Manuel de référence de la génération de code de SynDEx

Méthodologie et outils pour une génération de code générique

Yves Sorel, Simon Nivault
INRIA Rocquencourt,
BP 105 - 78153 Le Chesnay Cedex, France
yves.sorel@inria.fr
www.syndex.org

Table des matières

Introduction

Ce document a pour but d'expliquer la manière dont on génère du code à partir des fichiers produits par SynDEx. Il décrira les méthodes, les outils ainsi que les bibliothèques utilisées. Cette documentation cible toute personne qui compte générer du code en utilisant SynDEx et qui veut en savoir plus sur le fonctionnement de la génération de code ou qui souhaite compléter l'outil existant pour générer du code plus en phase avec ses besoins. Ce document cible également les développeurs SynDEx car il est le pendant "génération de code" du manuel de référence de SynDEx.

1 Principes généraux

Dans le monde du développement informatique et principalement dans le cadre industriel, le cycle de développement classique est le cycle en V. Le V matérialise une descente Spécifications / Modélisation vers le développement proprement dit et une remontée où on teste et où on valide les développements. L'existence de cette remontée est dû au fait que le développement est fait par des humains et peut donc être incorrect. Il est nécessaire de faire concorder à postériori le développement avec les spécifications et la modélisation afin d'obtenir une application conforme et sans erreurs.

L'objectif de SynDEx est de transformer ce cycle en V en cycle en I par la génération de code. L'idée est d'assurer par construction la conformité entre les spécifications et le développement. On gagne ainsi en sureté et en rapidité de développement, le cycle en I étant plus court que le cycle en V.

SynDEx réalise une adéquation entre deux spécifications initiales, la spécification fonctionnelle, également appelée algorithme et la spécification matérielle, également appelée architecture. A partir de cette adéquation, SynDEx produit dans un premier temps ce que l'on appelle un macro-code générique distribué [1] et hétérogène [2]. Ce macro-code est spécifique vis a vis de l'algorithme et de l'architecure pour lequel il a été produit mais il est générique vis-à-vis des diverses méthodes que l'on aurait pu employer pour réaliser ce programme si celui-ci avait du s'écrire manuellement.

Dans un second temps, et c'est précisément l'objet de ce document, il existe toute une chaîne de compilation au sens général du terme [3] qui permet de transformer le macro-code généré par SynDEx vers un code spécifique choisi par l'utilisateur. En effet le choix du langage cible est laissé à l'utilisateur, l'objectif étant de pouvoir intégrer SynDEx à sa propre chaîne de développement.

La figure 1 donne une vue d'ensemble de cette transformation de fichiers. Les flèches pleines noires représentent l'action de produire un fichier. Les flèches rouges en pointillés signifient "est inclus par".

SynDEx produit un macro-code constitué d'un ensemble de fichiers m4 (extension `.m4`), d'où le terme de macro-code, m4 étant un processeur de macros. Cet ensemble de fichiers est composé d'un fichier m4 décrivant l'architecture (macro-code applicatif) nommé `appli.m4`, `appli` étant le nom

1. L'architecture peut être composée de plusieurs opérateurs de calculs.
2. Les différents opérateurs de calcul et de communication de l'architecture spécifiée peuvent être de type diférent.
3. Transformation d'un ou plusieurs fichier d'un langage vers un autre.

de l'application SynDEx, et d'un fichier m4 par processeur de cette architecture nommé `proc.m4`, `proc` étant le nom du processeur, décrivant le programme qui devra être exécuté par le processeur en question (macro-code exécutif). Ainsi, SynDEx génère deux types de macro-code, un macro-code applicatif et un macro-code exécutif. A partir d'un Makefile initial écrit par l'utilisateur, le macro processeur m4 va transformer le macro-code applicatif en un makefile auxilliaire généré spécifiquement pour décrire comment transformer et compiler les macro-codes exécutifs.

Cette transformation de fichiers utilise des fichiers m4 (extension `.m4x` et `.m4m`) où sont définis les macros contenues dans les macro-codes. Ces fichiers m4 sont appelés des noyaux d'exécutif. Ceux-ci peuvent se classer selon deux axes :

- un axe relatif au deux types de macro-code concernés par le noyau d'exécutif,
- un axe relatif à la généricité du noyau d'exécutif.

Selon le premier axe, on a :

- les noyaux d'exécutif décrivant les macros utilisées dans le macro-code applicatif (génération de makefile, extension `.m4m`),
- les noyaux d'exécutif décrivant les macros utilisées dans le macro-code exécutif (génération de code, extension `.m4x`).

Selon le deuxième axe, on a :

- un noyau d'exécutif générique qui est le point d'entrée des autres noyaux d'exécutif, appelé `syndex.m4x` et `syndex.m4m`,
- des noyaux d'exécutif spécifiques à l'architecture qui décrivent la manière dont on va générer le code et dans quel contexte il devra s'exécuter (description du langage, de l'OS, des bibliothèques systèmes, etc.), appelés `Archi.m4x` et `Archi.m4m`, `Archi` étant le nom de l'architecture employée,
- des noyaux d'exécutif spécifiques à l'application appelés `Appli.m4x` et `Appli.m4m`, `Appli` étant le nom de l'application SynDEx, contenant principalement la définition des macros correspondantes au code métier de l'application. Ces définitions sont en général écrites spécifiquement pour une application donnée mais ce travail peut en principe être factorisé par la création de bibliothèques applicatives de noyaux d'exécutifs (certaines sont distribuées avec SynDEx). Ces noyaux d'exécutifs peuvent également contenir des options générales d'exécution (nombre d'itérations de l'application, nom des machines sur lesquelles exécuter l'application dans le cas d'applications distribuées), des options spécifiques modifiant le comportement d'autres noyaux d'exécutif, ou encore d'autres options, la liste n'étant pas limitative.

Plan L'outil principal est donc le processeur de macro m4. La première section y est donc consacrée et présente m4 dans ses principes généraux ainsi que dans ses fonctionnalités les plus utilisées dans le générateur. Le lecteur cherchant des détails est invité à regarder plus avant dans le manuel de l'utilisateur disponible sur le web [4]. La section suivante parle de la transformation proprement dite. Celle-ci aura lieu en deux étapes, toutes deux basées sur m4. La première étape est la génération du système de construction (Makefiles), la seconde est la génération du code, ainsi que sa compilation à l'aide du système de construction préalablement généré.

4. http ://www.gnu.org/software/m4/manual/

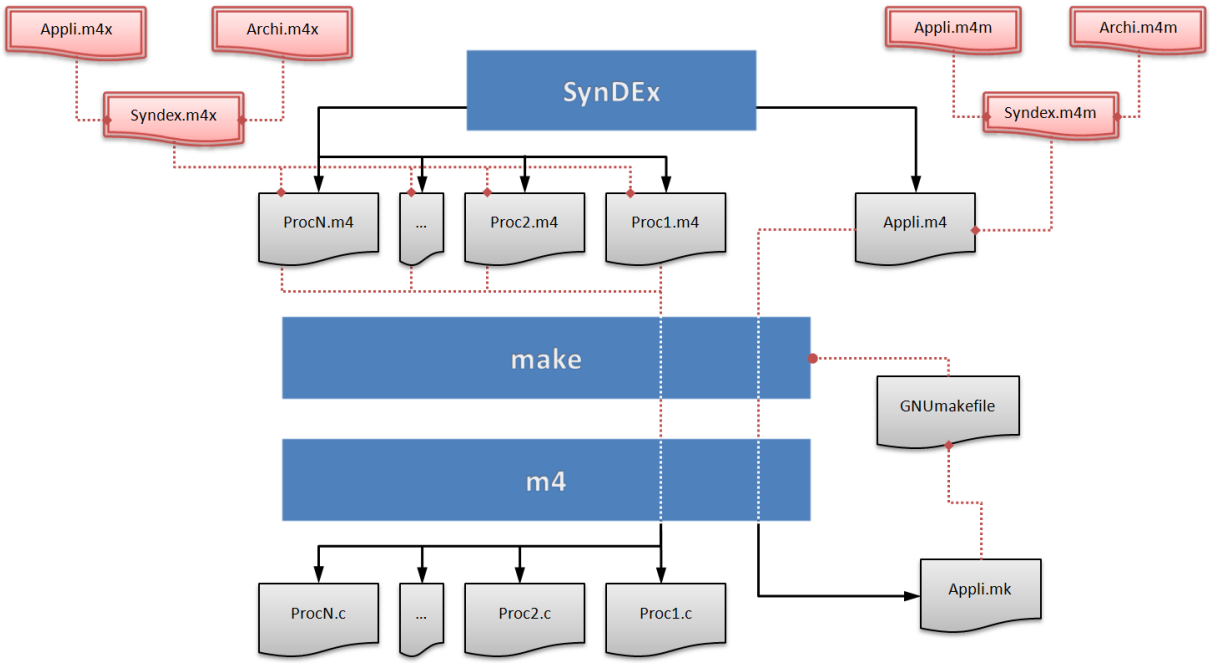

FIGURE 1 – Vue d'ensemble de la génération de code

2 Introduction à m4

Fondamentalement, m4 lit un texte en entrée et reproduit ce texte en sortie avec des modifications. Celles-ci sont définies dans le texte d'entrée lui-même sous forme de macros.

Invocation de m4 :

```
$ echo foo | m4
foo
$
```

Ici on appelle m4 en envoyant la chaine de caractère "foo" sur son entrée standard (ligne 1). m4 reçoit "foo" et l'écrit sur sa sortie standard (ligne 2).

m4 peut aller directement lire dans un fichier :

```
$ echo foo > test.m4
$ cat test.m4
foo
$ m4 test.m4
foo
$
```

Ici, on à préalablement mis la chaine de caractère "foo" dans le fichier `test.m4` puis on a invoqué m4 en lui donnant en paramètre le nom du fichier. Ainsi, au lieu de lire sur son entrée standard, m4 va lire le contenu du fichier en paramètre et écrire à nouveau "foo" sur sa sortie standard.

Puisque dans m4, tout repose sur les macros, rentrons directement dans le vif du sujet en expliquant ce que sont les macros.

2.1 Les Macros

Une macro m4 est une définition qui associe le nom de cette macro avec un texte de remplacement. Dès qu'une macro est définie, chaque fois que m4 passe sur le nom de cette macro, il substitue le nom de cette macro par son texte de remplacement.

Voyons un exemple :

Entrée :

```
1 foo
2 define(foo, bar)
3 foo
```

Sortie :

```
1 foo
2
3 bar
```

Sur la première ligne, aucune macro n'a été définie, par conséquent le mot `foo` est reproduit sur la sortie.

Sur la deuxième ligne, la macro `foo` est définie, le texte que constitue cette définition est supprimé, d'où une seconde ligne vide en sortie.

Enfin, sur la dernière ligne, puisque le mot `foo` correspond à une macro définie sur la ligne précédente, il est remplacé par le texte de remplacement correspondant, ici `bar`.

2.1.1 Les argument de macros

Les macros peuvent prendre des arguments pour bénéficier de subsitutions plus dynamiques.

Entrée :

```
1 define(OP1, var1)
2 define(OP2, var2)
3 define(add, $1 + $2)
4 add(OP1, OP2)
```

Sortie :

```
1
2
3
4 var1 + var2
```

Les arguments de la macro sont appelés par $1, $2, etc. Ces arguments sont donnés à la macro entre parenthèses et séparés par des virgules. Vous remarquerez sans doute que le mot `define` est lui même une macro, ce qui explique sa substitution par la chaîne vide.

Attention : Les parenthèses doivent suivrent immédiatement le nom de la macro quand celle-ci est appelée pour que les arguments soient pris en compte. Dans le cas contraire :

Entrée :

```
1 define(OP1, var1)
2 define(OP2, var2)
3 define(add, $1 + $2)
4 add (OP1, OP2)
```

Sortie :

```
1
2
3
4  +  (var1, var2)
```

2.1.2 Entre guillemets (‘’)

Les caractères `(backquote) et ’(quote) permettent d'encapsuler une partie d'un texte afin de le reproduire sans modifications, c'est-à-dire sans appeler les macros, ni tenir compte des caractères spéciaux qu'il contient.

Voici un exemple sans guillemets, le résultat n'est pas celui attendu :

Entrée :

```
1 define(add, $1 + $2)
2 add 1,2 and 3:
3 add(1,2 , 3)
```

Sortie :

```
1
2  +  1,2 and 3:
3 1 + 2
```

Voici comment obtenir le résultat attendu en utilisant les guillemets :

Entrée :

```
1 define(add, $1 + $2)
2 `add 1,2 and 3:'
3 add(`1,2' , 3)
```

Sortie :

```
1
2 add 1,2 and 3:
3 1,2 + 3
```

Néanmoins, les guillemets peuvent également introduire des comportements inattendus, comme celui-ci :

Entrée :

```
1 define(OP1, `1,2')
2 define(OP2, 3)
3 define(add, $1 + $2)
4 add(OP1, OP2)
```

Sortie :

```
1
2
3
4 1 + 2
```

Pour comprendre ce qui s'est passé, il est important de connaître la façon dont m4 lit l'entrée.

- Tout d'abord m4 lit mot à mot, par conséquent, dans l'exemple ci-dessus, il commence par reconnaître le mot `define`.
- Puisque `define` est une macro, et que le caractère suivant est une parenthèse ouvrante, m4 va décider de lire la suite pour récupérer les arguments de la macro, séparés par des virgules, jusqu'à ce qu'il trouve une parenthèse fermante.
  - M4 lit le mot suivant, ici `OP1`.
  - M4 lit une virgule, par conséquent il accepte `OP1` en tant que premier argument.
  - M4 lit ‘, il va directement chercher le ’ correspondant, puis accepte le contenu tel quel.
  - M4 lit la parenthèse fermante, il est prêt à appeler la macro `define` avec les argument `OP1` et `1,2`.

  Passons directement à la dernière ligne :
- M4 lit le mot `add`, puis la parenthèse ouvrante et va chercher les arguments...
  - M4 lit le mot OP1 et reconnaît le nom de macro. Comme il n'est pas suivi d'une parenthèse, la macro est aussitôt substituée par `1,2`. Or m4 traite à nouveau systématiquement le résultat d'une substitution de macro, par conséquent au lieu de passer à la virgule :
  - M4 lit le mot `1`, puis la virgule. Le premier paramètre est donc `1`.
  - Ainsi de suite, les paramètres suivant sont `2` et `3`.
  - La macro `add` est appelée avec `1` (resp. `2`) comme premier (resp. deuxième) argument. Le troisième argument est ignoré.

Pour éviter ces erreurs, il y a une règle simple à retenir : Toujours mettre une fois entre guillemets les arguments des appels de macros. Ainsi les arguments de macro sont traités une fois la macro substituée.

Entrée :

```
1 define(`OP1', `1,2')
2 define(`OP2', `3')
3 define(`add', `$1 + $2')
4 add(`OP1', `OP2')
```

Sortie :

```
1
2
3
4 1,2 + 3
```

Dernière chose à savoir à propos des appels de macros : les caractères d'espacement autour des arguments.

– Les espaces placés après l'argument sont comptés comme faisant partie de l'argument :

Entrée :

```
1 define(`add',`[$1] + [$2]')
2 add(OP1 ,OP2 )
```

Sortie :

```
1
2 [OP1 ] + [OP2 ]
```

– Les espaces placés avant l'argument sont ignorés :

Entrée :

```
1 define(`add',`[$1] + [$2]')
2 add( OP1 , OP2)
```

Sortie :

```
1
2 [OP1] + [OP2]
```

2.2 Les macros prédéfinies

Nous avons pu voir que m4 est basé sur du remplacement de texte à l'aide de macros. Ces macros peuvent être définies par l'utilisateur à l'aide de la macro `define`. Fort heuresement, m4 possède un grand nombre de macros prédéfinies qui permettent d'obtenir des comportements particuliers. Nous n'allons pas les aborder de façon exhaustive, mais seulement celles qui sont les plus utilisées dans SynDEx.

2.2.1 include

La macro `include` prend en paramètre un nom de fichier et insère son contenu afin de le traiter immédiatement. Typiquement, on aura souvent un fichier contenant un `include` en première ligne suivi des appels des macros définies dans le fichier inclu.

Fichier foo.m4

```
1 This file is foo.m4
2 define(`foo', `bar')
```

Entrée :

```
1 include(`foo.m4')
2 foo`'foo`'e
```

Sortie :

```
1 This file is foo.m4
2
3
4 barbare
```

Une chose à savoir : m4 cherche le fichier dans le dossier courant, ainsi que dans tous les dossiers déclarés dans la variable d'environnement `M4PATH`. Cette fonctionnalité est utilisée dans SynDEx car les fichiers définissant nos macros sont dans le répertoire d'installation de SynDEx, celui-ci est donc fourni dans la variable `M4PATH`.

2.2.2 dnl

Nous pouvons voir dans l'exemple précédent un effet génant. Les deux lignes vide (lignes 2 et 3) ne sont pas réellement désirées et pourtant elles ne sont pas là sans raisons. Ces deux lignes vides correspondent aux retours à la ligne après les macros `define` et `include`. Fort heureusement, il existe une autre macro pour "avaler" ce retour à la ligne : `dnl`. Cette macro se supprime elle-même ainsi que tous les caractères qui la séparent du prochain retour à la ligne, y compris celui-ci.

Entrée :

```
1 Texte pris en compte dnl Texte ign
2 Coucou
```

Sortie :

```
1 Texte pris en compte Coucou
```

Ou encore :

Fichier foo.m4

```
1 This file is foo.m4
2 define(`foo', `bar')dn
```

Entrée :

```
1 include(`foo.m4')dnl
2 foo`'foo`'e
```

Sortie :

```
1 This file is foo.m4
2 barbare
```

Dans le contexte de génération de code, il est important de bien maîtriser le retrait de ces retours à la ligne. Dans le cas contraire, on risque de se retrouver avec de nombreux retours à la ligne qui sont autant d'obstacles à la lisibilité du code généré.

2.2.3 divert

La macro `divert` est une des plus complexe de m4. En effet, elle crée des sortes de tampons parallèles autour de la sortie standard classique. En revanche, `divert` ne permet pas de rediriger la sortie standard de m4 au profit par exemple d'un fichier où de l'entrée d'un autre programme, cela n'est pas son rôle (c'est le rôle d'un shell). `divert` prend en paramètre un entier positif ou `-1` (0 par défaut) qui sélectionne le tampon dans lequel elle va écrire dorénavant, sachant que le tampon 0 est la sortie standard. Par conséquent, quand on appelle `divert` avec un entier différent de zéro, le texte qui devrait être écrit en sortie ne le sera plus. Exemple :

Entrée :

```
1 Ecriture en sortie standard.
2 divert(1)
3 Je suis incognito.
4 divert
5 Je suis revenu.
```

Sortie :

```
1 Ecriture en sortie standard.
2
3 Je suis revenu.
```

Comme on peut le constater, tout ce qui se trouve entre `divert(1)` et `divert` n'a pas été reproduit sur la sortie standard. La ligne vide correspond au retour à la ligne de la ligne 4 puisque celle-ci arrive après le `divert`. En général on ne souhaite pas ce retour à la ligne et on peut s'en débarrasser comme ceci :

Entrée :

```
1 Ecriture en sortie standard.
2 divert(-1)
3 Je suis incognito.
4 divert`'dnl
5 Je suis revenu.
```

Sortie :

```
1 Ecriture en sortie standard.
2 Je suis revenu.
```

En réalité ce qui est écrit dans un tampon parallèle n'est pas perdu. Au contraire il est stocké, prêt à être réutilisé plus tard grace à la macro `undivert`. Le tampon `-1` fait exception et fait office de poubelle, ainsi on peut y définir nos macros sans avoir à rajouter des `dnl` à chaque fin de ligne puisque ces lignes ne seront de toute façon pas écrites. Enfin et contrairement aux autres macros, m4 ne traite pas à nouveau le contenu d'un tampon quand celui-ci est rappelé par un `undivert`.

Entrée :

```
1 Ecriture en sortie standard.
2 divert(1)dnl
3 foo
4 divert(-1)
5 define(`foo',`bar')
6 divert`'dnl
7 Je suis revenu.
8 undivert(1)
```

Sortie :

```
1 Ecriture en sortie standard.
2 Je suis revenu.
3 foo
```

3 Génération de code

SynDEx génère deux types de fichier m4 :

- Un fichier [5] unique nommé `APPLI.m4` relatif à l'application. `APPLI` étant le nom de l'application
- Un fichier [6] par opérateur qui contient le macro-code de cet opérateur. Nous les nommerons par la suite `PROC1.m4`, `PROC2.m4`, ...`PROC1` et `PROC2` étant les noms des opérateurs de l'architecture.

Ces deux types de fichiers ont chacun leur rôle. Le fichier `APPLI.m4` sert à la génération du système de construction (makefiles) tandis que les fichiers `PROCN.m4` servent à la génération du code des exécutifs. Ces fichiers sont constitués d'appels de macros standards définies dans un fichier m4, lui même inclus en début de fichier. Ainsi, la première ligne du `APPLI.m4` est `include(syndex.m4m)` et la première ligne des `PROCN.m4` est `include(syndex.m4x)`. `syndex.m4m` et `syndex.m4x` sont les fichiers de macros génériques de la génération de code. Une de leurs principales attributions est d'inclure, selon le contenu des fichiers `APPLI.m4` et `PROCN.m4`, les fichiers de macros spécifiques à l'application. Les fichiers de macros relatifs à la génération du système de construction ont une extension en `.m4m` et ceux relatifs à la génération de code des exécutifs en `.m4x`. Le contenu de tous ces fichiers est abordé dans les sous-sections suivantes.

3.1 Génération du système de construction

Dans cette section, nous parlons beaucoup de Makefiles, car le système de construction en question est constitué de deux fichier de type Makefile, l'un étant inclus par l'autre. Pour autant nous n'expliquerons pas le concept de Makefile et nous considérerons que leur connaissance est acquise.

5. Le macro-code applicatif.
6. le macro-code exécutif

Tout d'abord il faut savoir que ces Makefiles ne sont qu'en partie générés. En effet, un fichier Makefile principal (ici GNUmakefile ce qui revient au même) minimaliste [7] doit être écrit à la main.

Ce Makefile principal peut être personnalisé selon vos besoins mais doit, pour être fonctionnel, respecter certaines conditions. Etant avant tout la passerelle vers le makefile généré, il doit contenir :

- la commande d'inclusion du makefile généré. Celui-ci se nomme par convention `APPLI.mk` (Le suffixe `mk` étant par convention le suffixe des makefiles non-autonomes, c'est-à-dire des makefiles qui n'ont pas été faits pour être utilisés seuls).
  Cette commande se retrouve ligne 18, annexe A ;
- une règle pour construire ce makefile. Pour cela on invoque `m4` sur le fichier `APPLI.m4` et on redirige la sortie dans le fichier `APPLI.mk`.
  On retrouve tout cela aux lignes 16 et 17 de l'annexe A ;
- une variable qui indique à `m4` l'emplacement des définitions de macros spécifiques à SynDEx. Cette variable se nomme `M4PATH` et doit contenir une liste de chemins, séparés par des " :", où se trouvent ces définitions de macros. (Voir la macro include 2.2.1).
  Les lignes de 5 à 8 dans l'annexe A servent à cela.

Enfin, le makefile généré l'est de manière à ce que toutes les opérations ayant pour but la génération de code et la compilation de ce code, soient déclarées comme étant des dépendances d'une cible "bidon" ("phony" en anglais) nommée `APPLI.all`. De même, les opérations ayant pour but l'exécution de l'application sont déclarées comme étant des dépendances de `APPLI.run`.

Afin de lancer ces commandes à partir des cibles `all` et `run` on rajoute les lignes 11 à 14 de l'annexe A.

Remarques sur l'exécution de l'application : Actuellement, dans plusieurs noyaux d'exécutif (bibliothèques m4), les macros et le code généré sont écrits de telle manière à ce que l'ensemble de l'application puisse être lancé à partir d'une seule commande et ce, même dans le cas d'applications distribuées. Ce mécanisme est appelé en interne "le loader". Actuellement ce loader est conçu uniquement de manière non modulaire dans le sens ou il est intégré au code exécutable de chaque opérateur. Ainsi l'ensemble des opérateurs de l'application forment un arbre dont chaque noeud-exécutable est censé être lancé par son parent, puis est chargé de lancer ses fils. Ce mécanisme a pour principal avantage le fait de pouvoir simplement exécuter une application distribuée à partir d'un unique poste de travail ce qui facilite le développement. En revanche il présente deux défauts majeurs :

- la capacité pour un exécutable d'en lancer un autre et ce sur une machine distante repose sur un service que les OS ne fournissent pas toujours simplement. Cela est relativement aisé sur un réseau d'ordinateur sous UNIX grâce à la commande `ssh` mais pose néanmoins quelques difficultés concernant la configuration de ces connexions ou lorsqu'une application est censée tourner sur des machines avec des configurations `ssh` différentes, ce qui diminue l'avantage de simplicité lorsque l'on change de système et de configuration. Enfin cela est encore plus complexe pour les systèmes qui ne proposent pas d'alternative efficace à `ssh` tels que Windows. Il est toujours possible de donner à un poste Windows un comportement proche d'UNIX avec des solutions tels que CygWin mais cette condition est trop intrusive vis-à-vis du système cible. Enfin, que faire pour une application distribuée sous UNIX et Windows ?
- l'implémentation actuelle du loader est également trop intrusive vis-à-vis des exécutables générés car l'exécutable renferme au même endroit les instructions spécifiques au "loader"

7. Un exemple de makefile minimaliste est donné à l'annexe A

et celles spécifiques à l'application. C'est pour cette raison qu'on ne peut pas se séparer du "loader" et choisir de lancer les différents exécutables manuellement. Ce défaut exacerbe le précédent car il empêche de faire abstraction du "loader" lorsque celui-ci devient trop difficile à mettre en place.

3.1.1 Génération du makefile APPLI.mk

Comme vu précédemment, le fichier `APPLI.mk` est généré par `m4` à partir du fichier `APPLI.m4` qui inclut `syndex.m4m`. Nous allons donc présenter les macros définies dans `syndex.m4m` et utilisées dans `APPLI.m4`.

3.1.1.1 Macro `architecture_` : Cette macro prend trois paramètres, dans l'ordre suivant :
- nom de l'application,
- copyright,
- date et heure de la création du fichier.

Elle génère la cible APPLI.run, APPLI étant substitué par le premier paramètre. La recette de cette cible est la concaténation de deux variables `prefix.run` et `APPLI.root`.

`prefix.run` contient par défaut les caractères `./` qui sont le début de la ligne de commande pour lancer un exécutable placé dans le dossier courant sous UNIX. Cette variable existe afin de supporter d'autres systèmes. Par exemple sous Windows/RTX, un exécutable est un fichier de type RTSS qui doit être lancé à l'aide du programme RTSSRun. Dans ce cas `prefix.run` doit être égal à `RTSSRun`, le noyau d'exécutif RTX.m4m est chargé de cette affectation.

3.1.1.2 Macro `processor_` : La macro `processor_` prend 2 + 2n paramètres. Les deux premiers sont respectivement le type et le nom d'un opérateur. Les paires de paramètres qui suivent sont respectivement le type de port de communication de cet opérateur ainsi que le nom de ce port.

Cette macro apparaît une fois pour chaque opérateur faisant partie de l'architecture pour laquelle on génère l'application. Ainsi, cette macro génère la partie du makefile qui se chargera de générer et de compiler le code associé à l'opérateur.

Dans la plupart des cas, nous avons un exécutable pour chaque opérateur, par conséquent, nous ajoutons ce fichier à la liste des dépendances de la cible `APPLI.ALL`.

Ensuite, afin de construire la ligne de commande, nous concaténons le nom de l'exécutable ainsi que l'adresse de l'opérateur (nom de machine, IP, etc.) à la variable `APPLI.root`. Ces informations sont en particulier utiles au loader (cf remarque page 11). En effet dans le cas où l'exécutable principal est censé lancer les autres, il obtient le nom de ces autres exécutables ainsi que l'emplacement pour les exécuter sur sa ligne de commande. Il faut noter également que ces morceaux de makefile étant écrits dans le même ordre que celui des différentes macros `processor_`, la macro de l'opérateur qui contient l'exécutable "root" (celui qui est chargé de lancer tous les autres) doit absolument être la première des macros `processor_`.

Enfin, une ou plusieurs cibles makefile doivent être écrites afin de générer l'exécutable à partir du fichier `PROC.m4`, où `PROC` est le nom de l'opérateur. En général cela se fait en plusieures étapes :
- Transformation du macro-code (`PROC.m4`) en code (par exemple `PROC.c`).
- Compilation du code en fichiers objets (`PROC.c` → `PROC.o`).
- Edition des liens du fichier objet (`PROC.o`, seul ou avec d'autres) en exécutable (`PROC.exe`).

3.1.1.3 Macro `connect_` et `endarchitecture_` : La macro `connect_` prend 2 + 2n paramètres. Les deux premiers paramètres sont respectivement le type et le nom d'un médium. Les paires de paramètres qui suivent sont respectivement le nom de l'opérateur connecté à ce médium ainsi que le nom du port par lequel l'opérateur se connecte au médium. Ces macros sont substituées par des chaînes vides et sont donc pour le moment inutiles.

3.2 Génération du code des exécutifs

A l'instar du fichier `APPLI.m4`, chaque fichier source de chaque opérateur est généré par `m4` à partir de son fichier `PROC.m4` correspondant, qui inclut `syndex.m4x`. Nous allons donc présenter les macros définies dans `syndex.m4x` et utilisées dans `PROC.m4` dans leur ordre d'arrivée.

Auparavant, en plus du fichier `syndex.m4x`, `PROC.m4` peut inclure également le cas échéant d'autres fichiers d'extension `.m4x` correspondant aux bibliothèques standard de SynDEx . Ces bibliothèques font partie du noyau applicatif du générateur.

3.2.1 Macros génériques

3.2.1.1 Macro `processor_` et `endprocessor_` Ces macros délimitent l'exécutif. Toutes les autres macros doivent se situer entre ces deux principales macros. La macro `processor_` prend 5 paramètres. Ceux-ci sont :

- le type du processeur de l'exécutif,
- le nom de ce processeur,
- le nom de l'application,
- la version de syndex qui a généré cet exécutif ainsi qu'un copyright,
- la date et l'heure de la génération.

L'appel à `processor_` provoque :

- La définition de deux nouvelles macros :
  - `processorType_` = le type du processeur,
  - `processorName_` = le nom du processeur,
- L'inclusion obligatoire[8] du fichier `TYPE.m4x` où `TYPE` est le premier paramètre de la macro,
- L'inclusion facultative des fichiers suivants :
  - `APPLI.m4x` où APPLI est le troisième paramètre de la macro,
  - `PROC.m4x` où PROC est le deuxième paramètre de la macro.

Le fichier `syndex.m4x` ne contient que des macros génériques. Toutes les macros définissant le code dépendant de l'architecture sont situées dans les fichiers `.m4x` inclus ici. Nous verrons ensuite (principalement pour `TYPE.m4x`) les macros que ces fichiers devront définir.

3.2.1.2 Macro `semaphores_` Cette macro sert à déclarer l'ensemble des sémaphores qui sont utilisés dans l'exécutif pour synchroniser le thread de calcul avec les différents threads de communication.

Le fichier `syndex.m4x` ne définit pas cette macro. Cette responsabilité est déléguée au noyau d'exécutif dépendant de l'architecture.

8. L'absence de ce fichier provoque une erreur.

Cette macro prend en paramètre :
- Pour chaque nom de thread de communication X, `Semaphore_Thread_X`. Ces paramètres sont historiques et n'ont jusqu'à preuve du contraire aucune utilité ;
- le nom de chaque sémaphore. Celui-ci est composé de la concaténation séparé par des underscores (`_`) des éléments suivants dans l'ordre inverse :
  - `empty` ou `full` selon si le sémaphore est censé indiqué si le tampon dont il protège l'accès est vide ou plein,
  - le nom du port de communication par lequel la donnée dont il protège l'accès transitera,
  - le nom de l'opérateur,
  - le nom complet du tampon dont il protège l'accès (voir ci-dessous le nom du tampon).

3.2.1.3 Macro `shared_` et `endshared_` Ces macros servent à délimiter les déclarations des sémaphores et des données qui seront utilisés dans le cadre d'une communication par mémoire partagée.

A l'origine cela permettait de distinguer les sémaphores et tampons des échanges de données intra-opérateurs de ceux inter-opérateurs. Aujourd'hui les macros pour les échanges inter-opérateurs (c'est-à-dire pour les communications par mémoire partagée) ont été renommées en `semaphoresR_` et `allocR_`

Le fichier `syndex.m4x` ne définit pas cette macro. Cette responsabilité est déléguée au noyau d'exécutif dépendant de l'architecture.

3.2.1.4 Macro `alloc_` Cette macro sert à déclarer un tampon de donnée. La règle est qu'un tampon est créé pour chaque production de donnée dans le graphe d'algorithme, ainsi à chaque port de sortie d'une fonction atomique du graphe d'architecture est associé une mémoire tampon.

Le nom de cette mémoire tampon est composé de la concaténation, séparée par des underscores (`_`), des éléments suivants dans l'ordre inverse :
- le nom du port,
- le nom de la définition de la fonction atomique,
- les noms des définitions des fonctions hiérarchiques dont fait partie l'instance de la fonction atomique,
- un underscore (`_`).

Cette macros prend trois paramètres :
- le type de la donnée,
- le nom du tampon,
- la cardinalité (le nombre de valeurs du type donné, autrement dit la taille du vecteur).

Le fichier `syndex.m4x` définit cette macro générique de manière à ce qu'elle appelle une macro plus spécifique pour effectuer l'allocation de cette mémoire :
- si une macro `NAME_alloc_` existe (ou `NAME` est le nom du tampon), elle est appelée,
- sinon, si une macro `TYPE_alloc_` existe (ou `TYPE` est le type de la donnée), elle est appelée,
- sinon, la macro `basicAlloc_` est appelée.

Dans tous les cas, la macro appelée reçoit le nom du tampon en paramètre.

Le fichier `syndex.m4x` ne définit pas `basicAlloc_`. Cette responsabilité est déléguée au noyau d'exécutif dépendant de l'architecture.

3.2.1.5 Macro `semaphoresR_` Voir `semaphore_`

3.2.1.6 Macro `allocR_` Voir `alloc_`

3.2.1.7 Macro `alias_` Cette macro permet de faire un alias sur un tampon mémoire. Aucune nouvelle mémoire n'est allouée mais un nouveau nom de tampon fera référence à un tampon existant.

Le fichier `syndex.m4x` définit cette macro générique de manière à ce qu'elle fasse une vérification de ses paramètres avant d'appeler une macro plus spécifique, `basicAlias_`, avec tous ses paramètres.

Ces paramètres sont :
- le nom de l'alias,
- le nom du tampon référencé,
- le décalage par rapport au tampon référencé,
- la taille de la donnée référencée.

Le fichier `syndex.m4x` ne définit pas `basicAlias_`. Cette responsabilité est déléguée au noyau d'exécutif dépendant de l'architecture.

3.2.1.8 Macro `thread_` et `endthread_` Ces macros delimitent et définissent un thread de communication.

La macro `thread_` reçoit les paramètres suivants :
- le type du médium par lequel ce thread va communiquer,
- le nom du port par lequel l'opérateur se connecte au médium,
- les noms des opérateurs qui sont connectés à ce médium, un nom par paramètre.

L'appel à cette macro a pour effet :
- la définition de nouvelles macros temporaires :
  - `mediaType_` contient le type du médium (ici le premier argument),
  - `mediaName_` contient le nom du port (ici le second argument),
  - `threadArgs_` contient l'ensemble des arguments donnés à la macro `thread_`,
- l'inclusion d'un noyau d'exécutif spécifique au type de médium : le nom du fichier doit être `TYPE.m4x`, où `TYPE` est le type du médium. Cette inclusion est facultative dans le sens où le fichier peut ne pas être présent ;
- l'inclusion d'un noyau d'exécutif spécifique au port de l'opérateur : le nom du fichier doit être `TYPE.m4x`, où `TYPE` est le nom du port de l'opérateur. Cette inclusion est également facultative.

Enfin cet appel de macro générique provoque l'appel d'autres macros spécifiques :
- `TYPE_shared_` où TYPE est le type du médium. La macro est appelée avec tous les paramètres de `thread_` sauf le premier (le type du médium) ;
- `basicThread_`, la macro est appelée avec le deuxième paramètre de `thread_` (le nom du port),
- `TYPE_ini_`, de la même manière que `TYPE_shared_`.

La macro `basicThread_` est appelée systématiquement, contrairement aux deux autres qui ne sont appelées que si elles existent.

La macro de fin `endthread_` à un comportement symétrique à `thread_` :
- appel de macro `TYPE_end_` de la même manière que `TYPE_shared_`,
- appel de macro `basicEndthread_` de la même manière que `basicThread_`,
- suppression des définitions `mediaType_`, `mediaName_` et `threadArgs_`.

3.2.1.9 Macro `loadDnto_` et `loadFrom_` Chaque section de thread d'un exécutif contient en première ligne l'une de ces deux macros. Celles-ci concernent entre autre le loader (cf remarque 3.1) et sont vouées à être substituées par un code qui sera principalement en charge de deux choses :
- le lancement à distance, ou non, du ou des autres opérateurs dont l'opérateur courant a la charge de lancer,
- l'établissement de la connexion entre l'opérateur courant et ceux avec qui il est censé communiquer.

3.2.1.9.1 Remarque sur une éventuelle disparition de ces dernières macros : La dissymétrie entre les deux macros est utile à ces deux objectifs. Le premier la requiert évidemment pour savoir lequel d'entre les deux opérateurs doit lancer l'autre. Le second la requiert aussi pour correspondre à la dissymétrie de la plupart des protocoles réseaux qui fonctionnent en client/serveur [9]. Or seul le premier objectif correspond au loader. Si à l'avenir on parvient à décorréler le loader de l'exécutif, cela n'entraînera pas la disparition de ces macros car leur dissymétrie est essentielle pour l'établissement de la connexion entre les opérateurs.

La macro `loadDnto_` est présente pour les opérateurs qui ont la charge de lancer un autre opérateur et qui tiendront le rôle de serveur pour l'établissement de la connexion avec l'opérateur qu'il a dû lancer. Son premier paramètre est vide et les suivants sont la liste des opérateurs qui devront être lancés automatiquement par cet opérateur.

La macro `loadFrom_` est présente pour les opérateurs qui ne sont pas lancés manuellement (mais par un autre opérateur qui en est chargé) et qui est censé se connecter à un opérateur qui attend sa connexion. Son unique paramètre contient l'opérateur qui doit le lancer et auquel il doit se connecter.

La macro `loadFrom_` ne peut apparaître au plus qu'une seule fois, en effet :
- l'opérateur ne doit être lancé que par un seul autre opérateur,
- la connexion qui s'établira aura été initialisée par un seul opérateur.

En revanche, la macro `loadDnto_` peut apparaître plusieurs fois car il est possible qu'un opérateur doivent lancer plusieurs autres opérateurs et que plusieurs de ces opérateurs se connectent au médium initialisé par ce premier opérateur.

Enfin rien n'empêche à ce que ces deux macros puissent apparaître ensemble, en commençant par la macro `loadFrom_`.

Ces deux macros sont génériques et définies dans `syndex.m4x`. Elles vérifient qu'elles sont bien appelées au sein d'une section `thread_` puis appelent les macros spécifiques `TYPE_loadFrom_` ou `TYPE_loadDnto_` selon le cas, `TYPE` étant le nom du type de médium utilisé [10]. Ces macros spécifiques sont appelées avec les mêmes paramètres que les macros génériques.

9. Dans le cas d'une mémoire partagée, un opérateur crée cette mémoire, les autres ne font qu'y acceder. Dans le cas d'une connection TCP, un opérateur attend et écoute le port sur lequel les autres opérateur vont se connecter
10. il correspond au premier paramètre de la macro `thread_`

3.2.1.10 Macro `saveFrom_` et `saveUpto_` Ces macros sont les symétriques des précédentes et apparaîssent en dernière ligne des sections `thread_`. Pour chaque `loadDnto_` il y a un `saveFrom_` avec les mêmes paramètres et s'il y a un `loadFrom_`, il y a un `saveUpto_` également avec les mêmes paramètres.

De la même manière, ces deux macros génériques vérifient qu'elles sont bien appelées au sein d'une section `thread_` puis appellent les macros spécifiques `TYPE_saveFrom_` ou `TYPE_saveUpto_` selon le cas.

3.2.1.11 Macro `loop_` et `endloop_` Ces deux macros apparaîssent au sein des sections `thread_` et `main_` et servent à délimiter le cycle d'instructions des threads (de communication et de calcul) qui doivent être exécutées.

`loop_` définit le MGC (voir annexe B) comme LOOP et appelle la macro spécifique `basicLoop_`. `endloop_` définit le MGC comme END et appelle la macro spécifique `basicEndloop_`

3.2.1.12 Macro `trans_` et `endtrans_` Ces macros sont absolument similaires à `loop_` et `endloop_` à ceci prêt qu'elles délimitent les phases transitoires quand elles existent au lieu des phases permanentes. Ces macros n'apparaissent donc que lorsque l'on génère du code relatif à une application en multipériode. Ces deux macros n'appellent pas de macros spécifiques et se cantonnent à définir le MGC (LOOP pour `trans_` et END pour `endtrans_`).

3.2.1.13 Macro `main_` et `endmain_` A l'instar des macros `thread_` et `endthread_`, ces macros servent à délimiter le thead principal ou encore thread de calcul.

`main_` définit le MGC comme INIT et appelle la macro spécifique `basicMain_`. `endmain_` supprime la définition du MGC et appelle la macro spécifique `basicEndmain_`

3.2.1.14 Macro `spawn_thread_` et `wait_endthread_` Les différents threads de communication étant gérés par le thread principal, celui-ci doit avoir des instructions pour créer ces threads et d'autres pour attendre leur terminaison. Ces macros sont là pour générer ces instructions.

Ces macros ne sont pas définies dans `syndex.m4x` et doivent donc être définies dans les noyaux d'exécutif dépendant de l'architecture.

La macro `spawn_thread_` prend en paramètre le nom du thread. La macro `wait_endthread_` prend en paramètre le nom du sémaphore du thread (voir la macro `semaphores_`)

3.2.1.15 Macro `Suc1_`, `Suc0_`, `Pre1_`, `Pre0_` Ces macros sont les macros de synchronisation intra-opérateur. Elles prennent en paramètre le sémaphore sur lequel elles agissent. Les macros commençant par `Suc` relachent leur sémaphore ; celles commençant par `Pre` les réservent. Les 0 et les 1 accolés à ces macros sont des précisions relatives aux priorités d'arbitrage des séquenceurs lorsque ceux-ci traitent les threads de calcul et les threads de communication de manière dissymétriques. [référence ?] Sur les systèmes gérant le multitâche comme Unix ou Windows, on peut ignorer ce détail et faire la même substitution pour les deux variantes de `Suc` et de `Pre`.

3.2.1.16 Macro `SucR1_et` `PreR1_` Ces macros fonctionnent de la même manière que les précédentes et sont dédiées aux synchronisations inter-opérateur dans le cadre de communications par mémoire partagée.

3.2.1.17 Macro `send_`, `recv_` et `sync_` Ces macros représentent les fonctions de communications et apparaissent au sein du ou des section thread. Elles prennent en premier paramètre le nom du tampon à envoyer ou à recevoir. Les deux paramètres suivants sont le type et le nom de l'opérateur qui produit la donnée (ce qui est redondant, elles semblent donc inutile jusqu'à preuve du contraire). Un dernier paramètre optionel indique l'opérateur avec lequel cette communication s'établit (ce qui semble également redondant, la macro `thread_` apportant déjà cette information). Dans le cas d'une communication multipoint, la macro `sync_` signifie que lors de l'envoi d'une donnée par un autre opérateur, l'opérateur courant n'est pas le destinataire de cette donnée et peut donc l'ignorer. Ainsi, au lieu du nom du tampon en premier paramètre, la macro `sync_` reçoit le type de la donnée ainsi que sa taille.

Ces deux macros sont génériques et définies dans `syndex.m4x`. Elles vérifient qu'elles sont bien appelées au sein d'une section `thread_` puis appelent les macros spécifiques `TYPE_send_`, `TYPE_recv_` ou `TYPE_sync_` selon le cas, `TYPE` étant le nom du type de médium utilisé [11]. Ces macros spécifiques sont appelées avec les mêmes paramètres que les macros génériques.

3.2.1.18 Macro `write_` et `read_` Ces macros apparaissent à la place des `send_` et `recv_` dans le cas des communications par mémoire partagée. Par conséquent elles reçoivent en paramètre, en plus du nom du tampon local, le nom du tampon de mémoire partagée dans lequel la donnée va transiter.

11. il correspond au premier paramètre de la macro `thread_`

Annexes

A Makefile minimaliste

```
A = control
M4 = m4

# these path have to be modified by user
SDX_MACRO_PATH=/path/to/syndex/macros
export Algo_Macros_Path = $(SDX_MACRO_PATH)/algo_libraries
export Archi_Macros_Path = $(SDX_MACRO_PATH)/archi_libraries
export M4PATH = $(Algo_Macros_Path):$(Archi_Macros_Path):$(S2s_Files_Path)
VPATH = $(M4PATH)

.PHONY: all $(A).all run $(A).run
all : $(A).mk $(A).all

run : all $(A).run

$(A).mk : $(A).m4
        $(M4) $< >$@
include $(A).mk
```

B MGC : Macro Generation Context

Le `MGC` est une variable servant à donner le contexte dans lequel une macro applicative est appelée. En effet ces macros applicatives peuvent être appelées pendant la phase d'initialisation, la phase transitoire ou permanente et après la phase permanente. Pour permettre à ces macros d'être substituées différemment selon la phase pendant laquelle elles sont appelées, elles peuvent tester le contenu de la variable `MGC`.

`MGC` contient :

- `INIT` à l'initialisation,
- `LOOP` pendant les phases transitoire et permanente,
- `END` après la phase permanente.

Ainsi, voici la manière générale d'écrire une macro applicative :

Patron de macro applicative

```
define(`ma_fonction', `ifelse(
MGC, `INIT', `// Initialisation',
MGC, `LOOP', `// Phase permanente',
MGC, `END', `// Finalisation')')
```