Présentation du projet Distributed SL (dSL)

Un environnement de programmation orienté–objet permettant la distribution semi–automatique et optimale des tâches de supervision et de contrôle d'automates

F. Geurts[1], F. Macq[2], T. Massart[1], A. Piron[1] et G. Vanstraelen[2]

[1] Département d'Informatique, Université Libre de Bruxelles
[2] Macq Electronique

1. INTRODUCTION

Ce projet est en collaboration entre le Département d'informatique, Faculté des Sciences, de l'ULB et Macq Electronique, une société Bruxelloise spécialisée en conception de systèmes de contrôle industriel à l'aide d'automates programmables..

Le projet vise à concevoir et développer un environnement de programmation permettant la conception intégrée de systèmes de contrôle automatisé d'équipements industriels. Les systèmes de contrôle sont distribués et principalement formés de serveurs et d'automates programmables. Les serveurs effectuent des tâches de configuration, de traitement « temps réel » et d'archivage de données. Ces serveurs sont en communication d'une part avec les utilisateurs par l'intermédiaire d'interfaces graphiques et d'autre part avec les automates contrôlant des équipements locaux.

En fonction de contraintes, l'environnement de développement devra établir, de façon semi–automatique et optimale, la répartition des tâches de contrôle et de supervision des automates industriels et des serveurs. L'environnement de développement devra alors compiler les différents codes requis et les transférer vers les serveurs et les automates. Pour des raisons d'efficacité et de sécurité, il est en effet utile d'affecter certaines tâches à un programme superviseur centralisé et d'autres aux programmes résidant dans les automates pilotant les maillons plus directement liés aux processus industriels à contrôler

L'aspect semi–automatique de l'outil à concevoir doit aider le programmeur à décider des parties d'un développement devant être affectées aux automates ou au programme de supervision sur seule base du code écrit. L'optimisation doit tenir compte de critères suivants :

- l'efficacité (temps de réaction et partitions de charge) ;
- la sécurité du processus industriel ;

- les règles données par le programmeur ;
- les possibilités de changer le programme en cours sans interruption du contrôle du processus.

Ces critères sont directement liés à la rapidité et à la fiabilité des communications entre automates et superviseur et des traitements devant être effectués. L'environnement de conception devra donc permettre au concepteur de systèmes de contrôle de se concentrer sur la logique de ce dernier en faisant abstraction, dans un premier temps, des aspects distribués. Ceux−ci feront l'objet d'une deuxième phase qui sera semi−automatisée : le concepteur donnant les contraintes de proximité à respecter.

Pour atteindre l'objectif, quatre phases doivent être réalisées :

- conception et développement du langage "distributed−SL" : un langage de programmation orienté−objet à distribution transparente ;
- conception d'un outil d'analyse de code destiné à l'identification de « modules » ou objets pouvant être distribués ;
- réalisation d'un outil de configuration et d'un compilateur traduisant les programmes écrits en distributed−SL en code distribué entre le ou les superviseurs et les automates ;
- réalisation d'un outil de diagnose pour assister le programmeur dans la mise au point de son application.

2. SITUATION DU PROBLÈME

2.1. « SL », un langage de supervision

SL est un langage de programmation dérivé du langage ST (Structured Text) décrit dans la norme IEC−61131−3. Cette norme propose des langages standardisés pour la programmation d'automates. La syntaxe de ST ressemble à Pascal ou ADA.

Pour concevoir le système de supervision OBviews, Macq Electronique a étendu le langage ST aux concepts de programmation « orienté−objet » et « événementielle ». La combinaison de ces deux concepts se retrouve dans la règle suivante du programme SL :

```
WHEN IN pump self.temperature > max_temperature THEN
        Self<−stop()
END_WHEN
```

Nous pouvons aussi écrire des invariants comme suit :

```
CONTROL alarm WITH temperature > 80 OR sensor_defect
```

Remarquons que le concept événementiel est à distinguer d'une simple machine à états finis. Il est aussi plus complexe qu'un moniteur. La règle WHEN n'est pas scannée cycliquement pour des raisons de performance. Au contraire, chaque fois qu'une variable change, toutes les règles dans lesquelles cette variable intervient sont réévaluées. A chaque changement de variable, les invariants sont aussi adaptés.

Dans OBviews , nous trouvons aussi une notion de valeur INCONNU. Cette valeur se propage dans les calculs. Par exemple :

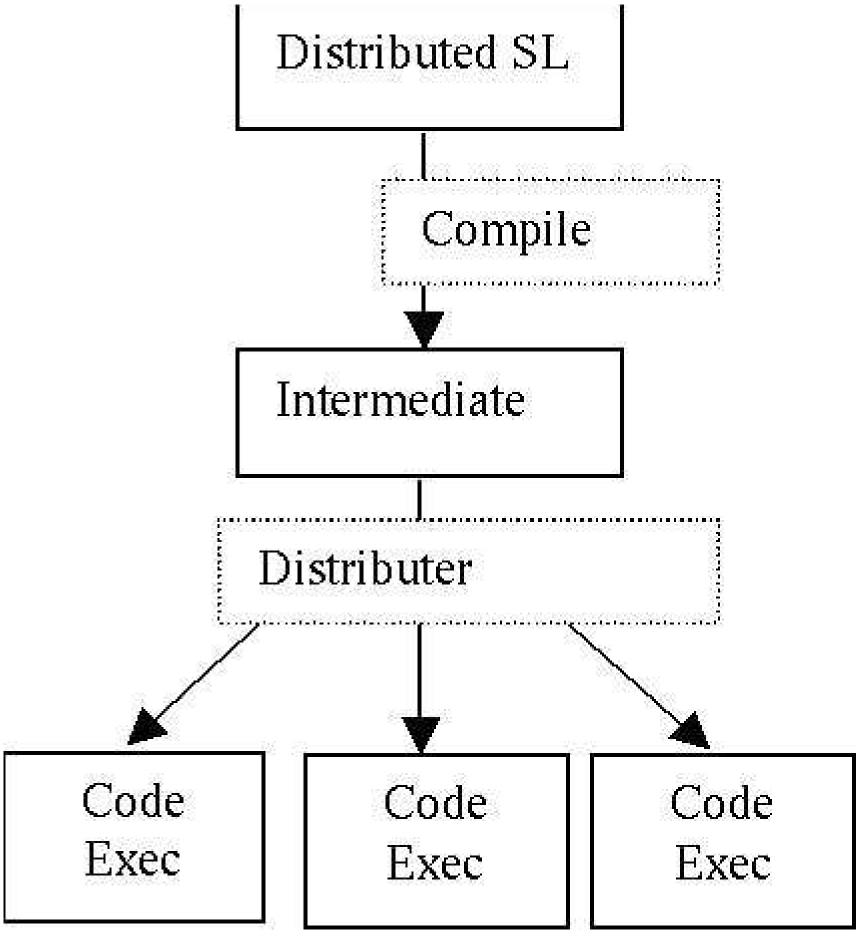
```
X,A :   BOOL ;
B,C :   INT ;
X := A AND B*C>10
```

Si une des valeurs B ou C est INCONNUE, la multiplication B*C donne INCONNU sauf si une des valeurs (B ou C) est 0. Si B * C est INCONNU La comparaison B * C > 10 donne INCONNU.
L'opérateur AND a la table de verité suivante :

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

2.2. Extension du langage SL : SL distribué

« SL distribué » signifie qu'un programme écrit à l'aide du langage SL de façon centralisée peut être découpé en plusieurs morceaux qui seront exécutés sur des machines hétérogènes, comme des serveurs de supervision et des automates programmables.



L'enchaînement des opérations est donc prévu comme suit : SL distribué est dans un premier temps traduit en code intermédiaire. C'est une compilation classique qui analyse lexicalement et sémantiquement le code et ajoute des informations permettant le debugging.
Le distributeur analyse les liaisons entre les variables et les différents morceaux de programmes. Sur base de cette analyse, de la configuration disponible et d'éventuelles directives du programmeur, le distributeur décide où les blocs de programmes seront exécutés.

Dans les automates et le système de supervision, les instructions sont exécutées dans une machine virtuelle qui interprète le code intermédiaire.

L'utilisation de système de supervision et d'automates au cœur de processus industriels exige que la distribution de programme respecte les contraintes suivantes :

- notion de temps réel ;
- possibilité de modification « on line » (à chaud) du programme lors de son exploitation ;
- nécessité de disposer d'une diagnose puissante et de bons outils de debugging ;
- robustesse du système de supervision.

Contraintes temps réel

Le système de supervision OBviews (disponible dans les environnements UNIX et WIN NT) n'est pas un système temps réel suivant la stricte définition de ce terme. Pourtant l'utilisateur attend naturellement de son système un temps de réponse déterminé. Les automates, par contre, fonctionnent en temps réel. Un programme y est exécuté cycliquement de façon très constante.
Le distributeur doit supporter les contraintes temps réel et offrir la possibilité de passer d'un vieux programme à un nouveau programme en synchronisant toutes les machines qui exécutent les morceaux du programme. Par la même occasion, il doit être capable de gérer différentes versions.

Changement « on line »

Il doit être possible de modifier un programme au cours de son exécution. En un temps très court, il doit être possible de passer de l'ancienne à la nouvelle version tout en assurant les points suivants :

- les valeurs des données doivent être conservées ;
- le nouveau programme doit redémarrer là où l'ancien s'était arrêté (dans la mesure du possible) ;
- le changement doit être assez rapide pour ne pas altérer le caractère temps réel des automates.

Diagnose et debugging

De bons outils de diagnose et de debugging doivent être disponibles. Par diagnose, nous entendons la recherche d'un problème au niveau du processus qui est supervisé. Le debugging recouvre la recherche de problèmes dans le programme exécuté.

Pour aider le programmeur (souvent un électricien de service) à effectuer une diagnose, les valeurs des variables lui sont présentées dans le programme source. Des fonctions de recherche doivent être disponibles pour retrouver les endroits où sont utilisées ou modifiées les variables. L'utilisateur part des cartes d'entrées–sorties d'un automate et remonte jusqu'aux endroits où elles sont utilisées.
La diagnose, tout comme le debugging, doit pouvoir être réalisée sans interrompre la supervision ni le processus en cours. Nous n'utiliserons pas de « break points » mais privilégierons les traces de programmes pour découvrir les anomalies. Nous devons donc définir des règles pour préciser ce qui doit être tracé et à quel moment cela doit être fait.

Robustesse

Un système de supervision et ses automates associés doivent garantir un fonctionnement le plus cohérent possible malgré les erreurs laissées par les programmeurs. Un programme ne doit en principe pas arrêter brutalement son exécution lorsqu'une erreur se produit. Par exemple, accéder à la onzième valeur d'un tableau contenant dix éléments devrait renvoyer la valeur INCONNU avant de poursuivre imperturbablement. De même, au niveau de la distribution de morceaux de programme, nous devrons garantir un niveau de sécurisé suffisant.

Par exemple, soit une machine 1 qui alimente une machine 2 en blocs ; la machine 2 marche lorsque la machine 1 marche. Une règle dangereuse est donc

```
IF NOT machine2.running THEN
        Machine1<–stop() ;
END_IF ;
```

Le reste du programme doit garantir que la machine 2 démarre avant la machine 1 !

Schéma complet

Nous pouvons construire un schéma plus complet (cf. figure ci–dessous).

Nous voyons ici que le distributeur tient compte des données de configuration du système et des différentes versions du programme. La tâche de charger les morceaux de programmes et de synchroniser le passage des anciens programmes aux nouveaux est prise en charge par un programme indépendant, dénommé « loader ».

Les outils de diagnose et de debugging utilisent les programmes source pour interagir avec l'utilisateur et pour interpréter les données du distributeur. Les valeurs des variables sont demandées aux divers modules.

Le distributeur décide aussi comment les communications entre le système de supervision et les automates doivent être configurées.

2.3. <u>State–of–the–art technologique</u>

Développement centralisé

Il existe plusieurs logiciels de supervision proposant un environnement de développement intégrant superviseurs et automates. Cela signifie que l'application y est écrite à l'aide d'un éditeur central et téléchargée vers les automates et le serveur. Chaque morceau de programme est écrit dans un langage spécifique à la machine cible et c'est le programmeur qui décide où les différentes parties de l'application doivent être exécutées et comment la communication entre ces parties doit être configurée. A notre connaissance, aucun logiciel de supervision ne propose une distribution semi–automatique optimale d'un programme uniforme à travers un ensemble de machines.

Systèmes distribués : langages, systèmes d'exploitation, équilibrage de charge, architectures, …

Le nombre de projets de recherche traitant de près ou de loin du concept de distribution est énorme. Notre comparaison avec ce qui existe doit donc être limitée à un groupe de caractéristiques spécifiques.

La recherche que nous voulons entreprendre se concentrera sur les aspects suivants :

- l'impact des caractères événementiel (supervision) et cyclique (automates) sur la distribution ;

- le changement on line d'un programme distribué en conservant les données et les divers « program counters » ; (Nous parlons bien de l'évolution du programme et non de sa reconfiguration pour réagir par exemple à une défaillance d'un équipement, ni à une migration de la tâche sur une autre machine. Les contraintes temps réel doivent aussi être respectées).
- la diagnose on line d'un système distribué sans interrompre le déroulement du programme. (S'il y a des recherches portant sur la diagnose de programmes distribués, il n'y en a pas, à notre connaissance, qui concerne un système de supervision industriel.)

La communication entre les processus de l'architecture que nous envisageons est totalement asynchrone. Il n'y a donc aucun rendez–vous entre les différents systèmes et tâches. Dans le cadre de SL distribué, nous n'aborderons pas la problématique de la programmation parallèle.

Le programmeur doit pouvoir interagir avec l'outil de distribution. Ce dernier demande au programmeur des informations qui lui permettra de solutionner des problèmes de distribution complexes. Par exemple :

Automate (cyclique)	Supervision (événementiel)

Pump.temperature := input_1 ;
Pump.alarm := Pump.temperature > 80 ;
IF Pump.alarm THEN
  Pump<–stop() ;
END_IF ;
	WHEN Pump.alarm THEN
	  PRINT 'Temperature too high :', Pump.temperature
	END_WHEN

est équivalent à :

Automate (cyclique)	Supervision (événementiel)

Pump.temperature := input_1 ;
Pump.alarm := pump.temperature > 80 ;
	WHEN Pump.alarm THEN
	  Pump<–stop() ;
	  PRINT 'Temperature too high :', Pump.temperature
	END_WHEN

si les conditions suivantes sont vérifiées :

- la pompe peut être arrêtée un instant après l'apparition de l'alarme correspondante ;
- il n'est pas possible de démarrer la pompe lorsqu'une alarme est présente (WHEN ne réagit que sur le flanc montant) ;
- dans le cas d'une coupure de communication entre automate et superviseur, le système doit se mettre en mode « sécurité ».

Nous reléguons au chapitre suivant les aspects plus techniques et scientifiques du projet ainsi que la description des avancées existantes sur lesquelles nous baserons notre étude bibliographique et notre recherche.

3. PROGRAMME DE RECHERCHE DÉTAILLÉ POUR 2 ANS

Le projet comprendra les trois phases suivantes :

- •conception et développement d'un langage de programmation orienté objet à distribution transparente ;
- •conception d'un outil d'analyse de code destiné à l'identification de « modules » ou objets pouvant être distribués ;
- •réalisation d'un compilateur traduisant les instructions de haut niveau du langage conçu vers un langage interprété sur les automates et développement d'un outil de diagnose.

Chaque phase comprend un certain nombre de points de recherche. Dans ce qui suit, nous détaillons, pour chacune d'elles, l'objectif, les voies de recherche et une étude préalable de l'existant.

3.1. <u>Description des 3 phases du projet</u>

<u>Phase 1</u> : Conception et développement d'un langage de programmation orienté objet à distribution transparente

But de la phase

Dans cette phase, un langage de programmation orienté−objet à distribution transparente doit être défini. Ce langage doit avoir les caractéristiques suivantes :

- •être orienté−objet ;
- •permettre la programmation évènementielle ;
- •permettre le contrôle de processus en temps−réel ;
- •permettre facilement la distribution de code ;
- •permettre éventuellement d'autres styles de programmation.

En effet les systèmes conçus fonctionnent généralement pour les parties de contrôle de façon évènementielle, et pour les parties automates par polling. Ces deux modes de fonctionnement doivent pouvoir être permis par le langage.

Situation du problème et voies de recherche

A notre connaissance, aucun environnement de conception de systèmes de contrôle n'intègre parfaitement l'ensemble de ces mécanismes.

Le choix du langage devra non seulement tenir compte de la facilité d'utilisation par le concepteur, mais également de la possibilité de compiler les programmes et d'obtenir un code efficace en particulier au niveau du traitement des évènements, des communications et des synchronisations entre processus.

Pour cette phase, la recherche débutera par une analyse précise des mécanismes implémentés dans des environnements de programmation évènementiels existants. On trouve ce type d'environnement dans la programmation système (Unix, NT, …)

ou dans certains langages de programmation tels que Java ou Ada où un mécanisme de traitement d'exception existe.

D'une manière moins directe, les systèmes réactifs sont modélisés par des formalismes d'automates (Hoare 1985, Milner 1989, Alur 1994, Bolognesi 1987, Henzinger 1996). Ces formalismes, quoique plus éloignés de l'implémentation, permettent un travail d'analyse dans le but de contrôler l'exactitude des systèmes définis. Développer des systèmes fiables peut être essentiel dans certains développements critiques. Notons à ce titre que le projet actuel ne propose pas de module de vérification permettant de prouver que le système défini est totalement correct. De façon générale, un tel module de vérification fonctionnant pour tous les systèmes analysés est impossible ou trop complexe. Des techniques d'abstraction et de composition peuvent alors être utilisées, généralement avec succès (de Roever 1998). Pour le présent projet, nous proposons de permettre des modules de test du système développé et de veiller pour le moins, à ce que ces aspects puissent être examinés ultérieurement.

Au niveau de l'aspect temps−réel, un certain nombre de mécanismes tel que le déclenchement de time−out ou de chien de garde (watch−dog) ainsi que la consultation d'horloge devront être permis.

Au niveau de l'existant en terme de langage de modélisation et d'environnement associé, le langage SDL et son environnement ObjectGeode (http://www.csverilog.com) seront étudiés pour déterminer les concepts qu'ils utilisent et leur adaptation à notre problématique. Notons tout de suite que SDL est un langage qui définit des systèmes à processus communiquant de façon asynchrone (file de communication entre processus) et est donc a priori inadapté tel quel pour développer des systèmes de contrôle en temps−réel dur (où les échéances ne peuvent pas être dépassées).

Enfin, en ce qui concerne l'aspect distribution « aisée », ce problème est lié à la synchronisation ou communication entre processus distribués ou même non distribués. Deux aspects sont à prendre en considération :

- distribution des données ;
- distribution du code.

Dans la mesure du possible, les aspects de distribution de données sont à éviter pour promouvoir une approche orienté−objet propre où les données sont privées au sein des objets. A priori, une approche possible sera celle d'objets distribués de façon transparente pour l'utilisateur. Les communications entre objets se feront soit sous forme de message (appel d'un objet) soit sous forme de signal (envoyé à l'objet).

Les objets distribués existent dans des environnements tels que celui du langage Distributed Oz (Haridi 1998, Van Roy 1997) où le concept d'objet mobile existe. Notre étude pourra déterminer les besoins en la matière.

Enfin, pour permettre au système de contrôle plusieurs « styles » de comportement, les notions d'événement « urgent » et « non urgent » seront étudiées. Un événement urgent devra directement être traité (programmation évènementiel) tandis qu'un événement non urgent pourra être traité dès que le système a « le temps » (traitement par polling).

Phase 2 : Conception d'un outil d'analyse de code destiné à l'identification de modules ou objets pouvant être distribués

But de la phase

Le produit principal de cette phase est un outil qui reçoit un certain nombre de contraintes de :

- localité en terme de partie de code ou en terme d'interactions entre le système et son environnement ;
- fiabilité ;
- cohésion ;
- performance en terme de temps de traitement.

L'outil devra essayer d'optimiser le système de contrôle produit en regard de ces critères.

Situation du problème et voies de recherche

Certains environnements de développement de systèmes proposent la génération automatique du code distribué à partir d'une description des composantes physiques et d'une assignation de chaque partie de code ou des données sur ces composantes.

Dans notre cas, l'assignation des parties de code est à choisir de façon « intelligente » par le système. Si l'on ne s'attarde qu'au problème de performance, ceci pourrait être vu comme un problème de « load–balancing » (Fox 1994). Ici, d'autres facteurs sont à prendre en compte. Pour résoudre le problème, une modélisation (abstraction) du code devra être faite sous forme de graphe. La modélisation devra être définie pour avoir le meilleur niveau de description et permettre de définir de façon optimale, la localisation de chaque morceau de code.

A partir de ce modèle, des algorithmes d'analyse de graphes du type recherche de composantes connexes pourraient être imaginés. Un certain nombre de techniques d'analyse statique (Kennedy 1981, Kosaraju 1974) ou même d'interprétation abstraite (Cousot 1977) pourraient être définies par exemple pour effectuer des analyses du pire des cas (dans l'exécution d'instructions répétitives par exemple).

Phase 3 : Réalisation d'un compilateur traduisant les instructions de haut niveau du langage conçu vers un langage interprété sur les automates et d'un outil de diagnose

But de la phase

Cette phase intègrera les notions définies dans les deux phases précédentes pour créer un compilateur effectuant de la distribution de code et de façon plus générale pour développer un environnement de conception et diagnose de systèmes de contrôle. Cette phase comprend donc une grande partie de développement :

- le développement d'un compilateur de code distribué ;

- le développement des librairies d’interface avec les équipements de contrôle et les pupitres de contrôle opérateur (interface graphique) ;
- le développement de l’outil de diagnose décrit au point 3.

Dans cette phase, une autre partie cruciale pour le projet consiste à trouver des techniques pour permettre d’effectuer de la compilation incrémentale. En effet, l’environnement doit permettre d’effectuer la maintenance des systèmes développés soit pour corriger des erreurs de conception soit pour étendre les systèmes à de nouvelles fonctionnalités. Une telle compilation incrémentale doit préserver la majorité du code précédemment généré ainsi que la distribution choisie, pour ne faire qu’un minimum de corrections nécessaires. Par ailleurs, l’environnement doit permettre une mise à jour du système de contrôle si possible sans devoir interrompre le contrôle.

Situation du problème et voies de recherche

Au niveau de la compilation, des techniques standard de compilation de programmes impératifs et orientés–objets pourront être utilisées (Aho 1986, Fischer 1991). Pour procéder à la compilation de façon incrémentale, deux problèmes précis sont à résoudre :

- génération d’un code respectant, pour une grande partie ou la totalité, une certaine distribution déjà imposée ;
- procédure de mise à jour automatique du nouveau code.

Pour résoudre ces deux problèmes assez spécifiques à notre projet, des procédures ad hoc devront être trouvées à travers une recherche spécifique.

3.2. <u>Planning et répartition des tâches</u>

Les tâches se répartissent sur une durée de deux ans ; elles impliquent deux chercheurs à l’ULB (notés U1 et U2) et trois personnes chez Macq Electronique (M1, M2, M3).

De façon schématique, les trois phases se développent en parallèle sur deux ans. Nous identifions une série de sous–produits constituant des objectifs intermédiaires à l’issue de chaque phase et de chaque année de travail.

	Phase 1	Phase 2	Phase3
Année 1	Conception d'une version distribuée du langage SL (U1 + M1)	Recherche de composantes « parallélisables » dans un langage de type SL étendu par analyse de code : étude bibliographique et conception d'heuristiques de sélection « optimale » de ces composantes sur base de critères (à définir) d'efficacité et de sécurité (U2)	Réalisation d'un compilateur traduisant le langage SL existant en un langage destiné à être interprété sur les nouveaux automates (M2 + M3)
Année 2	Conception et développement d'un outil supportant le développement de programmes dans le langage étendu (M1)	Suite et développement d'un outil prototype implémentant les heuristiques conçues (U2 + M2)	Compilateur pour le langage étendu, avec pour cibles les superviseurs et les automates (U1 + M3)

Au terme des deux ans, un outil intégrant l'édition de programmes en langage SL étendu, l'analyse du code pour la détection des composantes à distribuer et la compilation vers superviseurs et automates devrait être disponible.

3.3. <u>Extension future : vérification formelle</u>

Une des extensions futures du projet sera tournée vers la vérification de propriétés dynamiques à satisfaire par les programmes devant être exécutés sur les automates ou sur un superviseur. Actuellement, une telle vérification n'est effectuée que par le biais de tests en simulation.

Garantir formellement des propriétés spécifiées par le client constituera un avantage compétitif important que nous pouvons dores et déjà prendre en compte dans la conception du nouveau langage et des programmes qu'il permettra d'écrire.

4. REFERENCES BIBLIOGRAPHIQUES

1. A.V. Aho, R. Sethi, J.D. Ullman, Compilers : principles, techniques and tools, Addison−Wesley, 1986
2. R. Alur, D.L. Dill, A Theory of Timed Automata, Theoretical Computer Science 126:183−235, 1994
3. T. Bolognesi, E. Brinksma, Introduction to the ISO specification language LOTOS, Computer Networks and ISDN Systems, 14(1):25−59, 1987
4. P. Cousot, R. Cousot, Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints, Fourth ACM Symposium on Principles of Programming Languages, 238−252, 1977
5. W.P. de Roever, H. Langmaack, A. Pnueli, Compositionality: The Significant Difference, Lecture Notes in Computer Science 1536, Springer−Verlag, 1998
6. Ch. Fischer, R.J. Leblanc Jr, Crafting A Compiler With C, Benjamin/Cummings, 1991
7. G.C. Fox, R.D. Williams, P.C. Messina, Parallel Computing Works, Morgan Kaufmann Publishers Inc., 1994
8. S. Haridi, P. Van Roy, P. Brand, C. Schulte, Programming languages for distributed applications, New Generation Computing, 16(3) :223−1261, 1998
9. T.A. Henzinger, The Theory of Hybrid Automata, Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS 1996), pp. 278−292
10. C.A.R. Hoare, Communicating Sequential Processes, Prentice Hall, 1985
11. K. Kennedy, A survey of data flow analysis techniques, in S.S. Muchnick, N.D. Jones, Program Flow Analysis : Theory and Applications, 51−64, Prentice−Hall, 1981
12. S.R. Kosaraju, Analysis of structured programs, J. Computer and Systems Sciences, 9(3) :232−255, 1974
13. A.J.R.G. Milner, Communication and Concurrency, Prentice Hall, 1989
14. ObjectGeode: http://www.csverilog.com/
15. P. Van Roy, S. Haridi, P. Brand, G. Smolka, M. Mehl, R. Scheidhauer, Mobile objects in Distributed Oz, ACM Transactions on Programming Languages and Systems, 804−851, 1997.