RENPAR'16 / CFSE'4 / SympAAA'2005 / Journées Composants
Le Croisic, France, 5 au 8 avril 2005

Le projet Isaac : une alternative objet de haut niveau pour la programmation système

Benoît Sonntag

LORIA, UMR 7503
(INRIA - CNRS)
Campus Scientifique, BP 239,
54506 Vandœuvre-lès-Nancy Cedex, FRANCE
Email: bsonntag@loria.fr

Advanced System Technology
STMicroelectronics, Grenoble Lab.
12, rue Jules Horowitz
30019 Grenoble Cedex, FRANCE

Résumé
Le marché fleurissant de l'informatique embarquée impose une réutilisabilité croissante du code existant tout en ayant de fortes contraintes matérielles. Aussi, il est nécessaire de s'adapter de plus en plus rapidement à ce matériel hétéroclite. Nous pensons que l'utilisation de langages de haut niveau, en particulier les langages objets, apporte certaines réponses à une adaptabilité rapide au sein des systèmes d'exploitation et de la couche applicative grandissante. L'évolution des langages informatiques échappe encore à la programmation système. Ici, nous dénonçons l'utilisation quasi-systématique du langage C et apportons une réelle alternative.

Mots-clés : Isaac, Système d'exploitation orienté objet, Lisaac, adaptation matériel, modèle à prototypes

1. Introduction

Avec l'évolution des technologies, l'informatique a pris une place de plus en plus importante dans tous les domaines du quotidien : transports, télécommunications, appareils électroménagers, médecine, etc. La miniaturisation des composants électroniques a permis l'intégration de systèmes informatiques puissants dans toutes sortes d'appareils. La diversité des composants matériels permet également de pouvoir ajouter facilement de nouvelles fonctionnalités à ces appareils, que ce soit par l'ajout d'une caméra numérique, de carte mémoire externe ou d'un micro. Si cela permet de rendre de nombreux services dans la vie courante, la tâche des concepteurs est loin d'être simplifiée. En effet, la gamme des appareils embarqués étant sans cesse grandissante, et très hétéroclite, les systèmes d'exploitation qui pilotent ces architectures doivent être souvent développés au cas par cas, ou tout au moins en grandes parties. Depuis moins de 5 ans, nous assistons dans la téléphonie mobile ou dans l'automobile à une explosion d'intégration logiciels divers particulièrement lourds pour ce type d'appareil. Les compétences d'une équipe de développement système ayant pour but de rendre opérationnel un produit ne sont plus suffisantes dans ce domaine. Ces équipes doivent prendre en charge de nouveaux besoins comme l'intégration de *codec*[1] vidéo, de jeux vidéo ou de plateforme internet et Java. Évidemment, la réutilisabilité du code est alors fortement sollicitée. Néanmoins dans la pratique, l'équipe doit maitriser en profondeur ce code pour l'adapter aux différentes contraintes matérielles ayant des ressources encore trop limitées. Le code est essentiellement écrit en C pour des raisons d'efficacité, de portage (compilateur C présent sur toutes les architectures), d'intégration avec le système (gestion de bas niveau, proximité du matériel), mais aussi pour des raisons historiques. Ici, nous dénonçons une réutilisabilité peu fiable, parfois hasardeuse d'un code C non maîtrisé. Cela entraine de sérieux problèmes en terme de coût de développement, de complexité des applications développées, de maintenance, d'évolutivité et de fiabilité. En particulier dans la téléphonie mobile, la sonnette d'alarme est déjà donnée par les utilisateurs : de nombreux services et de fonctionnalité assemblés trop attivement par les constructeurs, nous parlons maintenant du *bug* au quotidien.

[1] Encodeur/Décodeur vidéo

Sans prétendre détenir une solution à ces problèmes, notre approche apporte de sérieux avantages que nous présentons dans cet article. Après un bref état de l'art en section 2 sur les modèles et les outils des systèmes d'exploitation actuels, cet article s'articule essentiellement en deux parties. La première partie développe notre modèle Isaac dans son ensemble. Nous présentons en section 3 notre projet qui s'inscrit comme une continuité de l'évolution informatique entre langage et système d'exploitation. La section 4 donne lieu à une brève présentation de notre langage Lisaac adaptée à la programmation système.
La seconde partie est une mise à l'épreuve concrète de notre approche. Nous montrons en section 5 l'utilisation de notre modèle pour une meilleure réutilisabilité et spécialisation du code par héritage statique. Dans un exemple plus précis concernant la gestion vidéo, nous développons en section 6 l'adaptabilité du matériel par héritage statique et liaison dynamique. Pour ce même exemple, nous développons en section 7 une adaptatibilité dynamique du matériel par utilisation de l'héritage dynamique. Pour finir, nous laissons place en section 8 a une expérience concrète de portage de notre système d'exploitation Isaac, sans oublier un ensemble de tests de performance concernant notre compilateur Lisaac en section 9, avant de conclure en section 10.

2. État de l'art

2.1. Les systèmes modulaires

Il existe de nombreux systèmes d'exploitation reflétant une certaine modularité et adaptabilité rapide au matériel. Ici, nous sélectionnons simplement un représentant particulièrement pertinant de chaque catégorie actuellement présent.

L4 : le système à micronoyau

Le projet L4 est une spécification de système d'exploitation à noyau restreint (micronoyau). Il existe plusieurs implantations de cette spécification. La philosophie de L4 est de sortir du noyau un maximum de services pour les mettre au niveau utilisateur (*user level*). Par exemple, le gestionnaire de mémoire ou l'ordonnenceur de tâches ne font pas partie du noyau, leurs politiques de fonctionnement sont donc facilement interchangeables comme n'importe quelles applications. Les IPCs sont au cœur de la communication entre le noyau et les applications. L'une des particuliarités de cette approche réside en la robustesse de ce noyau minimaliste et rapidement portable. Les implantations de L4 sont écrites en langage C++, mais il n'y a aucune utilisation des concepts objets. L'héritage et la liaison dynamique sont absents pour des raisons d'efficacité.

Merlin : le système à objets purs

Le projet *Merlin* (`http://www.lsi.usp.br/ jecel/merlin.html`) de l'université de Sao Paulo est proche conceptuellement de notre projet. Il utilise le langage Self à base de prototypes et intègre ses concepts à l'ensemble du système. Ici, la flexibilité et l'expressivité de l'ensemble du système (système d'exploitation et application) sont leur cheval de bataille [6]. Une étude sur les qualités de leur modèle pour l'utilisation de machines parallèles a été aussi réalisée [5], [4]. Il devrait fonctionner sur une machine virtuelle adaptée ou sur un système d'exploitation plus conventionnel de type Unix. Malheureusement, le projet a l'air d'être peu développé, un début d'interprèteur Self (appelé *Tiny*) fonctionne actuellement sous Linux. Les protections système (protection inter-objets) et les performances d'exécution ne sont pas abordées dans ce projet.

Camille : le système d'exploitation ouvert pour carte à microprocesseur

Le système *Camille* est destiné à des architectures très légères (ex. : *Smart Card*) ayant de fortes contraintes temps réels. Son noyau similaire au MIT *Exo-Kernel* est adaptatif, rechargeable, tout en conservant ses contraintes temps réels [2], [1].

2.2. Les outils pour les systèmes d'exploitation

Think : *THink Is Not a Kernel*

Think est un canevas et une méthodologie d'aide à la construction de système d'exploitation à base d'un nano-noyau et d'une bibliothèque de composants. Ces travaux sont menés dans le cadre du projet

Fractal [9], [3]. Le noyau, les drivers et les applications sont découpés en différentes entités de code appelées composants. Chaque composant est autonome et peut-être remplacé dynamiquement durant l'exécution du système. Un composant contient du code ou un ensemble de composants. La communication entre les composants est assurée par une interface décrite dans une IDL (*Interface Définition Language*) syntaxiquement proche du langage Java. L'un des avantages majeurs de ce modèle est dans la description explicite des services disponibles par un composant, et des services nécessaires au bon fonctionnement de ce composant. Ce modèle à composants n'est pas lié à un langage de programmation, néanmoins les composants restent malheureusement écrits en langage C. Les composants peuvent être considérés comme des objets, mais l'héritage et la liaison dynamique avec algorithme de *lookup*, essence même de la technologie objet, restent encore primitif dans ce modèle.

Flux OS Toolkit

L'outil *Flux OS Toolkit* développé par l'Université de *Utah* offre un ensemble d'interfaces bas niveau (`http://www.cs.utah.edu/flux/osk it/ht ml/os kit-ww w.htm l`). Il est utilisé dans de nombreux systèmes d'exploitation expérimentaux pour éviter le développement fastidieux des drivers de périphérique. Programmé en langage C, il suit la norme POSIX et offre des outils pour la gestion de la mémoire, l'ordonnancement et la prise en charge des périphériques (disques durs et disquettes, systèmes de fichiers, réseaux, claviers, souris, vidéo, ...). Actuellement, cet outil est limité à l'architecture *Intel x86*, son portage sur *Strong ARM* semble être en cours.

Devil

Devil est un IDL (Langage de définition d'interfaces) dédié à la spécification de l'interface de programmation des contrôleurs de périphériques [8], [10]. À l'aide de celui-ci, il devient possible de programmer les gestionnaires de périphériques avec plus de sécurité et de lisibilité qu'avec le langage C. Il permet de spécifier clairement les masques de bits dans la lecture et l'écriture des ports matériels. Agrémenté d'un typage précis, il réalise des vérifications de la sémantique du code, lors de la compilation, qui étaient impossibles avec le système de typage trop laxiste du langage C. Un bon nombre de vérifications permet de programmer avec plus de sécurité, comme l'absence d'omission, l'absence de redéfinition et l'absence de définition conflictuelle. À l'image de la programmation par contrat, certaines vérifications et tests sont réalisés dynamiquement durant l'exécution. Après validation du code, les tests peuvent être désactivés pour produire par compilation un code C efficace.

3. Présentation du projet Isaac

Ce projet défend l'utilisation des concepts objets et d'un langage de haut niveau pour la programmation système. À l'heure actuelle, l'intégration de l'objet reste encore trop souvent au niveau conceptuel, et surtout reléguée au niveau de l'utilisation de langage pour les applications. Il est nécessaire, pour répondre aux problèmes évoqués en introduction, d'aller plus loin et d'intégrer les concepts objets au cœur des systèmes d'exploitation. L'intérêt de cette intégration est multiple : modularité, expressivité, adaptabilité, sécurité mais également légèreté et élégance du code.

3.1. L'évolution des langages informatiques

L'innexistance de langage objet de haut niveau pouvant prétendre remplacer le langage C pour la programmation système est notre principal problématique. Java et C++ sont considérés, à juste titre, comme des langages objets non purs ayant une intégration faible, voir de bas niveau des concepts objets. Les programmeurs système utilisant actuellement le C++ pour garder la proximité du C avec le matériel n'exploitent pas la puissance de la technologie objet pour des raisons de performances. L'utilisation de VFTs[2] pour la liaison dynamique dans les compilateurs C++ provoque une baisse notable des performances. Le langage *SmallTalk*, référence incontestée du domaine objet, reste un langage interprété n'ayant aucune prédisposition pour la programmation système. Le langage Eiffel, considéré comme représentatif de l'âge d'or des langages à classes apporte beaucoup d'avantages dont la programmation par contrat et de bonne performance d'exécution avec l'implantation du compilateur SmartEiffel

[2] Virtual Function Table

(http://SmartEiffel.loria.fr). Mais ce langage reste peu adapté aux contraintes de la programmation de bas niveau.

Le projet Isaac est avant tout un nouveau langage objets de haut niveau adapté à la programmation système : le langage Lisaac[11]. Ce langage peut-être utilisé pour l'écriture de gestionnaire de périphériques (*drivers*), d'application ou d'un système d'exploitation complet. Pour étayer nos propos, nous l'avons utilisé pour concevoir le système d'exploitation Isaac reposant entièrement sur ces concepts.

Notre projet s'inscrit parfaitement dans l'évolution informatique depuis sa création (cf. fig. 1). En effet, nous observons deux phénomènes d'évolution, l'un touchant les langages, l'autre les systèmes d'exploitation.

Pour réduire le nombre de lignes de code et gagner en clarté, les langages informatiques n'ont cessé de se structurer. De l'assembleur, nous avons vu la création de langage plus "évolué" comme le Fortran66. Assez proche du langage machine par son côté très linéaire du code, le Fortran permet en quelques lignes d'effectuer un traitement complexe. Puis nous avons structuré le code linéaire en procédures et fonctions comme dans le langage C. L'utilisation de fonctions paramétriques permet une réduction et une généralisation du code. L'arrivée des concepts objets peut être vue comme un regroupement de procédures et fonctions en une entité appelée objet. L'héritage et les autres atouts des concepts objets sont encore là pour faciliter la réutilisabilité et la réduction du code. On peut considérer le découpage en objet comme une structuration macroscopique du code. Nous assistons donc à un phénomène de regroupement du code en entités de plus en plus grandes et abstraites permettant une vision plus macroscopique d'une application complexe.

Pour les systèmes d'exploitation, nous avons le phénomène inverse. Pour les mêmes raisons de réutilisabilité et de réduction du code, les systèmes n'ont cessé d'augmenter leur fragmentation. L'utilisation de librairies externes, l'insertion de gestionnaire de périphériques (*drivers*) découpent le système en parties indépendantes et communicantes. Ici, on parle plus de flexibilité et de communication avec le système et les applications. Mais il est surprenant de constater que l'évolution des langages et des systèmes d'exploitation suivent une progression différente vers les mêmes objectifs. La jonction des deux disciplines au niveau conceptuel paraît souhaitable.

3.2. Isaac/Lisaac : le modèle à prototypes

Dans l'ensemble de la famille des concepts objets, nous avons choisi le concept à prototypes. Les langages à prototypes représentent l'aboutissement et la généralisation des concepts objets. À la différence d'une classe, la description d'un prototype est déjà un objet vivant pouvant directement être utilisé (cf. fig. 2). L'instanciation d'une classe est remplacée par le clonage d'objet. La description d'un prototype est donc en réalité un objet modèle vivant, contrairement à une classe qui est une représentation statique, donc non modifiable, d'une catégorie d'objets.

Le modèle objet à prototypes est plus souple et plus uniforme que celui à classe, mais surtout, il s'adapte bien mieux à une représentation concrète de l'univers en objets.

Une autre différence marquante des prototypes par rapport aux classes se situe au niveau de l'héritage. Comme l'illustre la figure 2, une classe hérite d'une autre classe et l'instance d'une classe donne lieu à un seul objet représentatif de sa description, héritage inclus. Alors que dans le monde des prototypes, tout objet hérite d'autres objets. Ainsi, le parent d'un objet est un objet à part entière et plusieurs objets peuvent hériter d'un même objet physique (ce qui facilite le partage en mémoire). Avec les prototypes, le parent étant une entité physiquement séparée, il devient possible de changer de parent durant l'exécution (héritage dynamique). Cette possibilité apporte une flexibilité et un dynamisme supplémentaires qui devient un atout dans la programmation en général. Dans la suite de cet article, nous mettrons en avant cet avantage dans le cadre de la programmation système.

4. Présentation du Langage Lisaac

4.1. Vue générale

Le langage Lisaac est fortement inspiré de Self [12] et d'Eiffel [7]. Il hérite de la flexibilité et du côté minimaliste des langages à prototypes comme Self (héritage dynamique, absence de conditionnel et de boucles ou itérations, ...), tout en s'accaparant la sécurité du typage statique, la généricité (type paramétrique) et de la programmation par contrat d'Eiffel. Dans le cadre de cet article, nous ne pouvons

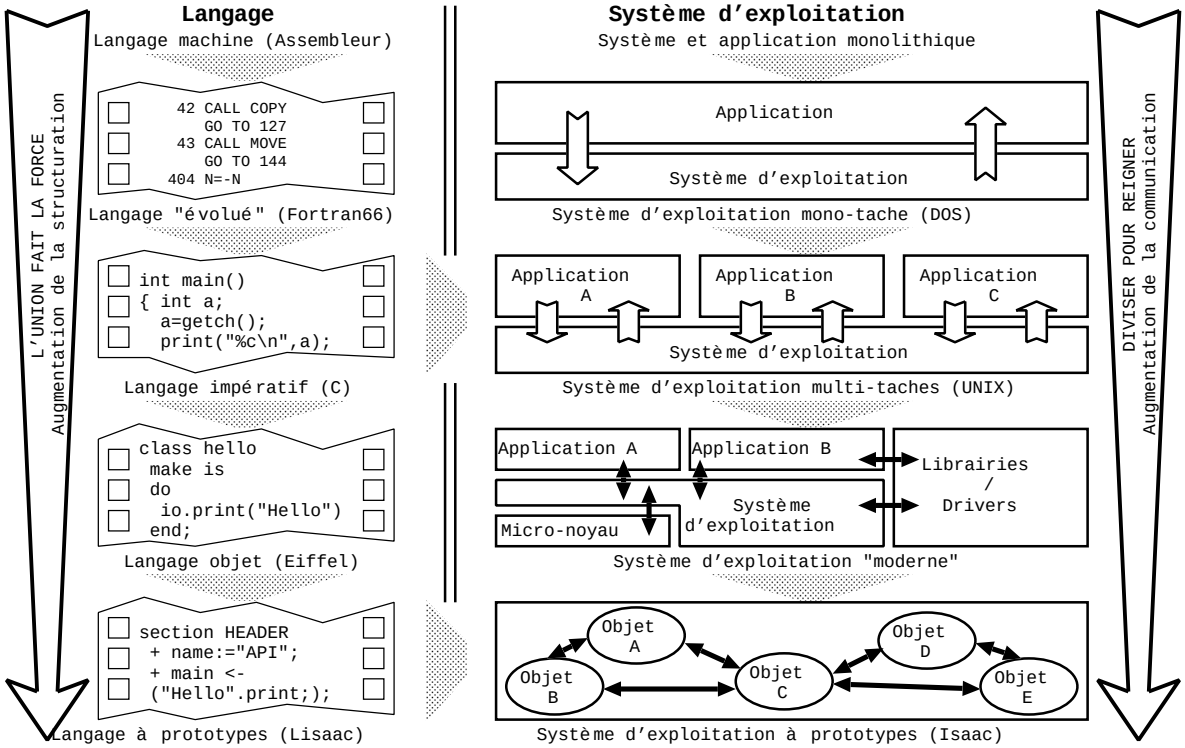


Figure 1: L'évolution informatique

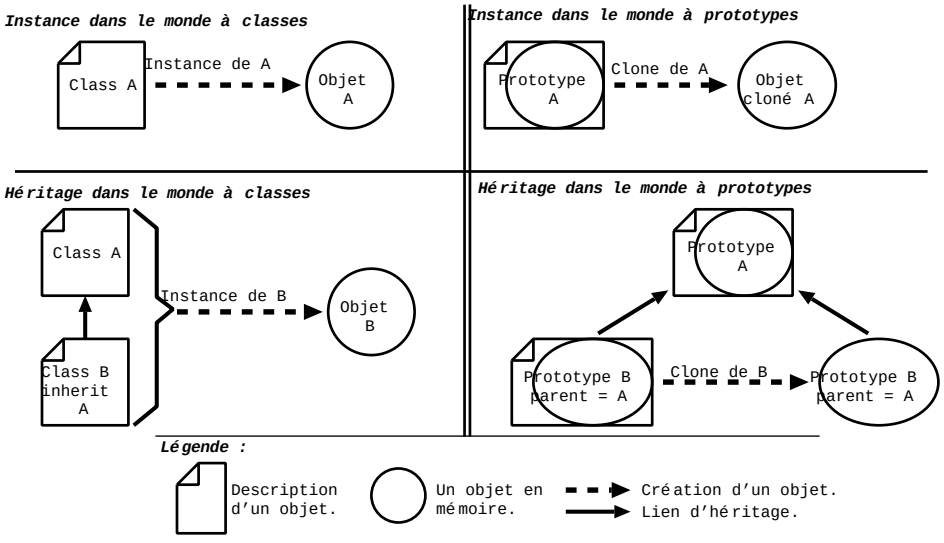


Figure 2: Différences Classes / Prototypes

pas détailler la sémantique de ce langage. Vous trouverez le manuel de références disponible sur le site `www.IsaacOS.com` .
Le compilateur Lisaac produit du code `C ansi` pour rester portable et profiter des optimisations de bas niveau présentes dans les compilateurs C actuels. De plus, cela nous permet de nous concentrer sur des optimisations de haut niveau concernant l'héritage dynamique, la liaison dynamique et la spécialisation de code par prédiction de type. Les techniques de compilation que nous utilisons sortent du cadre de cet article, néanmoins les performances, vecteurs essentiels de la validité de notre approche seront abordées en seconde partie.
Lisaac a été pensé pour répondre à certaines contraintes qu'impose la programmation de bas niveau.

4.2. Le *mapping* d'objets

Lisaac permet d'appliquer directement un objet sur une structure figée par le matériel ou une spécification quelconque. Cette structure est alors décrite dans une section *mapping*. Ce type de section oblige le compilateur à conserver l'ordre et l'intégralité des champs de données pour permettre de poser la structure d'un objet sur une structure interne au système d'exploitation. Ce mécanisme est particulièrement utile pour traiter les tables d'une composante physique (mémoire, disque, ...) comme un objet en bijection avec leur représentation en mémoire.

4.3. Les interruptions matérielles

La section INTERRUPT est réservée pour matérialiser les exceptions et les interruptions matérielles. Seules les procédures sont autorisées dans cette section afin d'associer leurs codes avec des interruptions du processeur. La particularité de ces procédures se situe au niveau du code généré en entête et sortie de code. Leurs invocations sont asynchrones, il est donc nécessaire de prendre les précautions qui s'imposent afin d'assurer la cohérence d'un processus brusquement interrompu. Le code doit être hors contexte d'exécution et le compilateur prend en compte les effets de bord possibles en positionnant la primitive `volatile` sur les données sensibles lors du code C généré.

4.4. Intégration du langage C

La communication entre le code Lisaac et le code C est facilitée dans les deux sens. Celle-ci est particulièrement importante pour l'inter-opérabilité entre un code C déjà existant et un nouveau code Lisaac. En entête de la description d'un prototype Lisaac, nous pouvons définir le type C correspondant à l'objet (cf. (1) dans l'exemple ci-dessous). Le besoin de l'intégration d'un code C nécessaire au bon fonctionnement du prototype peut-être aussi décrit en entête (cf. (2) dans l'exemple ci-dessous).
Par ailleurs, les cotes inverses (`) permettent d'incorporer du code C n'importe où dans un source Lisaac. À l'intérieur de ce code C, l'accès aux variables Lisaac est rendu possible par l'utilisation du symbole '@' précédent le nom de la variable. Un type de retour de ce code peut aussi être précisé (cf. (3) dans l'exemple ci-dessous).
Dans l'exemple ci-dessous, les objets de type DOUBLE sont directement traduits par le compilateur avec le type `double` du langage C. L'utilisation de cet objet dans un programme Lisaac ajoute le code `#include <math.h>` dans le code cible généré par le compilateur. L'opérateur binaire de soustraction est défini en Lisaac par un *external* C réalisant cette opération.

```
section HEADER
  * name := DOUBLE;

  - type := `double`;     // (1) Un objet DOUBLE est représenté par un 'double' C.
  - external := `#include <math.h>`;     // (2) Ajout de ce code si DOUBLE est nécessaire.
section INHERIT
  - parent_numeric:NUMERIC := NUMERIC;
section PUBLIC
  // L'opérateur binaire '-' d'associativité gauche et de
  // priorité 80 est réalisé par l'opérateur '-' du langage C.
  - '-' left 80 other: SELF :SELF <- `@self - @other`:SELF;     // (3)
```

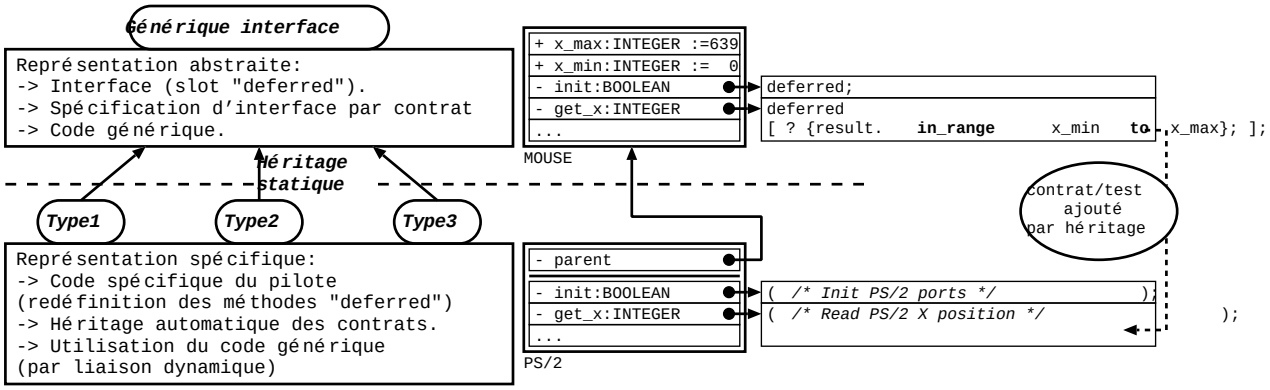


Figure 3: L'héritage statique

Dans le sens inverse, les méthodes décrites dans une section EXTERNAL ne seront pas expansées en ligne (*inlining*) par le compilateur et conserveront leurs profils exacts de déclaration. Ainsi, ce type de méthode peut-être directement appelé dans un source écrit en C. Cela permet notamment d'écrire des librairies en Lisaac utilisables dans un source C.

5. Réutilisabilité et spécialisation par héritage statique

L'utilisation des concepts de haut niveau de la technologie objet nous permet de tirer partie de l'héritage statique entre les prototypes du système (cf. fig. 3). L'héritage statique n'est plus simplement utilisé au niveau de la conception logicielle, mais devient un véritable outil et une caractéristique propres à la conception de la programmation système.
En utilisant de manière optimale ce mécanisme d'héritage statique, nous limitons au maximum le nombre de lignes de code réellement spécifiques à une architecture donnée. Le prototype spécifique réellement pilote d'un périphérique, hérite d'un prototype modèle, servant de description d'interface et de container de code générique. Ainsi, la partie spécifique est parfaitement détachée du code générique. L'association des deux est réalisée par héritage et par liaison dynamique.
L'intégration de la programmation par contrat dans le langage Lisaac permet de définir des contraintes (contrat) sur cette interface (au niveau du parent). Le non respect des contraintes automatiquement héritées dans l'objet fils (objet spécifique à une architecture) provoque une violation de contrat durant l'exécution. Cet outil de mise au point du code a déjà fait largement ses preuves dans le monde applicatif. Dans l'univers délicat de la programmation système où les effets de bord sont importants, une bonne utilisation de ce mécanisme permet de détecter une erreur le plus tôt possible et de réduire les coûts de développement. La fiabilité du code final est ainsi accrue par ces contrats, même si cela ne constitue en rien une garantie absolue.

6. Adaptabilité du matériel par héritage statique et liaison dynamique

Prenons comme exemple l'élaboration d'un gestionnaire vidéo. En principe, le mode vidéo d'un ordinateur est une matrice de pixels en mémoire, appelée plus communément *bitmap*.
Le gestionnaire vidéo est matérialisé par un prototype spécifique VIDEO pilotant le contrôleur vidéo de la machine. Il a pour objectif d'initialiser le mode graphique, d'offrir des informations sur celui-ci et de permettre l'affichage de pixels sur l'écran. Pour faciliter l'utilisation du contrôleur vidéo, l'ensemble des méthodes applicables sur une *bitmap* (ligne, polygone, ...) doit être accessible par VIDÉO. Naturellement, ce prototype hérite d'un exemplaire du prototype BITMAP qui implante ces méthodes. Ce prototype parent BITMAP représente physiquement la matrice de l'écran. Par cet héritage statique, l'ajout d'une méthode dans BITMAP est automatiquement accessible par l'objet VIDEO. À la manière des poupées russes, les méthodes complexes de BITMAP utilisent pour leur implantation d'autres méthodes

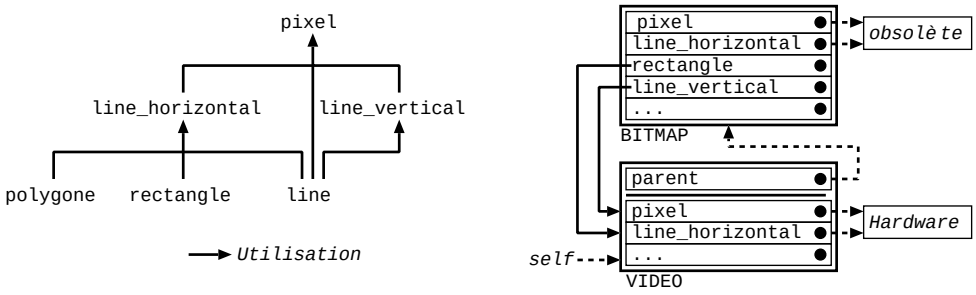


Figure 4: Dépendance des méthodes de BITMAP (à gauche) et l'utilisation de l'héritage statique pour les performances vidéo (à droite).

plus simples de BITMAP.
Sur la figure 4 à gauche, la méthode qui permet d'afficher une ligne horizontale (`line_horizontal`) utilise la méthode qui affiche un pixel (`pixel`). À son tour, la méthode chargée de l'affichage d'un rectangle plein (`rectangle`) utilise la méthode de ligne horizontale. Pour finir, toutes les méthodes d'affichage utilisent, directement ou par transition, l'unique méthode de l'écriture d'un pixel.
Cette interdépendance des méthodes a deux objectifs essentiels :

- Malgré les diverses formes possibles de codage des pixels, la majeure partie des méthodes reste identique et générique. Ainsi, l'ensemble des méthodes de BITMAP repose simplement sur l'écriture d'un pixel ou la lecture d'un pixel.
- L'optimisation d'une seule méthode clé permet d'accélérer les performances de tout un ensemble de méthodes.

Cette implantation du prototype BITMAP est particulièrement bien adaptée à l'évolution des cartes graphiques actuelles. Aujourd'hui, une carte vidéo ne se contente plus d'offrir la possibilité d'afficher ou de lire un pixel sur écran. L'affichage d'une droite, d'un polygone ou d'une texture est directement à la charge du contrôleur graphique. Une redéfinition dans le prototype VIDEO de certaines méthodes de BITMAP qui correspondent aux possibilités du contrôleur permet une prise en charge optimale du matériel. L'effet "château de cartes" ou "poupées russes" de l'implantation des méthodes de BITMAP permet d'accroître considérablement les performances globales d'affichage. L'exemple de la figure 4 à droite montre la redéfinition de l'affichage d'une droite horizontale. celle-ci accélère l'affichage d'un rectangle, d'un polygone, ...La fonction rectangle utilise automatiquement par liaison dynamique la nouvelle méthode d'affichage de ligne traitée directement par le contrôleur vidéo.
Ce type de mécanisme est aussi présent dans les systèmes actuels au prix de réindirection multiple du code (approche par délégation). Lorsque nous détaillons les gestionnaires graphiques actuels, une lourdeur du code est présente pour obtenir les mêmes qualités. Aussi, les performances offertes par le matériel ne sont pas toujours utilisées et répercutées dans l'ensemble des méthodes graphiques disponibles.

7. Adaptation dynamique par héritage dynamique

L'héritage dynamique des concepts objets à base de prototypes permet d'apporter une expressivité et une flexibilité dynamique nouvelles par rapport aux concepts objets à base de classe. L'utilisation de cet outil permet de matérialiser les changements de comportement d'une composante système de manière efficace. Dans l'exemple de la figure 5, selon la connexion dynamique d'un matériel sur le périphérique du port *USB*, le système s'adapte grâce à l'héritage dynamique. Le comportement du prototype *USB* se voit modifier sans pour autant être obligé de changer l'ensemble des pointeurs qui utilise ce prototype (objet A, B, ...).
Un exemple concret d'utilisation de cet héritage dynamique est la gestion de la variation des modes vidéo sur architecture *Intel x86*. Une carte graphique offre une multitude de modes vidéo qui se dis-

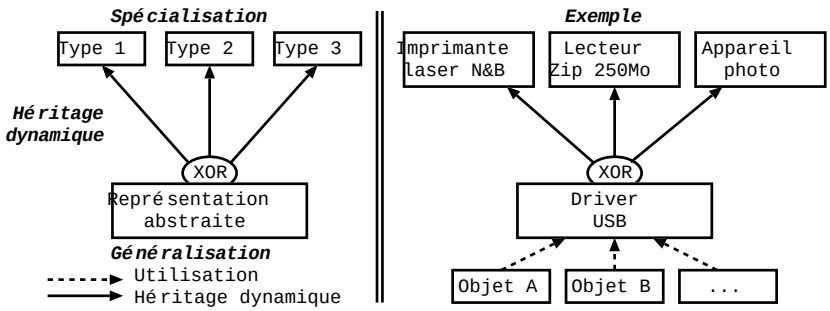


Figure 5: L'héritage dynamique à prototypes

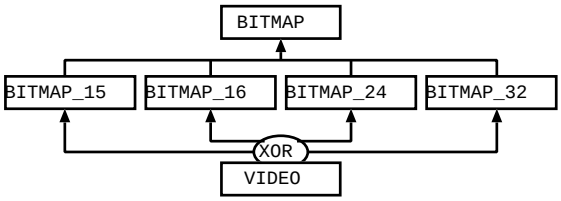


Figure 6: Première forme : par héritage dynamique

tinguent par leur résolution[3] ainsi que par la codification (ou profondeur) d'un pixel sur 8, 12, 16, 24 ou 32 bits. Le changement de résolution ne provoque que le redimensionnement de la BITMAP parent du pilote VIDEO. Par ailleurs, le changement de codification du pixel modifie massivement les méthodes clés de l'affichage. Deux solutions très proches sont envisageables.

7.1. Par simple héritage dynamique

Dans la figure 6, nous avons recours à l'héritage dynamique des prototypes pour changer et prendre en compte la modification de la géométrie du pixel durant l'exécution. Un changement de mode vidéo se traduit par une réaffectation du parent de VIDEO. Ainsi le code dédié à l'écriture et la lecture d'un pixel est automatiquement adapté au nouveau mode en vigueur.

Ceci est un bon exemple indiquant les avantages de l'héritage dynamique. En effet, en une seule affectation du parent de VIDEO, l'ensemble des modifications est pris en compte.

7.2. Par la généricité et l'héritage dynamique

La nouvelle version du prototype BITMAP du système Isaac utilise la généricité pour s'adapter aux différentes formes de pixels. La figure 7 décrit l'arbre d'héritage du moteur graphique avec cette implantation. Chaque objet PIXEL est un objet contenant une structure MAPPING décrivant le pixel en mémoire. Les méthodes de ces objets offrent une interface commune décrite par le prototype PIXEL pour appliquer ou saisir la couleur du pixel.

Par rapport à la première approche, cette implantation a fortement rendu le code plus lisible sans aucune baisse des performances : l'écran est physiquement matérialisé par un tableau à deux dimensions d'objets mappés de PIXELs.

L'héritage dynamique reste présent au niveau du parent de VIDEO pour conserver la flexibilité de la version précédente.

[3] nombre de pixels en abscisse, et nombre de pixels en ordonnée

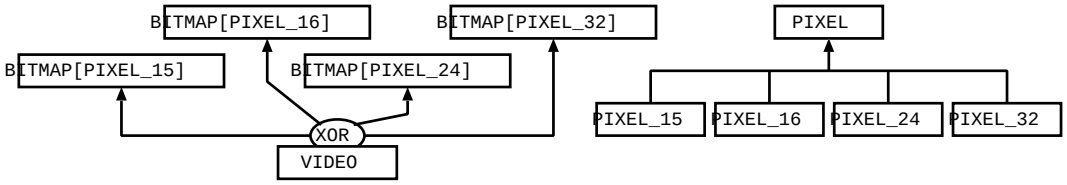


Figure 7: Deuxième forme: Généricité & héritage dynamique

7.3. Discussion entre modèle à prototype et celui à classe

Avec un modèle à classe, l'héritage serait inversé : une classe VIDEO générique se spécialiserait en plusieurs sous classes supportant chacune un mode vidéo particulier. Les appels seraient réalisés sur l'objet feuille du mode en vigueur. Cette approche a plusieurs contraintes par rapport à notre modèle à prototypes :

- Le changement du mode vidéo durant l'exécution provoque le changement de la feuille ainsi en vigueur dans l'arbre d'héritage. Les objets réalisant des appels sur cette feuille doivent être prévenus de son changement d'adresse.
- Dans le monde à classe, comme l'objet vivant est un ensemble monolithique en mémoire de la représentation hiérarchique de son arbre d'héritage, le changement d'un mode vidéo est plus coûteux en mémoire.
- La redéfinition des méthodes utilisant directement les possibilités matériel d'une carte vidéo doit être présente dans l'ensemble des objets feuilles par duplication du code ou par l'utilisation d'un héritage multiple.

8. Expérience de portage du système d'exploitation Isaac

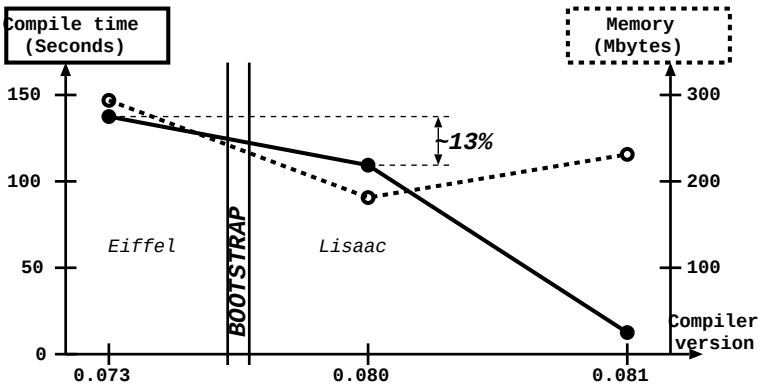
Aujourd'hui, le système Isaac fonctionne sur 5 architectures différentes. Des prototypes représentant le noyau jusqu'au prototype matérialisant son interface graphique, le projet est composé de 67 prototypes génériques et de 18 600 lignes de code Lisaac. Nous ne comptons pas la librairie standard (indépendante du système d'exploitation) composée de plus d'une centaine de prototypes et implantant des fonctionnalités de haut niveau comme les collections (dictionnaire, ensemble, tableau, liste chaînée, ...), manipulations graphiques (*bitmap*, vectoriel, police de caractère, ...), etc.

	Prototype spécifique	temps de développement	Taille des exécutables
Intel X86 *PC*	13 1232 lignes (63 ASM)	Première implantation	331 Ko
Motorola DragonBall *Sony PDA*	12 815 lignes (34 ASM)	3 mois	148 Ko
Intel StrongARM *Ipaq PDA*	11 738 lignes (230 ASM)	3 semaines	134 Ko
Unix *Linux*	6 315 lignes	8 heures	194 Ko
ST230 *Simulateur*	9 480 lignes (340 ASM)	2 semaines	283 Ko

9. Test de performances du compilateur Lisaac

9.1. Le *bootstrap* du compilateur: SmartEiffel *vs* Lisaac

Les premières versions du compilateur ont été écrites en Eiffel. Entre la version 0.073 écrite en Eiffel et la version 0.080 écrite en Lisaac, une simple traduction a été effectuée pour réaliser l'étape importante



Lisaac version	0.073	0.080	0.081
Time	135s	118s	11s
Memory	286Mb	192Mb	230Mb
Language	Eiffel V.1.1	Lisaac V.0.073	Lisaac V.0.080

Figure 8: Benchmark on a *Bootstrap* Lisaac compiler (Athlon 1.7GHz, 512Mb)

du *Bootstrap*[4]. Cette étape permet de prendre son indépendance, mais aussi de valider et de mettre en avant notre travail dans un cadre pratique et suffisamment conséquent pour être significatif (environ 15 000 lignes de Lisaac).
La figure 8 montre les résultats obtenus durant cette étape. Nous obtenons un gain d'environ 13% par rapport à SmartEiffel. Le gain de performance entre la version 0.080 et 0.081 est plus conséquent, mais moins significatif car elle est le résultat d'optimisation manuelle sur le code source. Nous utilisons notamment de manière plus fine la puissance des prototypes par rapport à une approche plus conventionnelle à classe.

9.2. Décodeur MPEG2 : C *vs* Lisaac

Pour réellement prétendre remplacer le langage C par un langage de haut niveau, les performances avec ce dernier se devaient être testées. Nous avons choisi de réaliser ces tests sur un décodeur MPEG1/2. Après une traduction en Lisaac rigoureusement identique au code C, nous avons obtenu le tableau suivant :

	Version C	Version Lisaac	Pourcentage
Lignes de code	9 852 lignes	6 176 lignes	37.31% en -
Taille de l'exécutable	99Ko	109Ko	10.10% en +
Taille à l'exécution	1 352Ko	1 332Ko	1.47% en -
Temps d'exécution	3,60s	3,67s	1.94% en +

(Athlon 2.4GHz, 512Mb, 190 frames en moyenne)

L'utilisation de langage de haut niveau réduit systématiquement de manière significative la taille du code source (37% même en conservant exactement les mêmes structures algorithmiques). Par ailleurs, les optimisations de spécialisation réalisées par notre compilateur pour gagner en efficacité augmentent de 10% la taille de l'exécutable. L'utilisation en mémoire durant l'exécution reste quasi-inchangée. En réalité, les structures en Lisaac sont légérement plus lourdes qu'en C. Mais bon nombre de tableaux statiquement alloués en C ne sont pas utilisés dans un contexte précis d'exécution. En revanche, ces tableaux sont alloués dynamiquement et à la demande en Lisaac. La perte en terme de temps d'exécution est presque négligeable (2%) et ce resultat est particulièrement prometteur pour un compilateur encore jeune.

[4] Dans le domaine de la compilation, cette opération correspond à écrire le compilateur en Lisaac pour pouvoir compiler le compilateur Lisaac à l'aide du compilateur Lisaac.

La comparaison avec un code C++ aurait été interessante à mener. Néanmoins, il est difficile dans ce langage de juger la part du code réellement objet par rapport au code C classique. De plus, le C++, comme le Java, utilise la technologie des VFTs[5] pour la résolution de la liaison dynamique. Cette approche paraît peu pertinante en vue des mauvaises performances de celle-ci.

10. Conclusion

L'évolution des langages, et par conséquent les avantages qu'elle procure échappent encore aujourd'hui à la programmation système. Le projet Isaac a pour objectif d'intégrer les concepts objets de haut niveau au cœur même des systèmes d'exploitation. Son langage objet à prototypes Lisaac a été réalisé à cet effet. Ses prédispositions à la programmation système permettent une programmation de bas niveau tout en profitant d'une technologie des langages objets de haut niveau. Ainsi, l'héritage statique, la liaison dynamique et l'héritage dynamique deviennent des outils puissants de modélisation de couche basse comme les pilotes de périphérique. La programmation par contrat et l'héritage automatique de ces contrats permettent de rendre plus robuste l'écriture de morceau de code spécifique à un matériel. Le système d'exploitation Isaac constitue une mise à l'épreuve de notre modèle avec une adaptation sur 5 architectures différentes. Les tests réalisés sur notre compilateur Lisaac face au compilateur SmartEiffel et C montrent qu'il est maintenant réellement possible d'utiliser des langages de haut niveau sans une perte de performance d'exécution conséquente (moins de 2%).

Bibliographie

1. Deville (D.), Rippert (C.) et Grimaud (G.). – *Trusted Collaborative Real Time Scheduling in a Smart Card Exokernel*. – Rapport technique n° RR-5161, INRIA, France, apr 2004.
2. Grimaud (G.). – *CAMILLE : un Système d'Exploitation Ouvert pour Carte à Microprocesseur*. – Thèse de PhD, Univ. Lille 1, France, dec 2000. in french.
3. Jean-Philippe Fassino (France Télécom R&D), Jean-Bernard Stefani (INRIA), Jean-Bernard Stefani (DIKU) et Gilles Muller (INRIA). – Think: A software framework for component-based. *In : USENIX 2002 Annual Conference*. – 2002.
4. Jr (Assumpcao) et M. (Jecel). – O sistema orientado a objetos merlin em máquinas paralelas (the merlin object oriented system in parallel machines). *In : in the V SBAC-PAD (V Brazilian Computer Architecture Conference - High Performance Computing) Conference Proceedings*, pp. 304–312. – 1993.
5. Jr (Assumpcao) et M. (Jecel). – Adaptive compilation in the merlin system for parallel machines. *In : in the WHPC'94 (IEEE/USP International Workshop on High Performance Computing) Conference Proceedings*, pp. 155–166. – March 1994.
6. Jr (Assumpcao), M. (Jecel), Kofuji et T. (Sergio). – Bootstrapping the object oriented operating system merlin: Just add reflection. *In : Chapter 5 in "Advances in Object-Oriented Metalevel Architectures and Reflection"*. – edited by Zimmerman, Chris, CRC Press, 1996. ISBN 0-8493-2663-X.
7. Meyer (B.). – *Eiffel, The Language*. – Englewood Cliffs, Prentice Hall, 1992. ISBN 0-13-247925-7.
8. Mérillon (F.), Réveillère (L.), Consel (C.), Marlet (R.) et Muller (G.). – Devil : un IDL pour les contrôleurs de périphériques. *In : 2ième Conférence Française sur les systèmes d'Exploitation, (CFSE'2)*. pp. 129–140. – ACM Press, 2001.
9. Rippert (Christophe). – Component isolation in the think architecture. *In : Proceedings of the 7th CaberNet Radicals workshop*. – October 13th-16th 2002.
10. Réveillère (L.), Mérillon (F.), Consel (C.), Marlet (R.) et Muller (G.). – *The Devil language*. – Rapport technique, Rennes, France, IRISA, mai 2000.
11. Sonntag (Benoît) et Colnet (Dominique). – Lisaac: the power of simplicity at work for operating system. *In : 40th conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific'2002), Sydney, Australia*. pp. 45–52. – Australian Computer Society, février 2002.
12. Ungar (David M.) et Smith (Randall B.). – Self: The Power of Simplicity. *In : 2nd Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87)*. pp. 227–241. – ACM Press, 1987.

[5] *Virtual Function Table*