VERBATIM

Projet exploratoire RNRT 2005-2006

Sous projet SP4
Analyses statiques pour la
validation de codes

Livrable Lot3, Version 1.0

Elaboration de modèles par abstraction de données et d'opérateurs

Livrable Lot4, Version 1.0

Vérification de modèles et Génération de scénarios de test de programmes

Auteurs: B.d'Ausbourg et G.Durrieu
ONERA/DTIM - Consortium Verbatim
mailto:ausbourg@cert.fr,durrieu@cert.fr
http://www.onera.fr

Réseau National de Recherche en Télécommunications - Agence Nationale de la Recherche

Table des matières

Chapitre 1

Introduction

Ce rapport présente les résultats des lots 3 et 4 du sous-Projet SP4 du projet Verbatim. Ces deux lots avaient pour objet de traiter :

- l'élaboration de modèles par abstraction de données et d'opérateurs ;
- la vérification des modèles et la génération de scénarios de tests des programmes

Ces deux points sont abordés par le biais des choix d'implantation réalisés pour outiller ces deux processus : élaboration de modèles et vérification de ces modèles. On présente donc les choix d'abstraction qui ont été réalisés, les choix d'implantation opérés pour la génération de scénarios de test des applications. Des exemples de mise en oeuvre de vérification ou de génération de scénarios de test illustrent le caractère opératoire de la méthode.

Chapitre 2

Structure des programmes de l'application

L'application test est l'application *Pages Jaunes* développée au laboratoire CLIPS/IMAG. Les programmes constituant cette application sont entièrement codés en Java.

La multimodalité de l'application est implantée à l'aide des composants *ICARE* dont le code constitue une part importante de l'application. Pour autant, ces codes ne constituent pas à proprement parler le code de l'application. Ils constituent une couche de traitement et de fusion des multimodalités.

L'application proprement dite effectue un traitement simple des données acquises selon différentes modalités.

2.1 Composants ICARE

L'application fait appel à trois types de composants ICARE :

- les composants *Devices* : ceux-ci traitent des données d'entrée directement fournies par l'usager au clavier, à la souris, au micro ;
- les composants dits *Interaction Language* : ils convertissent un mot d'un langage d'entrée en un mot d'un langage de sortie conformément à une relation de correspondance établie par programmation ou donnée sur fichier ;
- les composants *ICARE* à proprement parler qui implantent les propriétés *CARE* des multimodalités : Complémentarité, Redondance, Equivalence.

La programmation JAVA de ces différents composants est complexe. Les composants de type *Devices* font par ailleurs appel à des classes spécialisées dans le traitement des périphériques concernés. Ces classes ne sont pas fournies avec le code de l'application.

2.1.1 Composants *Devices*

Les composants *Devices* étendent tous la classe `ICAREDevice` et sont les suivants :

- *KeyboardDirectInput* : acquisition des données clavier ;
- *MouseCoordinates* : acquisition de coordonnées à la souris ;
- *MouseCoordinatesOnClick* : acquisition de coordonnées sur clic à la souris ;

- *MouseDirectInput* : acquisition des données souris ;
- *MouseDirection* : acquisition de valeurs de déplacements opérés à la souris ;
- *ViaVoice* : acquisition de données vocales au micro.

Le programme implantant chacun de ces composants *Devices* est décrit dans un fichier Java. On trouve ainsi les fichiers :

- `ICAREDevice.java` ;
- `KeyboardDirectInput.java` ;
- `MouseCoordinates.java` ;
- `MouseCoordinatesOnClick.java` ;
- `MouseDirectInput.java` ;
- `MouseDirection.java` ;
- `ViaVoice.java` ;

2.1.2 Composants *Interaction Language*

Les composants dénommés langages d'interaction (*Interaction Language*) ne comportent qu'une classe : la classe *SimpleCommand*. Celle-ci réalise une stricte correspondance entre mots d'un langage d'entrée et mots d'un langage de sortie. L'ensemble des fichiers permettant d'implanter ces composants sont les suivants :

- `IcareInteractionLanguage.java` est le fichier qui contient la déclaration de la classe générique `ICAREInteractionLanguage` ;
- `SimpleCommand.java` : contient la déclaration de la classe `SimpleCommand` qui étend et raffine la classe `ICAREInteractionLanguage` ;
- `SimpleCommandBeanInfo.java` ;
- `SimpleCommandListener.java` définit l'interface des objets listeners qui peuvent être attachés à un composant `SimpleCommand` ;
- `SimpleCommandSupport.java` : déclaration de la classe `SimpleCommandSupport` qui propose des services d'attachement de listeners aux objets `SimpleCommand` et des services de déclenchement d'événements auprès de ces listeners.

2.1.3 Composants *ICARE*

Les composants *ICARE* sont implantés par des threads. Chacun d'eux gère une file d'évènements acquis, dans une fenêtre temporelle donnée (paramétrable), selon des politiques dictées par le type de propriétés attendues des différentes modalités : redondance, équivalence, ou complémentarité. Les deux composants *ICARE* utilisés dans le contexte de cette application sont les composants :

- *Complementarity* : complémentarité des modalités ;
- *RedundancyEquivalence* : implantation de la redondance ou de l'équivalence (la redondance n'est pas exigée).

Chacun de ces composants est implanté par un ensemble de fichiers java qui inclut :

- le programme associé au composant ; on trouve ainsi les fichiers :
  - `Complementarity.java` ;

  - `RedundancyEquivalence.java` ;
- une classe `BeanInfo` associée à chacun des composants ; on trouve ainsi les fichiers :
  - `ComplementarityBeanInfo.java` ;
  - `RedundancyEquivalenceBeanInfo.java` ;
- une interface `Listener` associée à chacun des composants ; on trouve ainsi les fichiers :
  - `ComplementarityListener.java` ;
  - `RedundancyEquivalenceListener.java` ;
- une classe `Support`, gérant le déclenchement des événements auprès des listeners, associée à chacun des composants ; on trouve ainsi les fichiers :
  - `ComplementaritySupport.java` ;
  - `RedundancyEquivalenceSupport.java` ;

2.2 Application Page Jaunes

L'application proprement dite illustre ce que pourrait être une application de localisation d'une adresse et de navigation sur un plan géographique. La multimodalité s'exerce sur les entrées d'information. Un usager peut entrer des informations d'entrée :

- par la souris ;
- par le clavier ;
- par la voix.

L'usager, en se fondant sur l'une ou l'autre de ces modalités, entre la spécification d'un nom ou d'une adresse qu'il recherche sur le plan affiché. Il peut également effectuer des opérations de contrôle de l'affichage telles que le déplacement sur le plan ou le zoom. Les opérations d'entrée :

- spécification d'un nom ;
- spécification d'une adresse ;
- demande de recherche ;

peuvent être réalisées aussi par saisie des données au sein de champs de saisie textuelle et par utilisation d'un bouton de déclenchement de la recherche.

Les fichiers qui permettent l'implantation de ces fonctions sont les suivants :

- `AfficheImage.java` : contient la description d'une classe `Canvas` et toutes les opérations sur l'image du plan affichée ;
- `DialogControler.java` : il s'agit d'un squelette de contrôleur de dialogue dont les opérations sont vides et ne déclenchent que des demandes de déclenchement de *feedback* auprès de l'interface Icare ;
- `FunctionalCore.java` : ce programme implanterait le corps fonctionnel de l'application ; il est vide dans le cadre de l'application fournie ;
- `Icare.java` : c'est le module de programmation de l'architecture ICARE ; il décrit la programmation de l'ensemble des composants ICARE, des traductions opérées sur les langages, des actions déclenchées par la reconnaissance des modalités exercées ;

- `IcareEnd.java` : il s'agit de la description d'un thread qui déclenche la terminaison de l'application sur réception de la lettre q lue à la console ;
- `IcareInterface.java` : ce texte java réalise l'interface entre l'architecture ICARE et le contrôleur de dialogue. Il comporte les méthodes qui permettent d'invoquer le contrôleur de dialogue lors du traitement d'événements ICARE et les méthodes qui peuvent être invoquées par le contrôleur de dilaogue sur demande de l'application ;
- `myFrame.java` : gestion du Frame d'affichage de l'application, contenant les champs texte d'interaction, le bouton de déclenchement de la recherche et le tracé du plan ;
- `myMappyFrame.java` : il s'agit d'un module permettant d'introduire le Frame *myFrame* dans la structure des composants ICARE et de proposer un nouveau composant d'interaction prenant comme langage d'entrée les interactions effectuées sur les widgets d'interaction du Frame ;
- `myMappyFrameBeanInfo.java` : comme pour les composants ICARE, module `BeanInfo` associé au composant ;
- `myMappyFrameListener.java` : comme pour les composants ICARE, module `Listener` associé au composant ;
- `myMappyFrameSupport.java` : comme pour les composants ICARE, module `Support` associé au composant.

Chapitre 3

Elaboration des modèles

L'objectif est de pouvoir construire des modèles d'exécution Java qui soient complets pour être traités dans la chaîne des outils de Bandera. Il doivent comporter la déclaration, la description et donc le code de tous les objets utilisés dans le programme. En particulier, il n'y a pas de classe importées. Tout doit être déclaré dans le cadre du modèle d'exécution.

Note. Dans la suite on notera en `petits` caractères les instructions faisant partie du code source original de l'application et en caractères de taille `normale` les instructions faisant partie du modèle d'exécution généré.

3.1 Fichiers Java analysés

Tous les fichiers de l'application ne sont pas nécessairement analysés. Ainsi, les fichiers qui constituent la programmation des composants ICARE n'ont pas besoin d'être analysés car les composants font l'objet de modèles prédéfinis. Les fichiers :

- `ICAREDevice.java` ;
- `KeyboardDirectInput.java` ;
- `MouseCoordinates.java` ;
- `MouseCoordinatesOnClick.java` ;
- `MouseDirectInput.java` ;
- `MouseDirection.java` ;
- `ViaVoice.java` ;
- `IcareInteractionLanguage.java`
- `SimpleCommand.java`
- `SimpleCommandBeanInfo.java`
- `SimpleCommandListener.java`
- `SimpleCommandSupport.java`
- `Complementarity.java`
- `RedundancyEquivalence.java`
- `ComplementarityBeanInfo.java`
- `RedundancyEquivalenceBeanInfo.java`

- `ComplementarityListener.java`
- `RedundancyEquivalenceListener.java`
- `ComplementaritySupport.java`
- `RedundancyEquivalenceSupport.java`

sont ainsi des fichiers dont l'analyse ne sera pas effectuée. D'une certaine manière cela est cohérent avec le fait que ces fichiers puissent être considérés comme déclarant des classes importées et utilisées par les programmes d'application. Parmi ces derniers, tous ne sont pas non plus analysés. Le fichier `AfficheImage.java` contient la déclaration d'une classe `Canvas` qui offre des services ou des actions sur dessins. Ces actions ont un impact sur la présentation du dessin (du plan) mais n'ont pas réellement d'impact sur la structure d'interaction multimodale de l'usager avec l'application. Le fichier `IcareEnd.java` traite la terminaison de l'application. Il n'est pas traité dans un premier temps. Seuls font l'objet d'une analyse pour la construction du modèle d'exécution de l'application les fichiers suivants :

- `DialogControler.java`
- `FunctionalCore.java`
- `Icare.java`
- `IcareInterface.java`
- `myFrame.java`
- `myMappyFrame.java`
- `myMappyFrameBeanInfo.java`
- `myMappyFrameListener.java`
- `myMappyFrameSupport.java`

L'analyse de ces fichiers a pour objet la création d'un modèle d'exécution Java de l'application. Ce modèle d'exécution est lui-même un programme Java qui intègre certaines déclarations de classes héritées de schémas de classes prédéfinis qui, regroupés, pourraient constituer des bibliothèques de modèles d'exécution.

3.2 Bibliothèque de modèles

3.2.1 Modèles de composants d'interaction

Les programmes font appel à des classes Java importées dont, en particulier, les librairies implantant les objets d'interaction. On se donne donc un ensemble de classes Java qui constitueront des modèles d'exécution de ces composants d'interaction.

Par exemple, la classe `JButton` aura le modèle suivant :

```
class JButton {
    ActionListener al;
    boolean visible = false;
    public void setVisible(boolean v) {
        visible = v;
    }
    public void addActionListener(ActionListener al){
        this.al = al;
    }
```

```
}
```

Ces modèles constituent en fait des sortes de *patterns* dont certains éléments devront subir une forme d'*instanciation*, qui sera décrite ultérieurement, conduisant, dans le modèle final, à la définition de classes du type :

```
class JButton {
    ActionListener1 al1;
    boolean visible = false;
    public void setVisible(boolean v) {
        visible = v;
    }
    public void addActionListener1(ActionListener1 al1){
        this.al1 = al1;
    }
}
```

Un autre interacteur utilisé est le `TextField` dont le *pattern* de modèle utilisé sera le suivant :

```
class JTextField{
    FocusAdapter fa;
    KeyAdapter ka;
    int idText=0;
    boolean visible = false;

    public int getText() {
        return idText;
    }
    public void setVisible(boolean v) {
        visible = v;
    }
    public void setText(int idtxt) {
        this.idText = idtxt;
    }
    public void addFocusListener(FocusAdapter fa){
        this.fa = fa;
    }
    public void addKeyListener(KeyAdapter ka){
        this.ka = ka;
    }
}
```

Ce modèle est succinct. Il reflète l'état d'une donnée `TextField` à travers :

- la valeur du texte qu'il contient (un entier car les chaînes de caractères sont représentées par des entiers) ;
- son état `visible` ou pas.

En plus de l'état de l'interacteur la classe permet d'associer deux listeners : un listener de focus et un listener de frappe au clavier. Les méthodes qui permettent cette association (de type `add...Listener`) sont standards dans la librairie Java. On en donne une version abstraite et

concise dans le modèle d'exécution.

Un autre interacteur utilisé dans le programme d'application est le `JFrame` dont le modèle que l'on se donne est très simple :

```
class JFrame {
    boolean visible = false;
    public JFrame () {
    }
    public void setVisible(boolean v) {
        visible = v;
    }
```

Ce modèle mémorise exclusivement l'état visible ou invisible du cadre. `JButton`, `TextField` et `JFrame` sont les trois seuls interacteurs rencontrés dans cette application. On se donne des *patterns* de modèles sur ces trois interacteurs seulement.

3.2.2 Composants ICARE

Bien que les programmes des composants ICARE fassent partie de la distribution des programmes fournis avec l'application, on applique la même approche que pour les librairies de composants d'interaction, considérant qu'il s'agit ici de composants d'interaction multimodale. On définit donc des classes qui représentent le fonctionnement opératoire des composants ICARE réalisant :

- la complémentarité de modalités ;
- la redondance ou l'équivalence des modalités.

Les composants de type *Devices* ou *Interaction Language* ne sont pas modélisés dans le modèle d'exécution. En revanche l'utilisation qui en est faite dans la classe `Icare` décrite dans le fichier `Icare.java` est analysée pour construire en particulier le modèle d'environnement.

Concrètement, on se donne, comme pour les composants d'objets d'interaction, des *patterns* de modèles instanciés (cela revient à donner un numéro d'instanciation aux classes décrivant ces *patterns*).

Complementarity. Ainsi, le modèle du premier composant `Complementarity` (dont le numéro d'instanciation est 1) sera donné par exemple par le texte :

```
class Complementarity1 {
    boolean data_1_set;
    boolean data_2_set;
    int d1;
    int d2;
    boolean order = true;

    ComplementaritySupport1
    complementaritySpt = new ComplementaritySupport1();

    public Complementarity1 () {
        this.init();
    }
```

```
    public void init() {
        data_1_set = false;
        data_2_set = false;
    }

    public void setData(int nbPort,int d){
        if (nbPort == 1) {
            data_1_set = true;
            d1 = d;
        }
        else if (data_1_set || (!order)){
            data_2_set = true;
            d2 = d;
        }
        if (data_1_set && data_2_set) {
                complementaritySpt.fireNewComplementarityData(d1,d2);
                data_1_set = false;
                data_2_set = false;
        }
    }

    public void addComplementarityListener(ComplementarityListener1 l) {
        complementaritySpt.addComplementarityListener(l);
    }
}
```

Ce modèle instancie (donne un numéro d'instanciation) les classes `Support` et `Listener` qui sont associées.

Les composants sont décrits dans les textes java qui les concernent. Ainsi le composant de complémentarité est décrit dans le fichier `Complementarity.java`. A ces textes Java s'ajoutent trois autres textes.

Le premier est celui du fichier `ComplementarityBeanInfo.java`. On ne traite pas ce texte.

Le second est un texte décrivant l'interface listener associé au composant et est contenu dans le fichier `ComplementarityListener.java`. On ne traite pas non plus directement ce fichier qui ne comporte que la déclaration d'un interface et pas du code implantant cet interface.

Le troisième est un texte décrivant un ensemble de moyens de support permettant d'associer plusieurs listeners au composant. Le fichier `ComplementaritySupport.java` est le fichier qui contient ce texte. Il décrit une classe `ComplementaritySupport` qui sera introduite dans le modèle d'exécution. Celle-ci abstraira le support offert en ne décrivant l'attachement possible que d'un seul listener. La classe, dans le texte source original, comporte une déclaration de vecteur de listeners :

```
public class ComplementaritySupport implements java.io.Serializable {
        Vector complementarityListeners = new Vector();
```

L'examen des instructions du listener qui invoquent la méthode `addElement` sur ce vecteur fait apparaître que les éléments `l` ajoutés sont de la classe `ComplementarityListener`. Il s'agit donc d'un vecteur d'objets de cette classe et on traduit donc cette déclaration de vecteur

comme la déclaration d'un listener de la classe `ComplementarityListener1` :

```
class ComplementaritySupport1 {
        ComplementarityListener1 l1;
```

La classe comporte ensuite une déclaration de méthode `addComplementarityListener()` qui permet de déclarer un listener attaché au composant. Cette méthode est introduite dans le modèle d'exécution Java comme :

```
public void addComplementarityListener(ComplementarityListener1 l1) {
    this.l1 = l1;
}
```

La classe comporte enfin une déclaration de méthode `fireNewComplementarityData()` qui accepte comme paramètre un objet de type `ICAREEvent` et qui consiste à invoquer auprès de tous les listeners enregistrés leurs méthodes `newComplementarityData()` en fournissant cet événement en paramètre à ces méthodes. Conformément à ce qui est décrit au paragraphe 3.3.2, cet événement est considéré comme la donnée de deux valeurs entières qui sont fournies en paramètre à la méthode `fireNewComplementarityData()` introduite dans le modèle. Cette dernière invoque alors la méthode `newComplementarityData()` du listener concerné :

```
class ComplementaritySupport1 {
        ComplementarityListener1 l1;
        public void addComplementarityListener(ComplementarityListener1 l1) {
            this.l1 = l1;
        }
        public void fireNewComplementarityData(int v0, int v1) {
                l1.newComplementarityData(v0,v1);
        }
}
```

On effectue rigoureusement la même démarche pour l'ensemble des composants rencontrés en faisant progresser le numéro d'instanciation pour chacun d'eux. Ainsi définira-t-on également dans le modèle d'exécution les classes :

- `Complementarity2` ;
- `ComplementarityListener2` ;
- `ComplementaritySupport2` ;
- `RedundancyEquivalence1` ;
- `RedundancyEquivalenceListener1` ;
- `RedundancyEquivalenceSupport1` ;

RedundancyEquivalence Le modèle de base du composant de redondance équivalence est le suivant.

```
class RedundancyEquivalence {
    boolean data_1_set;
    int d1;
    RedundancyEquivalenceSupport redundancyEquivalenceSpt =
         new RedundancyEquivalenceSupport();
    public RedundancyEquivalence () {
        this.init();
```

```
    }
    public void init() {
        data_1_set = false;
    }
    public void setData(int d)
    {
        d1 = d;
        data_1_set = true;
        if (data_1_set) {
                redundancyEquivalenceSpt.fireNewRedundancyEquivalenceData(d1);
                data_1_set = false;
        }
    }
    public void
    addRedundancyEquivalenceListener(RedundancyEquivalenceListener l) {
        redundancyEquivalenceSpt.addRedundancyEquivalenceListener(l);
    }
}
```

L'abstraction réalisée est très simple et se fonde sur le comportement du composant tel qu'il est programmé dans ICARE. Il s'agit ici d'un *pattern* du modèle du composant. Il faudra le numéroter et introduire dans le modèle les différentes instances de ce pattern. Par exemple on inroduira la première instance lorsque l'on rencontrera dans le texte la première instanciation d'un composant de type `RedundancyEquivalence`. Celle-ci donnera lieu à l'introduction dans le modèle de la déclaration de la classe :

```
class RedundancyEquivalence1 {
    boolean data_1_set;
    int d1;
    RedundancyEquivalenceSupport1 redundancyEquivalenceSpt =
         new RedundancyEquivalenceSupport1();
    public RedundancyEquivalence1 () {
        this.init();
    }
    public void init() {
        data_1_set = false;
    }
    public void setData(int d)
    {
        d1 = d;
        data_1_set = true;
        if (data_1_set) {
                redundancyEquivalenceSpt.fireNewRedundancyEquivalenceData(d1);
                data_1_set = false;
        }
    }
    public void
    addRedundancyEquivalenceListener(RedundancyEquivalenceListener1 l) {
```

```
        redundancyEquivalenceSpt.addRedundancyEquivalenceListener(l);
    }
}
```

dans laquelle sont également numérotées les classes référencées et associées à ce composant :

- classe `RedundancyEquivalenceSupport1`
- classe `RedundancyEquivalenceListener1`

Comme pour le composant `Complementarity` on est amené à construire, par examen du fichier `RedundancyEquivalenceSupport.java`, la classe support associée au composant d'équivalence ou de redondance selon le même principe qui a guidé la construction de la classe support du composant de complémentarité. Cette classe propose deux méthodes. La première permet d'enregistrer un listener. Et la seconde d'invoquer la méthode de traitement de l'événement écouté par ce listener.

```
class RedundancyEquivalenceSupport1 {
    RedundancyEquivalenceListener1  l;

        public void addRedundancyEquivalenceListener
                        (RedundancyEquivalenceListener1 l) {
            this.l = l;
        }

        public void fireNewRedundancyEquivalenceData(int v) {
                l.newRedundancyEquivalenceData(v);
        }
}
```

On notera que l'objet de type `ICAREEvent` passé à `fireNewRedundancyEquivalenceData` comme paramètre de méthode dans le texte du fichier `RedundancyEquivalenceSupport.java` est traduit ici en une donnée entière (et non un couple) parce que le composant de redondance ou d'équivalence réagit uniquement à la fourniture d'une donnée identique sur deux canaux d'entrée : l'identité des valeurs fournies est abstraite dans le modèle en cette valeur unique. Cette façon de procéder n'est pas très rigoureuse ni très propre. Elle est néanmoins pragmatique et une technique plus générale devrait lui être substituée par la suite : par exemple traiter un vecteur systématiquement comme une suite de trois entiers.

3.3 Traitement des classes de base

On construira systématiquement les abstractions de certaines classes de base de Java ou de certaines classes ICARE. Principalement parce que les objets de ces classes sont manipulés par des opérateurs qui sont des méthodes de ces classes et pas des opérateurs de base du langage Java. On ne redéclarera pas exhaustivement ces classes dans le modèle d'exécution. Cela reviendrait à les recoder. On va plutôt pratiquer des abstractions qui permettront de traiter ces classes de manière commode dans les modèles.

3.3.1 String

Le contenu des chaines de caractères n'a pas d'intérêt en soi dès lors qu'on s'intéresse au contrôle de l'exécution de l'application. Le principe retenu est de numéroter les chaines rencontrées. Ainsi la chaîne nulle aura pour numéro le numéro *0*. La première chaine rencontrée aura le numéro *1*, la seconde le *2* etc .... Une instruction du type :

```
if ((String)v.equals("ZOOM IN"))
```

sera alors traduite, si la chaine `"ZOOM IN"` est la chaîne d'index *i*, en déclarant la variable `v` de type `int` par

```
if (v == i)
```

3.3.2 ICAREEvent et vecteurs

Les composants ICARE seront abstraits. Ceux-ci communiquent par le biais d'événements de la classe `ICAREEvent`. Ces événements véhiculent des vecteurs de deux (ou trois) valeurs entières (ou parfois une seule). Dans les modèles, on ne reproduira que ces valeurs entières : soit une valeur entière `int v`, soit un couple de valeurs entières `int v0,v1`.

Cela implique que les classes `ICAREEvent` et `Vector` ne seront pas introduites dans les modèles. Ainsi une référence du type `data_values.getVector()` où `data_values` est du type `ICAREEvent` sera traduite en une référence au couple `v1,v2` de deux variables de type entier composant le vecteur de cet événement.

Ainsi des instructions impliquant un vecteur `Vector v`, dans un contexte où le vecteur a été remplacé par un entier `int v` , du type :

```
if (((String)(v.elementAt(0))).equals("DOWN"))
```

seront traduites par

```
if (v==i)
```

si `i` est le numéro de la chaîne `"DOWN"` en considérant qu'à l'invocation de la méthode permettant d'obtenir l'élément 0 du vecteur `v` (on `elementAt(0)`) fait correspondre la valeur de l'entier `v` dans le modèle d'exécution.

Lorsqu'au vecteur `Vector v` correspond un couple `v1,v2` d'entiers, alors la traduction de l'instruction :

```
if (((String)(v.elementAt(0))).equals("DOWN"))
```

est donnée par

```
if (v1==i)
```

et la traduction de l'instruction

```
if (((String)(v.elementAt(1))).equals("DOWN"))
```

est donnée par :

```
if (v2==i)
```

puisque `elementAt(1)` désigne le second élément du vecteur et par conséquent le deuxième élément du couple d'entiers `v1,v2` utilisé pour abstraire ce vecteur.

3.4 Modélisation de l'environnement

On introduit dans le modèle d'exécution l'environnement de l'application comme un thread `Environment` dont le squelette, qui sera enrichi, est le suivant et dont la fonction est d'émettre des événement d'interaction correspondant à des actions de l'usager :

```
class Environment extends Thread {
    boolean stop = false;
    Icare icare;

    public Environment (Icare i) {
        this.icare = i;
    }

    public void run() {
        while (!stop) {
            if (Bandera.choose()) {
                Bandera.generateScenarioTest("Chaîne décrivant l'événement");
                action_sur_un_composant  ;
            }
            else if (Bandera.choose()) {
                Bandera.generateScenarioTest("Chaîne décrivant un événement");
                action_sur_un_composant ;
            }
            else if (Bandera.choose()) etc...
        }
    }
}
```

Les instructions que l'on appelle `action_sur_un_composant` peuvent consister à envoyer des données sur un composant ICARE en simulant ainsi l'acquisition de données d'entrée fournies par l'usager. Ces données dans le programme alimentent, à travers une chaîne de composants de types *Devices* ou *Internation Language* qui traduisent leurs données en commandes d'application, les composants ICARE multimodaux. Ces instructions peuvent également simuler le fonctionnement de la machine virtuelle Java et activer les méthodes de listeners déclenchables sur événements d'interaction. Par exemple, le modèle d'environnement simulera l'appui sur un bouton interactif par l'invocation de la méthode `actionPerformed()` déclarée dans le listener attaché au widget de type `JButton` chargé d'implanter ce bouton dans l'interface.

On notera aussi que l'on introduit une variable `icare` de type `Icare` qui contiendra l'instance `Icare` transmise en paramètre par le constructeur de la classe. A travers cette variable, on pourra atteindre, depuis le modèle d'environnement, l'ensemble des composants ICARE.

3.5 Modélisation du noyau fonctionnel

Très simple, le noyau fonctionnel est codé dans le texte du fichier `FunctionalCore.java`. Ce fichier est quasiment vide et ne contient que la déclaration de deux méthodes :

- le constructeur de la classe ;
- une méthode de génération d'une chaîne de caractères

Le modèle généré reproduit ce quasi-vide en générant le texte Java suivant :

```
class FunctionalCore {
        public FunctionalCore() {
        }
}
```

3.6 Modélisation de l'architecture ICARE

L'architecture ICARE du programme d'application est principalement décrite dans le fichier `Icare.java`. C'est l'analyse de ce texte qui permettra de dégager un modèle d'exécution Java de l'architecture ICARE composant le programme.

3.6.1 Traitement des déclarations

Les déclarations des objets référençant des composants ICARE non directement modélisés ne sont pas prises en compte. Par exemple les déclarations d'objets de classe `SimpleCommand` ne sont pas analysées. Il en va de même des objets de type *Devices*. Quant à `IcareEnd` on ne traitera pas cette classe dans un premier temps. En revanche, on peut conserver les informations relatives aux autres objets de l'application.

D'un point de vue syntaxique, on enlèvera toutes les références aux paquettages dans les déclarations pour ne conserver que les noms des classes.

Ainsi conserve-t-on les déclarations :

```
IcareInterface ii;
Complementarity1 c1;
Complementarity2 c2;
RedundancyEquivalence1 re;
myMappyFrame frame;
```

On notera toutefois que pour les composants ICARE on crée autant de classes différentes qu'il y a de variables associées aux composants. Ainsi il y a deux composants `Complementarity` créés dans le programme et l'on définit deux classes `Complementarity1` et `Complementarity2`. On définit aussi une classe `RedundancyEquivalence1` car il n'y a qu'une création d'instance de cette classe.

3.6.2 Traitement du constructeur *Icare()*

Instanciations. On conserve les instructions d'instanciations des variables déclarées dans le modèle telles qu'elles ont été identifiées au paragraphe 3.6.1. Par exemple on conserve l'instruction :

```
ii = new IcareInterface(this);
```

sera conservée. Et l'instruction :

```
ie = new IcareEnd(this);
```

ne sera pas conservée puisque la variable `ie` n'a pas été déclarée.

Composants multimodaux ICARE. Si l'on prend l'introduction du composant de complémentarité des modalités, la séquence de code qui réalise cette introduction est la suivante :

```
c1 = new Complementarity.Complementarity();
c1.setDeltaT(1000);
c1.setNbOfComponents(2);

c1.addComplementarityListener(new Complementarity.ComplementarityListener() {
    public void newComplementarityData(ICARE.ICAREEvent data_values)
    {
        ii.compositecommands(data_values.getVector());
    }
});
```

L'instruction d'instanciation de la classe `Complementarity` est reportée dans le modèle d'exécution. Je ne sais plus pour quelle raison mais je dupplique toutes ces classes en les instanciant et en leur affectant un numéro d'instanciation. Par exemple ici, on rencontre la première instanciation de cette classe et on la numérote *1*. Concrètement cela signifie que la variable `c1` est déclarée d'une classe `Complementarity1` :

```
Complementarity1 c1;
```

La classe `Complementarity1` sera introduite dans le modèle par une forme d'instanciation du modèle (du *pattern*) `Complementarity`. On détaillera cette instanciation plus loin. Pour l'heure on introduit donc l'instanciation de la variable `c1` par :

```
c1 = new Complementarity1();
```

Les méthodes `setDelta()` et `setNbOfComponents()` ne sont pas intégrées dans les modèles des composants ICARE. On ne reproduit pas leur invocation dans le modèle d'exécution.

En revanche, la déclaration de listener est plus complexe. Elle est réalisée par invocation de la méthode `addComplementarityListener` sur le composant `c1`. Cette méthode accepte en paramètre un objet de type listener. Dans le texte du programme source, ce paramètre est donné par la création d'une classe implicite (ordre `new` sur la classe `ComplementarityListener`) dont la méthode activable est la méthode déclarée dans cette classe : `newComplementarityData()`. Les classes implicites sont mal traitées par les outils de Bandera, et rendent plus difficiles les opérations de slicing sur les programmes. Pour cette raison, dans les modèles d'exécution on construit explicitement ces classes en instanciant un *pattern* de la classe considérée. Par exemple, pour traiter l'instruction

```
new Complementarity.ComplementarityListener()
```

on construit une classe `ComplementarityListener1`, qui est une implantation du modèle de l'interface `ComplementarityListener` avec le numéro *1* parce qu'il s'agit de la première création d'instance de cette classe. On observe que la méthode `newComplementarityData` référence une instance `ii` de la classe `IcareInterface`. Cette variable `ii` référence, dans le programme Java source, un objet `IcareInterface` déclaré. On transfère cette référence dans la classe `ComplementarityListener1` en la transmettant par le constructeur de classe. Ainsi, le corps de cette classe devient :

```
class ComplementarityListener1{
```

```
    IcareInterface ii;
    public ComplementarityListener1 (IcareInterface ii) {
        this.ii = ii;
    }
}
```

On introduit ensuite dans le modèle le corps de la méthode `newComplementarityData` qui est la méthode activable du listener. Il s'agit du texte du programme initial, à la modification du paramètre près : le vecteur obtenu par `getVector()` sur un objet de classe `ICAREEvent` est abstrait en deux entiers `v0` et `v1` conformément à ce qui est expliqué au paragraphe 3.3.2 :

```
class ComplementarityListener1{
    IcareInterface ii;
    public ComplementarityListener1 (IcareInterface ii) {
        this.ii = ii;
    }
    public void newComplementarityData(int v0, int v1)
    {
        ii.compositecommands(v0,v1);
    }
}
```

On procède exactement de la même manière pour les séquences de code qui créent les composants de complémentarité `c2` et de redondance équivalence `re1`.

Concernant le composant de redondance équivalence, on introduit la déclaration de la déclaration de la classe `RedundancyEquivalenceSupport1` à la rencontre de l'instruction :

```
re = new RedundancyEquivalence.RedundancyEquivalence();
```

On introduit alors la déclaration de la classe dans le modèle d'exécution Java en lui affectant le numéro *1* ainsi qu'aux classes associées à ce composant :

- classe `RedundancyEquivalenceSupport1` ;
- classe `RedundancyEquivalenceListener1`.

Le texte introduit est alors le suivant, en se fondant sur le *pattern* défini en 3.2.2 :

```
class RedundancyEquivalence1 {
    boolean data_1_set;
    int d1;
    RedundancyEquivalenceSupport1 redundancyEquivalenceSpt =
         new RedundancyEquivalenceSupport1();
    public RedundancyEquivalence1 () {
        this.init();
    }
    public void init() {
        data_1_set = false;
    }
    public void setData(int d)
    {
        d1 = d;
        data_1_set = true;
        if (data_1_set) {
```

```
                redundancyEquivalenceSpt.fireNewRedundancyEquivalenceData(d1);
                data_1_set = false;
        }
    }
    public void
    addRedundancyEquivalenceListener(RedundancyEquivalenceListener1 l) {
        redundancyEquivalenceSpt.addRedundancyEquivalenceListener(l);
    }
}
```

L'analyse des instructions d'attachement d'un listener au composant `re` est similaire à celle effectuée pour le composant de complémentarité. On trouve la déclaration d'une classe implicite :

```
re.addRedundancyEquivalenceListener(
    new RedundancyEquivalence.RedundancyEquivalenceListener() {
        public void newRedundancyEquivalenceData(ICARE.ICAREEvent data_values)
        {
                ii.simplecommands(data_values.getVector());
        }
});
```

On adopte alors la même démarche en construisant dans le modèe d'exécution Java une nouvelle classe `RedundancyEquivalenceListener1`. Comme plus haut, la méthode activable par le listener référence la variable `ii` de la classe `IcareInterface`. Il s'agit d'une référence à un objet déclaré dans la classe `Icare`. On transmet cette référence à travers le constructeur de la nouvelle classe, dont la structure est alors initialement :

```
class RedundancyEquivalenceListener1{
    IcareInterface ii;

    public  RedundancyEquivalenceListener1 (IcareInterface ii) {
        this.ii = ii;
    }
}
```

On introduit ensuite le corps de la méthode déclenchable sur réception d'un événement par le listener. Cette méthode `fireNewRedundancyEquivalenceData` reprend dans le modèle le même texte que dans le programme original au typage du paramètre près. L'événement `ICAREEvent` est ici remplacé par une donnée `int v` :

```
class RedundancyEquivalenceListener1{
    IcareInterface ii;
    public  RedundancyEquivalenceListener1 (IcareInterface ii) {
        this.ii = ii;
    }
    public void newRedundancyEquivalenceData(int v)
    {
        ii.simplecommands(v);
    }
}
```

Ceci achève la construction des modèles de composants ICARE.

Composants Devices. On n'introduira pas de modèles des périphériques dans le modèle d'exécution Java. En d'autres termes, il n'y a pas de traduction des instructions liées auxobjets périphériques du code applicatif dans le modèle. On observe et analyse quand même ces instructions pour repérer quels dispositifs de commande (`SimpleCommand`) ces périphériques attaquent et pour repérer par conséquent quelle est la modalité du message fourni en entrée de ce dispositif de commande.

Par exemple, la séquence d'instructions

```
vvi = new viaVoiceInput.ViaVoiceInput();
vvi.getViaVoice().addViaVoiceListener(new viaVoiceInput.ViaVoiceListener() {
        public void newViaVoiceData(ICARE.ICAREEvent data_values) {
                sc2.setData(data_values);
                sc4.setData(data_values);
                sc5.setData(data_values);
                sc6.setData(data_values);
                sc7.setData(data_values);
                sc8.setData(data_values);
                sc9.setData(data_values);
        }
});
```

permet d'établir les faits suivants :

- la première instruction indique que la variable `vvi` est un périphérique de la classe `ViaVoiceInput` ; la modalité est vocale ;
- la deuxième instruction crée, par le biais d'une classe abstraite, un listener dont la méthode `newViaVoiceData` envoie des valeurs (les chaînes de caractères correspondant au message vocal reconu) à des dispositifs de commandes ; les variables `sc2`, `sc4`, `sc5`, `sc6`, `sc7`, `sc8`, et `sc9` sont des variables déclarées de la classe `SimpleCommand` ;
- les invocations des méthodes `setData()` sur ces variables indiquent qu'une entrée micro, et donc vocale, est fournie aux dispositifs de commandes `sc2`, `sc4`, `sc5`, `sc6`, `sc7`, `sc8`, et `sc9` ;

On procède de la même façon avec les autres blocs d'instructions déclarant à la fois des objets périphériques et leurs connexions à des dipositifs de commandes. Le même type d'analyse permet d'établir à partir de la séquence :

```
kdi = new keyboardDirectInput.KeyboardDirectInput();
kdi.addKeyboardInputListener(new keyboardDirectInput.KeyboardInputListener() {
        public void NewKeyboardEvent(ICARE.ICAREEvent data_values) {
                sc1.setData(data_values);
                sc3.setData(data_values);
        }
});
```

que les dispositifs de commandes correspondant aux variables `sc1` et `sc3` reçoivent des données issues du clavier.

Un peu plus complexe est l'analyse des textes relatifs au dispositif souris.

```
mdi = new mouseDirectInput.mouseDirectInput();

mdi.addMouseDirectInputListener(new mouseDirectInput.MouseDirectInputListener() {
        public void newMouseEvent(ICARE.ICAREEvent data_values)
        {
                mc.setData(data_values);
```

```
                mcoc.setData(data_values);
        }
});
```

L'analyse de cette construction permet également d'établir que les données souris sont transmises aux deux dispositifs associés aux variables `mc` et `mcoc` dont les classes sont données par les instructions qui suivent :

```
mc = new MouseCoordinates.MouseCoordinates();
mc.addMouseCoordinatesListener(new MouseCoordinates.MouseCoordinatesListener() {
        public void newMouseCoordinatesData(ICARE.ICAREEvent data_values)
        {
                c1.setData(2,data_values);
        }
});
mcoc = new MouseCoordinatesOnClick.MouseCoordinatesOnClick();
mcoc.addMouseCoordinatesOnClickListener(
        new MouseCoordinatesOnClick.MouseCoordinatesOnClickListener() {
        public void newMouseCoordinatesOnClickData(ICARE.ICAREEvent data_values)
        {
                ii.selection(data_values.getVector());
        }
});
```

On repère alors, toujours en effectuant le même type d'analyse et en observant le contenu des instructions des listeners déclarés, deux faits importants :

- les coordonnées souris vont consituer une entrée du composant ICARE de complémentarité numéro *1* ; comme il s'agit d'un dispositif périphérique répondant à une action de l'usager on introduit dans le modèle d'environnement, au sein de la méthode `run()`, le choix possible de cette interaction en insérant la séquence d'instructions suivantes qui affecte une valeur de coordonnées souris quelconque (ici la valeur arbitraire `255` censée encoder un déplacement en x et y) :

  ```
  else if (Bandera.choose()) {
      Bandera.generateScenarioTest("Mouse Move");
      icare.c1.setData(2,255);
  }
  ```

- les coordonnées sur clic déclenchent directement l'invocation de la méthode `selection` de l'objet `ii` de la classe `IcareInterface`. Cette construction n'est pas très cohérente avec la démarche ICARE, aussi la laissons-nous tomber pour l'instant. En toute rigueur, il faudrait, de la même façon, introduire dans le modèle d'environnement une séquence d'instructions :

  ```
  else if (Bandera.choose()) {
      Bandera.generateScenarioTest("Mouse Move on click");
      ii.selection(255);
  }
  ```

  après avoir préalablement déclaré dans la classe `Environment` une variable `ii` de la classe `IcareInterface` et avoir affecté à cette variable une valeur d'instance de cette classe passée en paramètre dans le constructeur de la classe. Mais, dans une première version, on ne le fait pas.

Composants Interaction Language. Ces composants sont de la classe `SimpleCommand` dans le programme d'application. Concernant les instances de cette classe, on notera les déclarations de variables pour pouvoir repérer dans la séquence des instructions que les objets concernés sont de ce type. Par exemple on ne traduit pas les instructions de déclaration du type :

```
SimpleCommand.SimpleCommand sc1;
```

mais on conserve l'information que `sc1` désigne un objet `SimpleCommand` dans la suite. . On sautera également les instructions d'affectation et d'instanciation du type :

```
sc1 = new SimpleCommand.SimpleCommand();
```

Les instructions invoquant la méthode `setCommandsFile()` seront ignorées. En revanche, les instructions d'invocation de la méthode `mapOneCommand()` sont à prendre en compte pour la génération du modèle d'environnement. En particulier, on s'intéressera exclusivement aux dispositifs recevant des données de composants *Devices*. En l'occurrence, dans le programmes d'application, les dispositifs associés aux variables (comme montré dans le paragraphe de traitement des composants *Devices*) `sc1`, `sc2`, `sc3`, `sc4`, `sc5`, `sc6`, `sc7`, `sc8` et `sc9`. Ainsi par exemple, la séquence d'instructions

```
sc1.mapOneCommand("C9 PRESSED","ZOOM IN");
sc1.mapOneCommand("D1 PRESSED","ZOOM OUT");
sc1.mapOneCommand("C8 PRESSED","UP");
sc1.mapOneCommand("D0 PRESSED","DOWN");
sc1.mapOneCommand("CB PRESSED","LEFT");
sc1.mapOneCommand("CD PRESSED","RIGHT");
sc1.mapOneCommand("39 PRESSED","CENTER");
sc1.mapOneCommand("1C PRESSED","SEARCH");
```

est traitée ainsi : on ne prend en compte que la première instruction d'invocation à la méthode `mapOneCommand()`. Car ces méthodes établissent une correspondance entre chaînes de données d'entrée et chaînes de données applicatives. On ne s'intéresse pas directement, dans un premier temps, à ces valeurs car on ne travaillera pas sur un modèle du coeur fonctionnel de l'application. Du point de vue de l'interaction et de la multimodalité, on ne retient qu'une déclaration de mapping (la première ou une autre), sachant que le premier paramètre fourni est alors la donnée d'entrée fourni par l'utilisateur. Ainsi, dans le cas présent, on retiendra la première et, dans le modèle d'environnement, on génèrera alors la séquence d'instructions Java simulant cette entrée de l'utilisateur :

```
else if (Bandera.choose()) {
    Bandera.generateScenarioTest("C9 PRESSED");
}
```

Le second paramètre de la méthode (la chaîne `"ZOOM IN"`) est la chaîne générée par le dispositif. On enregistre (éventuellement dans une liste ou une table *ad hoc*) que cette chaîne fait partie des données d'entrée pouvant être fournies à un composant. On applique le traitement réservé aux chaînex de caractèrex en lui associant un numéro comme décrit au paragraphe 3.3.1. On trouvera l'action à exécuter dans le corps de la méthode du listener qui sera associé au dispositif `sc1`;

```
sc1.addSimpleCommandListener(new SimpleCommand.SimpleCommandListener() {
        public void newCommand(ICARE.ICAREEvent data_values)
        {
                re.setData(data_values);
         }
});
```

l'analyse du listener recevant l'événement envoie une valeur au composant ICARE de redondance équivalence, ce qui est reflété par l'ajout de l'instruction dans le modèle d'environnement :

```
else if (Bandera.choose()) {
        Bandera.generateScenarioTest("Press Key : C9");
        icare.re.setData(12);
}
```

si `12` est le numéro de la chaîne de caractères associée au second paramètre de la méthode `mapOneCommand()`.

Création du Frame. Suivent ensuite des instructions relatives à la création du Frame de classe `myMappyFrame`. La première instruction rencontrée est l'instanciation de la classe :

```
frame = new myMappyFrame.myMappyFrame();
```

On introduit dans le modèle la même instanciation en supprimant les préfixes associés aux paquettages :

```
frame = new myMappyFrame();
```

Suit ensuite l'invocation de la méthode `addmyMappyFrameListener` prenant en paramètre un objet issu d'une création de classe implicite. Il s'agit d'une nouvelle classe dont le nom est `myMappyFrameListener`.

On crée alors la déclaration de cette classe :

```
class myMappyFrameListener { }
```

que l'on va enrichir. Tout d'abord, cette classe implicite est créée au sein de la classe `Icare` et manipule donc des objets de cette classe. On passe donc la référence de cette classe à une variable `icare` de type `Icare` déclarée dans cette classe. Cette variable est affectée par le constructeur de classe `myMappyFrameListener` qui reçoit en paramètre la référence à l'objet `Icare`.

```
class myMappyFrameListener {
    Icare icare;

    public myMappyFrameListener(Icare i){
        this.icare = i;
    }
}
```

Suit ensuite le corps de la classe `myMappyFrameListener` avec la déclaration des trois méthodes que sont :

- la méthode `search()` ;
- la méthode `changeTextField()` ;
- la méthode `specifyTextField()` ;

Chacune de ces méthodes prend en paramètre un objet de type `ICAREEvent` dont on a vu au paragraphe 3.3.2 qu'on le représente par des entiers. Le corps de ces méthodes invoque aussi des méthodes des composants ICARE déclarés et créés par le contructeur `Icare()`. Dans le modèle on reproduit ces méthodes, au transtypage du paramètre près, et en accèdant aux composants ICARE à travers la référence à l'objet `icare` :

```
class myMappyFrameListener {
```

```
    Icare icare;
    public myMappyFrameListener(Icare i){
        this.icare = i;
    }
    public void search (int data_values)
    {
        icare.re.setData(data_values);
    }
    public void changeTextField(int data_values)
    {
        icare.c2.setData(1,data_values);
    }
    public void specifyTextField(int data_values)
    {
        icare.c2.setData(2,data_values);
    }
}
```

Une fois cette classe déclarée, on peut enrichir le code du modèle d'exécution et de la classe `Icare` de ce modèle. Pour cela on déclare un listener de la classe `myMappyFrameListener` nouvellement introduite :

```
myMappyFrameListener listener;
```

et l'on crée alors une instance de cette classe par l'introduction dans le modèle de l'instruction :

```
listener = new myMappyFrameListener(this);
```

où `this` désigne alors l'instance de classe `Icare` dont la référence sera transmise à l'instance de classe `myMappyFrameListener`.

Il reste alors à attacher le nouveau listener à l'objet `frame` comme le fait, dans le code de l'application, l'instruction :

```
frame.addmyMappyFrameListener(
     new myMappyFrame.myMappyFrameListener() {...} )
```

On réalise la modélisation de cet attachement par l'introduction dans le modèle de l'instruction :

```
frame.addMyMappyFrameListener(listener);
```

Traitement de startIcare(). Le code de l'application fait ensuite appel à la méthode `startIcare()`. Les instructions de cette méthode invoquent les méthodes `start` des différents composants ICARE car ces derniers sont implantés, dans l'application, par des threads. Ces invocations n'ont donc pas lieu d'être reportées dans le modèle d'exécution à l'exception de la dernière qui porte sur l'objet `frame`. On l'introduit donc directement par ajout, dans le modèle d'exécution, de cette instuction :

```
frame.startLanguage();
```

3.7 Modélisation du JFrame d'interaction

Outre les dispositifs d'entrée vocaux et clavier, on peut considérer l'utilisation d'un `JFrame` comme un dispositif d'entrée programmé par l'application. La programmation de ce dispositif est décrite dans l'ensemble des fichiers qui suivent :

- `myFrame.java`
- `myMappyFrame.java`
- `myMappyFrameBeanInfo.java`
- `myMappyFrameListener.java`
- `myMappyFrameSupport.java`

Chacun de ces fichiers définit une classe. On traitera chacun d'eux à l'exception du fichier `myMappyFrameBeanInfo.java`. On introduira dans le modèle d'exécution Java ces différentes classes dont l'abstraction sera construite à travers l'analyse des différents fichiers qui les déclarent dans le programme.

3.7.1 Analyse de `myFrame.java`

Ce fichier déclare une classe étendant la classe `JFrame` :

```
public class myFrame extends javax.swing.JFrame {
```

On introduira une déclaration de cette classe dans le modèle par :

```
class myFrame extends JFrame {}
```

Traitement des déclarations. Le texte de ce fichier comporte un certain nombre de déclarations de variables. On ne retient que les variables d'intérêt qui sont ici principalement les interacteurs élémentaires, sans prendre en compte leurs conteneurs éventuels. Les déclarations d'intérêt sont donc, dans ce texte, les suivantes :

```
public javax.swing.JTextField jTextField1 = new javax.swing.JTextField();
public javax.swing.JTextField jTextField2 = new javax.swing.JTextField();
javax.swing.JButton jButton1 = new javax.swing.JButton();
```

que l'on traduira dans le modèle par leur correspondant direct en créant une classe numérotée :

```
    JTextField1 jTextField1 = new JTextField1();
    JTextField2 jTextField2 = new JTextField2();
    JButton1 jButton1 = new JButton();
```

Il faut noter ici la déclaration, dans le modèle, des classes `JTextField1`, `JTextField2` et `JButton1` dont le texte sera alors donné en numérotant également les éventuelles classes associées (listeners, adapteurs ou supports) déclarées dans les déclarations des classes introduites :

```
class JButton1 {
    ActionListener1 al1;
    boolean visible = false;
    public void setVisible(boolean v) {
        visible = v;
    }
    public void addActionListener1(ActionListener1 al1){
        this.al1 = al1;
```

```
    }
}
class JTextField1{
    FocusAdapter1 fa1;
    KeyAdapter1 ka1;
    int idText=0;
    boolean visible = false;
    public int getText() {
        return idText;
    }
    public void setVisible(boolean v) {
        visible = v;
    }
    public void setText(int idtxt) {
        this.idText = idtxt;
    }
    public void addFocusListener1(FocusAdapter1 fa1){
        this.fa1 = fa1;
    }
    public void addKeyListener1(KeyAdapter1 ka1){
        this.ka1 = ka1;
    }
}

class JTextField2{
    FocusAdapter2 fa2;
    KeyAdapter2 ka2;
    int idText=0;
    boolean visible = false;
    public int getText() {
        return idText;
    }
    public void setVisible(boolean v) {
        visible = v;
    }
    public void setText(int idtxt) {
        this.idText = idtxt;
    }
    public void addFocusListener2(FocusAdapter2 fa2){
        this.fa2 = fa2;
    }
    public void addKeyListener2(KeyAdapter2 ka2){
        this.ka2 = ka2;
    }
}
```

On trouve ensuite une autre déclaration. Celle de la variable `mother` dans la déclaration :

```
myMappyFrame mother;
```

Cette déclaration touche à un objet applicatif dont la classe sera déclarée dans le modèle. On l'introduit donc dans l'ensemble des déclarations du modèle d'exécution Java.

Traitement du constructeur. Le constructeur réalise la création de la classe et affecte la variable `mother` avec la référence qui lui est transmise en paramètre. On conserve la structure du constructeur :

```
public myFrame(myMappyFrame theMappyFrame ) {
     mother = theMappyFrame;
}
```

Initialisation des composants. La méthode `initComponents()` est celle qui réalise cette initialisation. Elle comporte les instructions d'initialisation des composants d'interaction utilisés par l'application. On ne retient, dans un premier filtrage, que les instructions qui concernent les objets d'interaction d'intérêt dont on a conservé la déclaration et la création dans le modèle :

- `jTextField1` ;
- `jTextField2` ;
- `jButton1` ;

Dans ce sous-ensemble d'instructions, on ne retient que les instructions qui invoquent des méthodes déclarées dans les modèles des interacteurs. Par exemple, dans le modèle Java, la classe `jTextField1` comporte une méthode `setVisible()`. On traduira par conséquent l'instruction :

```
jTextField1.setVisible(true);
```

rencontrée dans le texte source du fichier `myFrame.java` par sa traduction à l'identique :

```
jTextField1.setVisible(true);
```

En revanche on ne traduira pas l'instruction source :

```
jTextField2.setLocation(new java.awt.Point(80, 60));
```

parce que le modèles de classe `jTextField1` ne déclare pas de méthode `setLocation()`. L'instruction d'affectation de texte :

```
jTextField1.setText("");
```

est traduite, en considérant que la chaîne de caractère vide `""` est la chaîne de numéro 0, par l'instruction suivante placée dans le modèle :

```
jTextField1.setText(0);
```

On applique ce procédé aux instructions relatives aux trois composants d'interaction retenus. Cela produit, dans le modèle, la séquence d'instructions :

```
public void initComponents() {
     jTextField1.setText(0);
     jTextField1.setVisible(true);
     jTextField2.setText(0);
     jTextField2.setVisible(true);
     jButton1.setVisible(true);
```

On trouve ensuite dans le texte du programme `myFrame.java` un ensemble d'attachements de listeners, créés à travers des déclarations de classes implicites. Le mécanisme de traitement est toujours le même. On prend, pour le décrire, le premier attachement déclaré :

```
jTextField1.addFocusListener(new java.awt.event.FocusAdapter() {
        public void focusGained(java.awt.event.FocusEvent e) {
                jTextField1FocusGained(e);
        }
});
```

L'invocation de `addFocusListener()` s'effectue avec un paramètre qui est un objet de la classe `java.awt.event.FocusAdapter`. Cette classe n'a pas été introduite dans le modèle. C'est un adapteur c'est à dire que c'est une classe déjà définie dont on peut redéfinir des méthodes. L'instruction `new` crée donc un objet de cette classe qui doit donc être introduite dans le modèle d'exécution. On introduit donc le squelette de cette classe en la numérotant avec le numéro 1 car il s'agit de la première création d'adapteur :

```
class FocusAdapter1 {
    public FocusAdapter1 () {}
}
```

La déclaration implicite de l'objet `FocusAdapter` dans le texte source construit la classe en redéfinissant la méthode `focusGained()`. Il est donc néecssaire de la redéfinir également dans le modèle. Cette méthode, dans le texte source, admet comme paramètre un objet `e` de classe `FocusEvent`. On doit donc également introduire dans le modèle une déclaration de cette classe et la déclaration de la méthode `focusGained()` dans la classe `FocusAdapter` :

```
class FocusEvent {}

class FocusAdapter1 {
    public FocusAdapter1 () {}
    public void focusGained(FocusEvent e) {}
}
```

Dans le texte source, le code de `focusGained()` se limite à l'invocation d'une méthode `jTextField1FocusGained()` déclarée par ailleurs dans le corps de la classe `myFrame`. Il s'agit donc d'une méthode de cette classe, qui, dans le modèle, sera invoquée depuis la méthode `focusGained()` déclarée dans la classe `FocusAdapter1` elle-même déclarée séparément de la classe `myFrame`. Il faut donc fournir à la classe `FocusAdapter1` une référence à la classe `myFrame` pour que la méthode `jTextField1FocusGained()` de la classe `myFrame` soit référençable. On fait passer cette référence en paramètre à travers le constructeur de la classe `FocusAdapter1` :

```
class FocusEvent {}

class FocusAdapter1 {
    myFrame frame;
    public FocusAdapter1 (myFrame f) {
        frame = f;
    }
    public void focusGained(FocusEvent e) {}
}
```

On peut ensuite introduire au sein du modèle et dans le corps de la méthode `focusGained()` la traduction de l'instruction trouvée dans le texte source. Il s'agit de l'invocation à une

méthode définie par ailleurs, `jTextField1FocusGained(e)`, invocation que l'on reproduit en précisant que la méthode s'invoque à travers une référence à l'objet `myFrame` déclaré :

```
class FocusEvent {}

class FocusAdapter1 {
    myFrame frame;
    public FocusAdapter1 (myFrame f) {
        frame = f;
    }
    public void focusGained(FocusEvent e) {
        frame.jTextField1FocusGained(e);
    }
}
```

Il reste enfin à réaliser l'attachement, dans le modèle et dans la classe `myFrame` déclarée dans ce modèle, d'un adapteur à l'objet d'interaction `jTextField1` de classe `JTextField1`. Cela est réalisé en créant une déclaration *explicite* d'un adapteur `f1` de la classe `FocusAdapter1` qui vient d'être créée

```
FocusAdapter1 f1;
```

et en attachant cet adapteur à l'objet d'interaction dans le corps des instructions d'initialisation des composants :

```
public void initComponents() {
     jTextField1.setText(0);
     jTextField1.setVisible(true);
     jTextField2.setText(0);
     jTextField2.setVisible(true);
     jButton1.setVisible(true);

     f1 = new FocusAdapter1(this);
     jTextField1.addFocusListener1(f1);
```

Ceci termine le traitement de l'attachement d'un listener créé implicitement à l'objet d'interaction `jTextField1`. On procède exactement de la même manière pour traiter les autres attachements de listeners, créés aussi implicitement, aux trois objets d'interaction retenus. On introduit ainsi une nouvelle classe similaire `FocusAdapter2` :

```
class FocusAdapter2 {
    myFrame frame;
    public FocusAdapter2 (myFrame f) {
        frame = f;
    }
    public void focusGained(FocusEvent e)
    {
        frame.jTextField2FocusGained(e);
    }
}
```

On introduit aussi successivement dans le modèle, et toujours en utilisant les mêmes procédés, les classes `KeyEvent`, `KeyAdapter1` et `KeyAdapter2` :

```
class KeyEvent {}
class KeyAdapter1 {
    myFrame frame;
    public KeyAdapter1 (myFrame f) {
        frame = f;
    }
    public void KeyReleased(KeyEvent e)
    {
        frame.jTextField1KeyReleased(e);
    }
}
class KeyAdapter2 {
    myFrame frame;
    public KeyAdapter2 (myFrame f) {
        frame = f;
    }
    public void KeyReleased(KeyEvent e)
    {
        frame.jTextField2KeyReleased(e);
    }
}
```

On introduit enfin la déclaration des classes permettant l'attachement d'un listener au bouton `jButton1`. Il s'agit des classes `ActionEvent` et `ActionListener1` :

```
class ActionEvent{}

class ActionListener1 {
    myFrame frame;
    public ActionListener1 (myFrame f) {
        frame = f;
    }
    public void actionPerformed(ActionEvent e)
    {
        frame.jButton1ActionPerformed(e);
    }
}
begin{verbatim}
```

Les instructions de déclaration et de création de tous ces listeners sont traduites en introduisant dans le modèle, dans la classe `myFrame` les déclarations :

```
KeyAdapter1 k1;
FocusAdapter2 f2;
KeyAdapter2 k2;
ActionListener1 al1;
```

ainsi que les instructions de création et d'attachement suivantes :

```
k1 = new KeyAdapter1(this);
jTextField1.addKeyListener1(k1);

f2 = new FocusAdapter2(this);
jTextField2.addFocusListener2(f2);

k2 = new KeyAdapter2(this);
jTextField2.addKeyListener2(k2);

al1 = new ActionListener1(this);
jButton1.addActionListener1(al1);
```

ce qui termine la modélisation de l'initialisation des composants. Il reste alors à introduire dans la classe `myFrame` la modélisation des méthodes invoquées par les listeners.

Traitement des méthodes des listeners. Les méthodes déclenchées par les listeners sont des méthodes déclarées dans le texte source de la classe `myFrame`. Ces méthodes sont les suivantes :

- `jTextField1FocusGained` ;
- `jTextField2FocusGained` ;
- `jTextField1KeyReleased` ;
- `jTextField2KeyReleased` ;
- `jButton1ActionPerformed` ;

Le corps et contenu de ces méthodes sont donnés par les lignes suivantes du texte source de `myFrame.java` :

```
public void jButton1ActionPerformed(java.awt.event.ActionEvent e)
{
        mother.search();

}
public void jTextField1FocusGained(java.awt.event.FocusEvent e) {
        mother.changeTextField("Name");
}
public void jTextField2FocusGained(java.awt.event.FocusEvent e) {
        mother.changeTextField("Address");
}
public void  jTextField1KeyReleased(java.awt.event.KeyEvent e)
{
        mother.changeTextField("Name");
        mother.specifyTextField(jTextField1.getText());
}
public void  jTextField2KeyReleased(java.awt.event.KeyEvent e)
{
        mother.changeTextField("Address");
        mother.specifyTextField(jTextField2.getText());
}
```

Ces méthodes font systématiquement appel à des méthodes de l'objet `mother` de la classe `myMappyFrame`. On reproduit ces instructions quasiment à l'identique dans le modèle d'exécution Java, à la représentation des chaînes de caractères près : la chaîne `"Name"` est la chaîne de numéro 1 tandis que la chaîne `"Address"` sera représentée par l'index 2. On introduit donc, dans le code de la classe `myFrame`, les déclarations de méthodes suivantes :

```
public void jTextField1FocusGained (FocusEvent e) {
    mother.changeTextField(1);
}
public void jTextField1KeyReleased(KeyEvent e) {
    int text;
    mother.changeTextField(1);
    text = jTextField1.idText;
    mother.specifyTextField(text);
}
public void jTextField2FocusGained (FocusEvent e) {
    mother.changeTextField(2);
}
public void jTextField2KeyReleased(KeyEvent e) {
    int text;
    mother.changeTextField(2);
    text = jTextField2.idText;
    mother.specifyTextField(text);
}
public void jButton1ActionPerformed(ActionEvent e) {
    mother.search();
}
```

On notera toutefois que l'expression `jTextField1.getText()` qui constitue le paramètre donné, dans le texte source, à la méthode `specifyTextField`, est remplacé dans le modèle par un accès direct à l'attribut `idText`. Il s'agit ici d'une modification qui est nécessaire et sans laquelle le processus de slicing (et de simplification des modèles Java d'exécution) risque de ne pas converger. En l'état, mieux vaut effectuer cette modification du texte qui conserve toute la sémantique du texte initial.

Le texte `myFrame.java` comporte également des déclarations de méthodes relatives à des manipulations ou des opérations de navigation sur l'image affichée. On ne retient pas a priori ces méthodes à l'exception d'une qui peut être invoquée par la réalisation d' une des actions possibles d'un usager : `zoomIn()`. On en propose une abstraction très succinte et très simple :

```
public void zoomIn() {
    doneZoomIn = true;
    doneZoomIn = false;
}
```

Cette abstraction ne fait qu'enregistrer le fait que la méthode est invoquée. Pour cela on utilise la variable booléenne `doneZoomIn` qui est positionnée à la valeur *true* puis *false*. Il aura fallu préalablement déclarer cette variable dans le modèle, au sein des déclarations de la classe `myFrame` par une instruction du type :

```
boolean doneZoomIn = false;
```

3.8 Modélisation de myMappyFrame

On analyse et abstrait les textes java contenus dans les deux fichiers :

- `myMappyFrame.java` ;
- `myMappyFrameSupport.java` ;

Traitement des déclarations. Le premier fichier, `myMappyFrame.java`, comporte la déclaration de la classe `myMappyFrame` extension de la classe `ICAREInteractionLanguage` :

```
public class myMappyFrame extends ICARE.ICAREInteractionLanguage
```

On introduit alors, dans le modèle, la déclaration de la classe `myMappyFrame` :

```
class myMappyFrame {}
```

Suivent, dans le texte source de `myMappyFrame.java` deux déclarations d'attributs :

```
myFrame frame;
private myMappyFrameSupport myMappyFrameSpt = new myMappyFrameSupport();
```

La première déclaration concerne une variable de la classe `myFrame` qui a déjà été introduite dans le modèle. On reproduit cette déclaration dans le modèle. La seconde déclaration concerne un objet de la classe `myMappyFrameSupport` qui est une classe qui n'a pas encore été introduite dans le modèle. Cette classe est décrite dans le fichier `myMappyFrameSupport.java` et sera introduite lorsque l'on analysera ce fichier. On la reproduit dans le modèle en supprimant néanmoins dans le modèle le restricteur d'accès `private`.

```
class myMappyFrame {
        myFrame frame;
        myMappyFrameSupport myMappyFrameSpt = new myMappyFrameSupport();
```

Viennent ensuite la déclaration de la méthode constructeur qui est vide dans le texte source et la déclaration d'une méthode `getFrame()` qui retourne la variable `frame` qui a fait l'objet d'une déclaration dans le modèle. On conserve donc dans le modèle ces deux déclarations de méthodes en l'état :

```
public myMappyFrame(){}

public myFrame getFrame()
{
        return frame;
}
```

Traitement de l'initialisation. La méthode `startLanguage` est une méthode d'initialisation dont le texte est le suivant :

```
public void startLanguage()
{
        try
        {
                try
                {
                UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
                }
                catch (Exception e) {
```

```
                }

                frame = new myFrame(this);
                frame.initComponents();
                frame.setVisible(true);
        }
        catch (Exception e) {
                e.printStackTrace();
        }
}
```

Les instructions de cette méthode sont contrôlées par des constructions `try ... catch` que l'on ne reproduit pas dans le modèle d'exécution. Par ailleurs, l'invocation de la méthode `setLookandFeel` de la classe `UIManager` ne sera pas non plus introduite puisque cette classe est absente du modèle. On ne conserve que les trois instructions qui affectent les objets retenus dans le modèle :

```
public void startLanguage()
{
        frame = new myFrame(this);
        frame.initComponents();
        frame.setVisible(true);
}
```

Traitement des méthodes de récupération d'événements. Le texte décrivant la classe `myMappyFrame` déclare ensuite un ensemble de méthodes qui sont les méthodes invoquées depuis les méthodes de l'objet `myFrame` elles-mêmes invoquées par les listeners d'événements déclenchés par les interacteurs de `myFrame`.

La première méthode est la méthode `search()` dont le contenu est le suivant :

```
public void search()
{
1       ICARE.ICAREEvent theEvent = new ICARE.ICAREEvent(this);

2       Vector vectorTemp = new Vector();
3       vectorTemp.addElement(new String("SEARCH"));

4       theEvent.setVector(vectorTemp);
5       theEvent.setNbOfValues(1);
6       theEvent.setConfidenceFactor(100);
7       theEvent.setTime(System.currentTimeMillis());
8       theEvent.setInitialTime(System.currentTimeMillis());

9       myMappyFrameSpt.fireSearch(theEvent);
}
```

Cette méthode comporte un ensemble d'instructions dont l'objectif est de constituer une structure d'événement `theEvent` de la classe `ICAREEvent` et de passer cette structure à la méthode `fireSearch` de l'objet support `myMappyFrameSpt`. Le paragraphe 3.3.2 expliquait que la classe `ICAREEvent` n'est pas introduite dans le modèle et que le vecteur qui accompagne tout objet de cette classe est représenté à travers une donnée entière ou un couple de données entières. L'examen des instructions de la méthode permet de faire les constatations suivantes.

La première instruction (1) permet de repérer que la variable `theEvent` désigne la structure d'événement de classe `ICAREEvent`. L'instruction 2 permet de repérer que la variable `vectorTemp` est une structure de vecteur de la classe `Vector`. L'instruction 3 ajoute un élément au vecteur par le biais de l'invocation de la méthode `addElement`. On prend le paramètre de cette méthode comme la valeur ajoutée au vecteur et donc la valeur à conserver : il s'agit d'une chaîne de caractère `"SEARCH"` que l'on transformera en valeur numérique d'index. L'instruction 4 permet de s'assurer que `vectorTemp` est bien le vecteur associé à l'événement `theEvent`. Les instructions `5,6,7` et 8 concernent la structure d'événement et ne sont donc pas reprises dans le modèle. L'instruction 9 est celle qui fait appel à la méthode de déclenchement des listeners par l'objet `myMappyFrameSpt`. On la conserve donc mais en lui affectant comme paramètre la valeur entière qui représente la chaîne transmise par l'événement :

```
public void search()
{
    myMappyFrameSpt.fireSearch(5);
}
```

La seconde méthode de récupération d'événement est la méthode `changeTextField` dont le texte est le suivant :

```
public void changeTextField(String theValue)
{
1       ICARE.ICAREEvent theEvent = new ICARE.ICAREEvent(this);

2       Vector vectorTemp = new Vector();
3       vectorTemp.addElement(theValue);

4       theEvent.setVector(vectorTemp);
5       theEvent.setNbOfValues(1);
6       theEvent.setConfidenceFactor(100);
7       theEvent.setTime(System.currentTimeMillis());
8       theEvent.setInitialTime(System.currentTimeMillis());

9       myMappyFrameSpt.fireChangeTextField(theEvent);
}
```

La structure de cette méthode ressemble à celle de la précédente. Elle en diffère cependant par le fait qu'elle accepte un paramètre de classe `String`. On a indiqué au paragraphe 3.3.1 que les objets de classe `String` sont représentés par des objets entiers `int`. Dans le modèle cette méthode reçoit un paramètre `int` :

```
public void changeTextField(int theValue) {}
```

La même analyse des instructions `1,2,3` et 4 que celle effectuée pour la méthode précédente `search` permet de conclure que le paramètre `theValue`, représenté dans le modèle par la valeur entière de la chaîne effectivement passée en paramètre, est la valeur également transmise à la méthode `fireChangeTextField` de déclenchement du listener associé. Dès lors cette méthode est traduite dans le modèle par :

```
public void changeTextField(int theValue)
{
        myMappyFrameSpt.fireChangeTextField(theValue);
}
```

Le même raisonnement appliqué à la méthode `specifyTextField` permet de conduire à

une représentation de cette méthode dans le modèle qui prend la forme :

```
public void specifyTextField(int theValue)
{
        myMappyFrameSpt.fireSpecifyTextField(theValue);
}
```

Les deux méthodes `selectNameLabel` et `selectAdressLabel` déclarées ensuite dans le texte source de `myMappyFrame.java` invoquent des méthodes `requestFocus` et `grabFocus` sur les objets `jTextfield1` ou `jTextField2`. Ces méthodes n'ont pas été définies dans les modèles `JTextField1` ni `JTextField2`. En conséquence ces méthodes ne sont pas introduites dans le modèle d'exécution.

Les deux dernières méthodes concernent l'ajout ou le retrait d'un listener de l'objet `MyMappyFrame`. Ces deux méthodes font appel à des méthodes définies au sein de l'objet support `myMappyFrameSpt`. On pourrait les reproduire telles quelles dans le modèle. En fait on ne reproduit que la méthode spécifiant quel est le listener ajouté car, du point de vue du modèle, c'est le lien qui importe plutôt que sa dynamique (attachement et retrait). On introduit donc dans le modèle la méthode :

```
public void addMyMappyFrameListener(myMappyFrameListener l) {
    myMappyFrameSpt.addMyMappyFrameListener(l);
}
```

ce qui achève la construction de la classe `myMappyFrame`.

Traitement de la classe support de myMappyFrame. `myMappyFrameSupport.java` est le fichier qui contient la déclaration de la classe `myMappyFrameSupport` qui décrit les objets supports des objets de classe `myMappyFrame`. On procède, pour les analyser, de la même façon que l'on a procédé au paragraphe 3.2.2. On introduira donc une description abstraite de cette classe dans le modèle. Là où la classe du texte source original permet l'enregistrement et la gestion d'un vecteur de listeners, dans le modèle, on ne dotera la classe que d'un seul listener. Le texte orginal introduit la classe par la déclaration d'un vecteur de listeners :

```
public class myMappyFrameSupport implements java.io.Serializable {
        Vector myMappyFrameListeners = new Vector();
```

L'examen des invocations de méthodes `addElement` sur `myMappyFrameListeners` montre que les objets additionnés au vecteur sont de la classe `myMappyFrameListener`. On remplace, dans le modèle, la déclaration du vecteur de listeners de ce type par la déclaration d'une variable listener de cette classe :

```
class myMappyFrameSupport {
    myMappyFrameListener l;
```

La déclaration de la méthode `addmyMappyFrameListener` suit ensuite : elle permet de déclarer un nouveau listener attaché au composant `myMappyFrame` et de l'intégrer dans la liste. Dans le modèle, la liste est réduite au seul listener déclaré et on affecte donc la référence du listener passé en paramètre à la variable `l` déclarée :

```
public void addMyMappyFrameListener(myMappyFrameListener l) {
    this.l=l;
}
```

La seconde méthode `removeMyMappyFrameListener` ne sera pas modélisée : elle retire, dans le texte original, un élément listener du vecteur des listeners. On n'incorpore pas ce fait dans le modèle où seul la liaison d'un listener au composant est reflétée.

La classe comporte enfin la déclarations de trois méthodes :

- `fireSearch` ;
- `fireChangeTextField` ;
- `fireSpecifyTextField` ;

Ces méthodes acceptent en entrée un paramètre `event` de la classe `ICAREEvent` et invoquent, en leur fournissant cet événement en paramètre, une méthode de chacun des listeners qui sont associés au composant. Ces méthodes peuvent être les méthodes :

- `search` ;
- `changeTextField` ;
- `specifyTextField` ;

On reproduit ce fait dans le modèle en définissant alors ces méthodes ainsi :

```
public void fireSearch(int event) {
    l.search(event);
}
public void fireChangeTextField(int event) {
    l.changeTextField(event);
}
public void fireSpecifyTextField(int event) {
    l.specifyTextField(event);
}
```

ce qui achève la constitution du modèle de la classe `myMappyFrameSupport`.

3.9 Modélisation de l'interface *ICARE*

Le texte `IcareInterface.java` comporte la déclaration de la classe `IcareInterface` qui constitue la couche logicielle réalisant l'interface entre le contrôleur de dialogue et les composants ICARE.

Traitement des déclarations. La déclaration de la classe, dans le texte source original est donnée avec la déclaration de différentes variables :

```
public class IcareInterface {
        DialogControler myDialog = null;
        Icare myIcare;
        boolean imageShown = false;
        String name;
        String address;
```

Les deux premières déclarations sont relatives à des objets de classes qui ont été introduites dans le modèle d'exécution Java. On maintient donc ces déclarations. La variable `imageShown` est utilisée dans des méthodes faisant appel à des méthodes de la classe `myFrame` relatives à la navigation sur image. On ne retient pas cette déclaration. Les deux variables `name` et `address` ne sont pas non plus reproduites dans le modèle. En fait, elles pourraient l'être (déclarées

alors `int` dans le modèle) mais ne seraient pas utilisées car les abstractions des méthodes qui utilisent ces données dans le texte original ne les référenceraient pas. On ne les abstrait donc pas et on obtient alors la déclaration de classe :

```
class IcareInterface {
    DialogControler myDialog;
    Icare myIcare;
```

Traitement du constructeur. Le constructeur de la classe utilise des données de classes définies dans le modèle, il est donc repris dans le modèle sans aucune transformation :

```
public IcareInterface(Icare theIcare)
{
    myIcare = theIcare;
}
```

Traitement du positionnement du contrôleur de dialogue. La méthode `setControler` définie dans le texte initial est reprise sans aucune transformation dans la mesure où elle opère sur des données qui sont reprises et modélisées :

```
    public IcareInterface setControler(DialogControler theDialog)
    {
        myDialog = theDialog;
        return this;
    }
```

Traitement des commandes. Les méthodes `simplecommands` et `compositecommands` invoquent les méthodes correspondantes du contrôleur de dialogue en fournissant un vecteur en paramètre. La section 3.3.2 indique que les vecteurs sont abstraits en un entier ou un couple d'entiers. En l'occurrence, on utilise un entier pour la commande simple et deux entiers pour la commande composée. Dès lors l'encodage dans le modèle est quasi le même que celui du texte d'origine, à ce transtypage près :

```
void simplecommands(int v) {
    myDialog.simplecommands(v);
}
void compositecommands(int v0, int v1) {
    myDialog.compositecommands(v0,v1);
}
```

On trouve ensuite, dans le texte d'origine, trois méthodes :

- `selection` ;
- `specifyName` ;
- `specifyAddress`

qui ne sont pas utilisées dans le programme d'application. On ne les reproduit pas.

Traitement de la méthode *nameAndAddress*. La méthode `nameAndAddress` a pour texte original le suivant :

```
public void nameAndAddress(Vector v)
```

```
        if ( ((String)v.elementAt(0)).equals("Name") )
        {
                myDialog.specifyName(((String)v.elementAt(1)));
        }
        else if ( ((String)v.elementAt(0)).equals("Address") )
        {
                myDialog.specifyAddress(((String)v.elementAt(1)));
        }
}
```

Cette méthode traite d'un objet vecteur `v` passé en paramètre et de chaînes de caractères. Les méthodes `elementAt` invoquées sur les objets vecteurs font apparaître que les deux éléments 0 et 1 sont référencés. On considère que le vecteur `v` sera abstrait par deux entiers `int v1` et `int v2`. Les termes `v.elementAt(0)` et `v.elementAt(1)` seront abstraits en `v0` et `v1`. De même l'expression

`((String)v.elementAt(0)).equals("Name")`

sera traduite, en retrouvant la valeur de l'index de la chaîne de caractères `"Name"` (ici cet index est 1), par l'expression `v0==1`. On fait la même opération pour l'expression :

`((String)v.elementAt(1)).equals("Address")`

qui est alors traduite par `v1==2`. La traduction de cette méthode est alors donnée dans le modèle par :

```
void nameAndAddress(int v0,int v1)
{
    if (v0==1) {
        myDialog.specifyName(v1);
    }
    else if (v0==2) {
        myDialog.specifyAddress(v1);
    }
}
```

Traitement des feedbacks. Un certain nombre de méthodes sont des méthodes de feedback : ce sont les méthodes invoquées par le contrôleur de Dialogue pour réaliser une opération de sortie, sur l'interface, consécutive à une opération d'entrée. Les méthodes de feedback décrites dans le texte `IcareInterface.java` sont les suivantes :

- `simpleCommandsFeedback` ;
- `compositecommandsFeedback`
- `searchFeedback` ;
- `specifyNameFeedback` ;
- `specifyAddressFeedback` ;
- `selectionFeedback` ;

Il convient de noter que, de la même façon que la méthode `selection` n'avait pas été modélisée car non utilisée, la méthode `selectionFeedback` ne sera pas non plus modélisée. La méthode `searchFeedback` ne sera également jamais invoquée dans le modèle de l'application. Cela parce que son invocation dans le texte original est située dans une section de code de la

méthode `simpleCommandsFeedback` qui ne sera pas modélisée. Aussi ne la modélise-t-on pas. Ne restent alors à modéliser que les méthodes :

- `simpleCommandsFeedback` ;
- `compositecommandsFeedback`
- `specifyNameFeedback` ;
- `specifyAddressFeedback` ;

Le texte source original de la méthode `simpleCommandsFeedback` est un aiguillage appelant des méthodes relatives à la gestion de l'image. On a fait comme hypothèse que, dans le modèle, l'image du plan sur lequel s'effectue la navigation est a priori affichée. En d'autres termes, cela revient à fixer a priori une valeur *true* à la variable `imageShown` du texte source original. Le texte de cette méthode est le suivant :

```
public void simpleCommandsFeedback(Vector v)
{
        if (imageShown)
        {
                if (((String)(v.elementAt(0))).equals("LEFT"))
                {
                        myIcare.frame.getFrame().move("left");
                }
                else if (((String)(v.elementAt(0))).equals("RIGHT"))
                {
                        myIcare.frame.getFrame().move("right");
                }
                else if (((String)(v.elementAt(0))).equals("UP"))
                {
                        myIcare.frame.getFrame().move("up");
                }
                else if (((String)(v.elementAt(0))).equals("DOWN"))
                {
                        myIcare.frame.getFrame().move("down");
                }
                else if (((String)(v.elementAt(0))).equals("ZOOM IN"))
                {
                        myIcare.frame.getFrame().zoomIn();
                }
                else if (((String)(v.elementAt(0))).equals("ZOOM OUT"))
                {
                        myIcare.frame.getFrame().zoomOut();
                }
                else if (((String)(v.elementAt(0))).equals("CENTER"))
                {
                        myIcare.frame.getFrame().center();
                }
                else if (((String)(v.elementAt(0))).equals("SEARCH"))
                {
                        searchFeedback();
                }
        }
        else if (((String)(v.elementAt(0))).equals("SEARCH")  && !imageShown)
```

```
        {
                searchFeedback();
        }
}
```

Cette méthode a un vecteur `v` comme paramètre. Nous traduisons ce vecteur en un entier ou en un couple d'entiers. Compte tenu de la valeur *true* affectée par défaut à `imageShown`, seule la branche *true* de l'instruction `if` sera évaluée. Celle-ci effectue des tests sur les valeurs possibles de l'expression `(String)(v.elementAt(0))`. Cela indique donc que le vecteur `v` ne comporte qu'un élément utile : son premier élément 0. Cet élément est une chaîne de caractère que l'on modélise comme la valeur entière correspondant à un numéro d'index de la chaîne. On peut en déduire que le paramètre de la méthode, dans le modèle, sera un entier `int v`.

L'expression est comparée à différentes valeurs de chaînes possibles. Ces valeurs de chaînes sont des valeurs qui ont pu être rencontrées lors de l'analyse du fichier `Icare.java` des composants ICARE. Ce sont des chaînes qui constituent des éléments des langages utilisés par les composants de gestion de la multimodalité. On ne retient, dans le cadre de l'analyse de la méthode courante, que les chaînes déjà rencontrées dans ce contexte. Ainsi la chaîne `"ZOOM IN"` a-t-elle été déjà rencontrée (paragraphe 3.6.2) dans le cadre de l'analyse du constructeur `Icare()` et fait-elle partie des chaînes pouvant constituer une entrée de composant. Son numéro d'identification est le 12. On retient donc et modélise le texte relatif à cette chaîne de caractère :

```
        else if (((String)(v.elementAt(0))).equals("ZOOM IN"))
        {
                myIcare.frame.getFrame().zoomIn();
        }
```

Si on fait l'hypothèse que l'index de la chaîne `"ZOOM IN"` est 12, alors on peut traduire cette méthode par :

```
void simpleCommandsFeedback(int v) {
    if (v==12)
    {
        myIcare.frame.getFrame().zoomIn();
    }
}
```

On décompose néanmoins (toujours pour faciliter les opérations de slicing des outils) les invocations à la méthode `getFrame()` en remplaçant cette instruction par sa valeur retournée : la vraible `frame` de classe `myFrame`. On introduit une déclaration locale de cette variable pour procéder alors à l'invocation de la méthode `zoomIn()` ainsi :

```
void simpleCommandsFeedback(int v) {
    myFrame frame;
    if (v==12)
    {
        frame =  myIcare.frame.frame;
        frame.zoomIn();
    }
}
```

Pour bien faire, il conviendrait de traiter tous les cas possibles. En particulier, dans le cadre de l'analyse des composants ICARE, il faudrait avoir enregistré toutes les chaînes du langage (`"LEFT"`, `"RIGHT"`, `"UP"`, ...) et introduire dans le modèles les comparaisons successives :

```
void simpleCommandsFeedback(int v) {
    myFrame frame;
    if (v==12)
    {
        frame =  myIcare.frame.frame;
        frame.zoomIn();
    }
    else if (v==13)
    {
        frame =  myIcare.frame.frame;
        frame.move();
    }
    else ...
}
```

Dans une première approche, on décide de restreindre le modèle et de faire au plus simple.

Ainsi, on observe que la méthode `compositecommandsFeedback` accepte aussi un vecteur `v` comme paramètre. Le corps des instructions de la méthode invoque le méthode `elementAt` avec des valeurs de paramètres 0, 1 et 2. Il y a plus d'un élément dans ce vecteur. On choisit donc de modéliser ce fait avec, dans le modèle, deux entiers `v1` et `v2` en paramètre. On considère que les paramètres 1 et 2 sont des valeurs concaténées et synthétisées dans le paramètre `v2` du modèle. Par ailleurs, le texte source de la méthode indique que les traitements sont effectués si la valeur de l'élément 0 est une chaîne donnée. Or, des deux chaînes qui sont testées (`"ZOOM IN POINT"` et `"ZOOM OUT POINT"`) aucune n'a été retenue dans la modélisation. Elles ne font ainsi pas partie de l'ensemble des chaînes enregistrées au paragraphe 3.6.2 comme pouvant constituer une entrée de composant. Le modèle de la méthode, dans une première approche, sera vide.

```
void compositecommandsFeedback(int v0, int v1) {}
```

Cet état de fait n'est pas très satisfaisant. Il est imputable non pas aux choix de modélisation qui ont été fait, mais au fait que l'application n'effectue pas de traitements pour les valeurs d'entrée qui ont pu être retenues dans le modèle (`"BIGGER POINT"` par exemple).

Les deux dernières méthodes de feedback sont les suivantes :

```
public void specifyNameFeedback(String theName)
{
        name = theName;
        myIcare.frame.getFrame().jTextField1.setText(name);
        myIcare.frame.getFrame().jTextField1.setCaretPosition(
               myIcare.frame.getFrame().jTextField1.getText().length());
}

public void specifyAddressFeedback(String theAddress)
{
        address = theAddress;
```

```
        myIcare.frame.getFrame().jTextField2.setText(address);
        myIcare.frame.getFrame().jTextField2.setCaretPosition(
            myIcare.frame.getFrame().jTextField2.getText().length());
}
```

Ces deux méthodes ont en commun d'accepter une chaîne de caractère en entrée. On traduit cela en `int`. La dernière instruction de chacune d'elle est par ailleurs une invocation à une méthode `setCaretPosition` de la classe Java Swing `JTextField` et qui n'a pas été modélisée dans les classes `JTextField1` ou `JTextField2` qui représentent, dans le modèle d'exécution, le comportement abstrait des objets d'interaction de classe `JTextField`. Par conséquent on ne retient pas ces instructions et on ne les introduit pas dans le modèle. Ce dernier abstrait alors ces deux méthodes en :

```
void specifyNameFeedback(int theName)
{
    myIcare.frame.getFrame().jTextField1.setText(theName);
}

void specifyAddressFeedback(int theAddress)
{
    myIcare.frame.getFrame().jTextField2.setText(theAddress);
}
```

Ceci achève la modélisation de l'interface Icare.

3.10 Modélisation du contrôleur de dialogue

Le fichier `DialogControler.java` comporte la déclaration des variables et des méthodes qui constituent le contrôleur de dialogue.

Traitement des déclarations. Du point de vue des déclarations, `DialogControler` est une classe qui présente quatre instructions :

```
public class DialogControler {
        String name = null;
        String address = null;

        FunctionalCore fc = new FunctionalCore();
        IcareInterface myInterface;
```

Les deux premières déclarations des chaînes de caractères `name` et `address` s'avèrent déclarer des variables qui sont assignées dans le corps des méthodes mais non utilisées. On ne les conserve pas dans le modèle. Les deux dernières en revanche sont des déclarations de variables de classes maintenues dans le modèle et on les conserve donc pour obtenir :

```
class DialogControler {
        IcareInterface myInterface;
        FunctionalCore fc = new FunctionalCore();
```

Traitement du constructeur. Le constructeur de la classe, dans le texte source original, est vide. Il le sera aussi dans le modèle :

```
public class DialogControler {

        FunctionalCore fc = new FunctionalCore();
        IcareInterface myInterface;

        public DialogControler()  {}
```

Traitement des autres méthodes. La méthode `setIcareInterface` est inchangée dans le modèle d'exécution :

```
public void setIcareInterface(IcareInterface ii)
{
        myInterface = ii;
}
```

Les autres méthodes, dans le texte source, sont de structure élémentaire. Les méthodes `setName` et `setAddress` agissent exclusivement sur les variables `name` et `address` qui n'ont pas été retenues dans le modèle. On ne retient donc pas non plus ces deux méthodes et on ne les introduit pas dans le modèle.

Les autres méthodes invoquent des méthodes de `myInterface` qui est un objet de la classe `IcareInterface`. Cette dernière a été modélisée comme indiqué au paragraphe 3.9. `selectionFeedback` n'a pas été modélisée. L'invocation de cette méthode constituant la seule instruction de la méthode `selection` du contrôleur de dialogue, celle-ci ne sera pas reprise dans le modèle d'exécution.

Les méthodes `specifyName` et `specifyAddress` acceptent en paramètre une chaîne `String` qui sera transformée en `int` dans le modèle. Le corps des deux instructions de ces méthodes comporte une affectation de l'une des variables `name` ou `address` qui n'ont pas été retenues dans le modèle : ces instructions ne sont donc pas reprises. La seconde instruction est conservée:

```
void specifyName(int theName)
{
    myInterface.specifyNameFeedback(theName);
}
void specifyAddress(int theAddress)
{
    myInterface.specifyAddressFeedback(theAddress);
}
```

Les deux dernières méthodes définies dans le texte source original sont les méthodes `compositecommands` et `simpleCommands`. Ces deux méthodes invoquent respectivement les méthodes `compositecommandsFeedback` et `simpleCommandsFeedback` qui sont définies dans la classe `IcareInterface`. On a modélisé ces méthodes (voir le paragraphe 3.9). On insère donc ces méthodes dans le modèle en modifiant les signatures. Les paramètres de classe `Vector` sont transformés en paramètres de classe `int` en se conformant aux signatures des méthodes définies dans la classe `IcareInterface` :

```
void simplecommands(int c)
```

```
{
    myInterface.simpleCommandsFeedback(c);
}
void compositecommands(int c1, int c2)
{
    myInterface.compositecommandsFeedback(c1,c2);
}
```

3.11 Modélisation de l'application

Le fichier `myMappy.java` comporte le corps du programme principal en décrivant la classe `myMappy`. Ce texte est court :

```
public class myMappy {
        public myMappy() {
                DialogControler dialog = new DialogControler();
                Icare icare = new Icare();
                IcareInterface ii = icare.getIcareInterface().setControler(dialog);
                dialog.setIcareInterface(ii);
        }

        // Main entry point
        static public void main(String[] args) {
                new myMappy();
        }
}
```

Il ne met en oeuvre que des objets modélisés dans le modèle d'exécution Java. On le conserve à l'identique en ajoutant simplement une instruction de trace pour les outils Bandera ainsi que l'instruction `start` de démarrage du modèle d'environnement :

```
public class myMappy {
        public myMappy() {
                Bandera.generateScenarioTest("Lancer et initialiser");
                DialogControler dialog = new DialogControler();
                Icare icare = new Icare();
                IcareInterface ii =
                        icare.getIcareInterface().setControler(dialog);
                dialog.setIcareInterface(ii);

                new Environment(icare).start();
        }

        // Main entry point
        static public void main(String[] args) {
                new myMappy();
        }
}
```

Chapitre 4

Vérification du modèle et génération de scénarios de tests

Lorsque le modèle d'exécution Java de l'application interactive multimodale est construit, il est alors possible de prendre le programme Java qui le constitue et d'appliquer sur ce programme les techniques et les outils de formalisation présentés en [dD06]. Ceux-ci ont pour objectif de construire un modèle formel de ce programme. Ce modèle doit pouvoir être soumis à la vérification de propriétés ou doit permettre de produire des scénarios susceptibles de constituer des tests de cette propriété.

Le rapport [dD06] indique comment la propriété envisagée peut être exprimée et formalisée dans une formule de logique temporelle linéaire (LTL). La formulation de cette propriété incorpore un certain nombre de prédicats sur des observables du programme (valeurs de variables, points d'exécution du programme). Ceux-ci permettent alors de définir un critère de slicing sur lequel se fondent les calculs de dépendances et les opérations de découpage du programme permettant d'obtenir un modèle d'exécution résiduel. C'est ce modèle d'exécution résiduel qui est formalisé. C'est lui qui est vérifié.

4.1 Vérifications

4.1.1 Définition des observables

Les opérations de vérification reposent sur la définition d'annotations du programme en forme Javadoc. Il est ainsi possible de définir des observables de plusieurs types.

Si l'on prend par exemple, dans le modèle d'exécution Java, la classe `Complemntarity2` définie par :

```
class Complementarity2 {
    boolean data_1_set;
    boolean data_2_set;
    int d1;
    int d2;
    boolean order = true;

    ComplementaritySupport2 complementaritySpt = new ComplementaritySupport2();
```

```
    public Complementarity2 () {
        this.init();
    }

    public void init() {
        data_1_set = false;
        data_2_set = false;
    }

    public void setData(int nbPort,int d){
        if (nbPort == 1) {
            data_1_set = true;
            d1 = d;
        }
        else if (data_1_set || (!order)){
            data_2_set = true;
            d2 = d;
        }
        if (data_1_set && data_2_set) {
                complementaritySpt.fireNewComplementarityData(d1,d2);
                data_1_set = false;
                data_2_set = false;
        }
    }

    public void addComplementarityListener(ComplementarityListener2 l) {
        complementaritySpt.addComplementarityListener(l);
    }
}
```

Il est possible de définir un certain nombre d'observables constitués par des prédicats sur les variables d'une classe. Ainsi, avant la déclaration de la classe `Complementarity2` est-il possible d'introduire les déclarations d'observables suivants :

```
/**
 *
 * @observable
 * EXP data_1_set(this) : (data_1_set == true);
 * EXP data_2_set(this) : (data_2_set == true);
 * EXP d1(this) : (d1 != 0);
 * EXP d2(this) : (d2 != 0);
 * EXP d12not0(this) : ((d1 != 0)&&(d2!=0));
 *
 */

class Complementarity2 {
    boolean data_1_set;
    boolean data_2_set;
    int d1;
    int d2;
    boolean order = true;
    .......
}
```

Le premier prédicat `data_1_set` sera vrai si, dans l'instance de classe `Complementarity2` considérée (ce qu'indique le paramètre `this` placé en paramètre), la variable `data_1_set` prend la valeur `true`. Les autres prédicats ainsi définis se comprennent aisément.

Il est aussi possible, dans le corps de la déclaration de classe, d'annoter les méthodes par de nouveaux observables ou par des assertions de pre et post-conditions d'expressions. On procède alors comme suit.

Une post-condition à la méthode `init()` peut être construite, qui exprime que les valeurs des deux variables `data_1_set` et `data_2_set` ont la valeur `false`. Cela se note :

```
/**
 * @assert
 *   POST TwoFalse: ((data_1_set == false) && (data_2_set == false=));
 */
 public void init() {
     data_1_set = false;
     data_2_set = false;
 }
```

De la même façon, une précondition pourra être notée comme la suivante, qui établit qu'un numéro de port passé en paramètre doit toujours être une valeur strictement positive :

```
/**
 * @assert
 *   PRE PositivePort: (nbPort > 0);
 */
 public void setData(int nbPort,int d){
   .....
 }
```

Il est également possible de définir des observables relatifs à des points d'exécution du programme Java. Par exemple les deux observables définis ci-dessous, `Call` et `Return` prennent la valeur `true` lorsque l'exécution du programme passe par la première instruction de la méthode (à son invocation) ou à sa dernière instruction (son retour) :

```
/**
 * @assert
 *   PRE PositivePort: (nbPort > 0);
 * @observable
 *   INVOKE Call(this);
 *   RETURN Return(this);
 */
 public void setData(int nbPort,int d){
   .....
 }
```

Enfin il est possible de définir des observables relatifs à des points d'exécution internes à des méthodes du programme Java. On peut par exemple ainsi définir un prédicat `initSet2(this)` qui prend la valeur vraie lorsque l'exécution de la méthode `init` ci-dessous passe par le point défini par l'étiquette `initSet2`.

```
/**
 * @observable
 *   LOCATION[initSet2] initSet2(this);
```

```
   */
     public void init() {
        data_1_set = false;
        initSet2:
        data_2_set = false;
    }
```

C'est l'ensemble de ces mécanismes d'annotations qui peuvent être utilisés pour établir des spécifications et pour formaliser ainsi les propriétés que l'on cherche à vérifier.

4.1.2 Vérification de propriétés

On dispose ainsi des outils permettant de réaliser un ensemble de vérifications ayant trait à des propriétés comportementales du programme. On peut donner ici quelques exemples de ces propriétés que l'on évalue sur le programme ayant fait office d'étude de cas.

Règle d'interaction. Par exemple, on peut vérifier sur le programme une propriété, non pas du programme, mais de l'interaction Java Swing : l'usager donne le focus à un widget visible uniquement. On peut vérifier cela sur chacun des widgets pour lesquels un `FocusAdapter` a été déclaré et programmé dans l'application. Pour le vérifier sur le champ de saisie textuelle désigné par la variable du programme `jTextField1` de classe `JTexfield`, on construit deux observables. Sur la classe `FocusAdapter1` implantant l'adapteur associé à la prise de focus du widget on construit l'observable ci-dessous qui permet de repérer l'activation de l'adapteur :

```
class FocusAdapter1 {
    myFrame frame;
    public FocusAdapter1 (myFrame f) {
        frame = f;
    }

   /**
   *
   * @observable
   * INVOKE  focusGained(this);
   *
   */
   public void focusGained(FocusEvent e)
    {
        frame.jTextField1FocusGained(e);
    }
}
```

De la même façon, on construit l'observable `visible` qui, dans le modèle, repère qu'un widget est dans l'état visible :

```
/**
 *
 * @observable
 * EXP  Visible(this): (visible == true);
 *
 */
```

```
class JTextField1{
    FocusAdapter1 fa1;
    KeyAdapter1 ka1;
    int idText=0;
    boolean visible = false;
    ...
}
```

Dès lors, on peut construire la propriété que l'on cherche à vérifier comme étant :

$$\forall j : JTextField1, f : FocusAdapter1, \Box\ (focusGaind(f) \rightarrow Visible(j))$$

Celle-ci exprime que l'on a toujours, pour tout objet `f` de classe `FocusAdapter` et pour tout objet `j` de classe `JTexField1`, l'énoncé suivant vérifié : si `f` est activé sur acquisition du focus alors `j` est un widget visible. Il s'agit là non d'une vérification du bon comportement de l'application à proprement parler, mais d'une vérification que le modèle produit est valide.

Composants multimodaux. Toujours pour valider le modèle, on peut aussi utiliser les capacités de vérification offertes pour vérifier la modélisation produite des composants multimodaux. Par exemple, le composant de complémentarité décrit par la classe `Complementarity2` peut être vérifié en s'assurant qu'il déclenche toujours la construction d'un énoncé dès lors qu'il a reçu deux données sur ses entrées.

Le déclenchement d'une action est repéré, dans le programme, par activiation de la méthode `newComplementarityData()` du listener associé au composant de complémentarité.

```
class ComplementarityListener2{
    IcareInterface ii;
    public ComplementarityListener2 (IcareInterface ii) {
        this.ii = ii;
    }
/**
 *
 * @observable
 * INVOKE fired(this);
 *
 */
    public void newComplementarityData(int v0, int v1)
    {
        ii.nameAndAddress(v0,v1);
    }
}
```

Le fait que deux données aient été présentées sur chacun des ports du composant de complémentarité est représenté dans le modèle du composant par le fait que les deux variables booléenes représentant le positionnement de données d'entrée sur chacun des ports sont à la valeur `true`. On peut construire l'observable suivant :

```
/**
 *
 * @observable
 * EXP twoData(this) : ((data_1_set == true)&&(data_2_set == true));
```

```
 *
 */
class Complementarity2 {
    boolean data_1_set;
    boolean data_2_set;
    ....
}
```

Dès lors, la formule qui permet de capturer la propriété que l'on cherche à vérfier sera :

$$\forall l : ComplementarityListener2, c : Complementarity2 \ \Box(twoData(c) \rightarrow \Diamond fired(l))$$

Comportement applicatif. On dispose aussi de moyens permettant de vérifier des propriétés applicatives. Par exemple, on peut utiliser ces moyens pour s'assurer par exemple du fait que l'exécution de la commande *Zoom In* de l'application est le résultat d'une redondance-équivalence de modalités.

On repèrera la redondance-équivalence des modalités par l'observable `AData` défini comme suit :

```
/**
 *
 * @observable
 * EXP AData(this) : (data_1_set == true);
 *
 */
class RedundancyEquivalence1 {
    boolean data_1_set;
    ...
}
```

Pour repérer l'activation de la fonction *zoomIn()*, on repère l'appel de la méthode `zoomIn()` de la classe `myFrame` du programme, reproduite dans le modèle d'exécution Java. Pour cela on construit l'observable suivant :

```
class myFrame extends JFrame{
  ...
   /**
   *
   * @observable
   * INVOKE  zoom(this);
   *
   */
        public void zoomIn() {

          ....
}
```

Dès lors on peut construire la formule suivante qui permet de s'assurer de la précédence d'une reconnaissance de redondance ou d'équivalence sur le déclenchement d'un zoom :

$$\forall r : RedudnancyEquivalence1,\ f : myFrame,\ (\neg zoom(f)\ \mathcal{U}\ AData(c))\ ||\ \Box\ (\neg zoom(f))$$

Cette formule indique que deux comportements sont admissibles. Soit il n'y a jamais d'activation de zoom. S'il y en a une, elle ne peut se produire avant le déclenchement d'une redondance-équivalence d'une donnée.

4.2 Génération de scénarios

La génération de scénarios se fonde sur l'utilisation des contre-exemples produits par les opérations de vérification.

4.2.1 Principe

On suppose que l'on cherche à produire un scénario permettant d'atteindre une configuration de l'interface C. Cette configuration C peut être caractérisée par une propriété exprimable par une formule LTL ϕ. Si M est le modèle d'exécution de l'application interactive, on va chercher donc à opérer la vérification

$$M \models \Box\neg\phi$$

On cherche donc à prouver que C est inatteignable car ϕ n'est jamais vérifiée.

Si la configuration C est atteignable, cette vérification doit échouer. Cela veut dire que, dans le système de transitions entre états associé au modèle d'exécution, un ou plusieurs chemins existent qui conduisent à un état conforme à C. Ce ou ces chemins constituent des contre-exemples.

A chacun des contre-exemples correspond une séquence d'exécution des instructions du modèle d'exécution de l'application. On peut ainsi retrouver les différentes étapes d'exécution qui conduisent à la configuration C. On peut associer à certaines de ces instructions des annotations particulières permettant de construire des traces pouvant constituer les éléments de séquences d'actions constituant un scénario d'exécution de l'application.

4.2.2 Mise en oeuvre dans les outils

Pour permettre la construction de scénarios pouvant être considérés comme des jeux de tests, une méthode particulière de la classe `Bandera` a été introduite dans les outils. Il s'agit de la méthode `generateScenarioTest()` qui accepte en paramètre une chaîne de caractère de classe `String`.

Il est ainsi possible de placer dans le modèle d'exécution des instructions du type :

```
Bandera.generateScenarioTest(``Action 1'');
```

Une telle instruction n'a pas de sens dans le modèle d'exécution de l'application. Elle constitue simplement une directive donnée aux outils de la suite Bandera de générer une trace `"Action 1"` si le déroulement du contre-exemple incorpore l'exécution de cette instruction du modèle.

C'est la raison pour laquelle on trouve ces instructions principalement dans la classe modélisant l'environnement de l'application. Celle-ci simule les actions que réalise l'utilisateur sur

les différents objets d'interaction de l'application. On trouve dans cette classe des séquences d'instructions telles que :

```
...
          else if (Bandera.choose()) {
              Bandera.generateScenarioTest("Press Key : 08");
              icare.c1.setData(1,13);
          }
          else if (Bandera.choose()) {
              Bandera.generateScenarioTest("Say bigger here");
              icare.c1.setData(1,13);
          }
...
```

Ces instructions sont la traduction opérée dans le modèle d'exécution d'actions de l'usager. Les commandes `generateScenarioTest` comportent en paramètre une donnée `String` qui peut contenir n'importe quelle information utile à la constitution d'un scénario. Il s'agit ici de simples commentaires décrivant les actions de l'usager. On pourrait tout aussi bien incorporer ici des appels à un synthétiseur d'entrées, des équations Lustre reflétant l'évolution de l'environnement, etc ....

L'introduction de cette méthode de classe `Bandera` a dû être prise en compte dans les outils de la suite Bandera. En particulier les outils de traduction en Jimple puis en Promela doivent traiter correctement cet appel de méthode. Plus délicate a été sa prise en compte dans les outils de slicing. Des erreurs, en particulier dans la constitution des programmes résiduels en présence de séquences ne comportant que cette seule instruction, peuvent conduire à la constitution de scénarios incohérents. Les analyses de dépendance et de constitution des résidus ont du être adaptés pour prendre en compte ces instructions de traces.

4.2.3 Exemples de génération de scénario

La génération de scénarios, dès lors qu'on dispose de l'outillage permettant de placer, dans les modèles, des directives de trace du déroulement des contre-exemples, revient à effectuer techniquement une vérification d'atteignabilité d'un état défini par une formule LTL et à réaliser une exécution imulée du contre-exemple produit. Cette simulation produit les traces correspondant à l'activation des ordres `generateScenarioStep` rencontrées.

On peut utiliser ce mécanisme pour diriger l'exécution de l'application lorsqu'elle est en test. On peut ainsi chercher à faire atteindre un état donné de l'application.

Par exemple, il est possible d'accroître la couverture des tests en produisant un scénario permettant l'activation d'une instruction du code donnée. On peut chercher ainsi à générer un scénario activant l'instruction d'affectation de la variable `data_set 2` dans la méthode `setData` de la classe `complementarity2`. Pour ce faire, on introduit une étiquette `initSet2` devant l'instruction d'affectation de cette variable. Et l'on définit un observable `initSet2` repérant le passage de l'exécution par cette instruction.

```
/**
 * @observable
 *   LOCATION[initSet2] initSet2(this);
 */
  public void setData(int nbPort,int d){
```

```
    if (nbPort == 1) {
        data_1_set = true;
        d1 = d;
    }
    else if (data_1_set || (!order)){
        initSet2: data_2_set = true;
        d2 = d;
    }
    if (data_1_set && data_2_set) {
            complementaritySpt.fireNewComplementarityData(d1,d2);
            data_1_set = false;
            data_2_set = false;
    }
}
```

La recherche d'un scénario provoquant le passage par cette instruction consiste à faire vérifier que cette instruction est inatteignable.

$$\forall c : complementarity2, \Box \neg initSet2(c)$$

La génération d'un contrexemple et la simulation de ce contrexemple produisent, pour le modèle d'exécution de l'application généré, la trace suivante :

```
SCENARIO_TEST_ETAT 1 Lancer et initialiser
SCENARIO_TEST_ETAT 2 Say C Grenoble
SCENARIO_TEST_ETAT 3 Say Address
SCENARIO_TEST_ETAT 4 Say C Grenoble
```

Ce scénario indique que les actions de l'usager conduisant à un état de l'application dans lequel le contrôle de l'application est passé par le point repéré par `initSet2` sont au nombre de 4. La première consiste à lancer l'application. La seconde à prononcer vocalement *C Grenoble*. La troisième à prononcer vocalement *Address*. La dernière à reprononcer *C Grenoble*.

L'examen approfondi du code de l'application montre que seules les actions 3 et 4 (et bien évidemment 1) suffisent à atteindre l'état recherché. L'action 2 est à cet égard sans effet. Elle apparaît pourtant dans le scénario car celui-ci n'est pas minimal. Il reflète un parcours et une exploration des états du modèle logique associé à l'application ayant conduit à l'état recherché : ce parcours n'est pas nécessairement le plus court. La recherche d'un contrexemple minimal (le plus court possible) nécessite une exploration exhaustive et s'avère coûteuse. Néanmoins elle permet d'obtenir par calcul le scénario suivant corroborant l'intuition précédente d'un scénario à 3 étapes :

```
SCENARIO_TEST_ETAT 1 Lancer et initialiser
SCENARIO_TEST_ETAT 2 Say Name
SCENARIO_TEST_ETAT 3 Say Laurence
```

Ce scénario indique que les actions de l'usager conduisant à un état de l'application dans lequel le contrôle de l'application est passé par le point repéré par `initSet2` sont au nombre à présent de 3. La première consiste à lancer l'application. La seconde à prononcer vocalement

Name. La troisième à prononcer vocalement *Laurence*. Effectivement cela revient aà constituer vocalement un couple de données vocales complémentaires.

On peut utiliser ce mécanisme de génération de scénarios pour toutes les propriétés mentionnées au paragraphe 4.1.2. Le principe de leur mise en oeuvre reste alors inchangé.

Chapitre 5

Conclusion

On a décrit ici un ensemble de procédés élémentaires qui permettent de construire le modèle d'exécution Java d'une application interactive multimodale. Une caractéristique de cette application est qu'elle se fonde sur l'utilisation d'une architecture de composants ICARE pour gérer l'ensemble des modalités disponibles pour les entrées de l'usager.

Les composants ICARE ont un caractère générique et peuvent être employés dans toutes les applications interactives qui ont pour objectif de mettre en oeuvre la multimodalité. On en fournit une modélisation relativement simple et abstraite. Le travail d'analyse consiste donc à retrouver, dans le code source original, les liens entre les différents composants et les flux de données qui transitent à travers ces liens.

Les choix de modélisation opérés ne se fondent pas toujours sur une technique rigoureuse et générale et s'appuient beaucoup sur une connaissance de l'application. Des techniques d'abstraction et de modélisation définies de manière plus systématique et plus générique auraient sans doute permis de définir un outil de traduction automatique général. Mais au prix sans doute d'une abstraction plus faible et d'un modèle sans doute trop détaillé pour pouvoir constituer un bon support à la vérification.

Il reste que les procédés présentés peuvent encore être améliorés. Ils ne constituent que la première ébauche de ce qui pourrait constituer un ensemble de transformations de programmes d'interaction Java.

Ce rapport a ensuite décrit comment le modèle d'exécution ainsi produit peut être utilisé à des fins de vérification et de génération de scénarios. Des annotations de type Javadoc permettent de définir des prédicats concernant les objets ou le contrôle du programme. Des formules LTL reflétant des propriétés de ce programme peuvent être construites et constituer des objectifs de vérification ou de tests.

Ces objectifs de vérification ou de tests peuvent être utilisés pour définir des critères de sclicing et permettre de réuire le modèle d'exécution Java produit comme décrit en [dD06]. Le programme résiduel ainsi obtenu est alors traduit en Promela. Les algorithmes, implantés pour une part dans la suite Bandera, de découpage des programmes, de constitution de programmes résiduels et de compilation en Promela ont été étendus pour traiter les constructions Java des modèles d'exécution et pour permettre l'introduction de mécanismes de génération de scénarios.

Annexe : modèle généré

```
/*-----------------------------------------------------------------
Constantes

1     Name
2     Address
3     Laurence
5     Search
11    C in Grenoble
12    ZOOM   IN
13    BIGGER POINT
-----------------------------------------------------------------*/

public class myMappy {
        public myMappy() {
                Bandera.generateScenarioTest("Lancer et initialiser");
                DialogControler dialog = new DialogControler();
                Icare icare = new Icare();
                IcareInterface ii = icare.getIcareInterface().setControler(dialog);
                dialog.setIcareInterface(ii);

                new Environment(icare).start();
        }

        // Main entry point
        static public void main(String[] args) {
                new myMappy();
        }
}

class ActionEvent{}

class ActionListener1 {
    myFrame frame;
    public ActionListener1 (myFrame f) {
        frame = f;
    }
    public void actionPerformed(ActionEvent e)
    {
        frame.jButton1ActionPerformed(e);
    }
}
```

```
class Complementarity1 {
    boolean data_1_set;
    boolean data_2_set;
    int d1;
    int d2;
    boolean order = true;

    ComplementaritySupport1 complementaritySpt = new ComplementaritySupport1();

    public Complementarity1 () {
        this.init();
    }

    public void init() {
        data_1_set = false;
        data_2_set = false;
    }

    public void setData(int nbPort,int d){
        if (nbPort == 1) {
            data_1_set = true;
            d1 = d;
        }
        else if (data_1_set || (!order)){
            data_2_set = true;
            d2 = d;
        }
        if (data_1_set && data_2_set) {
                complementaritySpt.fireNewComplementarityData(d1,d2);
                data_1_set = false;
                data_2_set = false;
        }
    }

    public void addComplementarityListener(ComplementarityListener1 l) {
        complementaritySpt.addComplementarityListener(l);
    }
}

class ComplementarityListener1{
    IcareInterface ii;
    boolean done = false;

    public ComplementarityListener1 (IcareInterface ii) {
        this.ii = ii;
    }

    public void newComplementarityData(int v0, int v1)
    {
        done = true;
        ii.compositecommands(v0,v1);
        done = false;
```

```
    }
}

class ComplementaritySupport1 {
        ComplementarityListener1 l1;

        public void addComplementarityListener(ComplementarityListener1 l1) {
            this.l1 = l1;
        }

        public void fireNewComplementarityData(int v0, int v1) {
                l1.newComplementarityData(v0,v1);
        }
}

class Complementarity2 {
    boolean data_1_set;
    boolean data_2_set;
    int d1;
    int d2;
    boolean order = true;

    ComplementaritySupport2 complementaritySpt = new ComplementaritySupport2();

    public Complementarity2 () {
        this.init();
    }

    public void init() {
        data_1_set = false;
        data_2_set = false;
    }

    public void setData(int nbPort,int d){
        if (nbPort == 1) {
            data_1_set = true;
            d1 = d;
        }
        else if (data_1_set || (!order)){
            data_2_set = true;
            d2 = d;
        }
        if (data_1_set && data_2_set) {
                complementaritySpt.fireNewComplementarityData(d1,d2);
                data_1_set = false;
                data_2_set = false;
        }
    }

    public void addComplementarityListener(ComplementarityListener2 l) {
        complementaritySpt.addComplementarityListener(l);
    }
}
```

```
class ComplementarityListener2{
    IcareInterface ii;
    boolean done = false;

    public ComplementarityListener2 (IcareInterface ii) {
        this.ii = ii;
    }

    public void newComplementarityData(int v0, int v1)
    {
        done = true;
        ii.nameAndAddress(v0,v1);
        done = false;
    }
}

class ComplementaritySupport2 {
        ComplementarityListener2 l2;

        public void addComplementarityListener(ComplementarityListener2 l2) {
            this.l2 = l2;
        }

        public void fireNewComplementarityData(int v0, int v1) {
                l2.newComplementarityData(v0,v1);
        }
}

class DialogControler {

        IcareInterface myInterface;
        FunctionalCore fc = new FunctionalCore();

        public DialogControler()
        {

        }

        void setIcareInterface(IcareInterface ii)
        {
                myInterface = ii;
        }

        void specifyName(int theName)
        {
            myInterface.specifyNameFeedback(theName);
        }

        void specifyAddress(int theAddress)
        {
            myInterface.specifyAddressFeedback(theAddress);
        }
```

```
        void simplecommands(int c)
        {
            myInterface.simpleCommandsFeedback(c);
        }

        void compositecommands(int c1, int c2)
        {
            myInterface.compositecommandsFeedback(c1,c2);
        }
}

class FocusAdapter1 {
    boolean done=false;
    myFrame frame;
    public FocusAdapter1 (myFrame f) {
        frame = f;
    }
    public void focusGained(FocusEvent e)
    {
        done = true;
        frame.jTextField1FocusGained(e);
        done = false;
    }
}

class FocusAdapter2 {
    boolean done=false;
    myFrame frame;
    public FocusAdapter2 (myFrame f) {
        frame = f;
    }
    public void focusGained(FocusEvent e)
    {
        done = true;
        frame.jTextField2FocusGained(e);
        done = false;
    }
}

class FocusEvent {}

class FunctionalCore {
        public FunctionalCore() {
        }
}


class Icare {
    IcareInterface ii;
    Complementarity1 c1;
    ComplementarityListener1 cListener1;
```

```
    Complementarity2 c2;
    ComplementarityListener2 cListener2;

    RedundancyEquivalence1 re;
    RedundancyEquivalenceListener1 reListener1;

    myMappyFrameListener listener;
    myMappyFrame frame;

    public Icare() {
        ii = new IcareInterface(this);

        //COMPOSITION
        c2 = new Complementarity2();
        cListener2 = new ComplementarityListener2(ii);
        c2.addComplementarityListener(cListener2);

        //COMPOSITION
        c1 = new Complementarity1();
        cListener1 = new ComplementarityListener1(ii);
        c1.addComplementarityListener(cListener1);

        //REDUNDANCE
        re = new RedundancyEquivalence1();
        reListener1 = new RedundancyEquivalenceListener1(ii);
        re.addRedundancyEquivalenceListener(reListener1);

        //Frame
        frame = new myMappyFrame();
        listener = new myMappyFrameListener(this);
        frame.addMyMappyFrameListener(listener);

        frame.startLanguage();
    }

    public IcareInterface getIcareInterface()
    {
        return ii;
    }
}


class IcareInterface {
    DialogControler myDialog;
    Icare myIcare;

    public IcareInterface(Icare theIcare)
    {
        myIcare = theIcare;
    }

    public IcareInterface setControler(DialogControler theDialog)
    {
```

```
        myDialog = theDialog;
        return this;
    }

    void compositecommands(int v0, int v1) {
        myDialog.compositecommands(v0,v1);
    }

    void nameAndAddress(int v0,int v1)
    {
        if (v0==1) {
            //name = v1;
            myDialog.specifyName(v1);
        }
        else if (v0==2) {
            //address = v1;
            myDialog.specifyAddress(v1);
        }
    }

    void simplecommands(int v) {
        myDialog.simplecommands(v);
    }

    void simpleCommandsFeedback(int v) {
        myFrame frame;
        if (v==12) //ZOOM IN
            {
                frame =  myIcare.frame.frame;
                frame.zoomIn();
            }
    }

    void specifyNameFeedback(int theName)
    {
        myIcare.frame.getFrame().jTextField1.setText(theName);
    }

    void specifyAddressFeedback(int theAddress)
    {
        myIcare.frame.getFrame().jTextField2.setText(theAddress);
    }

    void compositecommandsFeedback(int v0, int v1) {
    }
}


class JButton1 {
    ActionListener1 al1;
    boolean visible = false;
    public void setVisible(boolean v) {
        visible = v;
```

```
    }
    public void addActionListener1(ActionListener1 al1){
        this.al1 = al1;
    }
}

class JFrame {
    boolean visible = false;

    public JFrame () {
    }

    public void setVisible(boolean v) {
        visible = v;
    }
}

class JTextField1{
    FocusAdapter1 fa1;
    KeyAdapter1 ka1;
    int idText=0;
    boolean visible = false;

    public int getText() {
        return idText;
    }
    public void setVisible(boolean v) {
        visible = v;
    }
    public void setText(int idtxt) {
        this.idText = idtxt;
    }

    public void addFocusListener1(FocusAdapter1 fa1){
        this.fa1 = fa1;
    }
    public void addKeyListener1(KeyAdapter1 ka1){
        this.ka1 = ka1;
    }
}

class JTextField2{
    FocusAdapter2 fa2;
    KeyAdapter2 ka2;
    int idText=0;
    boolean visible = false;

    public int getText() {
        return idText;
    }
    public void setVisible(boolean v) {
        visible = v;
    }
```

```
    public void setText(int idtxt) {
        this.idText = idtxt;
    }

    public void addFocusListener2(FocusAdapter2 fa2){
        this.fa2 = fa2;
    }
    public void addKeyListener2(KeyAdapter2 ka2){
        this.ka2 = ka2;
    }
}

class KeyAdapter1 {
    myFrame frame;
    public KeyAdapter1 (myFrame f) {
        frame = f;
    }
    public void KeyReleased(KeyEvent e)
    {
        frame.jTextField1KeyReleased(e);
    }
}

class KeyAdapter2 {
    myFrame frame;
    public KeyAdapter2 (myFrame f) {
        frame = f;
    }
    public void KeyReleased(KeyEvent e)
    {
        frame.jTextField2KeyReleased(e);
    }
}

class KeyEvent {}

class myFrame extends JFrame{

    boolean doneZoomIn = false;


    JTextField1 jTextField1 = new JTextField1();
    FocusAdapter1 f1;
    KeyAdapter1 k1;


    JTextField2 jTextField2 = new JTextField2();
    FocusAdapter2 f2;
    KeyAdapter2 k2;


    JButton1 jButton1 = new JButton1();
    ActionListener1 al1;
```

```
myMappyFrame mother;

    public myFrame(myMappyFrame theMappyFrame ) {
        mother = theMappyFrame;
    }

    public void initComponents() {
        jTextField1.setText(0);
        jTextField1.setVisible(true);
        jTextField2.setText(0);
        jTextField2.setVisible(true);
        jButton1.setVisible(true);

        f1 = new FocusAdapter1(this);
        jTextField1.addFocusListener1(f1);

        k1 = new KeyAdapter1(this);
        jTextField1.addKeyListener1(k1);


        f2 = new FocusAdapter2(this);
        jTextField2.addFocusListener2(f2);

        k2 = new KeyAdapter2(this);
        jTextField2.addKeyListener2(k2);

        al1 = new ActionListener1(this);
        jButton1.addActionListener1(al1);
    }

public void jTextField1FocusGained (FocusEvent e) {
    mother.changeTextField(1);
}

public void jTextField1KeyReleased(KeyEvent e) {
    int text;
    mother.changeTextField(1);
    text = jTextField1.idText;
    mother.specifyTextField(text);
}

public void jTextField2FocusGained (FocusEvent e) {
    mother.changeTextField(2);
}

public void jTextField2KeyReleased(KeyEvent e) {
    int text;
    mother.changeTextField(2);
    text = jTextField2.idText;
    mother.specifyTextField(text);
}
```

```
    public void jButton1ActionPerformed(ActionEvent e) {
        mother.search();
    }

    public void zoomIn() {
        doneZoomIn = true;
        doneZoomIn = false;
    }

}


class myMappyFrame
{
        myFrame frame;
        myMappyFrameSupport myMappyFrameSpt = new myMappyFrameSupport();
        public myMappyFrame()
        {
        }

        public myFrame getFrame()
        {
                return frame;
        }

        public void startLanguage()
        {
                        frame = new myFrame(this);
                        frame.initComponents();
                        frame.setVisible(true);
        }

        public void changeTextField(int theValue)
        {
                myMappyFrameSpt.fireChangeTextField(theValue);
        }

        public void search()
        {
                myMappyFrameSpt.fireSearch(5);
        }

        public void specifyTextField(int theValue)
        {

                myMappyFrameSpt.fireSpecifyTextField(theValue);
        }


        public void addMyMappyFrameListener(myMappyFrameListener l) {
                myMappyFrameSpt.addMyMappyFrameListener(l);
        }
```

```
}

class myMappyFrameListener {
    boolean doneS = false;
    boolean doneChTf = false;
    boolean doneSpTf = false;
    Icare icare;

    public myMappyFrameListener(Icare i){
        this.icare = i;
    }

    public void search (int data_values)
    {
        doneS=true;
        icare.re.setData(data_values);
    }
    public void changeTextField(int data_values)
    {
        doneChTf = true;
        icare.c2.setData(1,data_values);
    }

    public void specifyTextField(int data_values)
    {
        doneSpTf = true;
        icare.c2.setData(2,data_values);
    }

}

class myMappyFrameSupport {
    myMappyFrameListener l;

    public void addMyMappyFrameListener(myMappyFrameListener l) {
        this.l=l;
    }

    public void fireSearch(int event) {
        l.search(event);
    }

    public void fireChangeTextField(int event) {
         l.changeTextField(event);
    }

    public void fireSpecifyTextField(int event) {
         l.specifyTextField(event);
    }
}

class RedundancyEquivalence1 {
    boolean data_1_set;
```

```
    int d1;


    RedundancyEquivalenceSupport1 redundancyEquivalenceSpt = new RedundancyEquivalenceSupport1();

    public RedundancyEquivalence1 () {
        this.init();
    }

    public void init() {
        data_1_set = false;
    }

    public void setData(int d)
    {
        d1 = d;
        data_1_set = true;
        if (data_1_set) {
                redundancyEquivalenceSpt.fireNewRedundancyEquivalenceData(d1);
                data_1_set = false;
        }

    }

    public void addRedundancyEquivalenceListener(RedundancyEquivalenceListener1 l) {
        redundancyEquivalenceSpt.addRedundancyEquivalenceListener(l);
    }
}


class RedundancyEquivalenceListener1{
    IcareInterface ii;

    public  RedundancyEquivalenceListener1 (IcareInterface ii) {
        this.ii = ii;
    }

    public void newRedundancyEquivalenceData(int v)
    {
        ii.simplecommands(v);
    }
}

class RedundancyEquivalenceSupport1 {
    RedundancyEquivalenceListener1  l;

        public void addRedundancyEquivalenceListener(RedundancyEquivalenceListener1 l) {
            this.l = l;
        }

        public void fireNewRedundancyEquivalenceData(int v) {
                l.newRedundancyEquivalenceData(v);
        }
```

```
}

class Environment extends Thread {
    boolean stop = false;
    Icare icare;
    FocusEvent fe = new FocusEvent();
    KeyEvent ke = new KeyEvent();
    ActionEvent ae = new ActionEvent();
    FocusAdapter1 f1;
    KeyAdapter1 k1;
    FocusAdapter2 f2;
    KeyAdapter2 k2;
    ActionListener1 l1;
    boolean focus_TF1=false;
    boolean focus_TF2=false;

    public Environment (Icare i) {
        this.icare = i;
    }
    public void run() {
        myFrame frame = icare.frame.getFrame();
        while (!stop) {
            if (Bandera.choose() && (!focus_TF1)) {
                Bandera.generateScenarioTest("FocusEvent Gained   jTextField 1");
                f1 = frame.jTextField1.fa1;
                f1.focusGained(fe);
                focus_TF1 = true;
            }
            else if (Bandera.choose() && focus_TF1) {
                Bandera.generateScenarioTest("KeyRelease Event in jTextField 1");
                k1 = frame.jTextField1.ka1;
                k1.KeyReleased(ke);
                focus_TF1 = false;
            }
            else if (Bandera.choose() && (!focus_TF2)) {
                Bandera.generateScenarioTest("FocusEvent Gained on jTextField 2");
                f2 = frame.jTextField2.fa2;
                f2.focusGained(fe);
                focus_TF2 = true;
            }
            else if (Bandera.choose() && focus_TF2) {
                Bandera.generateScenarioTest("Key Event on jTextField 2");
                k2 = frame.jTextField2.ka2;
                k2.KeyReleased(ke);
                focus_TF2 = false;
                }
            else if (Bandera.choose()) {
                Bandera.generateScenarioTest("Action Event on jButton1");
                l1 = frame.jButton1.al1;
                l1.actionPerformed(ae);
                }
            if (Bandera.choose()) {
                Bandera.generateScenarioTest("Say Name");
```

```
                icare.c2.setData(1,1);
            }
            else if (Bandera.choose()) {
                Bandera.generateScenarioTest("Say Address");
                icare.c2.setData(1,2);
            }
            else if (Bandera.choose()) {
                Bandera.generateScenarioTest("Say LAURENCE");
                icare.c2.setData(2,3);
            }
            else if (Bandera.choose()) {
                Bandera.generateScenarioTest("Say C Grenoble");
                icare.c2.setData(2,11);
            }
            else if (Bandera.choose()) {
                Bandera.generateScenarioTest("Say zoom in");
                icare.re.setData(12);
            }
            else if (Bandera.choose()) {
                Bandera.generateScenarioTest("Press Key : C9");
                icare.re.setData(12);
            }
            else if (Bandera.choose()) {
                Bandera.generateScenarioTest("Press Key : 08");
                icare.c1.setData(1,13);
            }
            else if (Bandera.choose()) {
                Bandera.generateScenarioTest("Say bigger here");
                icare.c1.setData(1,13);
            }
            else if (Bandera.choose()) {
                Bandera.generateScenarioTest("Mouse Move");
                icare.c1.setData(2,255);
            }
        }
    }
}
```

Bibliographie

[dD06] B. d'Ausbourg and G. Durrieu. Projet RNRT VERBATIM-SP4 Analyses statiques pour la vérification de codes - Lot2 : Abstraction de programmes guidée par les propriétés. Technical report, ONERA, Centre de Toulouse, Juin 2006.