L'expérience SCOL, un langage pour des applications internet multi-utilisateurs

Anne-Gwenn Bosser[1] & Francisco Alberti[2]

1: Anne-Gwenn.Bosser@pps.jussieu.fr
2: Francisco.Alberti@pps.jussieu.fr
Laboratoire PPS, CNRS UMR7126
Université Paris 7, case 7014, 2 Place Jussieu
75251 Paris Cedex 05

Résumé

Cet article présente un retour d'expérience sur l'utilisation de SCOL, un langage fonctionnel propriétaire développé par la société Cryonetworks qui a été utilisé pour réaliser des communautés virtuelles.

Les auteurs, anciens membres de l'équipe de Recherche et Développement de Cryonetworks, commencent par donner un bref aperçu des caractéristiques du langage. L'article propose ensuite une réflexion sur l'utilisation de SCOL dans le cadre de la réalisation d'une application critique en terme de performances attendues et faisant intervenir une importante équipe de développement.

1. Introduction

La société Cryonetworks, éditeur de technologies pour la réalisation de mondes virtuels et filiale technologique de Cryo Interactive, a déposé le bilan au mois de Juillet 2002 après cinq années d'existence, victime de la crise dans l'industrie française du jeu vidéo. Cette disparition risque de s'accompagner de celle de sa technologie propriétaire, créée par Sylvain Huet, à notre connaissance le seul exemple d'utilisation d'un langage avec polymorphisme paramétrique et inférence statique de type développé par et pour l'industrie (le langage Erlang [9] développé par la société Ericsson est un autre langage industriel polymorphe paramétrique, mais le typage y est dynamique). Depuis la naissance de cette technologie plus d'un million de *Voyagers*, les programmes clients permettant de visiter les sites SCOL (*Standard Cryo On-Line*), ont été téléchargés sur le site de Cryonetworks[1].

La technologie SCOL a été créée pour permettre de réaliser des applications internet et multimédia, dont le dénominateur commun est la représentation des utilisateurs connectés dans un espace en trois dimensions où ils peuvent se rencontrer et interagir. Parmi les réalisations de la société et de ses partenaires, on peut citer des communautés virtuelles comme Cryopolis[2], qui est aussi la vitrine technologique de Cryonetworks, des sites marchands (notamment pour la FNAC) dans lesquels l'acheteur pouvait s'informer auprès de vendeurs en ligne, des sites promotionnels comme ceux réalisés lors de la sortie du film *Scary Movie* ou de l'émission de télévision *Loft Story*, et des jeux multi-joueurs en ligne comme *Venise*, *Fog*, et *La chasse au trésor*. Plus récemment, la société finissait de mettre au point un jeu en ligne massivement multi-joueurs.

Les auteurs de cet article sont d'anciens membres de l'équipe Recherche et Développement de Cryonetworks, et ont travaillé sur les évolutions de la technologie de la société lors des deux dernières

[1] www.cryonetworks.com
[2] www.cryopolis.com

hello.pkg : un *package* de code SCOL

```
fun hello () =
  // Afficher un message sur la console
  _fooS "Hello, virtual world!";;
```

hello.scol : fichier exécuté à partir de la machine virtuelle

```
_load "hello.pkg" //Charge le package 'hello' dans le canal courant
hello             //Exécute la fonction nommée 'hello'
```

On obtient à l'affichage sur la console :

loading hello.pkg...
fun hello : fun [] S
Hello, virtual world!

Le type de `hello` trouvé par SCOL est `fun [] S`, une fonction qui ne prend aucun argument et qui renvoie une chaîne de caractères.

FIG. 1 – Exemple d'application SCOL

années de son existence. Dans cet article, ils reviennent sur le langage SCOL et leur expérience passée, tout en essayant d'ouvrir la discussion sur l'utilisation de cette famille de langages dans une industrie orientée internet et multimédia.

La première partie de cet article présentera donc l'essentiel de la technologie SCOL, en rapport avec les applications industrielles qu'elle vise. Les auteurs décriront ensuite les problèmes qu'ils ont rencontrés ou observés lors de l'utilisation de ce type de langage dans le cadre de la réalisation d'une application importante en terme d'équipe de développement et de performances attendues, comme c'est le cas pour un jeu massivement multi-joueurs.

2. La technologie SCOL

2.1. Généralités

La technologie SCOL est entièrement basée sur une machine virtuelle, qui exécute des programmes écrits dans le langage SCOL. Cette machine virtuelle est caractérisée par une gestion automatique de la mémoire (elle possède un *garbage collector*), une gestion des communications réseau (la façon dont celles-ci sont prises en charge est en effet masquée par le langage), et de nombreuses bibliothèques. Une application SCOL est donc un ensemble de programmes écrits dans le langage SCOL, que l'on donne à compiler (à la volée) et à interpréter à une ou plusieurs machines virtuelles communicantes, éventuellement distantes. La machine virtuelle possède également une console, qui permet de vérifier les données et types chargés par un programme.

L'approche de la technologie SCOL est contraire à celle de VRML (*Virtual Reality Modeling Language*) [21] : alors que cette dernière repose sur une immersion dans un espace en trois dimensions, à laquelle on superpose la notion de cohabitation des utilisateurs connectés (comme dans *Community Place* [14]), le cœur même de la technologie SCOL est le modèle de communication réseau auquel s'ajoutent les bibliothèques dédiées à la simulation d'un univers. Les plus importantes sont le moteur 3D et des bibliothèques 2D, son et SQL.

Pour lancer la machine virtuelle, on lui donne une ou plusieurs expressions d'un mini-langage de script (un sous-ensemble du langage SCOL) à évaluer (Figure 1).

Dans ce qui suit, nous allons brièvement décrire le noyau du langage et ce qui fait son originalité :

le chargement dynamique de code et son modèle de communication. Nous expliquerons ensuite le *framework* DMS (*Distributed Module System*), développé en SCOL, support de toutes les applications Cryonetworks.

2.2. Le noyau du langage

SCOL plonge ses racines dans la longue tradition des langages fonctionnels, nés au sein de la communauté académique à la fin des années 50. C'est un langage applicatif, polymorphe paramétrique avec inférence de types et qui manipule des valeurs fonctionnelles. En fait, SCOL a été directement inspiré par Caml [15] dont il adopte une grande partie de la syntaxe et l'inférence de types, et a voulu profiter de bon nombre des avantages de son grand-frère : il se devait donc d'être un langage d'un haut niveau d'abstraction, mettant à profit le polymorphisme paramétrique pour une meilleure réutilisabilité du code produit, et le typage statique pour détecter dès le chargement des incohérences éventuelles des programmes et permettre une compilation plus efficace. Certaines spécificités des applications destinées à être développées, comme la nature profondément impérative du code client, auraient cependant rendu l'utilisation de la même syntaxe que celle du langage Objective Caml [1] un peu encombrante (comme tout simplement l'obligation de déclarer les *mutables*) et CamlP4 [2], qui aurait permis de réutiliser la sémantique d'Objective Caml avec une syntaxe allégée n'existait pas encore à l'époque où SCOL est né. Mais ce sont surtout des impératifs bien industriels de maîtrise et de contrôle des évolutions qui ont motivé la création d'un nouveau langage.

La différence majeure entre Objective Caml et SCOL est que ce dernier ne manipule que des fonctions globales. La syntaxe de l'application (totale) d'une fonction `f` d'arité n est `(f e1 .. en)`, abbréviation de `(exec @f with [e1 .. en])` où la notation `@f` est une *référence* à la fonction `f`. Il est pourtant possible d'effectuer des applications partielles de fonctions : la bibliothèque du noyau du langage propose toute une famille de primitives (des fonctions nommées `mkfun`*n* pour n allant de 2 à 8) permettant d'appliquer partiellement une fonction d'arité n à son dernier argument. L'expression `(mkfun2 f e2)`, où la fonction `f` est d'arité 2, est une référence au résultat de l'application partielle de `f` à `e2` : c'est donc une référence à une fonction d'arité 1. Cette référence peut alors être utilisée à l'aide du mot clef `exec`.

Le noyau du langage propose les types de base courants dans les langages fonctionnels : entiers, réels, chaînes de caractères, n-uplets et listes. Mais, contrairement à Objective Caml, les listes y sont définies comme une suite de paires emboîtées, et sont terminées par la valeur spéciale `nil`. Elles sont typées en utilisant des types récursifs (Figure 2). Héritée de Lisp, `nil` est la valeur par défaut des variables SCOL et son type est monomorphe (Il y a un nil pour chaque type). La constante `nil` peut également être utilisée comme valeur d'un calcul, permettant ainsi de pallier l'absence d'exceptions dans le langage, et comme argument optionnel d'une fonction. Toute ambigüité sur le type d'une valeur `nil` est refusée par le typage. Par exemple, la déclaration d'une variable dont la valeur initiale est `nil` se fait en utilisant le mot-clef `typeof` pour déclarer son type. De même, on ne peut pas définir une fonction qui renvoie `nil` sans préciser son type de retour ou sans que la vérification de type ne puisse l'inférer.

Le noyau du langage SCOL se présente donc comme un langage simple, que l'on peut expliquer en quelques pages. Contrairement aux langages impératifs et orientés objet généralement adoptés par l'industrie, SCOL évite au programmeur le besoin d'écrire explicitement les types des variables et des fonctions sans pour autant mettre en danger l'exécution des applications. Cette légèreté permet également de produire un code source très compact, ce qui est un avantage certain quand ce code doit transiter par le réseau. L'objet de cet article n'étant pas de décrire exhaustivement le langage SCOL, nous engageons le lecteur à se reporter à la présentation semi-formelle [3] et au manuel de référence de SCOL [11] pour plus de détails sur sa sémantique.

En SCOL :

```
fun long (l) =
  if l == nil then 0
  else
    let l -> [_  nxt] in
      1 + (long nxt);;
```

fun long : fun [[u0 r1]] I
L'argument de la fonction `long` est une liste de récursion de niveau 1.

En Objective Caml :

```
let rec long l =
  match l with
     [] -> 0
   | _::nxt -> 1+(long nxt);;
```

val long : 'a list -> int = <fun>
L'argument de la fonction `long` est une liste polymorphe.

En Objective Caml avec types récursifs, un `nil` et des paires emboîtées.

```
let rec nil_int = 1,nil_int;;
let rec long l=
  if l==nil_int then 0
  else
   1 + (long (snd l));;
```

*val long : (int * 'a as 'a) -> int = <fun>*
L'argument de la fonction `long` est un type récursif sur le deuxième élément de la paire.

FIG. 2 – Exemple de typage en SCOL et en Objective Caml d'une fonction simple

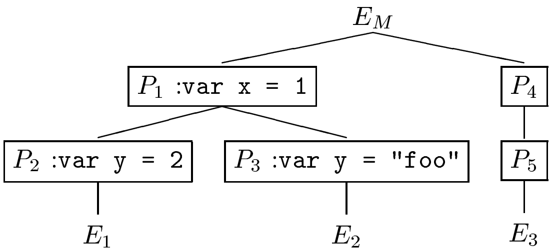
2.3. Les packages et le chargement dynamique

Le chargement dynamique est un grand avantage pour la réalisation d'un monde virtuel : lorsqu'un utilisateur se connecte, il n'a pas nécessairement besoin dès le départ de toutes les fonctionnalités de l'application, et peut donc se contenter de télécharger depuis le serveur dans un cache le code source correspondant à un minimum de fonctionnalités indispensables, puis de le charger dans la machine virtuelle (ces deux opérations étant prises en charge de manière transparente par l'application). Au fur et à mesure de sa visite, il se procurera, au besoin, le code SCOL nécessaire à ses activités : tout se passe exactement comme lorsqu'on *surfe* de page en page sur internet. Grâce au typage statique, les incohérences des programmes seront détectées dès le chargement. Il existe par exemple une zone dans Cryopolis, une communauté virtuelle réalisée en SCOL, où les utilisateurs peuvent s'affronter dans un jeu nommé *paintball*. Ils ne devront charger le code SCOL nécessaire au jeu que s'ils entrent dans la zone géographique de Cryopolis dédiée à cette activité. L'attente au lancement de l'application se trouve donc réduite, et n'est plus dépendante de la taille de celle-ci.

Le *package* est l'unité de chargement de la machine SCOL. C'est un fichier contenant une suite de définitions de types, de variables, et de fonctions. Contrairement aux modules d'Objective Caml, un *package* ne détermine pas un espace de nommage pour les valeurs qu'il définit (comme un `include` ou un `use` de la boucle d'interaction d'Objective Caml). La commande `_load`, que nous avons vue utilisée dans le mini-langage de script qui sert à lancer une machine virtuelle, appartient également au langage : elle peut être utilisée dans un *package*. C'est cette commande qui permet de charger dynamiquement un *package* dans la mémoire de la machine virtuelle, et d'augmenter ainsi l'environnement d'exécution du programme.

L'originalité de la machine SCOL est qu'elle sait gérer plusieurs environnements, qui ne sont pas forcément indépendants : ceci permet de partager certaines variables ou fonctions entre des environnements différents (Figure 3).

Un environnement peut donc évoluer au cours de l'exécution d'un programme :



Chaque chemin dans l'arbre définit un environnement E_i donné. Un nœud P_i correspond à un package chargé dans l'environnement formé par ses prédécesseurs dans l'arbre. Ainsi, P_2 est chargé dans l'environnement où P_1 défini la variable `x`, qui dans l'exemple est aussi partagée par P_3. Tous les packages chargés partagent les définitions de l'environnement minimal E_M (qui contient toutes les fonctions de la bibliothèque standard de SCOL).

FIG. 3 – Exemple des environnements partagés

- on peut l'agrandir en y ajoutant de nouvelles définitions ou des redéfinitions,
- on peut supprimer les définitions correspondant au dernier *package* compilé,
- on peut le substituer à un autre, dans un contexte d'exécution donné.

La relation de dépendance entre *packages* n'est pas explicite en SCOL. Le programmeur doit définir l'ordre correct du chargement des *packages* pour que la résolution de liens, nécessairement dynamique, puisse aboutir. Le chargement, dans le cas de SCOL, implique obligatoirement la compilation à la volée, car la sémantique d'un *package* dépend étroitement du contenu de l'environnement.

Le chargement dynamique de *packages* est utilisé de manière intensive dans la plupart des applications Cryonetworks car il est *naturellement* sûr : lorsque la machine virtuelle charge du code source, elle le compile et le type, ce qui est une garantie de sûreté du code chargé en évitant les erreurs de typage à l'exécution (il n'est pas nécessaire de faire de la vérification de bytecode comme en Java [5]).

2.4. Les canaux et les messages

Autre originalité de la machine SCOL, l'association d'un environnement avec une liaison réseau, que l'on appelle *canal*. Il est néanmoins possible de définir des canaux sans liaison réseau associée.

Le modèle d'exécution de SCOL repose fondamentalement sur cette notion : l'évaluation d'une expression s'effectue toujours dans le contexte d'un canal donné. La liaison réseau prend la forme d'une connexion TCP ou UDP avec une autre machine virtuelle. Dans le cas simple d'une application qui n'est pas liée au réseau, la connexion est vide et on se trouve alors dans un modèle d'évaluation non distribué. Si la connexion est définie, on se trouve dans le cas plus général d'une application SCOL répartie qui permet l'activation de fonctions définies dans des environnements possiblement distants. C'est une convention de nommage qui permet de décider si une fonction fait partie de l'interface de communication : une fonction est activée à distance par le message `mmm` si elle est nommée `__mmm`.

L'architecture de communication de la machine SCOL est fortement basée sur un modèle client-serveur. Afin que deux machines SCOL puissent communiquer, il faut que l'une d'entre elles adopte le rôle du serveur, et que l'autre ouvre une connexion vers ce serveur. Deux primitives très simples permettent d'initier ce dialogue. Une fois la connexion établie via la construction de canaux dédiés au dialogue, la communication peut se dérouler entre les deux machines, par envoi par l'une de *messages* faisant partie du vocabulaire de l'autre. Pour déclencher une fonction distante, il faut en plus de la convention de nommage que le message envoyé soit défini en déclarant les types des arguments de

vocabulaireAlice.pkg : fichier compilé dans chaque canal communiquant avec les clients

```
defcom Cwelcome = welcome;; //Constructeur d'un message sans arguments

typeof thisCli = I;;        //Variable locale à l'environnement du canal courant

// Fonction à exécuter dès la réception du message 'printThisString'
fun __printThisString (s) =
  _fooS (strcatn "Bob"::(itoa thisCli)::" said "::s::nil);;

fun welcome (nbCli) =
  set thisCli = nbCli;
  _on _channel Cwelcome [];; // envoyer le message sur le canal courant
```

Alice.pkg : fichier compilé dans le canal racine

```
var numClient = 0;;// Ordre du client connecté

fun main () =
  // Ouverture d'un serveur TCP en écoute sur le port 2000.
  // Lorsqu'un client se connecte, cela crée un canal associé à la connexion
  // qui hérite de l'environnement courant (1er paramètre), dans lequel
  // le script décrit par le 3ème paramètre s'exécute
  _setserver (_envchannel _channel) 2000
    (strcatn ("_load\"vocabulaireAlice.pkg\"\nwelcome "
      ::(itoh (set numClient = numClient + 1))::nil));;
```

Alice.scol

```
_load "Alice.pkg"
main
```

Fig. 4 – Application serveur pour **Alice**

cette fonction. Ce mécanisme s'apparente à un système de RPC (*Remote Procedure Call*) simplifié, où seuls des paramètres de type entier et chaîne peuvent être communiqués et où l'émetteur n'a aucune information sur le succès ou l'échec de l'appel. La machine virtuelle destinataire, à la réception du message, fera les vérifications de types nécessaires avant d'effectuer l'appel.

Exemple de client-serveur en SCOL : Les figures 4 et 5 décrivent une application client-serveur simple : les fichiers **Alice** (figure 4) décrivent le fonctionnement du serveur tandis que les fichiers **Bob** (figure 5) décrivent le client.

Les fonctions `_setserver` et `_openchannel` permettent respectivement d'ouvrir un serveur TCP et de créer une connexion vers un serveur TCP. Le mot clef `defcom` permet de définir un constructeur de communication, qui servira à créer des messages à envoyer sur un canal grâce à la fonction `_on`.

Chaque **Bob** qui se connecte possède un identifiant sur le serveur (qui correspond à son ordre de connexion). Cet identifiant est local au canal de communication d'**Alice** vers le **Bob** correspondant. **Bob** reçoit le message `welcome` dès que la connexion est effective, et envoie alors à **Alice** le message `printThisString`. **Alice** affiche la chaîne contenue dans le message et l'indice du client associé qui est une variable locale du canal de communication.

vocabulaireBob.scol : fichier compilé dans le canal communiquant avec les clients

```
defcom CprintThisString = printThisString S;;

fun __welcome () =
  _on _channel CprintThisString ["Thanks!"];;
```

Bob.pkg : fichier compilé dans le canal racine

```
fun main() =
  // Ouverture d'un canal vers Alice, en y chargeant le package vocBob.pkg
  _openchannel "212.157.130.74:2000" "_load \"vocabulaireBob.scol\"" nil;;
```

Bob.scol

```
_load "Bob.pkg"
main
```

FIG. 5 – Application client pour **Bob**

2.5. DMS et le SCS

La bibliothèque DMS (*Distributed Module System*), et l'outil associé, le SCS (*Site Construction Set*) illustrent bien la puissance du langage SCOL. Cette bibliothèque et l'outil d'intégration associé, écrits en SCOL, ont pour vocation de permettre d'assembler simplement des briques de base, implémentant des fonctionnalités de haut niveau, et de mettre ainsi la réalisation d'un monde virtuel à la portée d'un public intégrateur non spécialiste du domaine. DMS repose sur une architecture client-serveur, et utilise les protocoles TCP et HTTP pour faire communiquer les différents hôtes de l'application répartie.

Les briques de base d'une application réalisée à l'aide du SCS sont des programmes d'un haut-niveau d'intégration, appelés *modules*, et réalisant chacun une fonctionnalité fréquemment rencontrée dans un monde virtuel : par exemple, un *chat*, un mécanisme d'authentification, ou tout simplement un rendu 3D permettant de visualiser les avatars des utilisateurs connectés au même monde. Ces modules, dans le SCS, se présentent comme de simples boîtes noires, munies d'entrées (des fonctions particulières implémentées par le module et déclarées comme *actions*) et de sorties (qui sont des *événements* produits par le module), et comprenant éventuellement une interface utilisateur. L'intégrateur assemble les modules via le SCS, en reliant les entrées des uns aux sorties des autres, et définit l'interface utilisateur de son site en combinant entre elles les interfaces des différents modules qu'il utilise.

L'utilisateur du SCS n'a pas à se soucier de la répartition de l'application : les modules peuvent avoir une partie cliente et une partie serveur, la gestion de la distribution étant donc essentiellement réservée au développeur qui prend en charge la communication entre les parties clientes et serveur du module. Une fois le site assemblé, les communications générées par les liens tissés par l'intégrateur (un événement et une action liés pouvant indifféremment être client ou serveur) sont prises en charge par le *framework* DMS.

C'est donc dans le développement des modules que résidait la principale activité des développeurs d'une application DMS, et c'est sur l'observation de cette activité que se base l'analyse des auteurs.

3. Problèmes rencontrés dans le cadre du développement d'une application importante

Nous avons principalement observé trois types d'obstacles à la bonne utilisation du langage SCOL pour le développement d'applications de taille conséquente (impliquant une équipe de plus d'une demi-douzaine de développeurs), et ayant des contraintes de robustesse et d'efficacité, comme c'est le cas d'un jeu massivement multi-joueurs. Les deux premiers problèmes ont été couramment reprochés aux langages fonctionnels [22] : ce sont le manque de propriétés à même de faciliter l'ingénierie logicielle, et le manque d'outils pour assister le développement d'applications. Le troisième est lié aux particularités de l'industrie du jeu vidéo et aux habitudes de programmation des développeurs.

3.1. Langage et génie logiciel

Les langages les plus populaires de nos jours sont sans conteste les langages orientés objet, et ce pour une raison très pragmatique : le cœur du paradigme de programmation est la structuration du code par l'organisation des données. Le succès n'est pas tant celui d'un langage que celui d'un processus de développement extrêmement balisé et d'une méthode de conception quasi-hégémoniques dans l'industrie [17]. Bien sûr des langages comme ADA sont souvent utilisés sans s'appuyer sur une conception orientée objet, et dans les domaines industriels concernés (logiciels critiques, aéronautique), ce sont les normes (comme par exemple la DOD2167A, utilisée par l'armée) et le cycle de développement qui policent le bon déroulement du projet. Cependant, ces méthodes sont très coûteuses à mettre en œuvre, très lourdes à utiliser, et souvent inadéquates pour les autres industries.

Dans de nombreux domaines algorithmiques où le calcul a une grande importance, ce qui est le cas pour le serveur d'un monde virtuel (ou de manière encore plus flagrante, celui d'un jeu en ligne), le paradigme de la programmation fonctionnelle est plus approprié que le modèle objet. Les langages applicatifs comme SCOL ont une approche orientée *calcul* de la programmation : ce sont des langages purement algorithmiques qui n'orientent pas tout aussi naturellement le développement vers une structuration de l'application, ce qui a longtemps été un obstacle pour leur utilisation par le milieu industriel [22].

En SCOL, il n'existe pour l'instant aucun moyen autre que le *package* et la fonction pour structurer le code[3], et les limitations de ce modèle sont vite atteintes lorsqu'on s'atelle au développement d'une application importante. Aucun mécanisme du langage SCOL ne permet de définir des structures de données abstraites, car il n'y a pas de mécanisme d'encapsulation. Bien sûr, il est toujours possible, avec de la rigueur et des conventions de nommage, de venir à bout de ces difficultés, mais ces stratégies de conception sont vouées à l'échec : elles n'existent souvent que pour ne pas être respectées. De plus, en terme de génie logiciel, l'organisation des programmes a également pour but de permettre de réutiliser le code et de réaliser des tests unitaires. Or, nous l'avons vu dans la présentation du langage SCOL, la sémantique d'un *package* est fortement liée à celle de son environnement, les liaisons étant résolues dynamiquement. Il est donc très difficile de décrire le comportement d'un *package* sans tenir compte de son contexte et de tirer profit d'un code préexistant, tandis que la formalisation d'une batterie de tests unitaires pour ces unités de compilation s'avère être une tâche plus complexe qu'elle peut l'être avec un objet ou un module à la Caml dont les interfaces sont clairement définies.

Un langage, pour être un bon outil industriel, doit donc permettre de disposer des constructions nécessaires pour structurer le code et même refléter l'architecture de conception de l'application afin de faciliter sa maintenance et sa réutilisation. D'ailleurs, les langages issus du monde universitaire ont désormais pris en compte ces contraintes et intégré des mécanismes de structuration du code : en

[3] Les modules de DMS ne sont pas des entités du langage, mais des constructions abstraites, d'un haut niveau d'intégration fonctionnelle, du *framework*. Ils peuvent souvent atteindre une taille de code suffisament respectable pour nécessiter eux-même une structuration plus élaborée.

Objective Caml, par exemple, modules et objets permettent désormais d'organiser le développement d'importantes applications [7]. Cependant la structuration d'une application dont le développement s'appuie sur un langage à classes comme C++ a toujours un avantage sur les langages orientés *calcul* pour la réalisation d'une application à la limite des deux zones d'influence : les premiers reposent sur des mécanismes de structuration et d'encapsulation des données, tandis que les seconds les utilisent. Mis dans une situation d'urgence (phénomène habituel), le développeur pressé qui utilise un langage algorithmique ne passera souvent pas par l'étape conception. L'utilisation d'un langage reposant sur des concepts d'organisation et d'encapsulation l'obligera au minimum à se rappeler que cette étape existe. Si nous ajoutons à cela l'omniprésence de la méthodologie objet dans les cours de génie logiciel, il est facile de comprendre pourquoi les développeurs sont souvent convaincus que celle-ci est la base du génie logiciel[4]. La formation actuelle des développeurs paraît donc insuffisante de ce point de vue : il est vrai que la plupart d'entre eux connaissent différentes familles de langages et différents styles de programmation. Les cursus informatiques proposent tous des cours d'algorithmique basés sur l'étude des structures de données abstraites. Pourtant, et pour autant qu'il nous soit possible d'en juger, dès qu'il s'âgit d'apprendre à organiser un développement plus conséquent, ce sont les méthodologies orientées objet et le processus unifié qui ont le monopole : on apprend désormais rarement aux étudiants les techniques plus anciennes de décomposition modulaire [23], et bien que les méthodologies objet comportent de nombreux formalismes qui pourraient être adaptés à des langages basés sur un autre paradigme, les *cas d'école* sont le plus souvent en Java ou en C++. Alors que nous avons tous appris comment concevoir et analyser une application importante à l'aide de la méthodologie et de technologies orientées objet, la plupart d'entre nous semble ne jamais avoir été préparé à adapter ces méthodes ou à en utiliser une autre dans le cadre d'un gros projet utilisant un langage orienté *calcul*. Peut-être est-ce d'ailleurs l'une des raisons pour lesquelles beaucoup de développeurs ne considèrent pas ces langages adaptés à une utilisation industrielle.

3.2. Outils et environnements de développements

Qui dit génie logiciel dit également outils pour accompagner toutes les étapes du cycle de développement d'une application. Le manque d'environnements de développement, c'est-à-dire d'outils de mise au point et de bibliothèques riches et construites est un reproche qui a souvent été fait aux langages fonctionnels [22, 10]. Pendant de nombreuses années c'est la quête de performance qui a monopolisé l'attention des concepteurs de ces langages, au détriment des outils de développement. Le succès de Java a pourtant clairement démontré que l'adoption massive d'un langage par le monde industriel n'était pas seulement dû à son efficacité, critique désormais injustifiée pour la plupart des langages fonctionnels. Depuis quelques années cependant, la communauté fonctionnelle semble rattraper son retard en la matière : Objective Caml propose un debugger et un profiler [15], tandis que Bee [20] est un environnement de développement pour Scheme qui contient les outils indispensables au génie logiciel.

La lacune fréquente des langages fonctionnels au niveau des bibliothèques multimédia ne peut pas réellement être reprochée à SCOL qui est pourvu entre autre d'un riche moteur 3D et d'une bibliothèque 2D haut niveau. La question de la portabilité sur des systèmes d'exploitation autres que ceux de Microsoft n'a cependant jamais été complètement réglée, mais il est vrai que les applications SCOL visaient le grand public et les joueurs, principalement équipés de systèmes Windows, et ce n'était sans doute pas une priorité pour la société. Un enrichissement de l'éventail des protocoles de communications aurait néanmoins pu être souhaitable, s'agissant d'un langage qui tire sa force de son architecture de communication.

Cependant, alors même que de grands progrès en ce sens ont été effectués pour les langages fonctionnels, il n'y a pour SCOL aucun outil permettant de mettre au point du code : ni *debugger* digne

[4]Les utilisateurs de Scol (qui font presque tous partie de la *génération Objet*), interrogés sur leurs besoins en matière de structuration du code, demandaient presque invariablement un support du modèle objet

de ce nom, ni *profiler*. Il est vrai que les caractéristiques de SCOL ne facilitent pas la construction de ces outils (la perte des informations de type à l'exécution ne facilite pas la construction d'un debugger, même si ce n'est pas impossible à réaliser). Ce sont pourtant le minimum d'outils indispensables à la réalisation d'applications importantes, car ils permettent de découvrir le plus tôt possible dans le cycle de développement les éventuelles failles d'une application, et donc de gagner du temps à terme : sans *profiler*, il n'est pas rare de ne découvrir qu'une fonctionnalité critique est inefficace qu'au moment de la mise en service de l'application, d'autant plus que si celle-ci est de type monde virtuel, la faiblesse ne sera souvent découverte qu'à partir d'un certain nombre de clients connectés.

De plus, dans le monde de l'industrie et plus particulièrement dès qu'il s'agit d'internet et de multimédia, les développeurs sont désormais habitués à disposer d'environnements de développement intégrés, munis d'interfaces graphiques particulièrement sophistiquées, qui encadrent tout le cycle de vie de l'application : Rational Rose [4], qui permet de formaliser le cycle de développement suivant une méthodologie objet, simplifie le travail de conception et de codage (en produisant les squelettes de code correspondant à la conception, et inversement lorsque des modifications du code se répercutent sur un diagramme de classe). Microsoft Visual Studio est pourvu d'un debugger particulièrement confortable. De nombreux outils, à base de *drag and drop* permettent de générer rapidement une interface graphique (Ilog Views [13] par exemple). Un langage qui n'est pas à la hauteur de ces nouveaux standards de confort et d'outillage a souvent tendance à être perçu comme un langage-jouet pas vraiment candidat pour la réalisation d'applications industrielles et le SCS de la technologie SCOL, simple outil d'intégration, n'a pas pu rivaliser.

Cette nouvelle donne montre à quel point il est important que la question des outils de développement soit abordée au plus tôt et peut-être même influe sur la conception d'un langage voué à être utilisé pour d'importantes applications industrielles, afin de prévoir son adaptation aux environnements déjà existants, ou de faciliter la conception de nouveaux outils.

3.3. Langage évolué, prototypage et application efficace

L'utilisation d'un langage évolué (nous utiliserons ici ce terme pour désigner un langage qui abstrait la gestion des ressources systèmes et réseau) pour le développement d'une application pour laquelle ce langage est spécialisé semble être un cas de figure idéal. Les arguments de simplicité et de rapidité du développement semblent alors quasiment inattaquables, tellement ce type de langage facilite le prototypage d'une application. Pourtant, ce qui semble être un avantage peut vite se transformer en bombe à retardement au cœur du projet.

En effet, l'abstraction du langage ne fait pas que rendre la complexité des problèmes plus facile à manipuler. Dans une certaine mesure, il la masque : les problèmes complexes semblent tellement plus faciles à décrire, qu'on a souvent tendance à oublier qu'ils existent. L'opinion communément répandue est qu'il est assez délicat de développer en C, et beaucoup plus simple d'utiliser un langage évolué, puisque ce dernier prend en charge une partie des problèmes, et que les progrès des compilateurs sont tels que cela n'affecte plus l'efficacité d'un programme [12]. Prenons un exemple très simple et un peu caricatural : la gestion de la mémoire.

Le langage SCOL est muni d'un *garbage collector*, comme de nombreux langages modernes. Sauf pour certains cas bien particuliers [5], la mémoire n'est donc plus gérée explicitement par le développeur, souvent habitué par C et C++ (ces langages plus proches de la machine sont prépondérants dans l'industrie du jeu vidéo) à allouer et désallouer à la main au risque d'une fuite mémoire ou pire. Le libérer de cette activité, sans l'informer de la manière dont un *Garbage Collector* fonctionne revient souvent à lui donner l'impression qu'il n'est absolument pas responsable de le gestion mémoire. Tout naturellement, le développement ne sera pas orienté vers une utilisation optimale de cette ressource, d'autant plus qu'en SCOL, la facilité d'utilisation des listes et n-uplets a tendance à inciter le

[5] pour l'API 2D et le moteur 3D, qui utilisent de manière intensive des ressources système graphiques, les allocations et désallocations se font explicitement

fonction de codage d'une donnée de type `myStruct` en chaîne de caractère :

```
myStructToString: fun [myStruct] S
```

fonction de codage d'une liste de structures en chaîne de caractères :

```
fun myStructListToString(sList)=
  if sList == nil then
    nil
  else
    let l->[h nxt] in
       // fonction de concaténation de deux chaînes de caractères
       strcat (myStructToString h) (myStructListToString nxt)
;;
```

FIG. 6 – Sérialisation naïve d'une liste de structures

fonction de codage d'une donnée de type `myStruct` en chaîne de caractère :

```
myStructToString: fun [myStruct] S
```

fonction de codage d'une liste de structures en chaîne de caractères :

```
fun myStructListToString(sList)=
   // strcatn: fonction de concaténation d'une liste de chaînes de caractères
   // map_list: applique la fonction r\'ef\'erenc\'ee (premier argument)
   // à chaque élement de la la liste (deuxieme argument)
   strcatn (map_list @myStructToString sList)
;;
```

FIG. 7 – Sérialisation d'une liste de structures en optimisant l'utilisation de la mémoire

développeur pressé à en allouer un grand nombre. Et si, finalement, l'efficacité de l'application est mise en cause, la responsabilité sera tout aussi naturellement attribuée à la technologie.

Détaillons un exemple fréquemment rencontré dans le code SCOL : l'absence de sérialisation pour les structures du langage obligeait les développeurs à encoder eux mêmes les données à transmettre par le réseau en chaînes de caractères, et les solutions qu'ils retenaient étaient celles qui leur semblaient provoquer le moins de parcours de liste. La figure 6 présente un exemple naïf d'un tel traitement, qui se révèle bien plus coûteux en utilisation de la mémoire que le code présenté par la figure 7 si on étudie un peu les fonctions de concaténation de chaînes de caractères `strcat` et `strcatn`. En effet, chaque appel à `strcat` signifie une allocation de la taille du résultat ce qui donne un algorithme en N^2 pour la consommation de mémoire par rapport à la taille de la liste. La fonction `strcatn`, elle, ne provoque qu'une allocation pour la liste de départ. L'algorithme présenté par la figure 7 est donc en N pour l'occupation mémoire et parfois bien mieux adapté à la situation.

Développer une application efficace en terme de performances dans un langage évolué nécessite autant de maîtrise de ce langage que développer en C : la différence principale est qu'en C, il est relativement difficile à un développeur expérimenté d'écrire une application inefficace, puisqu'à devoir manipuler directement les ressources de la machine, on est plus sensibilisé à leur utilisation. On retrouve là un problème similaire à celui signalé il y a déjà longtemps par Gabriel dans [10] : comme LISP [16], SCOL permet un prototypage extrêmement rapide d'une application, mais ce style de développement n'est absolument pas adapté à la production d'une application efficace. En effet, il y a toujours un moment où il sera nécessaire de se pencher sur l'utilisation de la mémoire par quelques fonctionnalités critiques : l'implémentation d'*Ensemble*, une bibliothèque de communication réseau écrite en Objective Caml nous donne un autre exemple des avantages et inconvénients d'un GC, la réflexion ayant abouti à revenir à une gestion explicite de la mémoire dans certains cas bien précis [8]. Ce problème de prise de conscience pourrait être simplement résolu, comme le propose Manuel Serrano [19] par l'utilisation d'un *debugger* ou d'un *profiler*.

Cependant, compte tenu des contraintes de l'industrie du jeu vidéo, l'utilisation d'un langage autorisant un prototypage rapide a un grand interêt : c'est une industrie où tout va très vite, et où pour réussir, il faut être capable de montrer très rapidemment dans le cycle de vie de l'application à quoi ressemblera le produit final. L'urgence permanente dans cette industrie implique néanmoins également que peu de temps sera accordé au perfectionnement d'un code inefficace mais ne provoquant pas d'erreur. Nous continuons donc à défendre l'utilisation d'un langage évolué, qui semble d'ailleurs commencer à faire son chemin dans cette industrie [6], mais il nous semble évident qu'un effort de formation des développeurs est encore à faire : un ingénieur sortant de l'école n'est peut-être pas encore assez sensibilisé à l'utilisation de tels langages. Ceci est bien évidemment tout à fait inutile si on ne laisse pas aux développeurs le temps de revenir sur un prototypage rapide d'une fonctionnalité de l'application dès que celle-ci demande un certain niveau d'efficacité. Or, l'utilisation d'un langage évolué pour la réalisation d'une application est souvent prétexte à trop réduire les délais de production et d'apprentissage : il reste encore à faire prendre conscience aux chefs de projets et autres décideurs, que si gain de temps et de fiabilité il y a, plus l'application aura besoin de performances et plus les délais de production se rapprocheront de ceux qu'on aurait observé en utilisant une technologie moins abstraite.

4. Conclusion

Le langage SCOL s'est révélé très efficace pour le développement d'applications de type communautés virtuelles à petite échelle, et ceci grâce à son expressivité, au chargement dynamique, et à certaines autres caractéristiques propres à sa famille de langages, dont les plus évidentes sont sans doute la sûreté du typage statique et de l'inférence de type. De plus, contrairement à ce qui est reproché à beaucoup de langages fonctionnels [22], SCOL était muni de toutes les bibliothèques facilitant son utilisation pour les applications visées.

Mais la façon dont il a été pensé ressemble plus à la vision d'un internet communautaire en 3D, dans lequel on se déplacerait, en rencontrant les autres utilisateurs, d'un petit monde virtuel à l'autre, qu'à celle d'une application massivement multi-utilisateurs ayant des impératifs de performances. Les problèmes de facteur d'échelle qui se posent lorsqu'on cherche à réaliser une application telle qu'un jeu massivement multi-joueurs n'ont pas été pris en compte dans la conception initiale de la technologie. SCOL manquait également d'un véritable environnement de développement, ainsi que de constructions du langage propres à faciliter l'exercice du génie logiciel : s'il est possible de s'en passer lorsque les projets sont peu importants, il devient extrêmement difficile de contrôler le développement d'un projet impliquant plus d'une demi-douzaine de développeurs si on ne dispose pas de tels outils.

SCOL a également souffert de la perception souvent inexacte qu'a le monde industriel des langages fonctionnels : depuis longtemps maintenant, leur efficacité n'est plus un problème et il est possible d'écrire du code performant. Il n'en reste pas moins que cette famille de langages facilite un style de codage rapide et sûr, mais qui ne produit pas toujours l'exécutable le plus efficace. Nous avons constaté qu'un travail d'éducation reste à faire pour fournir aux développeurs des méthodes de conception et de développement adaptés à l'utilisation des langages applicatifs, et qu'il reste à faire disparaître les vieux mythes qui ont décidément la peau dure.

Finalement, le principal reproche qu'on puisse faire à SCOL est qu'il n'a jamais réussi à arriver à maturité. Pourtant, un projet de collaboration avec deux laboratoires de recherche universitaires (PPS et le LIP6), mis à mal par la faillite de Cryonetworks, promettait des évolutions qui auraient sans doute satisfait les utilisateurs du langage. Les extensions prévues comprenaient notamment un système de modules simples et le passage à une machine virtuelle multi-tâches pour offrir un support à un système multi-agents [18, 3]. Evoquée lors de la fermeture de Cryonetworks, la possibilité que le langage SCOL passe en licence *Open Source* dans les mois qui viennent permet cependant d'espérer que ces évolutions voient le jour dans un avenir proche...

Références

[1] Xavier Leroy, (avec Jacques Garrigue, Didier Rémy et Jérôme Vouillon). The Objective Caml system release 3.06, Août 2002. Institut National de Recherche en Informatique et Automatique.

[2] Daniel de Rauglaudre. Camlp4 - Reference Manual release 3.06, Août 2002. Institut National de Recherche en Informatique et Automatique.

[3] Rapport d'avancement du projet EDICA, livraison du lot 1, 2002. `www.industrie.gouv.fr/rntl/`.

[4] Khawar Ahmed. Building Reusable Architecture using Rational Rose Frameworks, 2000. `www.rational.com/products/rose/`.

[5] Ken Arnold, James Gosling, and David Holmes. The Java Programming Language. *Addison-Wesley*, troisième edition, Juin 2000.

[6] Jason Asbahr. Python for Massively Multiplayer Virtual Worlds. *Proceedings of the Ninth International Python Conference* `www.asbahr.com/paper2html/paper2.htm`.

[7] Emmanuel Chailloux, Pascal Manoury, et Bruno Pagano. Développement d'Applications avec Objective Caml. *O'Reilly, Paris, Avril 2000.*

[8] Mark Hayden. Distributed communication in ML *Journal of Functional Programming*, 10(1) :91-120, 2000.

[9] Joe Armstrong. Erlang - A survey of the language and its industrial applications *INAP'96* 16-18, October 1996.

[10] Richard P. Gabriel. Lisp : Good News, Bad News, and How to Win Big. *A.I. Expert*, pages 31–39, Juin 1991.

[11] Sylvain Huet. Le Langage SCOL : Tutorial (Version 3.0). Disponible sur le site de Cryonetworks, `www.cryonetworks.com`.

[12] John Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2) :98–107, 1989.

[13] ILOG. ILOG Views : White Paper, Novembre 1999. `www.ilog.com/products/views/`.

[14] Rodger Lea, Yasuaki Honda, Kouichi Matsuda, and Satoru Matsuda. Community Place : Architecture and performance. In *Proceedings of the Second Symposium on Virtual Reality Modeling Language*, pages 41–50, 1997.

[15] Xavier Leroy and Pierre Weis. Le Langage Caml. *Dunod*, deuxième edition, 1999.

[16] John L. McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM*, 3(4) :184–195, 1960.

[17] OMG. OMG Universal Modeling Language Specification (Version 1.4), Septembre 2001. `www.uml.org`.

[18] Nadine Richard. Description de comportements d'agents autonomes évoluant dans des mondes virtuels. *Thèse de doctorat*, ENST, Paris, Octobre 2001.

[19] Manuel Serrano. Vers une programmation fonctionnelle *praticable. Thèse d'habilitation à diriger la recherche*, Université de Nice, 2000.

[20] Manuel Serrano. Bee : an integrated development environment for the Scheme programming language. *Journal of Functional Programming*, 10(4) :353–395, Juillet 2000.

[21] The VRML Consortium Incorporated. The Virtual Reality Modeling Language, 1997. Standard International ISO/IEC 14772-1 :1997.

[22] Philip Wadler. Why no one uses functional languages. *SIGPLAN Notices*, 33(8) :23–27, 1998.

[23] David L. Parnas. On the criteria to be used in decomposing systems into modules. *CACM*, 15(12) :1053-1058, 1972.