Vers la certification de programmes interactifs Djnn

Pascal Béger, Sébastien Leriche, Daniel Prun

Vers la certification de programmes interactifs Djnn

Pascal Béger[1], Sébastien Leriche[1], and Daniel Prun[1]

[1]ENAC, Université de Toulouse - France

Résumé

Les systèmes critiques, particulièrement aéronautiques, contiennent de nouveaux dispositifs hautement interactifs. Pour certifier les logiciels qui les exploitent, les approches de vérification existantes ne sont pas toujours adaptées. Dans ce papier court, nous introduisons notre approche de construction d'applications interactives au moyen de Djnn (le modèle et l'API) et Smala (le langage réactif de haut niveau) puis nous discutons de la pertinence des outils et méthodes de vérifications dans ce contexte. Nous présentons enfin nos premiers résultats et perspectives de vérification formelle de programmes interactifs Djnn.

1 Introduction

Les systèmes critiques, particulièrement aéronautiques, contiennent de nouveaux dispositifs hautement interactifs : par exemple les cockpits de nouvelle génération font appel à une électronique sophistiquée pilotée par des applications logicielles complexes. Dans ce contexte, les processus de certification décrits dans la DO-178C/ED-12C offrent une place importante à la vérification formelle. Même si, depuis ces 3 dernières décennies, de nombreuses méthodes outillées ont été proposées pour le développement de logiciels sûrs et corrects d'un point de vue fonctionnel, les logiciels interactifs n'ont cependant pas bénéficié de la même attention : leur vérification repose encore principalement sur des évaluations de prototypes au cours de tests avec les utilisateurs finaux. Lorsque des techniques de vérification formelle sont utilisées, celles-ci ne proposent très souvent qu'une couverture partielle de la problématique des applications interactives. Par exemple les éléments issus de l'interface concrète sont peu pris en compte. La nature réactive, synchrone, mixant flot de contrôle et flot de données est difficile à appréhender à travers une unique approche et bien souvent plusieurs techniques doivent être utilisées de manière conjointes, ce qui complexifie la tâche.

Nous pensons qu'une part importante de ces difficultés est due à l'absence d'un langage spécifique permettant, par le biais d'une sémantique adaptée, aux concepteurs d'itérer sur la conception et aux développeurs de vérifier que leur produit est conforme à une spécification de référence.

Après avoir présenté le contexte de notre travail qui se concrétise par le développement du framework Djnn et du langage Smala, nous discutons du positionnement des principaux paradigmes de programmation ainsi que des approches de vérification formelles associées. Cela nous permet de positionner et justifier notre approche : Djnn repose sur un ensemble de concepts réduits et unifiés, ce qui nous semble pertinent pour décrire efficacement les applications interactives et facilite la création de passerelles vers les techniques usuelles de vérification formelle. Nous présenterons alors nos premiers résultats et les pistes envisagées pour nos travaux futurs.

2 Contexte

L'équipe « Informatique Interactive » de l'École Nationale de l'Aviation Civile travaille depuis plusieurs années sur une approche autour d'un modèle conceptuel d'exécution unifié dédié aux

applications interactives. Une implémentation sous la forme d'une API nommée **Djnn**[1] et des applications significatives sont venues valider notre approche [4]. Nous avons récemment montré comment ce modèle permettait la vérification et la validation de certaines propriétés inhérentes aux systèmes interactifs [10] et Djnn sera un des environnements d'exécution cible du projet FORMEDICIS[2].

Pour simplifier l'utilisation de Djnn, nous développons le langage **Smala**[3]. Smala permet de décrire de manière déclarative une application interactive selon le paradigme réactif. Une application en Smala contient un ensemble de composants où le contrôle est réduit à la transmission d'activation entre ces composants. A la base Smala contient 4 types de composants primitifs :

- La *property* est un composant permettant de stocker une valeur. La mise à jour de la valeur provoque l'activation du composant.
- Le *binding* (noté ->) permet la création d'un couplage entre 2 composants : quand le composant source est activé, alors le composant destination est immédiatement activé.
- L'*assignment* (noté =:) permet la transmission de valeur entre 2 *properties*.
- Le *comparator* (noté =) permet le test d'égalité de valeurs entre 2 *properties*.

A partir de ces composants primitifs, des composants de plus haut niveau sont définis par composition. Par exemple le *connector* contenant un *binding* et un *assignment* permet de transmettre simultanément une valeur et une activation ; le *if* constitué d'un *comparator* et d'un *binding* permet de transmettre une activation sous condition. En continuant ainsi, de nombreux autres composants plus complexes sont construits : le *switch*, la *fsm* (machine à état), le conteneur, les composants arithmétiques.... L'interface avec l'environnement est assurée par des composants dédiés fournis par Djnn. Par exemple les composants graphiques permettent de décrire les éléments à afficher (forme géométrique, position, couleur, opacité...) ainsi que les éventuels clics souris les concernant (position du clic, type de clic : press ou release...) ; les composants *clock* permettent de déclencher périodiquement une activation. L'exécution d'une telle application consiste à gérer l'activation des différents composants résultant soit de l'occurrence d'événements provenant de l'environnement (un clic souris, la modification de la valeur d'une propriété...), soit de la propagation explicite de l'activation décrite par le programmeur via les composants de *binding*.

Le compilateur Smala génère actuellement du code C ou Java utilisant l'API Djnn. Une illustration de Smala est proposée par la Figure 1.

Figure 1: Code Smala, extrait du code C/Djnn et résultat de l'exécution d'un chronomètre

Dans cet exemple, nous pourrions souhaiter valider les propriétés suivantes : l'aiguille des secondes est toujours visible (ie. elle n'est à aucun moment masquée par un autre élément graphique),

[1] `http://djnn.net`

[2] FORMEDICIS est un projet ANR en cours (ANR-16-CE25-0007) à l'origine du financement de ces travaux.

[3] `http://smala.io`

la valeur affichée par l'aiguille des secondes (liée à la perception humaine de la lecture d'une horloge) correspond à la valeur du chronomètre ou encore qu'un clic de souris sur le bouton gauche du chronomètre déclenche son départ. La spécificité de ce type de propriétés est d'être liée à la représentation concrète des éléments et aux interactions permises à l'utilisateur. De ce fait, leur vérification se fait traditionnellement au moyen de tests et nécessite un observateur humain.

3 Positionnement des principaux paradigmes de programmation

Les applications informatiques peuvent être développées en utilisant différents paradigmes de programmation. Même si nous nous intéressons à des applications réactives, cette section permet un positionnement qui nous permettra de discuter de l'impact de ces différences sur la vérification.

3.1 Analyse

Le paradigme impératif consiste en l'écriture d'une séquence de commandes devant être réalisées par l'application et modifiant son état [15]. Des structures de contrôle (conditionnelle, boucle...) permettent de faire évoluer le flot d'exécution par des changements d'états (mémoire/variables et pointeur sur l'instruction courante). Cet état permet l'interruption ou l'exécution pas à pas [23]. Dans ce paradigme, la vérification peut se focaliser sur l'état de la mémoire, le respect des bornes des variables ainsi qu'à l'absence de boucles infinies.

Un programme fonctionnel correspond à une fonction mathématique transformant son entrée en une sortie [24]. Il repose sur la conversion, consistant en un renommage des paramètres (pouvant être également des fonctions) et de réduction, représentant les appels de fonctions. Le flot d'exécution est décrit par les entrées et sorties des fonctions. En fonctionnel pur, les effets de bord sont interdits, l'affectation de valeur n'existant pas [24]. Le paradigme fonctionnel est donc sans état, ce qui simplifie la mise au point du programme puisqu'il est possible de tester et valider les fonctions indépendamment les unes des autres. Dans ce paradigme, les vérifications portent souvent sur la propriété de terminaison, les empilements des appels de fonction (déterminant l'utilisation de la pile).

La programmation orientée objet permet d'abstraire les données [13] de façon à obtenir une représentation proche des objets manipulés dans la réalité. Les concepts clés [17] sont une structuration du code et des données au sein de classes isolant ces données par encapsulation. La réutilisation est favorisée par l'héritage, le polymorphisme permet de manipuler des objets dont les types sont compatibles (issus d'une hiérarchie de classes). Les attributs d'un objet définissant son état interne, la programmation objet est de type avec état. Le flot d'exécution d'une application orientée objet suit les appels de méthodes entre objets [20]. Les propriétés intéressant la vérification sont par exemple le respect du principe d'encapsulation, des relations entre objets et de la causalité des appels de méthodes.

Dans une application réactive, des événements provenant de sources internes (horloge...) ou extérieures (clic de souris, appui sur le clavier...) déclenchent un comportement programmé. Ces événements se propagent en mettant à jour leurs dépendances par le concept de suite d'activation. Ainsi, la modification d'une variable peut provoquer la mise à jour d'autres variables et l'exécution de comportements qui en dépendent. L'exécution d'un programme réactif suit le flot de données, les événements extérieurs et la propagation automatique des changements se produisant en fonction des dépendances entre les composants [5]. Un programme réactif est donc sans état. Le respect des dépendances, le suivi du flot de données et de l'activation sont des propriétés pertinentes à vérifier pour les applications réactives.

3.2 Synthèse

Nous avons volontairement ciblé notre analyse (tableau 1) de manière à extraire les caractéristiques pertinentes à chaque paradigme, dans un objectif de vérification formelle. Toutefois, les langages récents offrent la possibilité de manipuler différents paradigmes. Par exemple, Scala.React est une couche réactive de Scala qui est une extension fonctionnelle de Java. Certains langages comme

Paradigme	Impératif	Fonctionnel	Orienté objet	Réactif
Flot d'exécution	Itératif, structure de contrôle	Suivi I/O fonctions	Contrôle, séquences appels méthodes	Données, arbre de dépendance
État	Avec état	Sans état	Avec état	Sans état
Définition de l'état	Variables en mémoire, locus de contrôle	*N/A*	État interne des objets (attributs) en mémoire	*N/A*
Propriétés pertinentes	État mémoire, bornes variables, boucles finies	Terminaison	Encapsulation, causalité appels méthodes	Dépendances, suivi flot de données et activation
Exemples de langages	Fortran, Pascal, C	Caml, Haskell,	Java, C#	FlapJax, Scala.React

Table 1: Synthèse des paradigmes de programmation

Oz ont été même inventés pour être multi-paradigme. Nous pensons que si de tels langages sont pertinents pour offrir au programmeur une forte expressivité, leur complexité augmente la difficulté des vérifications. Notre approche, ciblant les programmes interactifs et basée sur un modèle conceptuel simple devrait permettre de réduire la complexité des vérifications pour aboutir à de la certification.

4 Méthodes et outils pour la vérification

La vérification formelle consiste à s'assurer que l'application est conforme à une spécification de référence. Elle se base sur l'élaboration d'un modèle dont la syntaxe et la sémantique reposent sur des bases mathématiques permettant ainsi des raisonnements objectifs.

4.1 Moyens de modélisation

On distingue différentes catégories de modèles formels. Nous présentons ci-dessous les principales.

La logique de Hoare [16] repose sur la notion du triplet de Hoare $\{P\}Q\{R\}$ dans lequel P et R sont des formules logiques ou assertions représentant respectivement une précondition et une postcondition et Q le programme typiquement impératif. Cette modélisation permet de prouver qu'un programme vérifie une spécification, en garantissant la véracité de R à la fin de Q si P était vraie à l'initialisation de Q.

Un réseau de Petri est un triplet $\langle P, T, F\rangle$ avec P un ensemble de places p, T un ensemble de transitions t et F la relation de flot [22]. Les places peuvent contenir des jetons qui conditionnent l'exécution (on parle de tirage) des transitions : un tirage se concrétise par la consommation et la production de jetons. Ce modèle permet de vérifier des propriétés génériques d'atteignabilité d'état (vivacité, non-blocage) et de bornes.

Un automate [8] est défini par des ensembles finis d'états, de transitions associées à des labels, d'un état initial et d'une fonction de correspondance associant des propriétés à chaque état. Un tel automate avec une relation de transition et une fonction d'étiquetage est appelé structure de Kripke. Cette modélisation permet de mettre en oeuvre les techniques de *model checking*, où des propriétés exprimées sur les séquences de transitions franchies sont vérifiées.

L'interprétation abstraite [12] permet de décrire la sémantique opérationnelle d'un programme en ne conservant que les informations pertinentes pour le type de propriété à vérifier. Cette sémantique opérationnelle est représentée par un ensemble de traces sur lequel des vérifications par preuve sont possibles.

Le B au même titre que VDM ou Z, est une technique de raffinements successifs qui permet d'obtenir, à partir d'une abstraction haut niveau, une abstraction de plus en plus concrète. Cette méthode a évolué vers une approche plus générale, B-événementiel [2], prenant en compte les événements. A chaque raffinement, de nouvelles propriétés du système sont introduites. La vérification consiste à assurer (par preuve) que les propriétés exprimées dans les raffinements précédents

restent vraies pour le raffinement en cours. La plateforme Rodin permet de mettre en oeuvre cette approche.

Les propriétés à vérifier sont souvent exprimées sous la forme de formules de logique temporelle (CTL*, CTL et LTL). Elles reposent sur des combinateurs booléens, temporels ainsi que des quantificateurs de chemin [8]. CTL* et CTL s'intéressent à l'arbre de l'ensemble des exécutions tandis que LTL s'intéresse à une seule exécution. Ces propriétés peuvent être encodées par des observateurs qui permettent de reconnaitre des traces les respectant. De part leur grande expressivité, elles sont largement utilisées dans de nombreuses approches reposant sur la preuve ou le *model checking*.

4.2 Techniques de vérification

On distingue généralement 2 grandes techniques de vérification : le *model checking* et la preuve.

Le **model checking** est une technique permettant de vérifier quels états d'une structure de Kripke vérifient une formule de logique temporelle donnée [11]. Elle repose sur l'élaboration par simulation de l'ensemble des états possible du système, ainsi que des transitions entre ces états. Elle n'est utilisable que pour des applications dont le nombre d'états possibles est fini, pour des propriétés de sûreté. UPPAAL, SMV (Symbolic Model Verifier) ou encore SPIN sont des outils représentatifs de cette approche. TINA, Design/CPN ou GreatSPN supportent cette approche avec les réseaux de Petri.

La **preuve** est une approche où un ensemble de constructions et de règles mathématiques sont appliquées pour démontrer un résultat à atteindre. La théorie des graphes, la théorie des ensembles, les différentes logiques sont les principaux domaines utilisés où des techniques de déduction sont appliquées. Cette approche est utilisée pour des applications dont le nombre d'états possibles est potentiellement infini et permet de vérifier des propriétés de vivacité. PVS-Studio, Coq, HOL (a Higher Order Logic Theorem Prover) ou encore Isabelle sont les outils principaux utilisés pour cette approche.

4.3 Synthèse

Le paradigme des systèmes interactifs étant relativement récent, peu de travaux ont été réalisés concernant leur vérification formelle. Par exemple, concernant l'utilisation du *model checking*, [1] vérifie des interfaces utilisateur avec SMV (Symbolic Model Verifier) en utilisant CTL pour exprimer les propriétés à vérifier. [19] modélise le système avec un réseau de Petri orienté objet appelé ICO et mène des vérifications formelles limitées à un sous-ensemble du réseau. [14] a utilisé un langage à flots de données pour la validation automatique de systèmes interactifs. Dans le domaine des techniques de preuve, VDM, et Z ont été utilisées pour la définition de structures atomiques d'interaction [6]. HOL a été utilisée pour la vérification de spécifications d'interfaces utilisateur [7] et Event-B pour la modélisation et la validation de propriétés sur des systèmes multi-modaux [3]. Plus récemment, pour des systèmes médicaux, [18] ont utilisé un modèle en VDM sur lequel des propriétés de cohérence et de feedback ont été prouvées.

Historiquement, les techniques classiques d'analyse ont été développées pour être appliquées sur des langages impératifs ou fonctionnels. Celles-ci restent peu utilisées pour la prise en compte des spécificités des systèmes interactifs ainsi que celles des propriétés qui s'y rapportent. Ainsi, la modélisation concerne les parties abstraites de l'interaction (tâches, interface abstraite) et n'aborde pas les aspects concrets (objets graphiques, dispositif d'entrée, etc.). Démontrer des propriétés impliquant ce niveau de description (visibilité d'information par exemple) reste difficile à prendre en compte.

Dans ce contexte, notre objectif à moyen terme est d'étudier comment les techniques classiques de vérifications formelles peuvent être adaptées aux systèmes interactifs. Nous pensons que Djnn est un environnement pertinent pour réduire la complexité des activités de vérification formelle, grâce à sa capacité à exprimer de manière uniforme et avec peu de concepts, des éléments de l'interface aussi bien abstraite (contrôle...) que concrète (graphisme...).

La section suivante présente l'état de nos recherches ainsi que les perspectives que nous envisageons.

5 Travail réalisé et perspectives

Bien que Smala soit encore en cours de développement, nous avons déjà pu développer des techniques de vérification formelle de propriétés de systèmes interactifs, toutes s'appliquant au niveau du code Smala. Dans [9] nous avons exploité les caractéristiques du **graphe d'activation**. Ce graphe, déduit de la description en Smala, décrit toutes les relations d'activation possibles suite à l'occurrence d'un événement. Sur la base de son analyse, il est possible de vérifier formellement des propriétés d'atteignabilité (par exemple qu'une entrée finit toujours par générer une sortie attendue ou qu'une alarme est toujours déclenchée dans une configuration) ou encore relatives à la relation causale d'activation (par exemple qu'un message d'erreur affiché ne sera jamais recouvert par un autre).

Sur l'exemple introduit précédemment en figure 1, la propriété stipulant "qu'un clic de souris sur le bouton gauche du chronomètre déclenche son départ" est formellement vérifiée en s'assurant qu'il existe un chemin dans le graphe d'activation reliant l'événement "clic de souris" du composant "bouton gauche" à l'événement "départ" du composant "chronomètre". La propriété stipulant que "l'aiguille des secondes doit être toujours visible" est vérifiée en exploitant l'ordre de parcours des éléments du graphe suivi à l'exécution : en profondeur, de gauche à droite. Les éléments graphiques étant affichés selon cet ordre, vérifier la propriété consiste à s'assurer qu'aucun composant graphique est présent dans le graphe après le composant de l'aiguille des secondes, ou dans le cas contraire que le composant graphique ne recouvre pas l'aiguille[4].

Nous travaillons actuellement sur l'implantation d'**observateurs synchrones** en Smala : il s'agit de composants pouvant observer les activations au sein d'une application Smala, et reconnaitre l'occurrence de patterns pré-définis. D'autre part, nous étudions [21] la transformation de code Smala vers des **réseaux de Petri**, avec l'idée d'exprimer précisément une sémantique opérationnelle pour Smala et de profiter des possibilités de vérification qui sont disponibles sur ces réseaux. Nos perspectives à moyen terme concernent, d'une part, la prolongation des études précédentes (analyse du graphe d'activation et utilisation des réseaux de Petri) et, d'autre part, l'étude de l'utilisation de techniques de preuve formelle à partir de code Smala (traduction en Caml et utilisation de COQ, traduction en event-B).

References

[1] G. D. Abowd et al. A formal technique for automated dialogue development. In *Proceedings of the 1st Conference on Designing Interactive Systems: Processes, Practices, Methods, & Techniques*, DIS '95, pages 219–226, New York, NY, USA, 1995. ACM.

[2] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, New York, NY, USA, 1st edition, 2010.

[3] Y. Aït-Ameur et al. Vérification et validation formelles de systèmes interactifs fondées sur la preuve : application aux systèmes Multi-Modaux. *JIPS*, 1(1):1–30, September 2010.

[4] P. Antoine et al. Volta: the first all-electric conventional helicopter. In *MEA 2017, More Electric Aircraft* , Bordeaux, France, February 2017.

[5] E. Bainomugisha et al. A survey on reactive programming. *ACM Comput. Surv.*, 45(4):52:1–52:34, August 2013.

[6] J. Bowen and S. Reeves. Modelling safety properties of interactive medical systems. In *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '13, pages 91–100, New York, NY, USA, 2013. ACM.

[4] En toute rigueur, d'autres vérifications complémentaires sont effectuées : vérifier que l'aiguille des secondes n'est pas affichée de manière transparente, c'est à dire que ce composant graphique n'est pas précédé dans le graphe d'un composant d'opacité ayant une valeur faible ; vérifier que la couleur d'affichage de l'aiguille est différente de celle des composants affichés précédemment.

[7] P. Bumbulis et al. Validating properties of component-based graphical user interfaces. In Francois Bodart and Jean Vanderdonckt, editors, *Design, Specification and Verification of Interactive Systems '96*, pages 347–365, Vienna, 1996. Springer Vienna.

[8] B. Bérard et al. *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[9] S. Chatty et al. Verification of properties of interactive components from their executable code. In *Proceedings of the 7th ACM SIGCHI*, EICS '15, pages 276–285, New York, NY, USA, 2015. ACM.

[10] S. Chatty et al. Designing, developing and verifying interactive components iteratively with djnn. In *ERTS 2016*, TOULOUSE, France, January 2016.

[11] E. M. Clarke, Jr. et al. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[12] P. Cousot. Interprétation abstraite. *Technique et science informatiques*, 19(1):155–164, 2000.

[13] B. J. Cox et al. *Object-Oriented Programming; An Evolutionary Approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1991.

[14] B. d'Ausbourg. Using model checking for the automatic validation of user interface systems., 01 1998.

[15] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1997.

[16] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[17] M. Hristakeva et al. A survey of object oriented programming languages. Technical report, University of California, Santa Cruz, 2009.

[18] P. Masci et al. Formal verification of medical device user interfaces using pvs. In *Fundamental Approaches to Software Engineering*, pages 200–214, Berlin, Heidelberg, 2014.

[19] D. Navarre et al. A formal description of multimodal interaction techniques for immersive virtual reality applications. In *Proceedings of the 2005 IFIP TC13*, INTERACT'05, pages 170–183, Berlin, Heidelberg, 2005. Springer-Verlag.

[20] O. Nierstrasz. Object-oriented concepts, databases, and applications. chapter A Survey of Object-oriented Concepts, pages 3–21. ACM, New York, NY, USA, 1989.

[21] D. Prun, M. Magnaudet, and S. Chatty. Towards support for verification of adaptative systems with djnn. In *Proceedings of Cognitive 2015*, pages 191–194. IARIA, 03 2015.

[22] W. Reisig. *Understanding Petri Nets: Modeling Techniques, Analysis Methods, Case Studies*. Springer Publishing Company, Incorporated, 2013.

[23] G. Salvaneschi et al. Debugging for reactive programming. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 796–807, New York, NY, USA, 2016. ACM.

[24] P. Van Roy et al. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, 1st edition, 2004.