Zeitschrift: Bulletin Electrosuisse
Herausgeber: Electrosuisse, Verband für Elektro-, Energie- und Informationstechnik
Band: 94 (2003)
Heft: 9

Artikel: Un nouveau concept pour améliorer le développement des logiciels interactifs
Autor: Petitpierre, Claude
DOI: https://doi.org/10.5169/seals-857550

Nutzungsbedingungen
Die ETH-Bibliothek ist die Anbieterin der digitalisierten Zeitschriften auf E-Periodica. Sie besitzt keine Urheberrechte an den Zeitschriften und ist nicht verantwortlich für deren Inhalte. Die Rechte liegen in der Regel bei den Herausgebern beziehungsweise den externen Rechteinhabern. Das Veröffentlichen von Bildern in Print- und Online-Publikationen sowie auf Social Media-Kanälen oder Webseiten ist nur mit vorheriger Genehmigung der Rechteinhaber erlaubt. Mehr erfahren

Conditions d'utilisation
L'ETH Library est le fournisseur des revues numérisées. Elle ne détient aucun droit d'auteur sur les revues et n'est pas responsable de leur contenu. En règle générale, les droits sont détenus par les éditeurs ou les détenteurs de droits externes. La reproduction d'images dans des publications imprimées ou en ligne ainsi que sur des canaux de médias sociaux ou des sites web n'est autorisée qu'avec l'accord préalable des détenteurs des droits. En savoir plus

Terms of use
The ETH Library is the provider of the digitised journals. It does not own any copyrights to the journals and is not responsible for their content. The rights usually lie with the publishers or the external rights holders. Publishing images in print and online publications, as well as on social media channels or websites, is only permitted with the prior consent of the rights holders. Find out more

Download PDF: 04.07.2025

ETH-Bibliothek Zürich, E-Periodica, https://www.e-periodica.ch

Un nouveau concept pour améliorer le développement des logiciels interactifs

Pour traiter l'interactivité, la plupart des environnements de développement utilisent un concept de gestion des événements particulier (appelé *listener* en Java). Ce concept offre peu de contrôle car ces *listeners* peuvent en effet autoriser de très nombreuses séquences d'actions parasites que le développeur de logiciel a beaucoup de peine à identifier et qu'il ne peut donc pas vérifier proprement. Cet article présente une alternative aux *listeners,* basée sur le concept de processus ou pseudo-parallélisme, qui rend les programmes interactifs beaucoup plus précis et plus facile à mettre en œuvre.

Claude Petitpierre

La mise en service du système informatique utilisé par la société Billetel pour la vente de billets de théâtre et de concert fut un cauchemar. Alors que ce service devait introduire la vente par Internet, il n'a même pas été à même de fournir le service de base et a paralysé pendant de longs mois la vente des billets de la plupart des manifestations gérées par cette société. Cet exemple n'est malheureusement pas isolé et les applications d'informatique qui ne satisfont pas leurs utilisateurs ont englouti des fortunes. D'où peuvent donc provenir les problèmes de ces projets ?

Le laboratoire de téléinformatique (LTI) à l'EPFL n'a pas analysé le cas particulier mentionné ci-dessus, mais son expérience dans le domaine des systèmes interactifs montre que les difficultés de ces systèmes proviennent en bonne partie des pièges que cache la gestion de cette interactivité. En fait, les pannes des systèmes interactifs sont difficilement détectables avant qu'ils soient mis en charge. Elles ne surviennent pas toujours au même point des programmes, elles ne sont pas reproductibles et les méthodes habituelles de dépannages (dévermineur, impressions de traces) changent le programme suffisamment pour masquer les problèmes recherchés et pour en créer de nouveaux. Finalement les concepts utilisés pour développer ces systèmes sont très loin d'être adéquats [1, 2].

Dans les environnements basés sur Java, l'interactivité est actuellement traitée au moyen de *listeners*, un outil, décrit ci-dessous, qui permet de gérer les événements extérieurs (arrivée de messages, clics sur les boutons d'une fenêtre d'écran, etc.) quel que soit leur ordre d'arrivée. Les inconvénients des *listeners* sont liés au peu de contrôle qu'ils offrent. Ils autorisent en effet de très nombreuses séquences d'actions parasites (clic sur un bouton non prévu à un instant donné, message arrivant au mauvais moment, etc.) que le développeur a beaucoup de peine à identifier et qu'il ne peut donc pas vérifier proprement. Les considérations faites dans ce texte sont valables pour la plupart des autres langages et environnements de développement car la plupart d'entre eux utilisent des concepts semblables à ces *listeners*.

Dans cet article, une alternative aux *listeners* basée sur le concept de processus ou pseudo-parallélisme sera présentée. Cette alternative rend les programmes interactifs beaucoup plus précis et plus facile à mettre en œuvre. Bien que les systèmes multi-processus aient la réputation d'être particulièrement difficiles à dépanner, la façon proposée de les mettre en œuvre dans cet article évite ces difficultés. Les processus ne sont utilisés ici que pour introduire un peu de parallélisme, juste le minimum qui permet à l'application de ne pas attendre indûment, mais qui n'autorise pas de séquences d'actions imprévues.

La prochaine section présente un exemple très simple de programme qui lit des données dans des fenêtres (figure 1), puis elle démontre les problèmes créés par son codage au moyen de *listeners*. La section d'après présente un nouveau concept d'objet actif synchrone et son utilisation pour le codage du même exemple et finalement la dernière section montre que cette technique est compatible avec l'Unified Modelling Language (UML), le langage le plus couramment utilisé pour planifier la construction d'applications d'informatique.

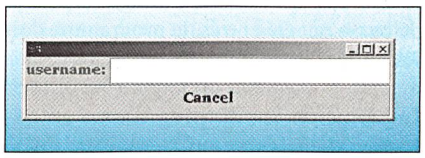

Figure 1 Une fenêtre pour entrer des données interactivement

Exemple d'un programme utilisant un *Graphical User Interface* (GUI)

L'exemple de programme très simple présenté ci-dessous utilise une interface pour entrer des données sur l'écran (notée GUI dans la suite, selon son abréviation anglaise). Cet exemple sera utilisé pour montrer comment coder cette application, tout d'abord à l'aide des *listeners* définis par Java, puis au moyen de l'approche du LTI.

Définition de l'exemple de référence

Le programme de référence doit lire un nom d'usager puis un mot de passe dans le même champ de texte, placé dans une boîte de dialogue semblable à celle qui est illustrée à la figure 1. L'utilisateur peut prendre autant de temps qu'il le désire pour entrer le nom d'usager, mais une fois qu'il a tapé son nom

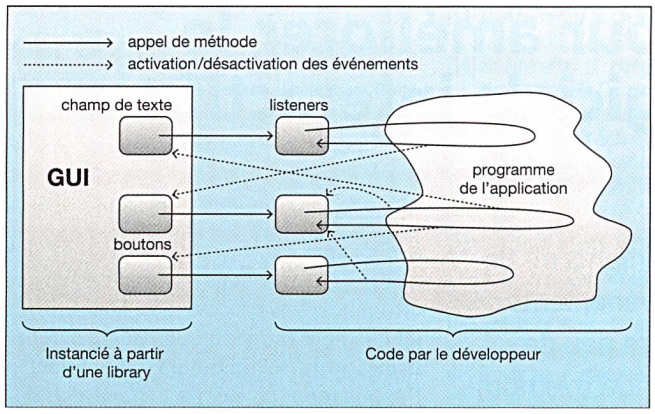

Figure 2 Programme basé sur les *listeners*

- soit il tape un mot de passe dans le champ;
- soit il appuie le bouton d'interruption (et la lecture du mot de passe est interrompue);
- soit une période de temps prédéfinie est écoulée (et la lecture est interrompue).

Une fois que le nom d'usager et le mot de passe ont été entrés, le programme doit les valider et continuer l'application.

Implémentation du programme au moyen de *listeners*

La figure 2 montre la structure d'un programme basé sur des *listeners*. Ces *listeners* sont des objets qui possèdent des interfaces prédéfinies connues par les éléments du GUI. Ils sont instanciés et enregistrés dans les éléments du GUI par le programme principal et contiennent soit des appels aux méthodes du programme principal qui gèrent les événements, soit directement le code qui gère les événements.

Le codage de l'exemple au moyen de ces *listeners* est imprimé dans les figure 3 et 4. Le programme principal placé dans la figure 3 instancie la classe *ADialog* qui lit le nom d'usager et le mot de passe, puis il se termine. Le traitement des résultats est contenu dans une méthode (*continuation*) définie dans la même classe que le programme principal. Cette continuation est appelée par un des *listeners* quand la récupération du nom de l'usager et du mot de passe est soit terminée soit interrompue.

La classe *ADialog* crée la structure statique de la boîte de dialogue et instancie et enregistre un *listener*. Les actions qui doivent être exécutées pour traiter le nom d'usager et le mot de passe (lignes 5 et 17 de la figure 4) sont placées dans la méthode *actionPerformed* (ligne 3) elle-même située dans le *listener ActionListener* instancié sur la première ligne. Ce *listener* est introduit dans l'objet champ de texte par la méthode *addActionListener* (ligne 1). La méthode *actionPerformed* est appelée par le système chaque fois que l'utilisateur du programme frappe la touche d'entrée sur son clavier.

En fait les structures de ces modules sont très mauvaises dans le sens où le premier (figure 3) requiert une méthode de continuation et que l'ordre dans lequel les lignes du second (figure 4) sont exécutées ne correspond pas du tout à l'ordre lexical (c'est-à-dire l'ordre du texte).

- Les lignes 1 et 2 de la figure 4 sont exécutées à l'initialisation du programme.
- Les lignes 3 à 9 et 15 (figure 4) sont exécutées quand le champ de texte a été rempli pour la première fois. La valeur booléenne *readUsername* a dû être ajoutée pour indiquer si la méthode est appelée pour entrer le nom d'utilisateur ou le mot de passe, bien que cette variable n'apparût pas dans la spécification du problème actuel. D'autre part, la ligne 9 modifie en fait le programme en cours d'exécution en introduisant un nouveau *listener*.
- Les lignes 16 à 20 (figure 4) sont exécutées après la deuxième fois que le champ de texte a été rempli.

```
 1 public class DialogMain {

 2     public static void main (String args [ ] ) {
 3         dm = new DialogMain ( );
 4         aDialog = new ADialog ( dm );
 5     }

 6     public void continuation ( String [ ] names ) {
 7         // continue ici après les événements du dialogue
 8         System.out.println ( names [0] + names [1] );
 9     }
10 }
```

Figure 3 Codage de l'exemple de référence

```
 1 textField.addActionListener (new ActionListener () {
 2    boolean readUsername = true;
 3    public void actionPerformed (ActionEvent e) {
 4       if (readUsername) {
 5          names [0] = textField.getText ();
 6          label.setText ("password: ");
 7          readUsername = false;
 8          cancelButton.addActionListener (
 9            new ActionListener () {
10              public void actionPerformed (ActionEvent e) .{
11                timer.stop ();
12                de.continuation (names);
13              }
14            });
15            timer.start (); // code executed somewhere else
16          } else {
17             names [1] = textField.getText ( );
18             timer.stop ();
19             de.continuation (names);
20          }
21       }
22    });
23 }
```

Figure 4 Codage: introduction d'un *listener*

```
// active object a                    // active object b

  run () {
      …
      b.myMethod();                   public void myMethod() { … }
      …
  }                                   run() {
                                          …
                                      accept myMethod;
                                          …
                                      }
```

Figure 5 Synchronisation par appel de méthode

```
// object a                           // object b

                                      public void myMethod() { … }

  run () {                            run() {
    …                                   …
    b.myMethod();                     accept myMethod;
    …                                   …
  }                                   }
```

Figure 6 Synchronisation par communication

- Les lignes 10 à 13 (figure 4) sont exécutées après que le bouton *cancel* a été cliqué.
- L'horloge est gérée par un *listener*. Ce dernier a été créé (ce qui n'est pas montré ci-dessus), mais il a quand même dû être activé et désactivé (lignes 11, 15 et 18, figure 4). De nouveau cela n'était pas explicité dans la spécification du problème.

Ainsi, si l'utilisation des *listeners* est une bonne chose à première vue, parce qu'elle permet l'élimination du programme principal, un examen plus approfondi révèle qu'il n'y a pas de miracle: le code est juste déplacé et de plus à un endroit où il est plus difficile à concevoir. Des actions non prévues peuvent être exécutées, telles l'entrée d'un troisième nom ou un clic sur le bouton *cancel* avant de taper le nom d'usager. Le développeur doit identifier toutes les séquences d'actions possibles et être sûr qu'elles ne provoquent pas de problèmes. Cependant, quand l'application devient plus grande, le nombre de cas se multiplie et dépasse rapidement les capacités de tout développeur.

Objets actifs synchrones

La technique décrite dans les derniers paragraphes a été introduite quand les GUIs ont introduit le non-déterminisme dans les applications (par le fait que le programme ne peut pas deviner quel élément du GUI sera activé le prochain) et a remplacé les appels bloquants. Cela est dommage car la lecture des données au moyen d'appels bloquants est très simple, elle assure que le résultat est disponible quand la méthode retourne, et en fait le non-déterminisme peut tout à fait être géré au moyen d'appels bloquants. La solution qui va être présentée repose sur le parallélisme et sur une façon standardisée de synchroniser les *listeners* avec les processus (*threads*).

Ces synchronisations pourraient être implantées à l'aide de librairies, mais le LTI a développé des instructions particulières, adaptées tout d'abord à C++ [3] et maintenant à Java. Ces instructions cachent les détails d'implémentation, présentent une bonne vue conceptuelle d'une application, mais peuvent être facilement traduites en pur Java.

Le concept d'objet actif

Un objet actif a une méthode (nommée *run* en Java) exécutée sur un processus (*thread*) attribué à l'objet. Un tel objet peut être opposé aux objets passifs (c'est-à-dire aux objets courants). Les objets passifs subissent des actions alors que les objets actifs peuvent agir de leur propre gré sans être appelés de l'extérieur. Un objet actif est conçu comme tel, et on peut considérer qu'un processus n'est pas un élément qui peut être introduit juste pour éviter un appel bloquant. La structure concurrente d'une application doit être pensée soigneusement dans les premières phases de développement d'un projet. Le LTI a donc proposé de définir un moyen de spécifier des classes pour objets actifs en décorant ces classes avec le mot-clé *active*. Le mot-clé *active* indique au compilateur qu'un processus doit être démarré à la fin de l'exécution du constructeur, quand l'objet est instancié, et que les appels à cet objet sont spéciaux, dans le sens décrit dans les prochains paragraphes.

Synchronisations entre objets actifs

La façon la plus naturelle de réaliser une communication entre deux objets actifs est évidemment de recourir à des appels de méthodes semblables à ceux qui sont utilisés d'un objet passif à l'autre. Cependant, comme les objets actifs introduisent de la concurrence, ces appels doivent être synchronisés. Les synchronisations que le LTI a définies peuvent être vues sous deux angles. Sous le premier angle, illustré dans la figure 5, le point central est l'appel de méthode: l'objet à gauche appelle une méthode de l'objet *b*, défini à droite. Comme *b* est un objet actif, cet appel est bloquant et son exécution est reportée jusqu'au moment où la méthode *run* de l'objet *b* accepte l'appel.

Sous le second angle (figure 6) l'attention est portée sur les communications. L'instruction *accept* dans l'objet *b* est aussi bloquante et elle n'est exécutée que quand l'appel à la méthode *myMethod* a été exécuté. Ainsi, une communication entre deux objets actifs synchrones exprime également le rendez-vous indiqué dans la figure 6. Le premier processus doit être arrivé à l'instruction d'appel et le second à l'instruction *accept* pour que le rendez-vous ait lieu. La méthode est exécutée pendant le rendez-vous, alors que les deux objets sont bloqués.

Le fonctionnement de l'instruction est le même dans les deux cas, mais suivant les situations, une explication ou l'autre est préférée. Quand on étudie le fonctionnement interne d'un objet, le point important est le fait que la méthode soit exécutée, alors que lorsque l'on analyse le comportement global d'une application comportant des objets actifs, l'aspect important est l'entrelacement des rendez-vous.

L'instruction select

Les appels synchrones simples sont évidemment trop contraignants et, par exemple, ne permettraient pas d'implan-

```
public void run () {
        for (;;) {
            select {
            case
                y = obj.method(x);
                ...
            case
                when (guard) accept myMethod;
                ...
            case
                waituntil (currentTimeMillis()+1000);
                ...
    }   }   }
```

Figure 7 Codage: instruction «*select-case*»

```
public class Main {
       static String [] name = new String[2];
       static Dislog dialog = new Dialog ();
       public static void main () {
          name = dialog.getNames();
          // the program obviously continues here
          System.out.println ( names [0] + names [1] );
      }
  }
  public class Dialog {
   JDialog dialog = new JDialog();            // initializations
   JLabel label = new JLabel("username: ");
   ATextField nameField = new ATextField(20);
   AButton cancel = new AButton("Cancel");
   Container pane;
   String [] name = new String[2];

   public String [] getNames () {
     pane = dialog.getContentPane();          // initializations
     pane.add(label,BorderLayout.WEST);
     pane.add(name,BorderLayout.CENTER);
     pane.add(cancel,BorderLayout.SOUTH);
     dialog.pack();
     dialog.setVisible(true);
     name[0] = nameField.read();
     name[1] = null ;
     label.setText("password: ");
     select { //appels parallèles: le premier prêt est exécuté
          case
               name[1] = nameField.read();
               System.out.println("U: "+username+" P: "+passwd);
          case
               waituntil(System.currentTimeMillis()+5000);
               System.out.println("Cancelled !");
          case
               cancel.pressed();
               System.out.println("Cancelled !");
     }
     return name;
  }  }
```

Figure 8 Code: Comment traiter le non-déterminisme?

ter le problème défini au debut de cet article (l'exemple de référence). Une instruction (*select*) a été introduite pour implémenter le choix défini par CCS[1] [4] ou par CSP[2] [5] de façon à permettre à un programme d'attendre autant d'appels, d'*accept* et de *délais* qu'il est nécessaire. De plus une notion d'instruction gardée a été introduite. Cette nouvelle notion est illustrée dans le deuxième cas du *select* décrit dans la figure 7. Les gardes sont optionnelles mais elles peuvent être introduites dans les trois sortes de cas.

La première instruction de chaque cas doit être une instruction synchronisante (qui peut être gardée) c'est-à-dire un appel à un autre objet actif, un *accept* ou un *waituntil*. Les cas qui ont une garde fausse quand le programme arrive sur l'instruction *select* sont ignorés pour cette exécution du *select*. Les instructions placées dans le corps d'un cas sont exécutées après que le rendez-vous qui synchronise le cas a été exécuté. Le corps d'un cas peut contenir des appels passifs ou des appels synchrones aussi bien que de nouvelles instructions *select*.

On note que l'instruction *select* est très proche de celle d'Ada[3]. La différence principale est que l'instruction proposé peut contenir autant d'appels synchrones, d'*accept* et de *waintuntil* que nécessaire. Le concept d'objet actif synchrone ressemble également au concept d'Actors[4] [6], mais les Actors exécutent les méthodes en parallèle avec les appelants. Elles ne sont donc pas bloquantes. Ces deux approches ne pourraient donc pas remplacer le concept proposé pour résoudre les problèmes considérés dans cet article.

Réalisation de l'exemple avec des appels synchrones

Le code source présenté dans la figure 8 montre comment les objets précédemment décrits permettent d'implémenter l'application de référence décrite au début de cet article. On peut noter que les mots mis en évidence dans la classe *Dialog* peuvent se reporter directement aux mots de la spécification, qu'à côté de l'initialisation des éléments du GUI il n'y a pas d'instruction supplémentaire et que la classe *Dialog* peut être réutilisée sans modification ailleurs dans la même application ou dans une autre. L'appelant ne doit introduire aucune instruction spéciale (nouvelle méthode, processus) pour continuer le programme principal.

Ce module de programme utilise les classes *AButton* et *ATextField* qui sont supposées accepter les appels synchrones (*nameField.read* et *cancel.pressed*) quand les éléments correspondants ont été activés (voir paragraphe suivant).

Un aspect très important du programme (figure 8) est la façon dont le programme traite le non-déterminisme: grâce à l'instruction *select* il est possible d'appeler les éléments (de classe *AButton* and *ATextField*) directement depuis le

```
public active class AButton extends JButton {
      public void pressed(){}
      public AButton() {
          addActionListener( new ActionListener() {
              public void actionPerformed( ActionEvent e) {
                  select {
                  case
                      accept pressed;
                  case
                      default;
              }   }
          )};
  }   }
```

Figure 9 La réalisation de la synchronisation d'un processus utilisant un *listener*

programme qui les a instanciés comme ce qu'on ferait avec des objets passifs instanciés d'une librairie – et à la différence de ce qui est fait avec les *listeners* présentés pour l'exemple de référence. Dans l'exemple des figures 3 et 4, les éléments de la librairie (les *listeners*) appellent eux-mêmes leur créateur (le programme principal) sous forme de *callbacks*. L'approche du LTI élimine cette inversion de contrôle et produit des programmes directs qui sont beaucoup plus faciles à comprendre et à dépanner. Le code présenté dans la figure 8 est compréhensible même pour quelqu'un qui n'a pas de notions de parallélisme.

Synchronisation des objets actifs avec les appels synchrones

Le code source dans la figure 9 montre comment réaliser la synchronisation d'un processus avec un *listener* d'une manière standard qui peut être reprise sans changement dans tout programme. Ce code peut être écrit par un spécialiste et mis à disposition des autres utilisateurs de Java par le biais d'une librairie qui cache les détails des *listeners*.

Du fait que la classe *AButton* génère un objet actif, la méthode *pressed* est bloquante. Elle n'est acceptée que lorsque le bouton a été pressé, que la méthode *actionPerformed* a été appelée, et donc que l'instruction *accept pressed* a été exécutée. La classe *AButton* hérite de la classe *JButton*, ce qui fait qu'elle peut être introduite dans un GUI de la même manière que la classe *JButton*.

Dans les programmes simples, ce code pourrait être écrit directement à partir des instructions *wait* et *notify* qui sont les instructions de synchronisation de base de la Java. Les instructions présentées dans la figure 9 sont par contre plus intéressantes que *wait/notify* si l'application contient également des communications sur un réseau ou d'autres sources d'évènements.

Implémentation d'une spécification UML

L'approche du LTI est également idéale pour implémenter les diagrammes d'activité, d'état ou de collaboration définis par UML (Unified Modelling Language), le langage standard utilisé par le génie logiciel. Cette section montre comment coder un diagramme d'état, les autres diagrammes pouvant être réalisés de façon semblable. Dans cette réalisation, on peut considérer que tous les éléments de l'environnement (GUI, accès au réseau) sont traités au moyen d'appels synchrones qui implémentent donc directement les transitions.

Implémentation d'un diagramme d'état

La figure 10 montre un diagramme d'état UML qui spécifie une machine d'états imbriquée dans une autre. Ces machines d'états exécutent des actions similaires à celles qui sont décrites dans les exemples précédents. Il y a beaucoup de possibilités de coder un tel diagramme. L'une d'entre elles est reportée dans le listing de la figure 11. Il est si simple de faire correspondre les différentes parties du diagramme de la figure 10 à ces codes sources que le code source est presque aussi expressif que le diagramme lui-même.

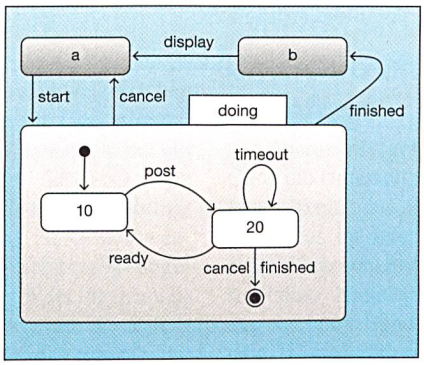

Figure 10 Diagramme d'états Unified Modelling Language (UML)

Conclusion

Dans l'article un concept d'appels synchrones basé sur les moyens de synchronisation décrits par les théories CCS [4] et CSP [5] a été présenté et on a montré que ce concept est très utile pour éviter les problèmes soulevés par l'utilisation des *listeners* et de l'inversion de contrôle que ceux-ci impliquent.

Le LTI a vérifié que ce concept pouvait être utilisé pour implémenter toutes les constructions habituelles utilisées en concurrence tels que les sémaphores, les boîtes aux lettres, l'exclusion mutuelle, les moniteurs, etc.

D'autres travaux proches de ceux du LTI [7, 8], également basés sur Java et CSP, mais ne permettant pas d'introduire des appels dans les *select* ont été poursuivis et leurs auteurs prétendent également qu'une telle approche simplifie considérablement le développement d'applications utilisant la concurrence.

L'article a finalement montré que le concept d'objet synchrone s'intégrait parfaitement dans les méthodes de développement logiciel basées sur UML.

Le concept présenté dans cet article fournit une contribution importante pour combler le fossé existant aujourd'hui entre la phase de design et les produits attendus par les clients souvent pendant beaucoup trop longtemps. Il peut être considéré également comme source d'inspiration pour réaliser des programmes simples, mais bien structurés au moyen des instructions de base de Java, *wait* et *notify*.

Outils disponibles

Le LTI a développé des compilateurs qui traitent les appels synchrones et les instructions *select* présentées dans ce papier tout d'abord pour C++ [3], puis pour Java[5]. Ce dernier compilateur, développé en collaboration avec Zenger[6], produit soit des codes binaires soit des codes sources en Java standard. En fait, les instructions qui traitent les synchronisations sont simples et peuvent aussi bien être codées à la main, soit en utilisant le noyau du LTI soit en utilisant directement les instructions *wait* et *notify*. Le déve-

loppeur n'est donc pas lié à l'utilisation du compilateur, mais la structure des programmes est naturellement plus claire quand le compilateur est utilisé.

Le LTI a également développé un analyseur d'états qui peut traiter un sous-ensemble de Java et des instructions et qui détermine si le code contient des interblocages (Deadlocks). Cet analyseur produit également une description CCS. En outre le LTI a réalisé quelques librairies. Les développements sont disponibles sur le site Web[5] du LTI.

Références

[1] *J. A. Whittaker, S. Atkin:* Software Engineering is not Enough. IEEE Software, July/August 2002.
[2] *P. B. Hansen:* Java's Insecure Parallelism. ACM Sigplan Notices, V. 34(4), April 1999, pp.38-44.
[3] *C. Petitpierre:* Synchronous C++, a Language for Interactive Applications. IEEE Computer, September 1998, pp 65-72.
[4] *R. Milner:* Communication and Concurrency. Prentice Hall, 1989.
[5] *C. A. R. Hoare:* Communicating Sequential Processes. Prentice Hall, 1984.
[6] *G. Agha, P. Wegner, A. Yonezawa:* Research Directions in Concurrent OO Programming. MIT Press, Cambridge, Mass., 1993.
[7] *P. Welch, P. D. Austin:* The JCSP Home Page. http://www.cs.ukc.ac.uk/projects/ofa/jcsp/, 2002.
[8] *G. Hilderink, A. Bakkers, J. Broenink:* A Distributed Real-Time Java System Based on CSP. ISORC 2000, Newport Beach, CA, pp 400-407.

Informations sur l'auteur

Prof. Dr **Claude Petitpierre** a obtenu un diplôme d'ingénieur électricien à l'EPFL. Il a ensuite passé 6 ans à développer des logiciels de contrôle de cimenteries avant d'entamer des recherches sur la conception des systèmes de communications à l'EPFL, puis pendant une année aux Laboratoires Bell de l'AT&T aux États-Unis. Depuis 1987 il dirige le Laboratoire de téléinformatique (LTI) à l'Ecole polytechnique fédérale de Lausanne (EPFL), CH-1015 Lausanne.
Contact: claude.petitpierre@epfl.ch

[1] CCS: Calculus of Communicating Systems. Une théorie permettant d'analyser le comportement de systèmes de processus parallèles et d'assurer par exemple qu'ils ne possèdent pas d'interblocages .
[2] CSP: Communicating Sequential Processes. Une théorie légèrement différente de CCS, mais visant les mêmes buts.
[3] ADA: Langage défini pour le Département de la défense américain, qui introduit un concept de rendez-vous pour la communication entre processus.
[4] Actor: Un concept particulier d'object actif.
[5] http://ltiwww.epfl.ch/sJava
[6] http://lamp.epfl.ch/~zenger/jaco/

```
public class StateDiagram {
    int doing () {                          // machine d'états interne
        state = 10;
        result = 0;
        for (;;) {
            switch (state) {
            case 10:
                proxy.post_remoteMeth ();
            case 20:
                select {
                case
                    proxy.ready_remoteMeth ();
                    result++;
                    state = 10;
                case
                    cancelButton.pressed ();
                    return -1;
                case
                    finishedButton.pressed ();
                    return result;
                case
                    waituntil (System.currentTimeMillis()+1000);
                    display ("I am alive");
    }   }   }   }

    public void run () {                    // machine d'états externe
        for (;;) {
        a:  startButton.pressed ();
            result = doing ();
            if (result>=0)
        b:      display (result);
}   }   }
```

Figure 11 Codage d'un diagramme d'état UML (Unified Modelling Language) qui spécifie une machine d'états imbriquée dans une autre.

Ein neues Konzept erleichtert die Entwicklung interaktiver Software

Um interaktive Prozesse in Programmen abarbeiten zu können, verwenden die meisten Entwicklungsumgebungen ein spezielles, ereignisgesteuertes Konzept, das bei Java Listener genannt wird. Dieses Konzept bietet allerdings nur geringe Kontrollmöglichkeiten, denn diese Listener lassen zahlreiche, unerwünschte Nebensequenzen zu, die der Programmentwickler nur schwer identifizieren kann, was eine korrekte Prüfung stark behindert. Der vorliegende Beitrag stellt eine Alternative zu den Listenern vor, die mittels des so genannten pseudo-parallelen Konzeptes eine wesentlich genauere und leichtere Umsetzung interaktiver Programme ermöglicht.