Un langage formalisable pour la définition d'applications interactives embarquées et critiques

RÉSUMÉ

Cet article présente un langage L en cours de définition et destiné à permettre aux concepteurs d'applications interactives embarquées et critiques, en particulier dans le domaine aéronautique, de décrire abstraitement leurs applications. Le langage permet de réaliser des opérations de vérification et de preuve formelles de manière à s'assurer, bien en amont des développements, que l'application présente un comportement conforme à celui attendu d'elle. Il permet ensuite de générer un code exécutable sur une plateforme cible. En particulier il permet de produire du code ARINC 661 embarquable au sein de cockpits d'avions.

Mots Clés

Systèmes interactifs embarqués ; systèmes critiques ; langages formels ; vérifications et preuves formelles ; logiciels aéronautiques ; ARINC 661.

ACM Classification Keywords

H.5.m. Information Interfaces and Presentation (e.g. HCI): Miscellaneous.

INTRODUCTION

Méthodes et techniques de descriptions formelles ont été, depuis longtemps, appliquées au domaine des systèmes critiques pour des besoins de sûreté. C'est particulièrement le cas des systèmes aéronautiques dont le fonctionnement peut mettre en jeu la vie de passagers. De nombreux travaux de recherche se sont attachés à élaborer et à proposer des moyens pour concevoir, programmer et vérifier les fonctionnalités de systèmes critiques en particulier aéronautiques. Ces systèmes et les processus qui encadrent leur développement se soumettent à des étapes de certification avec l'objectif d'établir que le produit et le développement qui a conduit à le construire sont conformes à des règles précises définissant des standards de certification, telle la DO178C [2] pour le logiciel aéronautique embarqué.

Les systèmes d'interaction homme-machine (IHM) n'ont bénéficié ni de la même attention ni du même effort de recherche. Pourtant des systèmes d'interaction mettant en oeuvre des capacités électroniques et numériques hautement sophistiquées ont fait leur apparition au sein de cockpits d'avions de nouvelle génération. Ces systèmes se fondent sur des logiciels à la fois volumineux et complexes et doivent répondre à des contraintes sévères en matière d'utilisabilité, de sécurité ou de sûreté. Ces logiciels mettent en oeuvre des applications dont le comportement, compte tenu de leur criticité, doit se conformer avec un niveau très élevé d'assurance à ce qui est attendu d'elles. En effet une erreur dans l'un des composants d'interaction logiciels peut conduire à une erreur humaine ou système dont les effets peuvent se révéler catastrophiques.

Le récent rapport du Bureau d'Enquêtes et d'Analyses [12] sur le crash de l'Airbus A330 d'Air France lors du vol AF447 Rio-Paris pointe un réel dysfonctionnement du système d'interaction, et plus particulièrement du directeur de vol, ayant affiché des indications qui n'auraient pas dû l'être et ayant ainsi conduit le pilote à cabrer son appareil alors que ce dernier était déjà en train de décrocher.

Le coeur du problème réside sans doute essentiellement dans le fait que les processus standards de développement des systèmes aéronautiques critiques se fondent la plupart du temps sur des cycles linéaires (du type du cycles en V ou de ses variantes) qui s'avèrent peu ou mal adaptés à la conception et au développement des logiciels d'interaction qui, eux, requièrent des cycles d'itérations impliquant les usagers et mêlant opérations de test et phases de conception. Une autre difficulté, propre au développement des systèmes d'interaction, est imputable à la multiplicité des communautés techniques et scientifiques intervenant dans le champ de la définition des systèmes d'IHM. Manquent à ces communautés diverses un langage et des notations qui soient, sinon communes, du moins intelligibles par toutes. Dès lors le traitement de la sûreté et de la sécurité de ces systèmes d'interaction critiques ne fait appel à aucune méthode ni aucun outil spécifiques. Dans ce contexte, les processus de validation demeurent pauvres car ils se limitent pratiquement à des phases intensives de test et d'évaluation en fin de processus de développement, qui plus est sans disposer de réelle référence, en particulier formelle, vis à vis de laquelle confronter le système développé.

Cet article suggère d'introduire cette référence formelle lors des étapes amont des processus de développement des systèmes d'IHM critiques en proposant un langage L, formalisable, qui permette la description des applications interactives, tant du point de vue de leur présentation que de leur comportement, ainsi que l'expression d'exigences des concepteurs concernant l'interaction encodée au sein de ces applications. Ce langage veut être suffisamment abstrait et convivial pour être compris et utilisables par tous les participants à la conception et au développement de l'IHM. Il veut également être, sinon formel, du moins formalisable de telle sorte qu'un texte rédigé dans ce langage

puisse être soumis à des opérations automatiques ou semi-automatiques de vérifications et de preuves formelles, de génération de tests pertinents du système à développer, et de génération de code plus sûr. L'objectif est d'accroître, in fine, l'assurance que le logiciel développé se comporte bien comme prévu.

TRAVAUX CONNEXES

La formalisation des systèmes d'interaction a fait l'objet de travaux importants lors de ces dernières années : bon nombre de modèles, de langages ou de méthodes formels ont vu le jour pour aider à spécifier, concevoir, vérifier et implanter ces systèmes.

Une première classe de modèles se fonde sur les systèmes de transitions et plus particulièrement sur les extensions d'automates d'états finis et de réseaux de Petri pour décrire des comportements interactifs de base. De manière à circonvenir les problèmes liés à l'explosion combinatoire de ces modèles de nombreuses extensions en ont été proposées. Les machines d'états hiérarchiques ou les statecharts décrivent des automates composables facilitant la spécification de systèmes complexes et permettant la réutilisation des modèles. Les automates à états finis ont également été utilisés dans le cadre d'outils de programmation [13, 14] ou ont pu être associés à d'autres formalismes pour décrire d'autres aspects de l'interaction. En [19] les automates modélisent les aspects de l'interaction liés à des entrées discrètes tandis que les flots de données en représentent les entrées continues.

Le modèle des capteurs formels [5] utilise quant à lui les réseaux de Petri de haut niveau [20] pour décrire les transformations successives d"événements d'entrée. De manière générale les réseaux de Petri ont été utilisés par plusieurs méthodes et techniques pour décrire les systèmes d'IHM. Le formalisme ICO [21, 22] se fonde sur une approche orientée objet pour modéliser les composants statiques d'un système d'interaction et sur les réseaux de Petri pour en décrire la dynamicité : PetShop [11, 24] est l'outil basé sur ICO.

D'autres descriptions formelles des systèmes interactifs se fondent sur la notion d'interacteurs. Les interacteurs peuvent se concevoir comme des entités autonomes et communicantes interagissant entre elles, de manière interne au système d'interaction, ou interagissant de façon externe avec le système interfacé ou l'usager par le biais d'événements (ou de stimuli). Les deux premiers modèles d'interacteurs développés le furent dans le cadre du projet AMODEUS. Le modèle de York [16] fournit un cadre pour la description de systèmes d'interaction à l'aide de notations orientées modèles telles que Z ou VDM. Le modèle du CNUCE [23] utilise quant à lui LOTOS. Des interacteurs formels ont également été décrits en utilisant le langage synchrone Lustre : [9] montre ainsi comment de tels interacteurs peuvent être dérivés d'une description informelle du système d'interaction. Ces interacteurs formels peuvent être composés pour constituer un modèle formel de l'IHM et peuvent être utilisés pour vérifier le système conçu ou générer des tests [10] ainsi que, plus généralement, pour valider le système d'interaction développé [15].

[7, 8] introduisent une approche modulaire basée sur l'usage de la méthode B. Les spécifications d'un système interactif opérationnel y sont formalisées en assurant qu'un certain nombre de propriétés ergonomiques sont préservées par le système. Diverses formes d'interaction ont ainsi été formalisées dans le cadre de cette approche en permettant la vérification de nombreuses propriétés de ces systèmes telles leur observabilité, leur honnêteté ou leur robustesse. Le travail présenté en [6] montre la description et la preuve, à l'aide de B, de toutes les étapes de raffinement d'un modèle abstrait de système jusqu'à son codage concret.

Le grand mérite de ces travaux est d'avoir démontré que l'interaction peut être formalisée et est donc formalisable. Toutefois ces approches construisent leurs modèles directement dans les formalismes ciblés. Cela nécessite donc une bonne maîtrise des notations proposées par ces formalismes. Il faut sans doute y voir la raison pour laquelle ces tentatives n'ont pas vraiment abouti dans la mesure où la communauté des concepteurs et développeurs ne s'est jamais approprié les techniques qu'elles proposaient.

En revanche beaucoup de travaux de recherche ont porté sur la définition et les transformations de modèles de tâches avec l'objectif général, partant d'un modèle abstrait des tâches assignées à l'utilisateur, d'en dériver des modèles d'IHM. Ces approches mettent ainsi l'accent sur la définition des tâches plutôt que sur la conception de l'IHM proprement dite. Même lorsqu'ils s'attachent à définir des langages d'interfaces abstraites, comme USIXML [18], ces derniers restent profondément orientés sur les tâches de l'usager plutôt que sur les détails d'interaction.

Il semble donc utile de proposer un langage de définition de systèmes d'interaction utilisable et disposant de la capacité d'offrir les mêmes services que les formalismes précédemment expérimentés.

OBJECTIFS D'UN LANGAGE DE DÉFINITION D'IHM

Constituer un langage commun, pivot, qui permette à tous les protagonistes de la conception et du développement d'un système interactif, designers, programmeurs, ergonomes, ingénieurs systèmes, ainsi que spécialistes des facteurs humains ou d'architectures logicielles de collaborer à la définition d'un même système, tel est sans doute l'un des premiers objectifs d'un langage de définition d'IHM.

Offrir un langage d'abstraction permet au concepteur de s'adapter aux évolutions permanentes des technologies dans le monde de l'IHM. Ce dernier vit actuellement une phase de grande diversification au point que de nombreux et nouveaux horizons s'ouvrent aujourd'hui devant les concepteurs de systèmes interactifs. Cet enrichissement des outils et des dispositifs d'interaction a cependant son revers. Celui de déplacer l'attention et la réflexion du concepteur depuis le problème conceptuel d'interaction qui lui est vraiment posé vers celui, qui ne lui est pas réellement posé, de l'implantation concrète de cette interaction à l'aide des moyens dont il dispose nouvellement. Passant trop rapidement du besoin à la solution et privilégiant la mise en oeuvre du moyen sur l'analyse de

Figure 1. Une application interactive jouet

la fin, le concepteur risque alors de définir un système inadapté au besoin de son utilisateur. L'abstraction fournie par un langage de définition conçu à cet effet peut lui fournir le moyen de se recentrer efficacement sur les aspects conceptuels de l'IHM et lui permettre ainsi de se poser les bonnes questions à propos de l'interaction envisagée.

Disposer d'un langage abstrait de définition de l'IHM doit également permettre la vérification et la preuve de proprietés ou exigences de sûreté le plus en amont possible du cycle de développement. A l'heure actuelle, ces vérifications ne peuvent être opérées, dans le meilleur des cas, qu'après une certaine concrétisation du concept d'IHM et, dans le pire des cas, qu'après implantation complète de l'IHM lors de campagnes de tests. Les erreurs, dont certaines peuvent s'avérer très graves, ne sont susceptibles d'être détectées que lors de la validation finale, obligeant alors à effectuer de nouveaux cycles de conception et de développement. Être sinon formel du moins formalisable de manière à permettre la vérification ou la preuve formelles de ces proprietés et exigences de sûreté dès les premières phases de conception du système, lorsque celui ci est encore abstrait, constitue un nouvel objectif assigné au langage de définition d'IHM ciblé.

UN LANGAGE L DE DÉFINITION D'IHM

Une application jouet

La Figure 1 illustre une application interactive très simple dont la définition abstraite exprimée dans le langage L peut être trouvée dans les tableaux de textes qui suivent. Cette application permet de contrôler deux valeurs réelles à l'aide de deux panneaux de boutons `-`, `+` et `R` permettant de décrémenter, d'incrémenter ou de remettre à zéro chacune d'elles. Un bouton `Reset All` permet de les remettre toutes les deux à zéro en même temps. La visualisation de ces valeurs est réalisée sous forme de labels exprimant leur valeur numérique ainsi qu'à travers deux dispositifs de présentation analogiques de valeurs réelles représentées comme des jauges. L'utilisateur sélectionne la jauge à afficher en agissant sur les boutons de sélection `View Left` ou `View Right`.

Sa définition en L

Le langage L proposé fait appel à la notion d'interacteur. Les interacteurs peuvent s'y composer pour définir un nouvel interacteur. Un interacteur peut donc être composé d'autres interacteurs qui deviennent ses sous-interacteurs. Ainsi, une IHM complète sera représentée sous la forme d'un interacteur racine, composé de ses sous interacteurs (fenêtres, contrôles, sons ...).

Le langage cherche à permettre la description des aspects conceptuels et abstraits de l'IHM sans avoir à se préoccuper des détails de leur implantation concrète et réelle. Ainsi différents types d'entités peuvent s'y voir abstraits sous un seul et même concept, simplifiant ainsi grandement le langage et son utilisation. Par exemple, les interacteurs tels que boutons à deux états (ToggleButton) et cases à cocher (CheckBox) offrent une même fonction : permettre à l'utilisateur de communiquer une valeur booléenne : le concept d' *entrée booléenne* les abstrait et les représente de manière unique dans le langage à l'aide du type prédéfini `BooleanInput`.

Beaucoup d'éléments d'IHM sont ainsi abstraits au sein de différents types de base représentant les briques élémentaires de l'interaction homme-machine. Ces types sont en nombre réduit tout en permettant de prendre en compte la plupart des dispositifs d'interaction homme-machine. On trouvera des entrées et sorties de textes, réels, entiers, booléens, choix de sous ensembles parmi un ensemble, ainsi que dispositifs de déclenchement d'événements. La nature du langage permet de combiner ces briques de base afin de définir des schémas d'interactions plus complexes.

La description en L d'un interacteur peut comporter quatre sections qui définissent pleinement cet interacteur :

- une section **`State`** présentant son état : celui-ci est constitué de la déclaration de ses variables d'état ;
- une section **`Components`** déclarant les sous-interacteurs qui le composent ;
- une section **`Interface`** qui décrit les données que cet interacteur peut recevoir en entrée ou les événements qu'il peut exporter en sortie ;
- une section **`Behavior`** qui décrit son comportement.

Le texte présenté dans la Définition 1 présente la définition de l'écran de contrôle principal qui constitue l'interacteur `ControlScreen` racine de l'application étudiée. Si l'on observe attentivement la structure de ce texte il apparaît que l'état de cet interacteur est donné par la valeur de deux variables réelles `left` et `right` qui sont les données effectivement contrôlées par l'application.

Cet interacteur se compose de cinq sous-interacteurs. Trois de ces sous-interacteurs sont d'un type prédéfini dans le langage : les composants `leftValueLabel` et `rightValueLabel` sont de type `NumericOutput`, qui abstrait tout dispositif de présentation d'une valeur numérique, et le composant `resetButton` est du type prédéfini `TriggerInput` qui abstrait tout dispositif d'entrée d'événement de déclenchement. En revanche les

```
Interactor ControlScreen // La page principale
State
  real left in [0. , 100.]
  real right in [0., 100.]
Components
  NumericOutput leftValueLabel
  NumericOutput rightValueLabel
  DualControlPanel controllers
  DualGaugePanel gauges
  TriggerInput resetButton
Behavior
  Init
    left = 0.0
    right = 0.0
  Always
    gauges.leftValue = left
    gauges.rightValue = right
    leftValueLabel.value = left
    rightValueLabel.value = right
  When controllers.incrementLeft
  left < 100 -> left = left +1
  When controllers.decrementLeft
  left > 0 -> left = left -1
  When controllers.resetLeft
  left = 0
  When controllers.incrementRight
  right < 100 -> right = right +1
  When controllers.decrementRight
  right > 0 -> right = right -1
  When controllers.resetRight
  right = 0
  When resetButton.triggered
  right = 0 ; left = 0
```

Définition 1. Écran de contrôle principal

```
Interactor DualControlPanel
Components
  ButtonControlPanel leftController
  ButtonControlPanel rightController
Interface
  Out incrementLeft
  Out decrementLeft
  Out resetLeft
  Out incrementRight
  Out decrementRight
  Out resetRight
Behavior
  When leftController.increment incrementLeft
  When leftController.decrement decrementLeft
  When leftController.reset resetLeft
  When rightController.increment incrementRight
  When rightController.decrement decrementRight
  When rightController.reset resetRight
```

Définition 2. Panneau de contrôle

interacteurs `controllers` et `gauges` sont respectivement du type `DualControlPanel` décrit dans le texte de la Définition 2 et du type `DualGaugePanel` défini par le texte de la Définition 3.

L'ensemble de ces déclarations définit la partie statique de l'interacteur `ControlScreen`. La dynamique de l'IHM est, elle, définie, au travers de la section **`Behavior`** décrivant le comportement de l'interacteur. Ce comportement s'exprime comme une machine à états. L'interacteur comprend donc un ensemble de variables d'état. Un ensemble de règles définit alors l'évolution de ces variables d'état en fonction de données directement perçues par l'interacteur ou restituées par le biais de ses sous-interacteurs. Ces règles d'évolution de l'état de l'interacteur revêtent plusieurs formes possibles.

```
Interactor DualGaugePanel
State
  String visibleGauge in {"LEFT", "RIGHT"}
  Number leftValue in [0. , 100.]
  Number rightValue in [0., 100.]
Components
  NumericOutput leftGauge
  NumericOutput rightGauge
  BooleanInput viewLeftGauge
  BooleanInput viewRightGauge
Behavior
  Init
    visibleGauge = ''LEFT''
  Always
    leftGauge.visible = (visibleGauge == "LEFT")
    leftGauge.value = leftValue
    viewLeftGauge.value = (visibleGauge == "LEFT")
    rightGauge.visible = (visibleGauge == "RIGHT")
    rightGauge.value = rightValue
    viewRightGauge.value = (visibleGauge == "RIGHT")
  When viewLeftGauge.changed(newValue)
  (visibleGauge != "LEFT" and newValue==true) ->
    visibleGauge = "LEFT"
  When viewRightGauge.changed(newValue)
  (visibleGauge != "RIGHT" and newValue==true) ->
    visibleGauge = "RIGHT"
```

Définition 3. Panneau de deux jauges

```
Interactor ButtonControlPanel
Interface:
    Out increment
    Out decrement
    Out reset
Components
  TriggerInput incButton
  TriggerInput decButton
  TriggerInput resetButton
Behavior
  When incButton.triggered increment
  When decButton.triggered decrement
  When resetButton.triggered reset
```

Définition 4. Panneau des boutons

La sous-section **`Init`** permet de représenter l'état initial de l'interacteur et des sous interacteurs. Ainsi, à l'initialisation, les variables d'état `left` et `right` sont toutes deux affectées d'une valeur nulle.

La sous-section **`Always`** définit des invariants de l'application. Elle représente des proprietés qui devront rester satisfaites à tout instant lors de l'exécution. Par exemple, dans le texte de la Définition 1, la variable d'état `leftValue` (respectivement `rightValue`) de l'interacteur `gauges` devra toujours avoir la valeur de la variables d'état `left` (respectivement `right`) de l'interacteur `controlScreen`. Il en va de même des valeurs des deux composants `leftValueLabel` et `rightValueLabel` qui afficheront en permanence les valeurs `left` et `right`. Ces invariants établissent donc des liens permanents entre des variables d'état de différents interacteurs de l'application. Leur formulation quoique très synthétique est cependant puissante.

Les sous-sections **When** permettent de définir les transitions d'états qui peuvent être déclenchées en réponse à des événements reçus. La clause **When** définit l'événement déclencheur. Cette clause est généralement suivie d'une garde, précédant le symbole `->`. Cette garde décrit sous quelles conditions la transition peut être déclenchée. Si la garde est satisfaite alors la transition est déclenchée et le nouvel état atteint est décrit par les affectations opérées sur les variables d'état.

Ainsi, dans le texte de la Définition 1 la première clause **When** indique que l'événement déclencheur est la réception de l'événement `incrementLeft` émis par l'interacteur `controllers` et défini comme signal de sortie au sein de la section `Interface` du texte de la Définition 2. Lorsque cet événement se produit, la garde `left < 100` est alors évaluée. Si elle est vraie, la transition est déclenchée et conduit l'interacteur à un nouvel état dans lequel la valeur `left` est incrémentée d'une unité.

Les autres textes de définitions adoptent une structure dictée par le langage équivalente à celle rencontrée dans le texte de la Définition 1. Ces autres textes de définitions appellent cependant un certain nombre de remarques ponctuelles.

Quelques remarques complémentaires

Le texte de la Définition 4 définit un type d'interacteur composé des trois boutons `-`,`+`,`R` et peut sembler trivial. Il définit simplement trois dispositifs déclencheurs `TriggerInput` et trois événements de sortie en `Interface`. Ces événements sont émis lorsque l'un des dispositifs déclencheurs est déclenché. Cette définition est donc relativement abstraite. Les dispositifs déclencheurs peuvent en effet être implantés par de nombreux et différents types de widgets proposés par diverses Toolkits. La conception en L ne s'attache pas à cela mais simplement à décrire comment les événements pourront être déclenchés à partir des dispositifs déclencheurs. Cette définition comporte ainsi très peu de choses. Mais, du point de vue de l'interaction et de son analyse, elle comporte l'essentiel.

Le texte de la Définition 2 n'offre d'autre intérêt que de définir l'association de deux interacteurs `ButtonControlPanel` décrit dans le texte de la Définition 4. Cet interacteur ne fait, au fond, que relayer les événements que lui transmettent ses deux sous-interacteurs. Ce schéma de composition particulier, mais fréquent en conception d'IHM, devrait ultérieurement être directement exprimé par un opérateur du langage décrivant ce multiplexage sans qu'il soit nécessaire pour le concepteur de définir explicitement un type d'interacteur pour cela.

Le texte de la Définition 3 décrit l'interacteur formé des deux jauges. Ces dernières sont définies comme des interacteurs de type `NumericOutput`. Il s'agit donc du même type abstrait utilisé pour définir les dispositifs de sortie numériques `leftValueLabel` et `rightValueLabel` dans le texte de la Définition 1. La figure 1 montre qu'il correspond à cette même abstraction deux concrétisations différentes : des labels numériques dans le premier cas et des jauges analogiques dans le second cas. En L ces deux dispositifs sont abstraits en un seul et même type d'interacteur.

Enfin cette Définition 3 montre que des événements peuvent *embarquer* avec eux des objets exprimés comme des paramètres. Ainsi le type de base prédéfini `BooleanInput`, comme d'autres types de base, comporte implicitement dans sa définition en L un attribut `changed`. Cet attribut reflète l'occurrence de l'événement signalant que le dispositif d'entrée auquel il est attaché a été modifié et qu'une nouvelle valeur a été entrée. Cette nouvelle valeur est associée à l'événement. Ainsi la construction syntaxique `changed(newValue)` exprime donc que la valeur de l'interacteur auquel `changed` est attaché a été modifiée et que sa nouvelle valeur est donnée par `newValue`.

EXPLOITER UNE DÉFINITION D'APPLICATION EN L

Le langage L permet de décrire abstraitement une application interactive comme l'association de composants dont chacun est défini par une machine d'états finis. Le comportement attendu de l'application peut être exprimé comme un ensemble de propriétés que le système final doit satisfaire. Ces propriétés peuvent préalablement être vérifiées sur un modèle formel de cette application. Mais il faut pour cela disposer d'un modèle formel de cette application ainsi que d'un modèle des propriétés à vérifier.

Le langage L revêt un certain nombre d'attributs sémantiques propres à ceux mis en oeuvre dans les langages formels. On retrouve explicitement dans L les notions d'états, de transitions, de gardes, d'événements et de modularité. C'est pourquoi il est relativement aisé de traduire mécaniquement des textes en L en notations formelles. Cette section présente trois exemples de traductions vers des langages formels et correspondant à trois cas d'utilisation.

Vérification sur modèle.

Promela [17] est un langage permettant de rapidement formaliser une définition en L. Promela est un langage de modélisation de processus dont l'objectif est de permettre de vérifier la logique de systèmes parallèles. Or une application interactive peut tout à fait être considérée comme un système asynchrone composé de processus parallèles (interacteurs) et communiquant (données et événements).

En disposant d'un modèle Promela de l'application interactive, l'outil Spin peut vérifier la correction de ce modèle en réalisant des simulations itératives ou aléatoires de l'exécution de l'application modélisée, ou en générant un programme C réalisant une vérification rapide et exhaustive de tout l'espace d'états de l'application. Durant ces simulations ou ces vérifications Spin vérifie l'absence de blocages. La vérification peut également être utilisée pour prouver la correction d'invariants du système ou pour détecter d'éventuelles boucles de non progression. Spin permet également la vérification de contraintes temporelles formulées en logique temporelle linéaire. Les exigences ou propriétés attendues du système doivent donc pour cela être formulées en logique temporelle linéaire

pour être alors vérifiées sur le modèle de l'application formalisé en Promela.

Ainsi, en reprenant la définition de l'application illustrée en figure 1 et définie en L, il est possible de montrer sur son modèle Promela que les formules

$$[]((rightGauge.value = right) \wedge (leftGauge.value = left))$$

et

$$[](resetButton.triggered- > <> (left = 0 \wedge right = 0))$$

sont toujours satisfaites. La première formule exprime que, dans tous les états de l'application, la valeur reflétée par les deux jauges est bien en permanence la valeur des données `left` et `right` contrôlées par l'application. Cela reflète une forme d'honnêteté du système d'interaction. La seconde formule exprime que le déclenchement opéré sur l'interacteur `resetButton` réinitialise bien à 0 les valeurs des deux données contrôlées.

Vérification par preuve et raffinement.

La conception d'une IHM en utilisant le langage L est un processus incrémental qui supporte des développements aussi bien *Bottom-Up* que *Top-Down*. En effet, le langage L s'appuie sur la composition de composants déjà existants ou bien sur la décomposition d'un composant en sous composants. Les liaisons établies par les invariants de la clause **`Always`** du langage L permettent de maintenir des liens entre composants et sous-composants.

Ces processus de développement peuvent également être vérifiés en utilisant des approches formelles supportant de telles opérations de composition et/ou de décomposition. La méthode Event B [4], successeur de la méthode B classique [3] est une des méthodes supportant ce type d'opérateurs.

En effet, l'état initial et les comportements décrits dans les composants L peuvent être décrits en Event B respectivement à l'aide de la clause *INITIALISATION* et par des événements de la clause *EVENTS*. Les compositions de composants définis en L sont décrites à l'aide de l'opérateur de raffinement *REFINES* qui permet de concevoir les modèles des sous-composants. Ces derniers sont reliés au modèle composé grâce aux invariants de collage (*gluing invariants*), décrits dans la clause *INVARIANTS* de Event B et extraits de la sous-section **`Always`** du texte en L. Ces invariants permettent de relier les variables des sous-composants aux variables du composant.

A titre d'exemple et à propos des deux propriétés ci-dessus,

1. la première constitue l'invariant de collage du sous-composant `gauges` de type `DualGaugePanel` et du composant global `ControlScreen`. Il lie les variables `left` et `right` du composant `ControlScreen` aux variables `left_value` et `right_value` du sous-composant `gauges` alors que
2. la seconde est un invariant de collage qui permet de relier le sous-composant `resetButton` de type `triggerButton` au composant global `ControlScreen`. Il lie les variables `left` et `right` du composant `ControlScreen` à la variable `triggered` du sous-composant `resetButton`.

L'intérêt de cette approche est double. D'une part, elle permet de formaliser le processus de construction du composant global d'une IHM par des opérations de composition/décomposition, et d'autre part, elle rend plus simple le processus de preuve et de vérification grâce aux invariants de collage et au raffinement qui assurent la préservation des propriétés établies et prouvées dans les phases précédentes du développement. Les preuves sont conduites par induction c'est à dire qu'il s'agit de prouver la préservation des invariants par l'initialisation et par la progression de tous les évènements des différents composants.

Génération de tests.

Enfin, un effet de bord intéressant de l'usage des techniques de vérification sur modèles résulte de la capacité qu'ont les outils de vérification de générer des contre exemples. Ces derniers peuvent être considérés comme des scnénarii conduisant à un état invalide de l'application. Un tel état peut être décrit et identifié par exemple par une formule en logique linéaire temporelle dont la vérification sera insatisfaite sur cet état. Il est dès lors possible d'utiliser ce mécanisme pour générer des scénarios de tests de propriétés de l'application. Ainsi, en reprenant l'exemple de la définition de l'application illustrée sur la figure 1 et en cherchant à vérifier $[]!(rightGauge.value = 3)$ sur son modèle en Promela, il est possible de générer les scénarios d'interaction qui conduisent aux états de l'application dans lesquels la jauge droite présente une valeur égale à 3. En effet de tels états existent et l'outil Spin construit les chemins qui, selon la définition de l'application donnée en L, conduisent à ces états. Ces chemins constituent des scénarios pour tester, sur le code généré de l'application, que la jauge droite peut effectivement correctement afficher la valeur 3. Pour cela, il est nécessaire au préalable de pouvoir générer un code exécutable correspondant à la définition de l'application donnée en L.

GÉNÉRER UN CODE APPLICATIF À PARTIR DE L

La définition de l'application interactive exprimée en L n'est pas directement exécutable, mais elle peut permettre, en revanche, de générer la structure d'un code exécutable pouvant être celui d'un prototype ou de l'application finale destinée à tourner sur une plateforme donnée. Différentes plateformes et différents langages de programmation peuvent être ciblés depuis le langage abstrait L. La génération de code conforme à la norme ARINC 661 [1] offre un exemple de génération possible auquel L n'est pas limité mais qui concerne directement les applications aéronautiques interactives critiques et embarquées à bord de cockpits tels que celui de la figure 2.

Le contexte ARINC 661

ARINC 661 est la norme, utilisée depuis l'A380, qui définit l'architecture de l'IHM à bord des cockpits

Figure 2. Dispositifs d'entrée et d'affichage à bord d'un cockpit

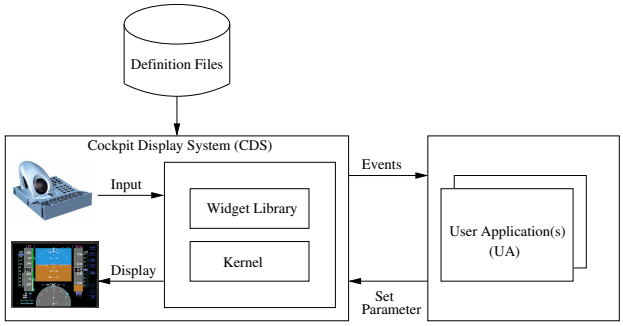

Figure 3. Architecture simplifiée de l'ARINC 661

d'avions. Cette architecture est une architecture client-serveur et l'ARINC 661 définit donc également le protocole de communication entre le client et le serveur qui, tous deux, réalisent le système interagissant avec l'équipage à bord de l'avion.

Le serveur est appellé le CDS (Cockpit Display System). Il comporte les dispositifs d'entrée (clavier et souris) et de sortie (écrans LCD). En se fondant sur une bibliothèque de widgets dont il dispose, le CDS intègre également dans son noyau un gestionnaire d'événements, d'affichage et de widgets répartis dans différentes fenêtres que le CDS gère également. Il est ainsi responsable, entre autres, de la création et de la gestion des widgets, de la gestion des curseurs graphiques, de l'affectation des entrées aux widgets concernés et de la présentation graphique de l'information.

La figure 3, qui décrit l'architecture proposée par l'ARINC 661, illustre en quoi le CDS joue le rôle de serveur. Il reçoit des requêtes d'entrée de l'équipage et restitue, en réponse à ces requêtes, des informations d'affichage sur les écrans. Mais le CDS interagit également avec ses clients logiques que sont les UA (User Applications). Ces dernières sont en prise avec l'état du système piloté et émettent à destination du CDS, par le biais de messages *Set Parameter*, des requêtes de mise à jour des états d'affichage de manière à ce que soit immédiatement présentée à l'équipage toute modification de l'état de ce système. Elles reçoivent en retour, par le biais de messages *Event Notification*, des notifications d'événements, émis par les widgets, qui reflètent les actions de l'équipage sur l'interface. Les UA traitent ces événements en envoyant, le cas échéant et pour le compte de l'équipage, des commandes au système piloté.

Le CDS, pour créer et gérer les widgets, fait appel à des fichiers de définition de l'IHM associée à chaque UA. Ces fichiers définissent la partie statique de l'IHM qu'utilise une UA, comme la disposition des éléments sur l'écran, leurs couleurs, la fonte des textes ...

Pour des raisons de sûreté, les interfaces ARINC sont donc relativement statiques. Sauf lors de l'initialisation, et sur la base des données fournies par les fichiers de définition, il est impossible de créer dynamiquement des widgets ou de les déplacer d'un conteneur à l'autre à la volée lors de l'exécution. L'interface est définie de manière immuable dans le CDS, et n'est modifiable à l'exécution que via les modifications, opérées par les UA, de certains paramètres de contrôle et de présentation.

La réalisation d'une IHM ARINC 661 consiste donc à créer deux entités distinctes. D'une part un fichier de définition, hébergé par le CDS, qui définit l'apparence de l'IHM et d'autre part l'application utilisateur qui en régit le comportement. Le langage L permet de décrire avec précision la partie comportementale de l'IHM sur des interacteurs abstraits. L'UA pourra donc être générée en grande partie automatiquement à partir de sa définition en L. Quand au fichier de définition, la définition en L permet d'en construire le squelette en permettant de construire l'imbrication de différents contrôles dans des conteneurs. Mais cette définition ne permet pas d'en construire les détails, comme la position et la couleur des contrôles. Il faut donc pour cela la concrétiser.

Concrétisation de la définition en L

La concrétisation consiste, pour chaque élément d'interaction abstrait, à choisir le type d'élément d'interaction concret (le widget) disponible qui répond le mieux au besoin défini. Ainsi, dans l'application présentée en exemple, certains interacteurs de type `NumericOutput` sont associés à des widgets labels, et certains autres à des widgets jauges graphiques. La concrétisation consiste également à régler finement les paramètres et l'apparence de l'ensemble de l'interface graphique. Elle se base principalement sur des critères ergonomiques, et permet de se focaliser, bien en aval des étapes de conception, sur les aspects graphiques et visuels de l'interface.

La génération de code ARINC 661

La génération du squelette de fichier de définition est relativement simple. Une fois les widgets choisis et associés aux interacteurs abstraits de la définition L, une première opération de génération consiste à placer les éléments d'IHM dans des conteneurs, imbriqués à la manière de l'arborescence des interacteurs, et à leur donner des identifiants uniques permettant à l'application utilisateur de les identifier correctement. Le fichier de définition ainsi genéré n'est qu'un squelette qui est ensuite complété et enrichi lors de l'étape de concrétisation.

La génération du code de l'application utilisateur est plus complexe. L'approche actuelle consiste dans un premier temps à aplanir l'arborescence des interacteurs et à reprendre les identifiants définis lors de la génération

du squelette de fichier de définition. Un premier bloc de code est généré à partir du comportement **`Init`** de chaque interacteur. Puis le bloc de gestion des événements en provenance du CDS est généré à partir de la définition des comportements de chaque interacteur. A chaque section **`When`** de chaque instance d'interacteur correspond un bloc de code traitant l'évènement correspondant. Dès que le traitement d'un événement nécessite la modification d'une variable impliquée dans la section **`Always`** d'un interacteur, les traitements nécessaires à la vérification de l'invariant sont ajoutés au traitement.

Lorsque la définition abstraite en L a été concrétisée et que les codes ont été générés, l'application est prête à être déployée. Dans le context ARINC 661, cela revient à charger le fichier de définition sur le CDS, le code de l'UA sur son système cible, et à exécuter le tout. La figure 1 représente la copie d'écran d'une UA générée à partir de sa définition L.

CONCLUSION

La définition du langage L est aujourd'hui un projet en cours de réalisation. Parallèlement à cette définition, la faisabilité des transformations et compilations de ce langages vers d'autres notations formelles est évaluée, de même que sa capacité à produire un code exécutable et embarquable. Un des objectifs des travaux en cours est de fonder ces transformations et de les valider en utilisant pour ce faire des techniques de réécriture de termes. Ces transformations validées pourront ensuite efficacement être outillées de manière à faciliter l'exploitation des définitions produites en L.

L'objectif reste néanmoins de contribuer à réaliser des logiciels interactifs embarqués plus sûrs. Il est donc important que le langage puisse incorporer les abstractions des objets et des structures de contrôle dont ont besoin les concepteurs et les développeurs d'IHM. Il doit en particulier permettre de définir des applications prenant en compte de nouveaux paradigmes d'interaction incluant des formes de dynamicité tels qu'ils peuvent être rencontrés dans les IHM post-WIMP.

La prise en compte de tels paradigmes est aujourd'hui impossible, pour des raisons liées à la recherche de la sûreté et aux exigences de certification, dans le domaine des logiciels interactifs aéronautiques embarqués. Les travaux consistant à définir un langage, des techniques et des outils formels permettant d'avoir un haut degré d'assurance sur la sûreté des logiciels produits pourront sans doute contribuer à améliorer les processus de certification et à alléger les contraintes qui pèsent aujourd'hui sur la définition d'interfaces embarquées sans pour autant ne rien perdre de la confiance à accorder aux systèmes développés. Et l'ouverture de ces systèmes d'interaction critiques à des paradigmes aujourd'hui interdits reste ainsi raisonnablement envisageable.

BIBLIOGRAPHIE

1. ARINC 661, Cockpit Display System Interfaces to User Systems. http://www.aviation-ia.com/aeec/projects/cds/index.html. Airlines Electronic Engineering Committee.
2. DO178C. Software Consideration in Airborn Systems and Equipment Certification, release c, 2012. RTCA,Inc.
3. Abrial, J.-R. *The B-book: assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996.
4. Abrial, J.-R. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.
5. Accot, J., Chatty, S., Maury, S., and Palanque, P. A. Formal transducers: Models of devices and building bricks for the design of highly interactive systems. In *DSV-IS*, M. D. Harrison and J. C. Torres, Eds., Springer (1997), 143–159.
6. Aït-Ameur, Y. Cooperation of formal methods in an engineering based software development process. In *Proceedings of the Second International Conference on Integrated Formal Methods*, IFM '00, Springer-Verlag (London, UK, UK, 2000), 136–155.
7. Ait-ameur, Y., Girard, P., and Jambon, F. A uniform approach for specification and design of interactive systems: the b method. In *in Eurographics Workshop on Design, Specification, and Verification of Interactive Systems (DSVIS'98), Vol. Proceedings (Eds, Markopoulos* (1998), 333–352.
8. Aït-Ameur, Y., Girard, P., and Jambon, F. Using the b formal approach for incremental specification design of interactiv systems. In *EHCI* (1998), 91–109.
9. Ausbourg (d'), B., Durrieu, G., and Roché, P. Deriving a formal model of an interactive system from its UIL description in order to verify and to test its behaviour. In *Proceedings of the Eurographics Workshop DSV-IS'96* (Namur, Belgium, June 1996).
10. Ausbourg(d'), B. Using Model Checking for the Automatic Validation of User Interfaces Systems. In *Design, Specification and Verification of Interactive Systems '98*, P. Markopoulos and P. Johnson, Eds., Eurographics, Springer (June 1998).
11. Bastide, R., Navarre, D., and Palanque, P. A model-based tool for interactive prototyping of highly interactive applications. In *CHI '02 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '02, ACM (New York, NY, USA, 2002), 516–517.
12. BEA. Rapport final sur l'accident survenu le 1er juin 2009 à l'Airbus A330-203 immatriculé F-GZCP exploité par Air France, vol AF 447 Rio de Janeiro - Paris. Tech. rep., Direction Générale de l'Aviation Civile, Juillet 2012. **`http://www.bea.aero/docspa/2009/f-cp090601/pdf/f-cp090601.pdf`**.
13. Blanch, R. Programmer l'interaction avec des machines à états hiérarchiques. In *Actes de la 14ème conférence francophone sur l'Interaction Homme-Machine (IHM 2002)* (2002), 129–136.
14. Blanch, R., and Beaudouin-Lafon, M. Programming rich interactions using the hierarchical state machine

toolkit. In *Proceedings of the working conference on Advanced Visual Interfaces (AVI 2006)* (2006), 51–58.

15. d'Ausbourg, B., Seguin, C., Durrieu, G., and Roché, P. Helping the automated validation process of user interfaces systems. In *ICSE*, K. Torii, K. Futatsugi, and R. A. Kemmerer, Eds., IEEE Computer Society (1998), 219–228.
16. Duke, D. J., and Harrison, M. D. Abstract Interaction Objects. *Computer Graphics Forum 12*, 3 (1993), 25–36.
17. Holzmann, G. *Spin model checker, the: primer and reference manual*, first ed. Addison-Wesley Professional, 2003.
18. USer Interface eXtended Markup Language. **http://www.usixml.eu/**.
19. Jacob, R. J. K., Deligiannidis, L., and Morrison, S. A software model and specification language for non-wimp user interfaces. *ACM Transactions on Computer-Human Interaction 6* (1999), 1–46.
20. Jensen, K. *Coloured Petri nets: basic concepts, analysis methods and practical use, vol. 2*. Springer-Verlag, London, UK, UK, 1995.
21. Palanque, P., and Bastide, R. Interactive cooperative objects : an object-oriented formalism based on petri nets for user interface design. In *Proceedings of the IEEE Conference on System Man and Cybernetics*, Elsevier Science Publisher (October 1993).
22. Palanque, P., and Bastide, R. Synergistic modelling of tasks, users and systems using formal specification techniques. *Interacting with Computers 9*, 2 (1997), 129–153.
23. Paternò, F., and Faconti, G. On the use of LOTOS to describe graphical interaction. In *Proceedings of the HCI'92 Conference on People and Computers* (1992), 155–173.
24. Schyn, A., Navarre, D., Palanque, P., and Porcher Nedel, L. Description Formelle d'une Technique d'Interaction Multimodale dans une Application de Réalité Virtuelle Immersive . In *IHM'2003: 15th French Speaking conference on human-computer interaction, Caen, France, 24/11/2003-28/11/2003*, ACM Press (novembre 2003), 150–157.