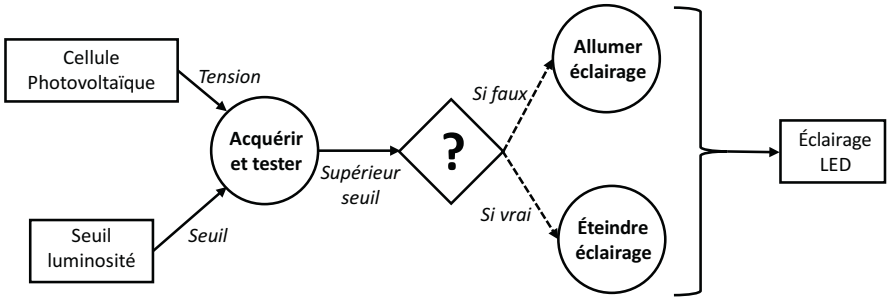

Figure 1.13 – Application « allumer un éclairage en fonction de la luminosité ambiante » nécessitant une structure alternative symbolisée par le losange « **?** ».