MEMOIRE DE STAGE

Couplage de codes dans une plateforme pour applications de simulation numérique

Pierre ’khorben’ Pronchery

MEMOIRE DE STAGE: Couplage de codes dans une plateforme pour applications de simulation numérique

par Pierre ’khorben’ Pronchery

Historique des versions

Version 0.3.0 3 Juin 2002
on the way to release candidate
Version 0.2.0 31 Mai 2002
needs little more content: dia functions, cast explanations, ...
Version 0.1.1 30 Mai 2002
Got justified text, page numbering, A4 format, still pdf tables problems
Version 0.1.0 29 Mai 2002
Approaching definitive plan; justified text, page numbering, pdf tables and A4 format problems
Version 0.0.6 28 Mai 2002
Dia plug-ins
Version 0.0.5 27 Mai 2002
Intro, Cast, toolkits
Version 0.0.4 21 Mai 2002
Translated dia_plugin
Version 0.0.3 21 Mai 2002
Merged dia_plugin.sgml inside as a plug-in ;-)
Version 0.0.2 17 Mai 2002
Ported to DocBook SGML
Version 0.0.1 16 Mai 2002
First draft.

Remerciements

Je remercie Toan Nguyen, directeur de recherches à l'INRIA, de m'avoir permis d'effectuer ce stage.

Je remercie également Laurent Bonnaud, enseignant-chercheur à l'IUT2 de Grenoble, pour l'aide qu'il m'a apporté au cours de ce stage.

Table des matières

Liste des tableaux

Liste des illustrations

Introduction

Présentation de l'INRIA

L'INRIA, Institut National de Recherche en Informatique et en Automatique, a été créé en 1967. C'est un établissement public à caractère scientifique et technologique (EPST), sous la tutelle du ministère de la recherche et du ministère de l'économie, des finances et de l'industrie. L'INRIA a ainsi été chargé de missions par l'État, dont la réalisation de systèmes expérimentaux, l'organisation d'échanges scientifiques, la diffusion des connaissances, la contribution à la normalisation. Pour cela l'INRIA crée des unités de recherche, prend des participations dans des entreprises et accueille des chercheurs de toute nationalité.

L'unité de recherche INRIA Rhône-Alpes est située à Montbonnot, regroupant 22 équipes de recherche. Les thèmes abordés sont les réseaux et systèmes informatiques, les interactions homme machine et la simulation de systèmes complexes, la moitié de ces projets étant en partenariat direct avec des acteurs de l'industrie et de l'enseignement.

Le projet Opale

Le projet Opale, ou "OPtimisation et contrôle, ALgorithmes numériquEs et intégration" (succédant au projet SINUS, "SImulation NUmérique dans les Sciences de l'ingénieur") consiste à faire progresser les méthodes de modélisation numérique. Le domaine couvert va de l'analyse des modèles à leur mise en oeuvre sur ordinateur, avec optimisation. Une des applications actuelles consiste par exemple à optimiser l'écoulement de l'air sur un profil d'aile d'avion. Ce type de calcul très exigeant est grandement accéléré grâce à l'utilisation de plusieurs grappes d'ordinateurs interconnectées par un réseau à haut débit.

Sujet du stage

Le stage devait initialement porter sur le développement de la plateforme d'intégration de logiciel, Cast, qui gère et fait communiquer les modèles et programmes numériques lors de leur exécution. Il s'agissait de mettre en oeuvre un couplage de code entre deux modèles numériques, et de les paralléliser pour une exécution sur grappe de PC.

Pour des raisons techniques, le sujet du stage a évolué: il s'agit de reprendre l'interface graphique de Cast, afin de rendre celui-ci utilisable sur un maximum de systèmes possibles.

Chapitre 1. Description du problème

1.1. Analyse de l'existant

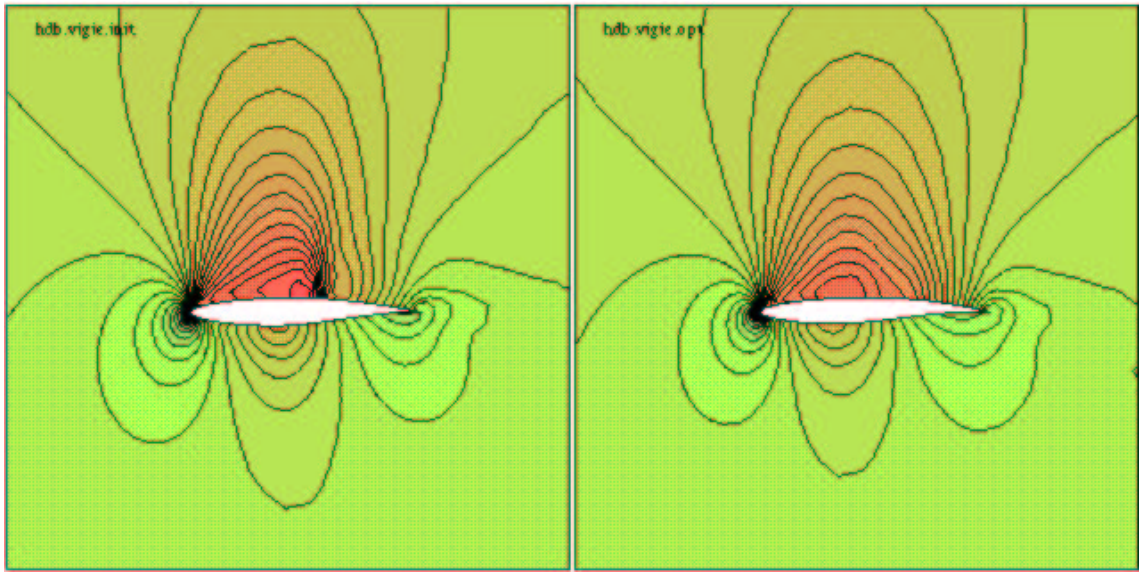
1.1.1. Description de Cast

Fonctionnalités offertes

Cast[1] est une plateforme d'intégration de logiciels. Elle permet de définir une succession dynamique d'applications, ou tâches, à l'aide d'opérateurs de choix, itération, séquence par exemple. Ceci est proposé sous une forme graphique, permettant une définition intuitive des tâches et de leur déroulement. L'objectif est de synchroniser et répartir l'exécution de code, tout en le faisant communiquer.

Ainsi Cast est aujourd'hui capable de faire de la simulation numérique distribuée, en utilisant autant de grappes d'ordinateurs que disponibles. Interconnectées, Cast peut y exécuter des applications réparties de calcul parallèle, dépassant les performances de certains calculateurs plus coûteux. Dans le calcul de l'exemple suivant, l'optimisation du profil d'une aile d'avion, il a fallu 4h30 de travail pour une station de travail, et 3 minutes pour une grappe de 40 ordinateurs (à gauche se trouve le profil initial, à droite l'optimisé).

Figure 1-1. Calcul d'un profil d'aile d'avion



Modèle mathématique manipulé

La représentation empruntée correspond en fait au modèle de l'algèbre SCCS de Milner. Ce modèle permet de définir une suite algébrique de données suivant leurs caractéristiques temporelles: par exemple le produit de deux données marque leur simultanéité, la somme leur exclusion mutuelle.

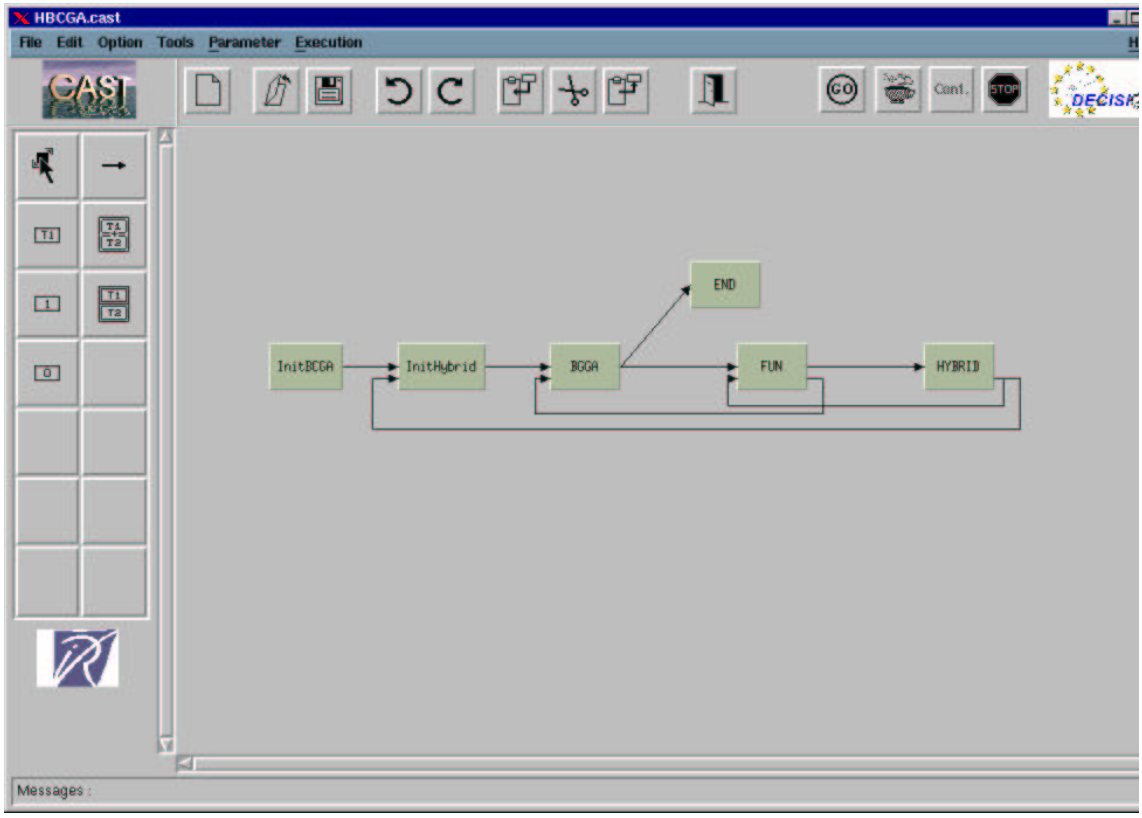
Aspects techniques

CORBA[2] est une infrastructure et architecture logicielle, indépendante du matériel et logiciel utilisés. Elle permet l'exécution et la communication d'applications à travers n'importe quel type de réseau. Ce standard est défini par l'OMG[3], et il est actuellement utilisable sur la quasi-totalité des matériels existants, depuis les systèmes embarqués jusqu'aux super-calculateurs.

Cast est exclusivement programmé en C++. Il utilise diverses librairies pour son fonctionnement:

- une implémentation de CORBA gérant les objets parallèles, PACO, développée par l'INRIA en extension à l'implémentation CORBA MICO.
- une librairie graphique, wxMotif, issue du projet wxWindows, permettant la compilation d'une interface graphique sur de multiples plateformes différentes

Figure 1-2. Interface de Cast



1.1.2. Les précédents développeurs

Cast a été développé successivement par plusieurs personnes depuis 1995. Il a été adapté à PACO

en 1999. Des scripts d'autoconfiguration ont ensuite été rajoutés afin de permettre la compilation sur la plupart des Unix existants: ils génèrent dynamiquement le fichier Makefile, en étudiant la disponibilité et l'emplacement des librairies nécessaires.

Le projet a ensuite été placé sur l'hébergeur Sourceforge, qui permet le développement par CVS[4]. Il s'agit en fait d'un serveur, accessible exclusivement par réseau, qui contient l'ensemble des sources du projet. Il conserve de manière appropriée chaque modification, par personne et date, ce qui le rend capable de régénérer d'anciennes versions du projet sans avoir à les garder séparément. Son interrogation, soumise à autorisation, permet le téléchargement des sources, et aux développeurs la mise à jour du serveur. De cette manière le développement en groupe d'un logiciel est rendu complètement transparent, les opérations courantes étant réduites à une simple commande (téléchargement, mise à jour d'un téléchargement, mise à jour du serveur, administration, historique, ...).

Les derniers développements ont été réalisés en Août 2001, avec le travail de Mickaël Marchand. Un module a été rajouté à Cast, appelé "Observator". Celui-ci, exécuté sur un serveur chargé d'une tâche de Cast, est capable de suivre son déroulement et la charge du système, et de les reporter à Cast: l'utilisateur peut alors mettre en pause, reprendre et arrêter l'exécution d'un calcul à tout moment.

Cast est mis gratuitement à disposition de tous par l'INRIA, sur autorisation écrite. Ses sources le sont de la même manière.

1.1.3. Fonctionnement de Cast

Pendant la prise en main de Cast j'ai procédé à la mise à jour du manuel de Jérôme Blachon, alors dépassé: elle est disponible en Annexe A.

Dans Cast la représentation des tâches est faite au moyen de boîtes rectangulaires, auxquelles on peut assigner un nom. Un double clic sur une tâche ouvre sa boîte de dialogue, dans laquelle on peut définir ses propriétés: type, arguments d'entrée/sortie, nom du serveur, et si c'est le cas les propriétés CORBA (nom d'objet, de méthode et ses paramètres).

La définition des successions de tâches se fait donc graphiquement. Différents éléments sont disponibles: tâche simple, tâche de fin, "somme" de deux tâches, "produit" de plusieurs tâches, et bien sûr les liaisons.

- **Tâche simple:** représentée sous la forme d'une boîte, elle peut être de deux types, primitive (fichier exécutable), ou composite (abstraction d'au moins une sous-tâche, CORBA ou non).
- **Tâche de fin:** cette tâche en fait n'exécute rien, elle marque la fin du processus.
- **Somme de tâches:** cette entité comprend deux tâches simples, et n'en permet l'exécution que d'une à la fois, selon plusieurs critères, comme le résultat de tâches précédentes.
- **Produit de tâches:** cette entité permet l'exécution concurrente d'autant de tâches simples que désirées.
- **Liaisons:** cette entité permet de relier les tâches entre elles, dans leur ordre d'exécution.

1.2. Définition du problème

1.2.1. Les limites de Cast

La seule version de Cast disponible en binaire était datée du 9 Mai 2001. À l'exécution, on se rend rapidement compte que cette version est difficilement exploitable: elle plante presque à chaque opération, et ne ressemble pas tout à fait aux copies d'écrans visibles sur le site du projet. Suivant les documents fournis par Mr Nguyen, la version la plus récente de Cast est toujours disponible sur CVS et date de Septembre 2001. Sa compilation a été périlleuse, car elle nécessitait une version de MICO particulière, PACO, en plus de wxWindows et d'une version de Motif compatible avec Motif-2.0 (en l'occurence, lesstif).

Toutes les opérations de compilation et de programmation ont été effectuées sur une machine personelle, mise à disposition par l'INRIA. La compilation et l'intégration de Cast à un système d'exploitation Unix courant a été effectuée sous Linux (Debian 3.0). Les détails de la procédure ont été rédigés à l'attention du projet et sont disponibles en Annexe B.

Enfin, quelques fonctionnalités sont présentes dans les menus mais pas encore implémentées: le copier/coller, annuler/répéter par exemple. D'autre part, selon ses objectifs Cast devrait se voir intégrée une gestion dynamique de répartition de charge.

1.2.2. Migration de librairie graphique

La programmation de l'interface graphique de Cast a été réalisée à l'aide de wxMotif. Celle-ci est peu rencontrée sur les systèmes d'exploitation existants, c'est pourquoi dans la poursuite du développement de Cast, la migration vers une autre librairie graphique était souhaitable. Or, wxMotif fait partie du projet wxWindows (http://www.wxwindows.org), qui a pour but de créer une interface commune aux librairies MFC (windows), Gtk+, MacOS et Motif. Ainsi en théorie, la migration vers wxGtk devrait être immédiate, simple fait d'une directive de compilation.

Notes

1. Collaborative Applications Specification Tool, http://www.inrialpes.fr/sinus/cast
2. Common Object Request Broker Architecture, http://www.corba.org
3. Object Management Group, http://www.omg.org
4. Concurrent Version System, http://www.cvshome.org

Chapitre 2. Les différentes approches possibles

2.1. Recompilation avec wxGtk

Dans wxWindows, le changement de librairie est effectué après le simple changement de directive de compilation suivant:

```
#define __WXGTK__
```

au lieu de:

```
#define __WXMOTIF__
```

Malheureusement, Cast comprend également des portions de code réalisées directement avec Motif, rendant la compilation avec wxGtk impossible. Ceci empêche la compilation de 4 fichiers C++ sur 81:

- combobox.cpp
- castappli.cpp
- verticalToolBar.cpp
- spinCtrl.cpp

En regardant de plus près, ces fichiers comprennent des appels à des fonctions de wxWindows spécifiques à Motif, ainsi qu'à Motif directement.

La prise en main concurrente de la programmation en wxWindows, en Motif, et de Cast m'a semblée trop périlleuse pour envisager le passage direct vers wxGtk. De plus selon mon prédécesseur, Christine Plumejeaud, le port de Cast de Solaris vers Linux avait entraîné des instabilités. J'ai donc cherché si des solutions alternatives étaient envisageables.

2.2. Réécriture de l'interface avec Gtk

Par suite il m'a semblé plus adéquat de procéder à la réécriture de l'interface de Cast. Avec l'approbation de Mr Nguyen, je me suis penché sur cette possibilité.

2.2.1. Réutilisation du code de Cast

Il s'agit de ne réécrire que l'interface de Cast: le reste de son code étant existant et fonctionnel, il faut le réutiliser au maximum. Cast ayant été programmé en C++, la librairie graphique à utiliser doit

être utilisable dans ce langage. De plus elle devra si possible être libre (idéalement avec un minimum de restrictions). Enfin elle doit être présente sur un maximum de plateformes disponibles.

2.2.2. Choix de librairie graphique

Tableau 2-1. Librairies graphiques les plus répandues

Librarie	Langage(s)	Plateforme(s)	Licence	Commentaire
Gtk	C, C++, Ada, Perl, Python, Eiffel	Unix, Windows, BeOS	LGPL	Utilisé dans Gnome[a]
MFC	C++	Windows	Payant ou limité	Une partie de la librairie Microsoft est utilisable librement
Motif (lesstif)	C, C++	Unix, MacOS X, OS/2	LGPL	Motif est payant mais lesstif en est une implémentation libre (OpenMotif également)
Qt (Qt/X11)	C++	Unix, Windows, MacOS X	GPL	Version libre d'un projet commercial (utilisée dans KDE[b])
Tk	Tcl	Unix, Windows, MacOS X	BSD	Tcl est un langage interprété

a. Environnement complet, comprenant un bureau, un navigateur et une suite d'applications

b. Projet similaire à Gnome

Ici intervient le problème de licence. En effet la plupart des librairies graphiques multi-plateformes et gratuites sont protégées par des licences dites "libres". Leur code source est disponible pour tous, mais leur utilisation soumise parfois à des obligations. La GPL[1], créée par la FSF[2], oblige tout logiciel utilisant du code sous cette licence à être aussi soumis à la GPL, et donc à diffuser le code source. En contrepartie, le logiciel reste la propriété de son auteur, et tout travail effectué dessus sera disponible avec son code source. La LGPL[3] autorise quant à elle les travaux propriétaires à utiliser une librairie sous son régime, sans aucune obligation. La licence BSD[4] enfin, autorise toute réutilisation de code sous son régime.

Pour résumer les conséquences des licences, Cast en utilisant une librairie sous licence GPL, devra également être en GPL (et donc, de code source ouvert), tandis que la LGPL et la BSD lui autorisent n'importe quel régime. Ceci a une grande importance dans un tel choix, mais le code source de Cast étant déjà disponible assez facilement, son passage en GPL ne pourrait que le protéger.

Les possibilités des différentes librairies sont généralement équivalentes, étant toutes disponibles au moins en version finale stable. La difficulté majeure dans l'interface de Cast est la reproduction de l'aire de travail. Mais en utilisant par exemple Gtk+ ou Gtk--, interfaces respectives C et C++ de Gtk, cette opération me semblait faisable, pour les raisons suivantes:

- Gtk est multi-plateforme
- sa licence en autorise une utilisation peu restreinte
- l'intégration d'objets Gnome est immédiate, en particulier le GnomeCanvas qui permettrait de réaliser à moindre coût l'aire de travail

- très utilisé, Gtk dispose d'une documentation importante
- enfin, j'avais déjà des connaissances sur la programmation avec Gtk+

Néanmoins, pour me lancer dans cette opération je devais acquérir plus de connaissances sur la création et la manipulation d'objets graphiques. C'est pourquoi Je me suis penché sur un logiciel, Dia, que j'utilise régulièrement.

2.3. Approche du problème avec Dia

2.3.1. Présentation de Dia

Dia est un éditeur de diagrammes, en cours de développement (version 0.90rc3). Il comprend déjà de nombreuses possibilités de création de diagrammes: diagrammes de flux, UML, Sybase par exemple, qui peuvent être étendues grâce à un système de plug-ins.

Son principal intérêt réside dans son interface, comportant une aire de travail multi-documents complète. Elle est composée d'une fenêtre principale, comportant un accès direct aux formes standard, et à celles du plug-in sélectionné, ainsi que d'une fenêtre de diagramme par diagramme ouvert.

Figure 2-1. Fenêtre principale

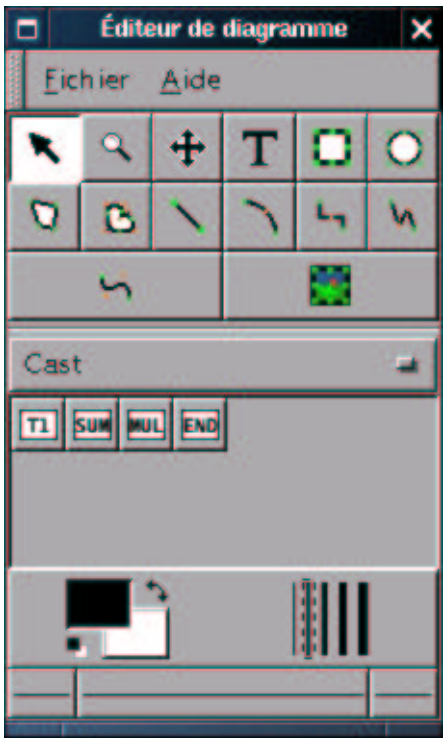
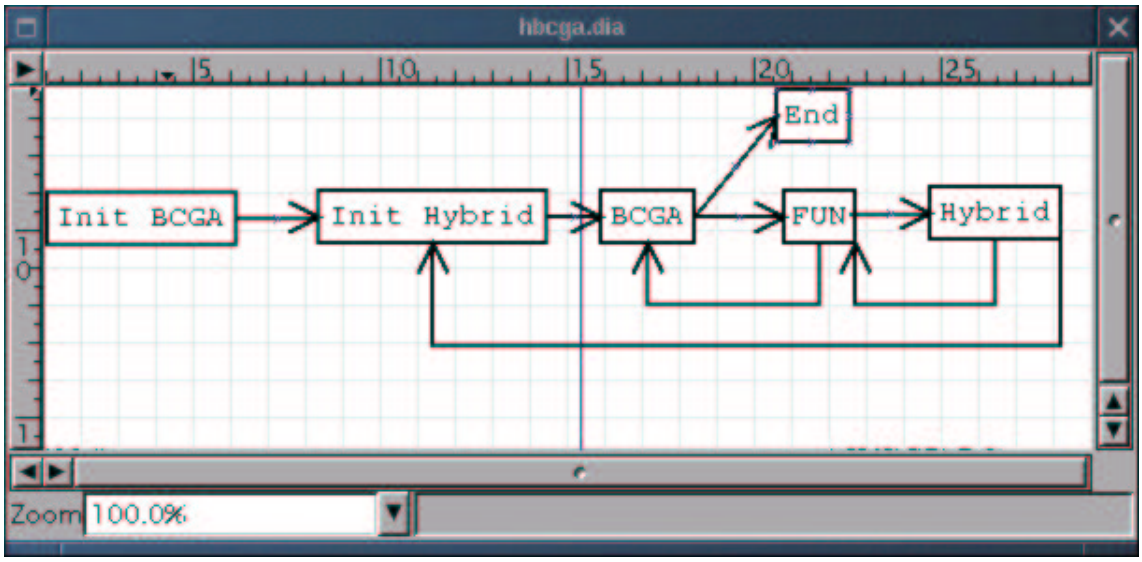


Figure 2-2. Édition de diagrammes



Dia est donc capable d'utiliser des formes simples: texte, polygones, ellipses, lignes droites, courbes, flèches et même des images. Plus précisément, un diagramme est constitué de formes rectangulaires (pouvant être transparentes), reliées par des flèches. Il est ensuite possible de rajouter des plug-ins, pouvant produire des entités complexes, les déplacer, redimensionner, relier, dupliquer grâce à de simples clics de souris.

La construction de diagrammes est assez intuitive, et ressemble à celle de Cast. Les différentes formes disponibles sont regroupées par thème, et une fenêtre de propriétés s'ouvre après un double clic sur une forme. De plus les diagrammes peuvent être imprimés ou sauvegardés sous divers formats.

D'un point de vue plus technique, Dia présente d'autres propriétés très intéressantes. Dia est réalisé en Gtk, et complètement compilable sur plusieurs plateformes (dont Unix et Windows). Il est déjà traduit dans une trentaine de langues. Son système de plug-in accepte le C, le C++ et le Python.

2.3.2. Similarités avec Cast

L'édition des diagrammes présentée dans Dia ressemble à celle présente dans Cast, et possède même plus de fonctionnalités. Un point négatif cependant concerne la façon de relier les formes entre elles: dans Cast les flèches sont automatiquement liées et placées, tandis que dans Dia il faut le faire manuellement. Cependant des développeurs travaillent en ce moment à l'ajout de cette fonctionnalité dans Dia.

De plus, au cours du développement de Dia un autre projet est né, visant à séparer l'implémentation de l'interface graphique de Dia du reste de son code. Il comporte donc une aire de travail complète, accessible directement par le développeur. Ce projet est encore très jeune, mais a

contribué à mon intérêt pour Dia. En effet l'utilisation de cette aire de travail permettrait de se passer de Dia pour la réécriture de l'interface de Cast. Ceci pourrait régler des problèmes de licence.

En effet Dia est protégé par la licence GPL, ce qui autorise sa modification, mais oblige les travaux réalisés ainsi à être également protégés par cette licence, et donc à dévoiler le code source. Mr Nguyen a accepté de placer cette nouvelle version de Cast sous licence GPL.

2.3.3. Estimation de charge de développement

Différents choix techniques sont alors possibles, en voici une estimation, dans un ordre croissant de temps de développement:

- Écriture d'un plug-in de Dia: capable de faire au moins toute opération s'appliquant à un diagramme, la possibilité de pouvoir ajouter toutes les fonctionnalités de Cast n'est pas assurée.
- Idem avec modification en profondeur de ce logiciel: dans ce cas il est effectivement assuré de pouvoir implémenter Cast avec Dia, mais la durée du développement est potentiellement allongée.
- Reprogrammation complète de l'interface de Cast: perspective la plus souple, mais également la plus exigeante en ressources temporelles et techniques.

En m'inspirant de plug-ins simples, fournis dans Dia, je me suis donc penché sur la réalisation de l'interface de Cast, sous la forme d'un plug-in pour Dia. De plus les diagrammes générés par Dia sont exportables assez facilement, car au format XML. Ce format présente des données textuelles sous forme structurée: ceci en permet une lecture humaine comme informatique facile et sans ambiguités. Ainsi, dans le pire des cas les diagrammes générés seraient exportables dans Cast et directement exécutables. Par ailleurs, un fichier de données au format XML répond à un besoin particulier, qui doit être défini informatiquement. Ces définitions sont appelées DTD, et peuvent valider la conformité d'un fichier concerné (bon nommage et bonne succession des entités par exemple) Les différentes DTD correspondant aux fichiers XML rencontrés sont situées en Annexe C.

Notes

1. General Public License, http://www.gnu.org/licenses/licenses.html
2. Free Software Foundation, http://www.gnu.org
3. Lesser General Public License, idem GPL
4. Berkeley Software Distribution, système Unix de l'université de Berkeley (Californie), http://www.freebsd.org/copyright/index.html

Chapitre 3. Réalisation du plug-in Dia

3.1. Prise en main du logiciel

Pour commencer, il faut télécharger le logiciel. À la mi-Avril, il y avait deux possibilités: la version stable, 0.88.1, sortie en Mai 2001, et la version CVS "0.89". Le développement de Dia est très actif, donc la version CVS avait beaucoup évolué. La dernière version stable étant de surcroît âgée, la réalisation a été effectuée en utilisant la version CVS.

Dia faisant partie du projet Gnome, il est hébergé sur leur serveur CVS. Il faut donc s'identifier (une fois suffit) sur le serveur CVS anonyme:

```
$ cvs -d:pserver:anonymous@anoncvs.gnome.org:/cvs/gnome login
```

et enfin, le télécharger:

```
$ cvs -z3 -d:pserver:anonymous@anoncvs.gnome.org:/cvs/gnome co dia
```

pour appliquer les changements du CVS à sa copie, la commande est la suivante:

```
$ cvs -z3 -d:pserver:anonymous@anoncvs.gnome.org:/cvs/gnome update -PAd
```

en se plaçant bien dans la racine du projet.

De plus une liste de diffusion (système d'échange par mail) est mise à disposition de tous, elle aussi hébergée par le projet Gnome. Après inscription, on se rend rapidement compte qu'elle est très active, utilisée autant pour les questions venant des utilisateurs, les rapports de bugs et les prises de décisions pour les orientations du projet. Ainsi j'ai pu entrer directement en contact avec les développeurs, et rester informé des mises à jour de la version CVS.

Ayant informé les développeurs de mon intention de réaliser un document sur l'écriture de plug-ins pour leur logiciel, la suite de ce chapitre devrait être intégrée au projet Dia. En effet ceci répond à plusieurs demandes formulées sur la mailing-liste, et il n'existait pas encore de document sur ce sujet dans le projet.

3.2. Création des formes simples

Ceci nécessite un fichier global, décrivant les formes créées, et un fichier par forme créée. Chaque forme doit être accompagnée d'une image.

3.2.1. Création de formes avec Dia

Dia facilite la création de nouvelles formes, grâce à la possibilité de les réaliser directement dans Dia, de la même manière qu'un diagramme.

Il est plus aisé de commencer avec un exemple simple, tel qu'un court texte encadré par une ligne polygonale. Alors, à l'aide du menu contextuel (ou de la barre de menu si celle-ci est activée), il faut sélectionner l'option "File->Export...". Alors interrogé pour l'emplacement de l'export, ne pas oublier de sélectionner le format "dia shape file (*.shape)" dans le menu "file type". Dia va alors automatiquement générer un fichier .png (image), élaboré à partir de l'aspect de la forme et qui apparaitra dans le bouton de sélection de la forme. (il suffit de confirmer la fenêtre "PNG Export Options" qui apparait, et d'ajuster la taille de l'image à ses besoins).

Une fois sauvegardés, il faut copier les deux fichiers dans le répertoire `~/.dia/shapes`.

3.2.2. Création et modification manuelle des formes

Les formes sont écrites sous la forme de fichiers XML. Il est tout à fait possible, et parfois nécessaire, d'éditer ce fichier manuellement afin de le corriger ou améliorer. En voici un exemple commenté:

```
<?xml version="1.0" encoding="UTF-8"?>
<shape xmlns="http://www.daa.com.au/~james/dia-shape-ns"
xmlns:svg="http://www.w3.org/2000/svg">
  <name>Plug-in - Shape</name>
  <icon>Shape.png</icon>
  <!-- remplacer le nom et l'icône par les valeurs correctes -->
  <connections>
    <point x="0" y="0"/>
    <!-- placer tous les points de connections -->
  </connections>
  <aspectratio type="fixed"/>
    <!-- divers attributs -->
  <svg:svg>
    <svg:polyline style="stroke-width: 0.1; stroke: #000000" points="0,0 4,0 ... "/>
    <!-- placer les points a la suite, sous la forme x1,y1 x2,y2 -->
    <svg:text style="fill: #000000; font-size: 0.8" x="6" y="5.3">text</svg:text>
    <!-- la liste de possibilités n'est pas exhaustive -->
  </svg:svg>
</shape>
```

3.2.3. Création du fichier de description

Dia a besoin d'un fichier de description afin de savoir quelle forme chercher, et où elle se trouve. C'est également un fichier XML, situé dans le répertoire `~/.dia/sheets`.

Suit un exemple de fichier:

```
<?xml version="1.0" encoding="iso-8859-1"?>

<sheet xmlns="http://www.lysator.liu.se/~alla/dia/dia-sheet-ns">
  <name>Plug-in</name>
  <name xml:lang="fr">Greffon</name>
  <description xml:lang="fr">Diagrammes selon mon greffon</description>
  <contents>
    <object name="Plug-in - Forme">
```

```
      <description>Forme</description>
    </object>
    <object name="Plug-in - Forme 2">
      <description>Forme 2</description>
    </object>
  </contents>
</sheet>
```

le fichier commence par le nom du plug-in. Il est possible d'y intégrer autant de traductions que désiré. Pour chaque forme voulue, il faut insérer une section "object", comportant une description.

3.3. Écriture d'un plug-in élaboré

Afin de disposer de plus de fonctionnalités il est nécessaire d'employer un langage de programmation. Les instructions données ci-après font référence au langage C, mais il est également possible d'utiliser python, et bien sûr C++ (les différences permettant la compilation avec C++ sont à la fin de ce chapitre). Dans un premier temps est détaillé le processus de la création du plug-in, puis un exemple concret, inspiré d'un plug-in existant, appelé "ER" pour "Entité/Relations".

3.3.1. Les instructions nécessaires

Un plug-in est un fichier, plus précisément une librairie, qui peut être chargé par le programme pour lequel il a été prévu. Pour ce faire le programme cherche des points d'ancrage spécifiques dans le plug-in, conformément à sa structure de données. C'est pourquoi le plug-in doit nécessairement comporter les instructions suivantes:

Déclaration du plug-in Dia. Ceci est accompli ainsi:

```
DIA_PLUGIN_CHECK_INIT

PluginInitResult dia_plugin_init(PluginInfo *info)
{
    /* vérifier l'initialisation */
    if(!dia_plugin_info_init(info, "Plug-in",
            "The description of the plug-in",
            NULL, NULL))
return DIA_PLUGIN_INIT_ERROR;

    /* suite: déclaration des formes, etc. */

    return DIA_PLUGIN_INIT_OK;
}
```

Inclusion du fichier image. Il en faut un par forme:

```
#include "shape.xpm"
```

Déclaration de l'"ObjectType" de la forme. Il s'agit habituellement d'une structure renommée, dont les attributs peuvent être des types C quelconques. Néanmoins, Dia propose une liste renommée de ces types afin d'être multi-plateforme, il vaut mieux donc les utiliser (comme DiaFont, Color, la liste est conséquente mais malheureusement éparpillée). La structure comporte au moins les champs suivants:

```
typedef struct _Shape Shape;
struct _Shape
{
    Element element;
    /* contient le type d'objet, sa position, zone affichée, handles,
    * ses connections, ainsi que le coin supérieur gauche,
    * largeur et hauteur */

    /* autres attributs */
};
```

Déclaration des fonctions... chargées de:

Les fonctions sont accompagnées de leur prototype.

```
ObjectTypeOps shape_type_ops =
{
```

`(CreateFunc)` Créer une nouvelle instance d'une forme:

```
Shape* shape_create(Point *startpoint, void *user_data,
Handle **handle1, Handle **handle2)
```

`(LoadFunc)` Charger une forme depuis un fichier Dia:

```
Object* shape_load(ObjectNode obj_node, int version,
const char *filename)
```

`(SaveFunc)` Sauver une forme dans un fichier Dia:

```
void shape_save(Shape* shape, ObjectNode obj_node,
const char *filename)
```

```
/* un exemple complet:
(CreateFunc)    shape_create,
(LoadFunc)      shape_load,
(SaveFunc)      shape_save
*/
};
```

Déclaration des fonctions... chargées de:

Les fonctions sont accompagnées de leur prototype.

```
ObjectOps shape_ops =
{
```

`(DestroyFunc)` Retirer une tâche de la mémoire:
`void shape_destroy(Shape *shape)`

`(DrawFunc)` Dessiner une tâche dans l'aire de travail:
`void shape_draw(Shape *shape, Renderer *renderer)`

`(DistanceFunc)` Renvoyer la distance depuis un point à la forme:
`real shape_distance_from(Shape *shape, Point *point)`

`(SelectFunc)` Mettre à jour l'affichage lors de la sélection:
`void shape_select(Shape *shape, Point *clicked, Renderer *interactive_renderer)`

`(CopyFunc)` Copier une forme en mémoire:
`Object* shape_copy(Shape *shape)`

`(MoveFunc)` Mettre à jour la structure de données de la forme lorsqu'elle est déplacée:
`void shape_move(Shape *shape, Point *to)`

`(MoveHandleFunc)` Mettre à jour l'affichage de la forme lorsqu'elle est déplacée:
`void shape_move_handle(Shape *shape, Handle *handle, Point *to, HandleMoveReason reason, ModifierKeys modifiers)`

`(GetPropertiesFunc)` Renvoyer la boîte de dialogue des propriétés (sans les boutons "OK", "Apply" et "Close", gérés par Dia):
`GtkWidget* shape_get_properties(Shape *shape, Property *props, guint nprops)`

`(ApplyPropertiesFunc)` Appliquer les nouvelles propriétés depuis la boîte de dialogue des propriétés:
`ObjectChange* shape_apply_props(Shape *shape)`

`(ObjectMenuFunc)` Afficher le menu contextuel d'objet (invoqué à l'aide du bouton du milieu):
`DiaMenu* shape_object_menu(Shape *shape, Point *clicked)`

`(DescribePropsFunc)` Pas encore implémenté dans mon plug-in:
`PropDescription* shape_describe_props(Shape *shape)`

`(GetPropsFunc)` Non plus:
`void shape_get_props(Shape *shape, GPtrArray *props)`

`(SetPropsFunc)` Non plus:
`void shape_set_props(Shape *shape, GPtrArray *props)`

```
/* un exemple raccourci:
(CopyFunc)      shape_copy,
*/
};
```

Déclaration de la forme:

```
ObjectType shape_type =
{
    "Plug-in - Shape",    /* name */
    0,                    /* version */
    (char **) shape_xpm   /* pixmap */
    &shape_type_ops       /* ops */
};
```

3.3.2. Corps des fonctions

Dia propose au programmeur diverses fonctions, qui lui facilitent la tâche et permettent d'uniformiser les plug-ins. Certaines permettent de remplacer tout le corps d'une fonction, Dia

reconnaissant ce qu'il a à faire. Dans d'autres cas, ce n'est pas possible ou suffisant. Dans la liste suivante, seules les premières fonctions sont décrites, et celles que Dia est capable d'effectuer mentionnées. En effet le code de Dia est bien commenté, et il est assez aisé d'écrire les corps des fonctions à partir des en-têtes ou des plug-ins existants (sous-répertoires "lib" et "objects" du projet).

Création d'une forme. Voici un exemple typique, commenté.

```
Object* shape_create(Point *startpoint, void *user_data,
                Handle **handle1,
                Handle **handle2)
{
    Shape *shape;
    Element *elem;
    Object *obj;
    int i;

    /* allocation de l'espace nécessaire */
    shape = g_malloc0(sizeof(Shape));
    /* pour alléger l'écriture par la suite */
    elem = &shape->element;
    obj = &elem->object;

    obj->type = &shape_type;
    obj->ops = &elem->object;

    elem->corner = *startpoint;
    /* bien sûr il faut définir les valeurs au préalable */
    elem->width = SHAPE_DEFAULT_WIDTH;
    elem->height = SHAPE_DEFAULT_HEIGHT;

    /* changer 8 au nombre de points de connection */
    element_init(elem, 8, 8);
    for(i = 0; i < 8; i++)
    {
        obj->connections[i] = &shape->connections[i];
        shape->connections[i].object = obj;
        shape->connections[i].connected = NULL;
    }

    /* initialiser les autres attributs */

    shape_update_data(shape);

    /* changer 8 au nombre de points de connection */
    for(i = 0; i < 8; i++)
        obj->handles[i]->type = HANDLE_NON_MOVABLE;

    *handle1 = NULL;
    *handle2 = obj->handles[0];
    return &shape->element.object;
}
```

Chargement de la structure de données d'une forme. Avant chaque extraction de valeur d'un champ il est conseillé de l'initialiser à une valeur par défaut acceptable, dans le cas où Dia n'arriverait pas à le repérer. Il faut alors appeler la fonction:

```
AttributeNode object_find_attribute(ObjectNode obj_node, const char* attrname)
```

Ensuite après vérification de la validité de l'attribut renvoyé (NULL est renvoyé en cas d'échec), on utilise l'une des fonctions suivantes afin d'extraire la valeur de l'attribut:

Tableau 3-1. Fonctions de chargement

Type de données	Fonction et arguments
boolean	`int data_boolean(DataNode data)`
color	`void data_color(DataNode data, Color *col)`
enum	`int data_enum(DataNode data)`
font	`DiaFont *data_font(DataNode data)`
int	`int data_int(DataNode data)`
point	`void data_point(DataNode data, Point *point)`
real	`real data_real(DataNode data)`
rectangle	`void data_rectangle(DataNode data, Rectangle *rect)`
string	`char* data_string(DataNode data)`

Ce qui donne par exemple:

```
AttributeNode attr;
Shape *shape;
shape = g_new0(Shape, 1);

//load name
shape->name = NULL;
attr = object_find_attribute(obj_node, "name");
if(attr != NULL)
    shape->name = data_string(attribute_first_data(attr));

//load type
shape->type = SHAPE_COMPOSITE;
attr = object_find_attribute(obj_node, "type");
if(attr != NULL)
    shape->type = (ShapeType)data_enum(attribute_first_data(attr));
```

Sauvegarde d'une forme. La sauvegarde est très similaire au point précédent, mais ici on a aucun test préalable à effectuer. La liste de fonctions est la suivante:

Tableau 3-2. Fonctions de sauvegarde

Type de données	Fonction et arguments
boolean	`void data_add_boolean(AttributeNode attr, int data)`
color	`void data_add_color(AttributeNode attr, const Color *col)`
enum	`void data_add_enum(AttributeNode attr, int data)`
font	`void data_add_font(AttributeNode attr,const DiaFont *font)`
int	`void data_add_int(AttributeNode attr, int data)`
point	`void data_add_point(AttributeNode attr,const Point *point)`
real	`void data_add_real(AttributeNode attr, real data)`
rectangle	`void data_add_rectangle(AttributeNode attr, const Rectangle *rect)`
string	`void data_add_string(AttributeNode attr, const char *str)`

Ce qui donne par exemple:

```
data_add_string(new_attribute(obj_node, "name", shape->name);
data_add_enum(new_attribute(obj_node, "type", shape->type);
```

Dessin d'une forme. Le dessin d'une forme est effectué en invoquant le "renderer". On distingue les fonctions de choix de trait, de celles qui dessinent effectivement. Les fonctions sont disponibles en les préfixant par:

```
renderer->ops->
```

Tableau 3-3. Fonctions de choix de trait

Type de trait	Fonction et arguments	Énumération
épaisseur de ligne	`void set_linewidth( Renderer *renderer, real linewidth)`	Aucune (valeur réelle)
longueur des tirets	`void set_dashlength( Renderer *renderer, real length)`	Aucune (valeur réelle)
remplissage de forme	`void set_fillstyle( Renderer *renderer, FillStyle mode)`	FILLSTYLE_SOLID (solide)
style de ligne	`void set_linestyle( Renderer *renderer, LineStyle mode)`	LINESTYLE_SOLID (trait), LINESTYLE_DASHED (tirets), LINESTYLE_DASH_DOT (pointillés), LINESTYLE_DASH_DOT_DOT (tiret point point), LINESTYLE_DOTTED (tramé)

Tableau 3-4. Fonctions de dessin

Type de dessin	Fonction et arguments
arc plein	`void fill_arc(Renderer *renderer, Point *center, real width, real height, real angle1, real angle2, Color *color)`
arc vide	`void draw_arc(Renderer *renderer, Point *center, real width, real height, real angle1, real angle2, Color *color)`
cercle plein	`void fill_ellipse(Renderer *renderer, Point *center, real width, real height, Color *color)`
cercle vide	`void draw_ellipse(Renderer *renderer, Point *center, real width, real height, Color *color)`
image	`void draw_image(Renderer *renderer, Point *point, real width, real height, DiaImage image)`
ligne	`void draw_line(Renderer *renderer, Point *start, Point *end, Color *line_color)`
ligne en zig-zag	`void draw_polyline(Renderer *renderer, Point *points, int num_points,Color *line_color)`

Type de dessin	Fonction et arguments
rectangle plein	`void fill_rect(Renderer *renderer, Point *ul_corner, Point *lr_corner,Color *color)`
rectangle vide	`void draw_rect(Renderer *renderer, Point *ul_corner, Point *lr_corner,Color *color)`
texte	`void draw_string(Renderer *renderer, const utfchar *text, Point *pos, Alignment alignment, Color *color)`

Illustration par l'exemple:

```
Point ul_corner, lr_corner;

/* mettre à jour les coordonnées des points */

renderer->ops->set_fillstyle(renderer, FILLSTYLE_SOLID);
renderer->ops->fill_rect(renderer, &ul_corner, &lr_corner, &shape->inner_color);
renderer->ops->draw_rect(renderer, &ul_corner, &lr_corner, &shape->border_color);
```

Fonctions proposées par Dia. Enfin, voici la liste des fonctions évoquées en début de section, permettant d'éviter l'écriture du corps de certaines fonctions.

Tableau 3-5. Corps proposés par Dia

Fonction du plug-in	Nom du corps fourni
GetPropertiesFunc	object_create_props_dialog
ApplyPropertiesFunc	object_apply_props_from_dialog

Pour les utiliser, il suffit de placer le nom du corps Dia dans l'emplacement correspondant de la structure "ObjectOps" ou "ObjectTypeOps".

3.4. Plus loin dans la création des plug-ins

3.4.1. Un exemple concret

Supposons que le nom du plug-in est "ER", et que nous souhaitons disposer de deux formes, appelées "entity" et "relationship". Voici les différents fichiers correspondants:

Fichier principal du plug-in

```
#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include "intl.h"
#include "object.h"
#include "sheet.h"
#include "plug-ins.h"

/* il faut un ObjectType par forme, référencé ici: */
```

```
extern ObjectType entity_type;
extern ObjectType relationship_type;

DIA_PLUGIN_CHECK_INIT

PluginInitResult dia_plugin_init(PluginInfo *info)
{
    if(!dia_plugin_info_init(info, "ER", _("Description du plug-in"), NULL, NULL))
        return DIA_PLUGIN_INIT_ERROR;

    /* il faut déclarer tous les objets */
    object_register_type(&entity_type);
    object_register_type(&relationship_type);

    return DIA_PLUGIN_INIT_OK;
}
```

Fichier de forme

```
#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

/* inclure l'image xpm associée au bouton de la forme */
#include "pixmaps/entity.xpm"

/* pour faciliter la programmation */
typedef struct _Entity Entity;

struct _Entity
{
    /* champ obligatoire */
    Element element;

    /* tableau nécessaire pour conserver les connections */
    ConnectionPoint connections[8];

    /* et enfin, les attributs voulus */
}
    /* idem pour RelationShip */

    /* il faut alors réaliser le corps des fonctions */
```

3.4.2. Compilation avec C++

La création d'un plug-in Dia avec C++ est tout à fait possible. Les problèmes se posent:

- au niveau de la conception objet: les prototypes des fonctions doivent rester les mêmes qu'en C, et donc ne peuvent appartenir à une classe.
- au moment de l'édition de liens, différente en C++: il faut définir les fonctions interfaçant avec Dia au sein d'une section "extern "C" { }", qui indique à l'éditeur de liens la méthode à utiliser.

Conclusion

Avec du recul les choix techniques effectués m'ont semblé un temps moins évidents. En particulier, à certaines étapes de la réalisation où j'ai pû être confronté à des blocages. En fait ceux-ci une fois franchis marquent les progrès réalisés, ces expériences ont été très enrichissantes et me serviront pleinement à l'avenir.

Je regretterais de n'avoir eu plus de pratique préalable en programmation, surtout en applications graphiques C/C++, si les bases acquises en algorithmie, modèles relationnels et conception orientée objet n'avaient pas été suffisantes pour finalement l'appréhender sans trop de difficultés.

Au-delà des simples notions de programmation, j'ai redécouvert le monde du travail, sous un nouvel angle. En effet j'ai pu me rendre compte du temps et des investissements nécessaires à l'élaboration d'un projet informatique, ce dans une grande autonomie grâce à la confiance de Mr Nguyen que je remercie particulièrement. D'autre part la prise de contact m'a été facilitée par les documents réalisés par les précédents "intermittents", ceci ajoute positivement la nécessité de maintenir une documentation au projet, tant à destination interne qu'externe.

Ces différents éléments combinés à l'enjeu du stage m'ont motivé à m'y investir, et je suis ravi d'avoir pu apprendre et partager cette nouvelle expérience, tout en contribuant à faire avancer le projet.

Résumé et abstract

Cast est un logiciel facilitant la mise en oeuvre de calculs sur grappes de PC, typiquement de modélisation numérique. Développé par l'INRIA depuis 1995, le logiciel est complètement fonctionnel, mais son interface graphique est difficile à compiler, car la librairie utilisée est rarement rencontrée sur les systèmes actuels.

Le choix d'une nouvelle librairie graphique a ainsi été fait suivant sa popularité, mais aussi selon ses possibilités en termes de temps de développement, réutilisation du maximum de code de Cast, et droit d'utilisation par exemple. Ce choix s'est finalement porté sur l'écriture d'un plug-in à un logiciel existant, Dia, dont l'interface est similaire à celle de Cast.

Ce mémoire présente donc la démarche ayant amené à faire ce choix, puis détaille les différentes manières de réaliser un plug-in au logiciel Dia.

The software Cast aims to ease the implementation of calculations on PC clusters, typically of numeric modelisation. Developped by the INRIA since 1995, it is fully functional, but its graphic interface library is rarely found on current systems and so it's hard to compile.

The choice of a new graphic library has been determined by its popularity, but also in terms of development time, licensing, and ability to reuse most original Cast code for example. It was finally decided to write a plug-in to an existing software, called Dia, because its interface is very close to Cast's.

Therefore this report deals with the steps which led to this choice, and then explains how to develop different kinds of plug-ins for Dia.

Annexe A. Manuel de l’utilisateur de Cast

Annexe B. Procédure d’installation de Cast

Annexe C. Fichiers de définition XML