Produire un logiciel de confiance : quelles hypothèses, quelles limites?

T. Hardin[1]

INRIA, Projet Moscova et
Laboratoire d'Informatique de Paris 6 (LIP6),
8, Rue du Capitaine Scott,
75015 PARIS, France
`Therese.Hardin@lip6.fr`

Les composants logiciels sont devenus des éléments essentiels dans la plupart des systèmes complexes qui concourent à instrumenter notre vie quotidienne, que ce soit dans le cadre des transports, des moyens de communication, de la médecine, des services bancaires, etc. Les concepteurs de tels systèmes ne peuvent les mettre en exploitation qu'après avoir fourni un certain nombre de garanties aux autorités de tutelle : garanties de respect des cahiers des charges, garanties de tolérance aux fautes, garanties de confidentialité, etc. qui sont souvent décrites par des normes comme la (famille de) norme européenne CEI IEC 61508[7] ou les critères communs[12]. Le recensement des problèmes et les exigences à respecter pour satisfaire à ces normes font l'objet d'un nombre élevé de communications. Mais les moyens à mettre en œuvre n'y sont pas toujours décrits avec une grande précision, pour des raisons de propriété industrielle mais aussi parce qu'ils reposent encore beaucoup sur des notions très empiriques. L'informatique n'est pas perçue comme un corpus de connaissances fondamentales, contrairement à la physique ou à d'autres sciences fondant la pratique de l'ingénieur. Par exemple, il est acquis que la mise sur orbite d'un satellite est fondée sur des connaissances théoriques apportées par la mécanique, la physique et les mathématiques. L'apport de l'informatique, absolument essentiel dans ce cas, n'est souvent perçu que comme une collection de savoir-faire. Mieux maîtriser la complexité des systèmes à logiciel prépondérant ne se fera, me semble-t-il, qu'en accentuant l'apport et donc le développement de la recherche fondamentale en informatique. Cet apport est bien illustré par les méthodes dites "formelles" qui en sont issues, certaines d'entre elles ayant d'ores et déjà une place reconnue dans l'industrie des systèmes (l'exemple de METEOR construit en partie avec le système B[1] est convaincant).

Je voudrais dans la suite de ces notes cerner la notion de "logiciel de confiance". Tout comme une "personne de confiance", il s'agit d'un logiciel que l'on *croit* : ses réactions vis-à-vis de l'environnement sont prévisibles, contrôlées, non-dangereuses, les résultats de son utilisation sont garantis. Prévoir, démontrer, contrôler les propriétés mais d'abord construire un tel logiciel ne peut se faire que de manière raisonnée. Il faut pour cela une bonne connaissance de la logique, de la calculabilité, de la sémantique, tous savoirs qui fondent les outils de compilation, d'analyse des programmes et plus généralement les méthodes formelles.

Ces notes ne prétendent aucunement à l'originalité et toutes les remarques ci-dessous figurent déjà dans un certain nombre d'articles. Il me semble néanmoins utile de les redire encore car la pratique quotidienne, tant au niveau de l'enseignement que dans l'industrie du logiciel, montre que leur contenu n'est toujours pas passé dans les faits. Mon souhait est qu'elles soient le point de départ à une réflexion sur l'enseignement et la pratique de la programmation.

Par "logiciel", je désigne un ensemble de programmes écrits dans un ou plusieurs langages source. Il ne s'agit donc pas d'un système embarquant du logiciel : le sujet est trop vaste et de nombreuses réponses ont déjà été apportées dans la littérature concernant la sûreté de fonctionnement. Le logiciel considéré ici peut être dédié à une application donnée ou être inclus dans une librairie. On rejoint ainsi la problèmatique des COTS (Composants Off The Shelves) dans le cadre de la sûreté. Notons au passage que la notion de "système certifié" fait souvent référence à une norme et à un niveau dans cette norme. Il semble difficile de dire qu'un logiciel certifié au niveau le moins exigeant de certaines normes soit vraiment un "logiciel de confiance".

Dans tout ce qui suit, l'utilisation de tests ne sera pas mentionnée explicitement. Quelles que soient les méthodes utilisées pour augmenter la confiance dans un logiciel, les tests restent toujours nécessaires, même si certains tests unitaires peuvent parfois être allégés. Le test est en quelque sorte la validation de l'action "créer la confiance", un peu comme la validation numérique confirme la confiance en des résultats de calculs exacts.

Ces notes sont issues de notre expérience dans le projet Foc[4, 5, 6], qui porte sur le développement d'un logiciel de confiance pour le calcul formel. Nous avons quotidiennement à choisir entre une vision idéale de la confiance et une vision plus pragmatique, mais qui permet d'avancer en temps raisonnable.

Donner confiance requiert d'identifier tous les points dangereux dans les étapes de construction et d'exploitation d'un logiciel. Dans ce qui suit, je vais en recenser quelques uns, sans prétendre à l'exhaustivité, en ne donnant que quelques commentaires. Je vais pour cela partir de l'exploitation du logiciel pour remonter vers le cahier des charges. J'essaierai de dégager quelques hypothèses (mises en italique) permettant de simplifier — ou même tout simplement de mettre en œuvre — l'étude du logiciel. Je tenterai aussi d'indiquer les limites des choix effectués.

Ultimement, un logiciel conduit à du code binaire qui est exécuté par un processeur donné dans un environnement matériel donné. Si on ne peut pas faire abstraction de l'environnement, alors on retrouve la problématique générale des systèmes critiques embarqués. Sinon, on fera l'hypothèse que *l'exécution du logiciel ne dépend pas de l'environnement matériel*. Abstraire le processeur de la même manière est moins simple. Même si on fait l'hypothèse que *le processeur fonctionne correctement*, il faut encore décider de la manière dont le logiciel dépend des choix de sa conception (des arithmétiques par exemple).

Le binaire est produit à partir du code objet par un gestionnaire d'exécution qui coopère avec le système d'exploitation. Cette coopération n'est pas nécessairement harmonieuse et on a un certain nombre d'exemples de désaccord sur les bibliothèques partagées, sur la gestion des processus légers, etc. Peut-on accepter qu'un logiciel soit de confiance en faisant abstraction du système d'exploitation sur lequel il est exécuté ? en faisant abstraction du gestionnaire d'exécution du langage ? Nous cernons ici une limite importante de la notion de "logiciel de confiance". Pour ne pas être dédié à une seule version d'un seul système d'exploitation sur un seul processeur, il faut faire l'hypothèse que *le système d'exploitation et le gestionnaire d'exécution coopèrent de manière pleinement correcte à l'exécution du logiciel*.

La compilation traduit le source en du "code objet", vocable qui recouvre ici du vrai code-objet, du code trois-adresses (byte-code et son interpréteur), langage d'assemblage, l'hexadécimal. On souhaiterait pouvoir affirmer que les compilateurs les plus utilisés d'un langage lui-même beaucoup utilisé parviennent au bout d'un certain temps à être exempts d'erreurs. Malheureusement, c'est rarement le cas. De plus, la compilation peut engendrer un certain nombre d'erreurs dûes au fait que compilateur et programmeur ne donnent pas toujours la même sémantique à une même construction syntaxique. Si on peut faire l'hypothèse que *le langage source utilisé est sémantiquement fondé, les compilateurs respectent scrupuleusement la sémantique du langage*, alors on peut considérer que les propriétés démontrées sur le source restent valides au niveau objet. C'est une hypothèse très forte, qui doit être rediscutée pour chaque logiciel étudié. Si on ne peut pas faire cette hypothèse, force est de se tourner vers le code objet pour augmenter la confiance. C'est possible mais cela présente plusieurs inconvénients. Tout d'abord, on étudie un couple (source + compilateur) et l'étude est à refaire pour chaque compilateur. Ensuite, les codes objet sont en général beaucoup plus gros que les sources correspondantes. Enfin, les outils d'analyse du code objet, bien que s'améliorant tous les jours, ne savent traiter ni des structures de donnée complexes ni des propriétés fines des structures de contrôle (terminaison de boucles par exemple). Le handicap majeur réside dans la distance entre la spécification du logiciel, dès qu'elle est un peu riche, et sa dissémination dans le code.

Supposons avoir toutes garanties (ou accepter de possibles erreurs) sur la sémantique du langage et la chaîne de compilation/exécution. Il faut maintenant préciser quelles sont les propriétés à démontrer sur le source pour que le logiciel soit de confiance. On peut les partager grosso-modo en deux catégories. Certaines propriétés assurent que le logiciel respecte bien sa spécification. Les autres propriétés assurent la correction des opérations effectuées. Par exemple, l'implantation de l'algorithme d'Euclide doit respecter la spécification (mathématique) du pgcd de deux entiers. On démontrera donc que le résultat est bien le pgcd et qu'il est obtenu en un nombre fini d'étapes. De plus, si les entiers manipulés sont les entiers natifs, il faut assurer que les opérations effectuées ne donnent pas lieu à des débordements.

Un programme peut utiliser une ou plusieurs arithmétiques entières et/ou flottantes en précision donnée à l'avance. Ces arithmétiques dépendent du langage de programmation choisi et du processeur visé. S'affranchir de cette dépendance est difficile et parait pourtant nécessaire, en particulier pour des librairies. Lorsque plusieurs arithmétiques sont utilisées en même temps, le langage peut autoriser des coercions implicites. Faut-il les expliciter systématiquement ? Sinon, dans quel cas ? Faut-il de même considérer tous les débordements/sous-débordements possibles et démontrer qu'ils ne peuvent jamais se produire ? Par exemple, dans la librairie Foc, les degrés de polynômes sont implantés par des entiers natifs, avec un risque potentiel de débordement. Il semble qu'en l'état actuel des implantations, le débordement ne se produirait que longtemps après la saturation totale de la mémoire. Peut-on pour autant présager qu'il en sera toujours de même, pour n'importe quelle manipulation des polynômes ? La réponse est difficile car le risque de débordement peut rarement être éliminé par une analyse statique et les tests à l'exécution ont toujours un coût. La seule hypothèse possible est la suivante :*les opérations des arithmétiques fournies par le langage sont correctes.*. Cette hypothèse ne peut être étendue à n'importe quelle opération arithmétique. Par exemple, le reste de la division de deux entiers négatifs fournit des résultats différents d'un langage à l'autre.

Un programme utilise aussi des types de données. Si le langage n'est pas typé, il est nécessaire de démontrer que les données vérifient les invariants de typage. *Si le langage est fortement typé, si la correction du typage par rapport à la sémantique d'exécution est démontrée*, alors on peut faire l'hypothèse que *tout traitement sur les données correctement typé respecte de fait les invariants de typage.*

Le respect des invariants de typage étant obtenu, il reste à démontrer un certain nombre de propriétés propres à chacun des types. Par exemple, les traitements de chaînes de caractères peuvent être décrits à des niveaux différents, les considérant soit comme des tableaux soit comme un type abstrait et peuvent conduire à des débordements. Si une troncature dans l'affichage d'un message n'entamera que peu la confiance dans le logiciel, une troncature dans une donnée XML peut avoir des conséquences sérieuses.

Un logiciel de confiance peut-il se passer de manipuler des pointeurs ? Les risques d'erreur d'allocation/désallocation dynamique de pointeurs sont parfaitement mesurés[1] dans les systèmes critiques où ils sont simplement interdits. Implanter des structures de données complexes sans utiliser des pointeurs revient à compiler à la main les opérations sur les pointeurs. Cela semble encore plus dangereux. Pour accroître la confiance dans le source, il est conseillé d'utiliser des langages permettant de représenter des données complexes sans manipulation de pointeurs, soit à l'aide de types construits soit à l'aide de classes (à condition bien sûr que la sémantique des traits objet du langage soit bien étudiée).

[1]Ils semblent en revanche sous-estimés dans la plupart des ouvrages d'algorithmique ou de programmation

Un source se construit à l'aide de structures de contrôle et d'appels de fonctions/procédures/méthodes. C'est sur eux que s'articulent les preuves de correction de l'implantation par rapport à la spécification. Ces preuves comprennent bien sûr les preuves de terminaison des différents appels récursifs et boucles du programme. On admet en général que le programme se poursuit correctement après l'achèvement d'une boucle et d'un appel de fonction. Pour les boucles, se pose cependant le problème des sorties prématurées, à l'aide de constructions break ou exit, réminiscentes des goto étiquetés. Il faut assurer que la continuation du programme après une telle sortie s'effectue correctement par rapport à la spécification. Cette démonstration repose directement sur la sémantique du langage et nécessite de manipuler les états d'exécution. Le problème est le même pour les exceptions. Pour toute instruction de levée d'exception, il faut démontrer que soit elle n'est jamais exécutée soit qu'elle est correctement reprise et traitée dans l'environnement d'exécution approprié. Celui-ci dépend de la sémantique donnée à la reprise d'exception et aussi du traitement des continuations. Soit on peut faire confiance sur ce point au langage utilisé soit il faudrait s'abstenir d'utiliser des exceptions ou, dans le cas des boucles, s'abstenir des sorties de boucles prématurées. Mais une telle abstention peut avoir des effets dévastateurs dans la mesure où elle peut amener à compliquer beaucoup le code.

Les différentes structures de contrôle examinées ci-dessus portent sur la partie séquentielle du contrôle. Les contrôles de la concurrence synchrone ou asynchrone, de la distribution ou de la programmation par évènements sont trop complexes pour être abordés dans cette simple note. Leurs propriétés sont fortement liées à l'interaction entre la gestion de l'exécution du langage et le système d'exploitation. Elles reposent donc fortement sur la correction des compilateurs par rapport à la sémantique de ces traits dans le langage. Malheureusement, cette sémantique est rarement suffisamment formalisée pour lever toute ambiguité et permettre un raisonnement.

Toutes ces remarques sur le source amènent aux conclusions suivantes. Il est assez illusoire de vouloir construire un logiciel de confiance en utilisant un langage source dont la sémantique n'est pas définie formellement, soigneusement étudiée et dont le respect par les compilateurs est garanti. En effet, l'effort de démonstration doit alors porter sur le code généré, donc sur une implantation de très bas niveau de la spécification. Même si le langage source présente toutes garanties, il ne faut pas oublier que plus les constructions utilisées sont simples, plus les preuves sont faciles. Par exemple, n'utiliser dans l'implantation que des traits fonctionnels facilite les preuves de correction (la notion d'état d'exécution n'a pas à être introduite) sans pour autant nuire à l'efficacité[?]. L'utilisation de bibliothèques éventuellement externes au langage pose un véritable problème de confiance. Supposons par exemple avoir à implanter un algorithme utilisant des entiers en précision arbitraire. Il semble déraisonnable de développer préalablement une bibliothèque de grands entiers, parce que cela nécessite une compétence pointue dans le domaine et demande de toute façon un certain temps. Force est donc d'utiliser une bibliothèque de grands entiers comme BigNum, GMP, ... et de lui faire un peu confiance. Pour accroître la confiance, on peut inclure des vérifications dans le programme lui-même mais cela peut être impossible (difficile de vérifier qu'un polynôme est irréductible) ou tout simplement trop coûteux.

Les commentaires qui précèdent concernent les différents moyens utilisés pour aboutir à l'exploitation du logiciel (environnement, processeur, code objet, code source). Ils portent surtout sur l'aspect "ne pas trahir l'intention" du programmeur qui souhaite suivre fidèlement la conception du logiciel.

La phase de conception définit l'architecture du logiciel puis celles de ses différents sous-systèmes. Un logiciel ne peut être de confiance qu'après démonstration que l'architecture choisie valide les propriétés requises par la spécification. Ces propriétés sont à vérifier en s'appuyant sur l'interface du logiciel avec l'extérieur, sur les échanges de données entre les sous-systèmes, sur le flôt de contrôle du système et sur les propriétés ds différents composants. Dès que la conception est un peu compliquée (conception objet par exemple), ce travail est difficile car il demande de raisonner sur les liens entre les composants et donc de les décrire de manière tout à fait précise et suffisamment formelle. Mais ce travail est essentiel.

Ce qui ultimement fonde la confiance dans un logiciel est l'assurance qu'il est en tous points conforme à sa spécification. Cela suppose que celle-ci soit complètement décrite, que les propriétés requises soient énoncées de manière non ambigüe et qu'elles soient cohérentes entre elles.

La spécification elle-même se doit de respecter le cahier des charges. L'élaboration de la spécification se fait à partir de documents fournis par une personne du métier concerné, exprimant un besoin sans considération à priori des limites des outils informatiques et sans que ce besoin soit obligatoirement complètement cerné. Etablir la conformité de la spécification par rapport au cahier des charges doit se faire de manière rigoureuse mais ne peut pas être complètement formalisé. Comment formaliser le fait que le logiciel doit être efficace ? qu'il doit offrir une interface homme-machine adaptée aux habitudes d'un métier ?

Pour susciter la confiance dans le logiciel, il est donc important de faire figurer dans le cahier des charges les propriétés essentielles au bon fonctionnement du logiciel, en les décrivant en langue naturelle de manière très méticuleuse. En s'appuyant sur cette description, il est alors possible de modéliser au moins une partie de ces propriétés pour établir formellement que la spécification les respecte (avec par exemple des outils comme AGFL et RCSPL[14], B[1], Coq[10], ...) et pour traquer des imprécisions et des incohérences dans le cahier des charges ou dans l'établissement de la spécification. On peut ensuite s'appuyer sur les énoncés de ces propriétés pour valider la conception et le développement du logiciel.

Ces notes sont issues de notre expérience dans le projet Foc, qui porte sur le développement d'un logiciel de confiance pour le calcul formel. Nous avons quotidiennement à choisir entre une vision idéale de la certification et une vision plus pragmatique, mais qui permet d'avancer en temps raisonnable.

Il s'agit donc de développer un outil de calcul formel de confiance, qui soit efficace et bien adapté aux besoins des ingénieurs de calcul formel. Ce cahier des charges doit être précisé. Un outil peut être destiné soit à un utilisateur non averti des difficultés du calcul formel soit à un utilisateur souhaitant développer ses propres implantations d'algorithmes. Ces deux versions possibles de l'outil s'appuient toutes deux sur une librairie implantant les structures algébriques les plus courantes. Etendre cette librairie revient donc à se comporter comme un utilisateur averti. Le choix est donc fait de construire un outil pour un utilisateur averti et donc d'adopter une vision de compilation plutôt que d'interprétation. Cependant, les choix effectués ne doivent pas handicaper une future utilisation en "calculette".

Pour le projet Foc, la confiance est l'élément primordial du cahier des charges et elle s'obtient en démontrant que les implantations des algorithmes respectent bien leurs propriétés mathématiques, donc que les hypothèses nécessaires à leur application sont bien vérifiées, etc.. Autrement dit, l'implantation ne doit pas trahir les mathématiques. En explicitant cette notion de confiance, on réalise très vite que n'importe quel langage de programmation ne peut convenir. Assurer de telles propriétés est difficile par nature. S'il faut y méler des vérifications sur la production de code objet, la tâche devient impossible. Donc, le cahier des charges impose l'utilisation d'un langage sémantiquement fondé et de compilateurs corrects par rapport à la sémantique. D'autre part, un utilisateur averti doit être en mesure de valider ses implantations. Il faut donc que l'outil lui permette aussi de réaliser des preuves, sans qu'il ait à trop s'éloigner de sa pratique mathématique.

Le cahier des charges doit encore préciser certains attendus sur la librairie. La librairie doit constituer une hiérachie d'unités superposable à la hiérarchie des structures algébriques. Elle doit offrir plusieurs niveaux d'abstraction pour une même notion, autoriser des définitions par défaut de fonctions partageables par toute une famille de structures et des redéfinitions. Un processus de raffinement est nécessaire pour pouvoir définir par exemple une version abstraite de $Z/2Z$ et deux versions concrètes ZI et ZB, implantées respectivement par des entiers et des booléens. De plus, ces deux implantations doivent pouvoir partager les propriétés de $Z/2Z$ mais doivent être distinguées au niveau des opérations (mieux vaut éviter d'additionner un entier et un booléen).

Ce cahier des charges oriente vers une spécification orientée-objet mais il contient quelques points apparemment contradictoires (partage de propriétés mais distinction d'opérations). Pour être sûr que ces contradictions ne sont qu'apparentes, une spécification formelle de la hiérarchie et des outils pour l'étendre a été faite en Coq par S. Boulmé[2, 3]. En même temps, la spécification de la discipline de programmation était établie à partir de la réalisation d'un certain nombre de prototypes assez conséquents[4, 6]. Ce travail de spécification nous a conduit à affiner le cahier des charges en distinguant deux sortes d'unités, les espèces qui décrivent des implantations de structures algébriques pouvant encore être étendues et les collections, obtenues à partir d'espèces complètement développées par un processus d'abstraction des types, garantissant que leur utiisation ne peut pas rompre les invariants de représentation. Nous retrouvons là la distinction entre utilisateurs avertis (qui créent de nouvelles espèces) et utilisateurs qui souhaitent disposer seulement d'un interpréteur de confiance.

Au cours de la phase de conception, nous avons choisi de clairement séparer les aspects propres à la librairie de ceux venant du langage hôte. La hiérachie est conçue sous l'hypothèse que les outils de représentation des données (nombres, listes, tableaux, arbres) sont offerts par le langage hôte. Elle repose sur un mécanisme d'héritage et de paramètrage, conforme à la spécification formelle de S. Boulmé. La sémantique-objet de ce mécanisme, assez semblable à celui d'Objective ML, est en cours d'étude par S. Fechter[?]. Nous avons aussi défini notre propre syntaxe concrète, qui permet de construire de nouvelles espèces à l'aide du jeu de primitives spécifié par S. Boulmé. Les espèces possèdent toutes un champ caractérisant l'ensemble sous-jacent à la structure par un type (à la ML) et un invariant de représentation. Les autres champs peuvent être des déclarations de fonctions ou des énoncés de propriétés ainsi que des définitions de fonctions ou des preuves de propriétés. La syntaxe ne permet aucun écart par rapport à la discipline de programmation. Les preuves sont conçues (travaux de D. Doligez et V. Prévosto) comme des enchaînements de lemmes, vérifiés interactivement. La preuve construit un arbre de preuve soumis à Coq[10] pour vérification.

Le développement côté programmation, conçu et effectué par V. Prévosto[13], comporte une phase d'analyse syntaxique, une phase de typage, une phase d'analyse statique vérifiant que les propriétés de cohérence identifiées par S. Boulmé sont bien respectées, enfin une phase de compilation vers du source Ocaml. En effet, nous avons choisi le langage Ocaml[11] car il est l'un de ceux répondant le mieux aux exigences du cahier des charges (sémantique affirmée et efficacité). Côté preuves, un travail conséquent reste à faire pour concevoir et implanter des outils spécialisés pour l'induction, la réécriture, etc.

Au long de ces travaux, nous avons constamment à réfléchir à la manière la plus appropriée pour donner confiance. Il nous faut par exemple traiter de la partialité. On peut modifier la représentation en utilisant un type `option`. On peut lever une exception si le paramètre effectif n'est pas dans le domaine de définition de la fonction. On peut prouver que le paramètre effectif appartient bien au domaine de la fonction et ne rien tester à l'exécution. Le choix entre ces trois méthodes dépend beaucoup de la fonction et de l'environnement des appels. Il n'est pas sans conséquence sur la lourdeur de preuves et sur l'efficacité des exécutions[8].

Une implantation des entiers utilise les petits entiers de Ocaml. Elle sert par exemple pour coder les degrés des polynômes. Il est par exemple impossible de démontrer que l'addition des entiers de Ocaml (et donc de la machine sous-jacente) est associative. Force est de faire confiance à Ocaml et au processeur et de mettre cette propriété en postulat. En revanche, les débordements sont traités à l'aide d'un invariant de représentation. Une autre implantation des entiers utilise la bibliothèque GMP et fait également le postulat que cette bibliothèque est correcte vis-à-vis des mathématiques. Ces derniers exemples soulèvent le problème de l'évolution des langages et matériels utilisés. Si l'arithmétique du processeur évolue, alors le débordement aura une définition différente et les preuves de validité de l'invariant de représentation seront à reconsidérer. La documentation du sytème doit mettre l'emphase sur ces points.

1. Conclusion

Ces quelques notes avaient pour principal objectif de rassembler quelques éléments de discussion sur la notion de logiciel de confiance. Les difficultés soulevées sont souvent bien identifiées dans le cas particulier des systèmes critiques, avec pour conséquence des restrictions fortes sur les outils de programmation autorisés. Ces restrictions ont pour but de permettre la validation de propriétés de sécurité avec les outils existants. Pour lever ces restrictions, handicapantes lorqu'il s'agit de développer des logiciels sortant du cadre du contrôle-commande, il faut disposer d'outils sémantiques puissants, faciles d'utilisation et d'ingénieurs aptes à les utiliser. Le premier point requiert un renforcement de la recherche en informatique tant fondamentale qu'appliquée. Par exemple, comprendre comment utiliser la déduction modulo[**?**] pour construire les preuves en Foc permettrait d'effectuer des calculs et des récurrences de manière quasi-transparente pour l'utilisateur.Le second point demande de revoir les contenus des enseignements d'informatique. Les enseignements de logique, de sémantique des langages, de compilation doivent acquérir une place primordiale dans les cursus d'informatique, pour servir de fondements à la pratique des ingénieurs informaticiens qui, même s'ils n'ont pas à programmer, utilisent quotidiennement du logiciel. Cela nécessite de développer des méthodes et outils pédagogiques appropriés. Cela nécessite d'abord de convaincre les concepteurs de contenus de cursus.

Références

[1] J.R. Abrial. The B Book, Assigning programs to meanings *Cambridge University Press*, 1996.

[2] S. Boulmé. Spécification d'un environnement dédié à la programmation certifiée de bibliothèques de calcul formel Thèse `http ://www-spi.lip6.fr/~boulme/bin/these.ps.gz.cgi`, Université Paris 6, Décembre 2000.

[3] S. Boulmé. Specifying in Coq inheritance used in computer algebra. Rapport de Recherche `http://www.lip6.fr/reports/lip6.2000.013.html` 013, LIP6, May 2000.

[4] S. Boulmé, T. Hardin, and R. Rioboo. Modules, objets et calcul formel. *Journées Francophones des Langages Applicatifs*, janvier 1999.

[5] S. Boulmé, T. Hardin, D. Hirschkoff, V. Ménissier-Morain, and R. Rioboo. On the way to certify computer algebra systems. In *Calculemus 99 Systems for Integrated Computation and Deduction*, volume 23 of *ENTCS*. Elsevier, 1999.

[6] S. Boulmé, T. Hardin, and R. Rioboo. Polymorphic data types, objects, modules and functors : is it too much ? *Research report*, LIP6, 2000. `http ://www.lip6.fr/reports/lip6.2000.014.html`

[7] Recueil de normes CEI IEC 61508 `http ://www.iec.ch/etech/ed4.2000/fprod-f.htm`

[8] C. Dubois et V. Viguié Donzeau-Gouge A step towards mechanization of partial functions : domains as inductive predicates. *CADE Workshop on partiality*, Lindau, 1998.

[9] S. Fechter. Objective Foc : une sémantique objet pour Foc *Rapport de DEA*, septembre 2001

[10] Coq `http ://coq.inria.fr`

[11] Ocaml `http ://caml.inria.fr`

[12] The COMMON CRITERIA for Information Technology Security Evaluation, `http ://www.commoncriteria.org`.

[13] V. Prévosto. Prototype d'interface utilisateur de la librairie foc. In *Actes des Journées Francophones des Langages Applicatifs*, 2001.

[14] B. Le Trung, M. C. Monégier du Sorbier, B. Soubiès et O. Elsensohn. Logiciels de Sécurité, nouvelle approche pour optimiser la vérification et la validation. Real Time Systems 1996. ISBN 2-87717-054-3 TEKNEA : B.P. 18 – F-31660 BESSIERES `http ://www.surlog.com`

[15] The Precise UML (PUML) group `http ://www.cs.york.ac.uk/puml/`