B : passé, présent, futur.

J.-R. Abrial.

Décembre 2002.

Version 4

B : passé, présent, futur

1 Introduction

Ce texte s'adresse à des lecteurs qui auraient entendu parler de B et qui auraient le désir d'en savoir un peu plus.

Voici, par exemple, quelques questions que l'on a pu entendre ici ou là : Que signifie B ? Que peut-on faire avec B ? Quels sont les principes du langage B ? Quelle approche progressive de la réalisation des systèmes est mise en oeuvre en B ? On parle de construction prouvée avec B : qu'est-ce que cela signifie ? Comment B a-t-il évolué ? Evolue-t-il encore ? Où en est l'Atelier B ? Comment l'usage de B s'intègre-t-il dans un projet industriel opérationnel ? etc.

Nous souhaitons que ceux qui se posent ces questions puissent trouver ici des éléments de réponse et ... qu'ils s'en posent d'autres.

Ce texte n'est pas technique : il ne contient ni formules mathématiques, ni descriptions de langage, ni développements formels. Les éventuels lecteurs théoriciens ne sont donc pas concernés. Mais il ne cède pas pour autant à une facilité, qui ne serait d'ailleurs qu'apparente. Autrement dit, vous aurez besoin au cours de votre lecture d'une certaine concentration d'esprit, car seront exposés ici des concepts et des principes qui requièrent attention et réflexion. Cependant, si vous possédez déjà une certaine sensibilité à la pratique de la réalisation des systèmes complexes (informatiques ou autres) vous ne devriez pas éprouver de difficultés à le comprendre. Pour les lecteurs pressés, nous indiquerons cependant ci-dessous les parties qu'il est possible de sauter.

2 L'évolution de B

Initialement, c'est-à-dire vers la mi-80, B a été conçu comme une *méthode de développement de logiciels prouvés*. Il n'est donc pas étonnant qu'il ait d'abord été utilisé pour la construction de logiciels où la sûreté de fonctionnement était considérée comme vitale. C'est effectivement le cas du projet de métro sans conducteur METEOR réalisé à l'époque par Matra Transport International (maintenant Siemens Transportation System) pour le compte de la RATP. Dans ce cadre-là, B a été utilisé pour s'assurer *par la preuve* que les parties du logiciel qui concernaient directement la sécurité des voyageurs étaient bien totalement sûres.

Peu à peu, nous nous sommes cependant rendus compte que B pouvait aussi être utilisé avec profit beaucoup plus en amont dans le cycle de développement, par exemple dès la phase de rédaction du cahier des charges d'un système, puis ensuite éventuellement dans les phases de spécification générale et de conception.

Notons à ce sujet que B ne requiert pas du tout d'être utilisé de bout en bout dans le cycle de vie d'un système. Il peut, par exemple, n'être utilisé que dans les phases initiales puis passer la main à d'autres approches. Inversement, comme dans le cas de METEOR évoqué ci- dessus, B peut n'être utilisé que dans les dernières phases du cycle de vie.

Mais l'évolution que nous venons d'évoquer ne s'est pas arrêtée là. En effet, B est aujourd'hui mis en oeuvre sur des systèmes opérationnels qui ne sont plus nécéssairement basés sur l'informatique. Ici l'utilisation de B consiste à construire différents modèles du système concerné, dans le fonctionnement duquel on essaie d'y voir ainsi plus clair. Cette approche peut être mise en oeuvre sur des systèmes déjà construits ou au contraire seulement encore au niveau du projet. Nous pensons qu'elle constitue une voie importante de l'utilisation de B. Ces applications plus larges ont été rendues possible grâce à divers développements techniques récents de B (certains sont d'ailleurs encore en cours) qui ont donné naissance à ce qu'on a pris l'habitude d'appeler maintenant le "B-événementiel".

Nous reviendrons, bien entendu, sur tous ces thèmes dans la suite de ce document. Retenons donc bien ceci pour l'instant : au départ, B était une méthode de développement de *logiciels prouvés* qui est aussi devenue au fil du temps une méthode d'analyse et de construction de *systèmes prouvés*.

3 Comment lire cet article ?

Tout au long de cet article, nous allons suivre plus ou moins le schéma historique qui vient d'être esquissé.

Nous partirons (section 4) de la démarche initiale (construire des logiciels sûrs) pour présenter la raison d'être du concept principal de la méthode B, à savoir celui de *modélisation*. Nous expliquerons pourquoi nous devons modéliser un logiciel avant de l'écrire et, bien entendu, nous préciserons ce que modéliser veut dire.

Nous illustrerons ensuite (section 5) notre propos par le *survol de quelques modèles* pris dans différents domaines. On commencera à se rendre compte à ce sujet que le concept de modélisation peut s'appliquer aussi avec profit en dehors de la stricte problématique du développement de logiciels. Et nous observerons à cette occasion qu'un logiciel n'est, bien évidemment, qu'un composant parmi d'autres dans un ensemble plus vaste, qui peut comporter des éléments matériels, des éléments de communication, d'autres logiciels, et aussi des utilisateurs. Nous verrons qu'il est très profitable pour la bonne construction de l'ensemble que chacune des parties ainsi répertoriées soit en définitive modélisée quelle qu'en soit la destinée future (logiciel, matériel, etc.).

Nous reviendrons alors sur des points plus techniques (sections 6 et 7) destinés à montrer comment se présente un modèle dans la pratique et comment sa construction est nécessairement progressive et structurée : les importants concepts de *raffinement*, de *décomposition* et de *généricité* seront introduits et précisés à cette occasion. Les lecteurs pressés pourront sauter ces deux parties.

On verra ensuite (section 8) de quelle façon un modèle B peut s'adapter soit à une *vision locale* comme c'est le cas pour un logiciel, soit à une *vision globale* comme c'est le cas pour un système pris dans son ensemble. Cette vision globale sera développée plus en détail que la première, de façon à en souligner les spécificités. C'est ici que l'on abordera les différents concepts du B-événementiel auquel nous avons fait allusion dans la section précédente. Nous illustrerons cette approche par un exemple.

Nous parcourerons ensuite (section 9) la question de la *preuve* en B : que prouve-t-on ? pourquoi ? comment ? Et nous poursuivrons cette étude (section 10) par un rapide coup d'oeil sur l'Atelier B, qui est l'outil de l'utilisateur de la méthode B. De même que précédemment pour les sections 6 et 7, les lecteurs pressés peuvent sauter ces deux sections.

Enfin nous terminerons (section 11) par une discussion plus générale sur la manière d'intégrer B dans le *contexte industriel* de la construction de systèmes opérationnels.

4 Présentation informelle de B

Dans le B-Book, qui est l'ouvrage théorique de référence sur la méthode B, une introduction ayant pour titre "What is B ?" contient une présentation succincte de cette approche. Il n'est pas question de reprendre ici cette introduction mais plutôt de montrer pourquoi et comment B peut être utilisé pour développer des systèmes robustes dans un contexte industriel.

4.1 A quoi B sert-il ?

Le but de B est très pratique. Son ambition initiale est de fournir à l'industriel concerné une série de techniques et d'outils capables de l'aider à construire des logiciels et, plus généralement comme nous l'avons vu, des systèmes, et ce de la façon la plus sûre possible.

Tout le monde sait bien que la complexité de tels systèmes, de même que le contexte dans lequel ils sont réalisés, les rend très sensibles aux erreurs. Dans ce cadre-là, le but de B est d'accompagner certaines parties du processus de développement, de façon à ce que ces inévitables erreurs produites durant les phases initiales (cahier des charges, spécifications générales), médianes (spécifications détaillées, conception générale), ou finales (codage, intégration, maintenance), que ces erreurs donc soient découvertes le plus tôt possible sur le lieu et dans le temps même de leur production.

Le but qui vient d'être assigné à B a induit un certain usage, qui avait donc été pensé initialement pour le développement de logiciels. Nous allons voir cependant que l'approche qui en est résultée n'est pas spécialement liée à cette problématique-là, qu'elle peut en fait être aussi étendue à d'autres types de développements plus généraux.

4.2 La technique classique de développement est basée sur l'exécution

Dans les développements classiques de logiciels, les erreurs sont (partiellement) découvertes et en principe corrigées après la phase de codage proprement dite. Ceci est réalisé dans la pratique en soumettant le produit final (ou mieux de petites portions de celui-ci) à toute une série de *tests exécutables* supposés couvrir la plus grande majorité des comportements attendus.

Une autre technique de validation, plus récente, appelée *model checking*, a pour but de montrer qu'un logiciel possède bien certaines propriétés jugées intéressantes. Pour cela, on pratique une recherche exhaustive sur tous les chemins d'exécutions attendues.

Une dernière technique enfin, plus élaborée que les précedentes, l'*interprétation abstraite*, consiste, comme son nom l'indique, à abstraire le programme que l'on veut valider en ne lui faisant exécuter que ce qui à trait à certains aspects qui sont précisément l'objet d'une étude particulière.

Comme on le voit, ces techniques sont toutes basées sur une sorte *d'exécution en laboratoire* du programme qu'on veut rendre sûr. Comme nous pensons qu'il est fondamental de découvrir les erreurs commises à tout niveau dès leur production, il est clair que ces techniques basées sur l'exécution sont inadéquates dès lors que cette possibilité d'exécution n'apparaît que dans les toutes dernières phases du développement proprement dit.

4.3 La technique B de développement et de validation est basée sur la preuve

Comme nous venons de le voir, la technologie de découverte et de correction d'erreurs proposée par B ne peut donc pas être fondée sur l'exécution. Elle est basée en fait sur l'élaboration d'un certain nombre de *preuves*, réalisées au fur et à mesure du développement, et effectuées, non pas sur le programme lui-même puisqu'il n'existe pas encore, mais plutôt sur les différents modèles de plus en plus précis que l'on est amené à en élaborer.

A noter d'emblée que cette technique basée sur la construction de modèles prouvés permet d'élargir la vision conceptuelle du logiciel que nous voulons développer à l'ensemble du système dont il fait partie.

Cette approche, relativement courante dans d'autres disciplines de l'ingénieur, est assez nouvelle en informatique. En génie civil, par exemple, on soumet les différents modèles, progressivement

élaborés, d'un projet de pont à toute une série de preuves basées sur la résistance des matériaux et la mécanique des sols. De même en aéronautique, soumet-on les différents modèles, de plus en plus détaillés, d'un projet d'avion à toute une série de preuves basées, entre autres, sur la mécanique du vol.

En informatique et plus généralement dans le développement des systèmes complexes, la mise en oeuvre de cette approche par la preuve pose un certain nombre de questions spécifiques. Qu'entend-on exactement par modèle d'un programme ou même par modèle d'un système ? En quoi le modèle d'un programme se distingue-t-il du programme lui-même ? Comment et par qui est élaboré ce que l'on doit prouver ? Comment ces preuves sont-elles réalisées dans la pratique ? Autant de questions auxquelles nous allons maintenant tenter de répondre.

4.4 Qu'entend-on par modèle ?

Lorsque le critère de validation d'un programme est son exécution, le développeur tend à concevoir ce programme avec cette idée-là en tête. Dès la phase préparatoire à l'écriture de son programme, il raisonne plus ou moins en terme de *données déjà concrètes* (c'est-à-dire informatisables) qui sont destinées à être modifiées lors de l'exécution.

En B, le développeur ne doit donc pas (ne peut donc pas), au moins pour un temps, raisonner en terme d'exécution. Puisque le critère de validation est la preuve, il doit raisonner directement en terme de *propriétés* auxquelles le futur système doit obéir et dont il lui faudra prouver qu'il en est bien ainsi. Il va donc définir le quoi (voire le pourquoi) plutôt que le comment du logiciel qu'il veut réaliser.

Le modèle initial d'un programme est donc constitué essentiellement par *l'énoncé des propriétés qui permettent de le qualifier*. Si ces propriétés sont trop faibles, beaucoup de réalisations seront possibles. Si, par contre, elles sont trop fortes, on n'aura pas beaucoup de choix. Il faudra, dans la pratique, trouver le juste milieu entre ces deux extrêmes. Les modèles que l'on élabore ensuite par *raffinement* du modèle initial (nous reviendrons sur le raffinement à la section 7) doivent bien entendu toujours vérifier (sous une forme plus concrète) les propriétés définies dans le modèle initial.

4.5 Comparaison entre la preuve et le test

Avant de poursuivre plus avant, nous voudrions nous arrêter un moment sur cette comparaison entre la preuve et le test comme méthodes de validation d'un programme. Un test mené sérieusement comprend plusieurs moments : (1) la définition du scénario qui definit ce que l'on veut tester dans le programme, (2) l'élaboration *a priori* du résultat attendu de l'exécution du programme sur le scénario retenu (à noter que cette élaboration pose un problème difficile puisqu'il faut définir ce résultat attendu à partir d'une source qui soit évidemment indépendante du programme lui-même sous peine d'un biais évident, source qui bien souvent n'existe malheureusement pas en dehors de la tête de celui qui fait le test), (4) l'execution du test proprement dit, (5) la comparaison du résultat du test avec ce que l'on attendait a priori. Lorsque cette comparaison est bonne, le test indique que le programme répond correctement au scénario qu'on lui a soumis (si tant est que le résultat attendu ait été lui-même correctement élaboré), il n'indique rien de plus et en tout cas, en aucune façon que le programme est "correct". Lorsque cette comparaison est mauvaise, on peut penser que le programme est erroné (ou aussi, comme nous venons de le voir, que le résultat attendu n'était pas correct) et l'on doit corriger le programme.

Dans le cas de la construction de modèles validés par les preuves qui y sont associées, le processus décrit ci-dessus pour le test est en partie inversé. On se concentre, en effet, *en premier lieu* sur le point (2) ci dessus. Autrement dit, avant de commencer la conception puis l'écriture du programme proprement dit, on définit avec précision les propriétés qui permettront de le qualifier.

A noter que l'on peut s'arrêter relativement longtemps sur cette phase fondamentale (nous y reviendrons dans la suite de ce texte). Ce qui est important ici de bien voir c'est que contrairement au test où les propriétés à tester sont imaginées après que le programme soit effectivement codé, dans le cas de la modélisation, les propriétés que l'on valide *font partie* du modèle. La preuve peut d'ailleurs rentrer en jeu sur ce modèle initial, on peut en effet être amené à prouver qu'il ne contient pas certaines incohérences. Ensuite, tout au long du processus de raffinement qui mène petit à petit à la réalisation finale, on prouve que les propriétés initiales ainsi que celles qui ont pu être incorporées en cours de route, sont bien conservées. La preuve accompagne donc de bout en bout le processus de développement.

Les preuves sont conduites, soit de façon automatique, soit de façon interactive, avec un outil spécialisé, que l'on appelle le prouveur (nous y reviendrons à la section 9). Le résultat de la preuve a des conséquences très intéressantes. L'énoncé à prouver peut en effet rentrer dans l'une des quatre catégories suivantes : (1) l'énoncé est vrai, (2) il est faux, (3) la preuve ne réussit pas mais l'énoncé est néanmoins *probablement prouvable ou réfutable* (on n'a cependant pas réussi avec les moyens à disposition), (4) la preuve ne réussit pas et l'énoncé n'est *probablement ni prouvable ni réfutable*. Noter que la discrimination entre les cas (3) et (4) nécéssite une réflexion et une interprétation.

Les conséquences de ces différentes issues sont intéressantes : dans le cas (1), on continue la suite du processus ; dans le cas (2), on doit revoir le modèle en cours pour le corriger puis le reprouver (on sait, en tout cas, quelle est la propriété qui est mise en défaut) ; dans le cas (3), on peut être amené à revoir le modèle pour le rendre "mieux prouvable", la difficulté apparente de la preuve apparaissant alors comme un signe de la "mauvaise qualité" du modèle (cette utilisation, non prévue à l'origine, de la preuve comme indice de qualité s'est avérée à l'expérience extrêmement intéressante) ; dans le cas (4), le modèle est aussi à revoir, mais il n'est pas, cette fois-ci, erroné, il est plutôt probablement "trop pauvre" (souvent la propriété non prouvable, ou une propriété plus forte, est alors intégrée dans les propriétés invariantes du modèle, ce qui a pour effet de rendre prouvable la preuve auparavant infaisable, mais ce qui, bien entendu, génère aussi d'autres énoncés qu'il va falloir maintenant prouver). Ces différents cas sont donc extrêmement riches d'enseignements car leurs conséquences s'intègrent complétement *dans le processus de développement lui-même*.

En fin de compte, modéliser puis prouver ces modèles n'est pas un but en soi, cela sert avant tout à (se) poser des questions et à révéler des problèmes.

4.6 Généralisation à l'analyse et au développement de systèmes

Dans les deux sections précédentes, on pourrait remplacer les mots "programme" par le mot "système" sans que la pertinence de notre propos n'en souffre vraiment.

C'est exactement ce qui s'est passé en définitive dans l'évolution de l'usage de B. On a petit à petit étendu la pratique de la modélisation des logiciels à celle de la modélisation de systèmes beaucoup plus vastes. Et ceci s'est fait en continuant à procéder de la même façon, c'est-à-dire en construisant des modèles déclinant les propriétés du futur sytème.

A noter que si I'on désirait pousser l'analogie plus loin on devrait aussi se poser la question de savoir par quoi remplacer les mots "test" et "exécution", valables pour un programme. Il semble que dans le cas des systèmes on parle des "essais" et que l'on emploie le mot "fonctionnement" à la place du mot "exécution".

Ce qui est intéressant en tout cas c'est qu'au delà des mots qui sont différents, la problématique reste exactement la même, à savoir que les essais de fonctionnement d'un système ne peuvent s'effectuer que lorsque il est en état de marche, ce qui ne saurait se produire avant la fin du processus de construction. Et pourtant I'on sent là aussi le besoin d'en savoir plus bien avant. La

modélisation en B (éventuellement accompagnée de preuves) est alors le prétexte à une véritable réflexion se déroulant beaucoup plus tôt dans le cycle de développement.

On notera que la réalisation des essais d'un système complet demande des procédures pratiques qui sont nécessairement beaucoup plus lourdes que celles mises en oeuvre pour des tests de logiciels. Se pose alors la question de l'exhaustivité de tels essais. Alors que l'on pouvait (en principe) envisager des tests logiciels véritablement complets, ce n'est évidemment plus le cas pour les systèmes incluant des éléments matériels. Ceux-ci induisent en effet des comportements nouveaux liés à l'usure des pièces mécaniques et au caractère aléatoire des pannes matérielles, tout phénomène qu'il est impossible de déclencher artificiellement et systématiquement au cours d'essais. Comment être sûr alors que le système qu'on veut réaliser est capable de bien réagir lorsque de tels événements imprévisibles se produisent .

La modélisation B permet d'apporter des réponses à ces questions. Grosso modo, elles consistent à modéliser les comportements hostiles envisageables, à proposer et à exprimer dans le modèle B les propriétés des éléments matériels (par exemple la redondance et la synchronisation entre éléments redondants) qui permettent d'en limiter les conséquences malheureuses et à prouver enfin (sur le modèle) qu'il en est bien ainsi. Il ne restera plus alors qu'à vérifier au cours des essais que ces propriétés matérielles sont bien satisfaites dans la pratique sur le système final. On aura donc remplacé une impossible couverture systématique par la vérification de quelques propriétés, vérification qui constitue la *recette* des matériels en question.

5 Quelques exemples de modélisation

Afin de fixer par la pratique les quelques idées que nous venons seulement d'esquisser abstraitement, nous allons maintenant présenter un certain nombre d'exemples visant à montrer quelles sont les propriétés qu'on peut trouver dans les modèles de différentes classes de systèmes, et ce sans définir pour l'instant la façon pratique de les écrire. Nous voulons simplement nous concentrer d'abord sur le sens plutôt que sur la forme.

Comme précédemment, nous commencerons d'abord par des logiciels, puis nous dériverons insensiblement vers des systèmes beaucoup plus vastes.

5.1 Un programme de tri

Lorsqu'on s'occupe de développer un petit programme, par exemple le programme de tri d'un fichier séquentiel, le modèle initial (le quoi) est relativement simple à définir. Les propriétés que nous allons énoncer caractérisent en effet la relation existant entre le résultat final du programme (le fichier trié) et sa donnée initiale (le fichier à trier) : ce résultat final doit comprendre les mêmes éléments que la donnée initiale, éléments qui doivent de plus être classés entre eux suivant un certain critère. On a bien indiqué ici les deux propriétés fondamentales nous permettant de qualifier le futur programme de tri sans dire pour autant comment il trie effectivement. Et il existe, en fait, des centaines de programmes qui réalisent avec plus ou moins d'efficacité une telle spécification.

Ce modèle initial purement abstrait peut maintenant être affiné de façon à être transformé petit à petit en un modèle exécutable, c'est-à-dire en définitive en un programme. Mais, est-il toujours aussi simple de définir un tel modèle initial ?

5.2 Un protocole de transmission

Par exemple, quel est le modèle initial d'un protocole de transmission de données fonctionnant sur un réseau ? Voilà en effet un "programme" dont la difficulté d'appréhension est manifeste :

elle réside essentiellement dans l'aspect distribué de son exécution qui met en oeuvre deux partenaires, voire plus, ne se trouvant pas sur le même site.

La plupart du temps, ces protocoles sont définis avec un luxe de détails par la description du séquencement des différents *messages* que doivent s'échanger les deux partenaires en question. Ces messages concernent généralement aussi bien l'envoi des données proprement dites et leur acquittement que les éléments dits de négociation qui les concernent. Cette négociation peut être initiale ("on va commencer, est-on bien d'accord là- dessus ?") ou terminale ("de mon côté c'est fini, est-ce bien aussi le cas pour vous ?"). Ces messages peuvent enfin concerner les procédures de resynchronisation (mettant en jeu des horloges) en cas de difficultés dans la transmission.

Le lecteur d'une telle description peut certes imaginer comment le protocole fonctionne (quelle est son exécution) mais il ne sait pas au juste quel est son but. Ces messages ne lui disent rien de la véritable mission du protocole. Et, en fin de compte, *il ne sait pas dire pourquoi le séquencement proposé est correct*.

En y réfléchissant bien, on s'aperçoit que ce qui est gênant dans une telle approche c'est la prise en compte de l'aspect distribué du protocole comme une donnée initiale incontournable. Cette prise en compte, notons-le, est essentiellement matérialisée par la description des messages. Dans le modèle abstrait initial, il nous faudra donc d'abord nous affranchir complètement de cette contrainte, qui ne s'introduira que plus tard comme une décision d'implantation parmi d'autres.

Cette démarche d'abstraction étant effectuée, on se sent beaucoup plus à l'aise pour définir le "quoi" du protocole : il s'agit tout simplement de transmettre totalement ou partiellement les éléments d'un fichier. On indique aussi que l'ordre et la continuité des données ne doivent pas être modifiés. On précise enfin que chacun des deux partenaires doit, à la fin de l'exécution, savoir dans quel état l'autre se trouve (s'il sait ou non que le fichier a bien été transmis totalement ou partiellement et, dans ce dernier cas, jusqu'où).

A ce niveau, il est important de définir aussi avec rigueur les hypothèses hostiles qu'on peut être amené à envisager concernant "l'environnement" du protocole. Par exemple, quel est le niveau de fiabilité des canaux de transmission (de façon à pouvoir définir ensuite le degré de redondance qu'on devra adopter dans les transmissions) ? Ou bien, quel est le niveau de confidentialité à élaborer (au moyen de clés par exemple) de façon à protéger éventuellement l'exécution future du protocole de certaines "oreilles indiscrètes" ? A noter qu'il existe toute sorte de normes (européennes notamment) qui traitent de ces questions. On voudrait pouvoir garantir que le modèle d'un certain protocole les suit bien.

A partir du modèle initial ainsi élaboré, la suite du développement va pouvoir alors se dérouler par affinages successifs prouvés pour aboutir à une description fine du protocole en terme de messages comme précisé ci-dessus. Cette description constitue maintenant, comme on le voit, l'état final de notre développement et non plus son point de départ. Et, grâce à la preuve, on est donc maintenant complètement sûr que cette même description réalise bien la mission qui avait été précisée et clarifiée au tout début du développement.

A cette occasion, on commence à voir apparaitre un phénomène qui est très largement répandu, à savoir que les descriptions informelles ou semi-formelles des cahiers des charges de systèmes sont la plupart du temps *beaucoup trop concrètes*. Nous avons maintenant pris l'habitude de presque systématiquement re-écrire ces textes-là, de façon a ce qu'ils ne contiennent plus de "pseudo-réalisations" mais bien, comme nous l'avons déjà signalé plus haut, l'énoncé des différentes propriétés que le futur système doit satisfaire.

Ce deuxième exemple était toujours petit, même si l'aspect distribué du contexte de ce problème apportait tout de même une relative complexité par rapport au cas précédent. On peut se poser

maintenant la question de savoir s'il est effectivement possible de pousser plus loin le facteur d'échelle de façon à envisager la modélisation de programmes vraiment beaucoup plus importants.

5.3 Un système transactionnel

Par exemple, quel est le modèle initial qu'on pourrait être amené à construire pour un système bancaire transactionnel ? Il est clair que dans ce cas, la prétention de tout redévelopper à partir de rien n'a guère de sens. En effet, le marché offre des systèmes généraux de base de données ainsi que des systèmes généraux de construction d'interfaces interactives très élaborés, qu'il n'est pas question de réinventer pour les besoins de la cause. On ne doit donc pas envisager comme dans les cas précédents de formaliser le problème dans sa totalité.

On peut alors se demander où pourrait se situer l'apport de B dans un tel contexte ? En fait, la réponse se trouve précisément entre ces deux pôles, c'est-à-dire entre l'Interface Hommes-Machines (IHM) et la Base de Donnée (BD). C'est généralement là que se trouve le noeud des difficultés. En effet, à l'exécution, l'IHM (excitée par un utilisateur) peut effectuer correctement (de son point de vue) un certain nombre de demandes à la BD (en particulier des demandes de modifications), et, de son côté, la BD peut exécuter techniquement ces demandes de façon parfaitement correcte (également de son propre point de vue). Mais rien ne garantit que, dans ce processus, l'intégrité globale des données ait bien été respectée, et ce pour la bonne raison qu'aucun des deux acteurs précédents n'en a vraiment ni la responsabilité ni les moyens.

Le modèle B initial a donc pour but de définir avec rigueur les propriétés vitales de ces données du point de vue de l'usage qu'en fait l'institution, c'est-à-dire ici la banque. La cohérence de chaque transaction doit alors être modélisée de façon à garantir le respect de ces propriétés. On doit aussi préciser les propriétés de l'indispensable procédure de "backup" à mettre en oeuvre au cas où un incident arriverait au milieu d'une transaction l'empêchant ainsi d'arriver à son terme. Pour toutes ces raisons, il est nécessaire en définitive de bien modéliser la signification précise de toutes ces données et surtout de préciser leurs contraintes de cohérence.

Le système que nous venons d'évoquer était un système à réaction lente (à l'échelle humaine). Mais qu'en est-il des systèmes plus réactifs et largement automatiques tels qu'on les trouve très fréquemment aujourd'hui dans l'industrie ? Comment modéliser de tels systèmes sans en pasticher l'exécution ?

5.4 Un système réactif

Prenons par exemple le cas d'un système de contrôle de poste d'aiguillage ferroviaire et essayons d'envisager les diverses propriétés qui permettent de le qualifier.

D'un point de vue pratique très simplifié, un tel système est commandé par une personne, l'aiguilleur, qui le met en oeuvre en réservant des itinéraires prédéfinis pour les trains qui peuvent traverser simultanément l'espace ferroviaire soumis à son contrôle. Le système doit alors positionner des aiguilles et actionner des feux tout en garantissant la sécurité des trains mais aussi une bonne disponibilité. De plus, ceci doit s'accomplir à l'intérieur de certaines contraintes hostiles de défaillance : par exemple, les appareils de voie ou les circuits détecteurs de présence des trains peuvent tomber en panne, les trains peuvent être en retard ou tout simplement annulés au dernier moment, etc.

Pour mener à bien le développement d'un tel système on va être amené à décliner un certain nombre de propriétés qui appartiennent à diverses catégories bien distinctes : la sécurité d'abord, la disponibilité ensuite, le fonctionnement proprement dit enfin.

Les propriétés de sécurité (plutôt négatives) vont stipuler que le système doit avant tout prévenir les diverses collisions qui pourraient se produire (collision frontale, collision par rattrapage, ou enfin par prise en écharpe) ; de même, le système doit-il garantir qu'il n'y aura jamais de déraillements dus à un changement de position d'une aiguille sous (ou juste devant) un train en mouvement.

Les propriétés de disponibilité (généralement positives) vont par exemple stipuler que l'on désire optimiser l'occupation des portions de voie par les trains simultanément en mouvement dans la zone de contrôle du poste d'aiguillage. Il pourrait en résulter que dès qu'une telle portion n'est plus occupée par un train, elle peut être réutilisée par un autre.

A noter que, *dès ce niveau*, la modélisation peut permettre de discuter puis de valider (par la preuve) les choix et arbitrages délicats qu'il y a toujours lieu de considérer entre sécurité et disponibilité.

Enfin, on trouve les propriétés fonctionnelles définissant par exemple avec précision les règles à mettre en oeuvre pour commander les moteurs d'aiguilles ainsi que les contrôles qui doivent en résulter.

C'est aussi à ce niveau que peuvent s'introduire les premiers résultats mis en évidence par des spécialistes de l'analyse des risques ou des dangers encourus par ce système. Cette introduction est concrétisée par une éventuelle modélisation de certains événements redoutés (par exemple une défaillance d'un capteur de présence de trains) et la façon dont le système s'en défend. On pourra ainsi montrer sur le modèle que, dans l'hypothèse où ils se produisent, de tels événements ne peuvent pas avoir de conséquences désastreuses.

Le lecteur aura évidemment noté que le modèle qu'on développe à ce niveau est déjà beaucoup plus large que celui d'un simple logiciel de contrôle. Il s'agit vraiment ici d'un modèle global décrivant l'ensemble du système : matériel, logiciel et même communication entre les deux. Nous reviendrons ci-dessous sur ce type de modélisation.

Il semble donc qu'on puisse modéliser avec profit des systèmes d'une certaine importance, mais qu'en est-il des systèmes pour lesquels les contraintes de type "temps-réel" sont primordiales ? Comment définir de façon abstraite des propriétés qui semblent appartenir cette fois définitivement à l'aspect exécutoire d'un système ?

5.5 Un système "temps-réel"

Lors de la mission d'un lanceur spatial, le système informatique central doit contrôler certaines transitions extrêmement délicates entre les différentes phases du vol. Par exemple, on peut passer d'une phase au cours de laquelle la fusée est propulsée par des accélérateurs à poudre, les fameux "boosters", à un autre qui correspond au régime de croisière où la propulsion est assurée par le moteur principal.

De part et d'autre d'une telle transition, les régimes de mesures, de navigation et de pilotage du lanceur sont très différents avec, en particulier, des timings périodiques de consultation des capteurs, de calcul des positions des gouvernes, de contrôle du roulis, etc. qui sont très différents. Après la transition proprement dite, correspondant au largage des boosters, il faut donc resynchroniser les différentes tâches périodiques sur une base complètement différente de celle qui avait cours lors de la phase précédente.

De même, la détermination du moment précis où la transition doit avoir lieu (correspondant, par exemple, à l'instant où il faut larguer les boosters) est-elle basée sur différents critères relativement indépendants correspondants à un cahier des charges très précis. On doit observer, d'un côté, la décroissance de l'accélération jusqu'à ce qu'elle atteigne un certain seuil rendant le

largage nécessaire. On doit surveiller, par ailleurs, l'horloge générale du système de façon à ce que ce largage ait effectivement lieu entre une certaine date "au plus tôt" et une autre date "au plus tard".

La détermination précise des caractéristiques des différents régimes et transitions que nous venons d'envisager pour un certain lanceur ne fait pas partie du domaine de compétence de l'informaticien qui participe à l'élaboration du contrôleur logiciel. Il s'agit là plutôt d'une compétence qui appartient à des "gens de métier", par exemple à des automaticiens ou à des spécialistes des moteurs de fusées, qui ont déterminé les propriétés en question après des études particulières, éventuellement des simulations, ou même des essais au banc.

Il est par contre de la responsabilité de l'informaticien d'intégrer les propriétés que nous venons d'envisager à l'intérieur de contraintes plus vastes qui ont à voir avec la réussite globale de la mission du lanceur. Par exemple, celle-ci doit pouvoir survivre à une panne éventuelle de la centrale inertielle nominale survenant en plein milieu de la transition que nous venons de considérer, sachant que le passage à la centrale de secours ne saurait être instantané.

Comme on le voit, les contraintes "temps-réel" présentées ici sont relativement sévères. Cependant, le modèle proposé ne doit pas être une pseudo-implantation, il doit définir avec précision tous ces critères et leurs articulations, il doit montrer la faisabilité de l'ensemble, il doit enfin pouvoir être affiné en un système sain dont le bon fonctionnement sera validé par la preuve.

La liste d'exemples que nous venons très succinctement de parcourir n'est évidemment pas exhaustive : elle pourrait même être poursuivie indéfiniment. Nous avons simplement voulu présenter ici une certaine palette correspondant à une large diversité de systèmes à propos desquels on retrouve néanmoins un même besoin : celui de modéliser avec précision les propriétés fondamentales d'un système avant de le concevoir et a fortiori de le réaliser. La garantie que ces propriétés sont bien respectées par la réalisation finale est fondamentale, elle doit pouvoir être fournie à qui le demande.

6 Forme pratique d'un modèle

Il est temps maintenant de présenter à grands traits la façon dont un modèle peut être écrit pratiquement à l'aide du langage B. Nous ne rentrerons pas cependant ici dans les détails du langage, notre intention est plutôt de décrire les principes de base qui ont dicté sa conception. Comme nous l'avons déjà signalé ci-dessus, *cette partie peut être sautée en première lecture.*

Le domaine commun qui caractérise *tous les exemples* étudiés ci-dessus est le suivant : ces systèmes sont tous, en première approximation, des *systèmes de transitions discrètes* (automates). Autrement dit, leur comportement peut etre caractérisé par une suite d'états stables entrecoupés de sauts brutaux qui font passer d'un état au suivant. Au lieu donc d'envisager des modèles adaptés à tel ou tel catégorie de systèmes, nous allons plutôt favoriser une *formulation unique* correspondant à cette problématique des systèmes de transition. C'est un choix qui relève avant tout de la simplicité.

Un modèle B est donc simplement un texte formel qui décrit les propriétés d'un système de transition. Il comprend essentiellement deux parties : une partie définissant les éléments statiques, c'est à dire l'*état* de ce système et une autre contenant sa dynamique, c'est-à-dire ses *transitions*.

6.1 La partie statique

Cette partie correspond aux déclarations des variables qui caractérisent l'état du système que l'on désire modéliser. En ce sens, elle ressemble un peu à ce qu'on trouve dans les structures analogues des langages de programmation. Mais l'analogie s'arrête là. En effet, les variables qu'on

peut déclarer en B ne sont pas du tout de la même nature que celle qu'on trouve habituellement en informatique. Rappelons en effet que ces variables peuvent caractériser l'état global d'un système, état qui n'a aucune raison d'être nécéssairement représenté par des variables informatiques (scalaires limités en taille, tableaux, fichiers).

On peut ainsi déclarer comme "variables" n'importe quel objet mathématique descriptible dans la Théorie des Ensembles (ensembles, relations, fonctions, nombres etc. construits eux-mêmes sur d'autres ensembles, etc.). En fait, si on y regarde de près, c'est plus ou moins ce que l'on trouve dans des langages semi-formels qui sont largement utilisés actuellement. On a l'impression cependant que les concepteurs de ces langages s'ingénient à inventer de nouvelles notations (la plupart du temps mal conçues) qui correspondent en fait à des notions mathématiques ultra-classiques : pourquoi ne pas prendre directement ces notations-là ? On entend dire qu'il faudrait cacher ces notations par trop mathématiques car elles ne seraient pas compréhensibles par les ingénieurs. L'expérience nous a montré, bien au contraire, que des notations mathématiques précises, dont la signification est parfaitement explicable, sont en fait rapidement maitrisées par les ingénieurs. Après tout, ils ont fait des études scientifiques. C'est plutôt l'inverse qui se passe dans la pratique de ces langages : on a l'impression trompeuse de comprendre des notations mal définies. Combien de discussion oiseuses n'a-t-on pas alors entendu les concernant.

Ces variables, comme on l'a dit plus haut, permettent de représenter l'état du système que nous voulons modéliser. Ces déclarations de variables sont complétées par toute une série d'*invariants* qui explicitent formellement les propriétés qui les relient les unes aux autres. Ces propriétés sont celles que l'état du modèle doit toujours satisfaire et auxquelles nous avons fait allusion dans les exemples décrits dans la section 5. L'écriture de ces propriétés s'effectue à l'aide du langage de la Théorie des Ensembles (appartenance, inclusion, composition des relations ou des fonctions, etc.), langage qui s'avère être particulièrement bien adapté à la construction de modèles formels. Là encore, pourquoi innover en inventant des notations baroques, qu'il faut d'abord apprendre, ensuite retenir et dont la signification n'est pas précise ? On ne comprendrait pas que les physiciens inventent maintenant d'autres notations que celles utilisées depuis plusieurs siècles déjà pour écrire, par exemple, des équations différentielles.

6.2 La partie dynamique

Il existe aussi dans un modèle une partie précisant quelle est la dynamique du système envisagé. Elle contient la description des propriétés (plus ou moins détaillées suivant le niveau d'abstraction) des transitions supportées par le futur système et que le modèle est chargé de spécifier.

Ces transitions décrivent la façon dont le sytème qui est modélisé peut évoluer. Leur forme ressemble un peu aux structures de contrôle de la programmation impérative. Mais, comme ci-dessus pour les variables, l'analogie avec la programmation classique s'arrête cependant là. En règle générale (au moins pour les modèles les plus abstraits), une transition contient des descriptions d'évolutions qui peuvent s'effectuer simultanément et aussi, la plupart du temps, de façon non-déterministe. Un modèle de système, dans ses descriptions les plus abstraites, n'a en effet aucune raison, comme c'est le cas pour un programme, de définir dans le détail la façon d'évoluer du futur système. Il faut laisser la place à un certain flou qui sera levé plus tard lors des raffinements à venir.

7 A propos du raffinement, de la décomposition et de la généricité

Le modèle d'un système complexe ne saurait être construit de façon monolithique, c'est-à-dire en une seule étape. En fait, il est toujours plus ou moins réalisé par *approximations successives*. Ces différentes approximations entretenant entre elles une relation dite "de raffinement". Par ailleurs, qui dit raffinement dit, à l'évidence, complexification. Il est donc très important de pouvoir découper un modèle devenu par trop complexe en composantes plus petites aux fins d'analyses

ultérieures (pour les soumettre en fait à d'autres raffinements et décompositions) indépendantes les unes des autres. Nous allons maintenant expliciter ces notions fondamentales et voir qu'elles correspondent en fait à deux approches bien distinctes. A la fin de cette section, nous présenterons succintement un dernier aspect très prometteur de la structuration des modèles B, à savoir celui de *généricité*. De même que la précédente, *cette partie peut être sautée en première lecture*.

Avant d'aller plus loin nous voudrions cependant attirer ici l'attention du lecteur sur un point très important : ces concepts de raffinement et de décomposition sont absolument centraux en B. Ils sont d'ailleurs centraux, notons-le en passant, dans toutes les disciplines de l'ingénieur. On sait bien depuis longtemps qu'aucune réalisation technique d'importance ne saurait en effet avoir la prétention d'être effectuée d'un seul coup : un architecte dresse les plans des bâtiments qu'il désire construire en les affinant petit à petit au fur et à mesure de l'avancement de son travail ; un avioneur "dessine" progressivement son aéronef sous tous les angles ; un constructeur d'autoroutes dresse des profils d'abord généraux puis de plus en plus détaillés du tracé de ses projets, etc. Comment se fait-il donc que ces concepts de raffinement et de décomposition mettent si longtemps à se manifester dans la pratique en informatique et plus généralement dans la réalisation des systèmes complexes ?

7.1 Raffiner pour ajouter des détails pris dans le cahier des charges

Dans un premier temps, les différentes approximations du futur système sont obtenues en ajoutant petit à petit des propriétés qui sont extraites du cahier des charges initial (éventuellement au préalable ré-écrit comme nous l'avons vu). Chaque ajout de détails à un modèle se traduit donc par un nouveau modèle qui doit rester cohérent avec le précédent. Si c'est bien le cas, on dit que le deuxième modèle raffine le premier.

7.2 Raffiner pour se diriger vers une réalisation finale

Dans un deuxième temps, le raffinement ne concrétise plus l'ajout de détails. Lorsqu'on a épuisé le cahier des charges initial, on obtient une certaine structure qui, en principe, contient la formalisation de toutes les propriétés du futur système. Il est alors temps de songer à une réalisation efficace. Pour cela, on va aussi utiliser la technique du raffinement, mais cette fois-ci dans un but tout à fait différent. Il s'agit maintenant en effet de transformer les différents modèles abstraits auxquels on est arrivé en un ou plusieurs modules concrets (du moins si l'on se dirige vers la réalisation d'un logiciel).

Le raffinement correspond alors à la transformation des "données" utilisées jusqu'ici pour décrire les modèles abstraits (sans aucune considération d'implémentabilité), en des variables plus concrètes qui ont une correspondance informatique dans les langages de programmation classiques.

Les parties "transitions" des modèles sont elles-mêmes raffinées pour arriver petit à petit à des structures physiquement réalisables (le non-déterminisme et la simultanéité des traitements envisagés précédemment peuvent donc être amenés à disparaître peu à peu).

7.3 Décomposer

Quelle que soit l'usage que l'on fasse du raffinement, il est clair qu'au bout d'un certain nombre d'entre eux le modèle qu'on obtient commence à devenir assez volumineux : on peut alors songer à le décomposer en plusieurs sous-modèles relativement indépendants, qui vont pouvoir être eux-mêmes raffinés jusqu'à de prochaines décompositions, etc. On commence en fait à définir ainsi les éléments d'une certaine *architecture*, qui est donc un peu fonction de l'ordre dans lequel on a pris en compte les diverses propriétés du futur système.

7.4 Après les raffinements et les décompositions

A la fin de ce processus, on obtient des modèles qui (dans le cas de la réalisation d'un logiciel) sont presque des programmes. Il ne reste plus en fait qu'à les transformer de façon complètement automatique en de "vrais" programmes écrits dans un langage de programmation classique (il s'agit en général d'une transformation très simple). Il existe, parmi les outils de B, plusieurs traducteurs vers C++ ou ADA, d'autres langages sont prévus (nous y reviendrons).

On peut aussi être amené à implanter directement les modèles obtenus à la fin d'une série de raffinements B sur les "primitives" d'un noyau sous-jacent (par exemple, un moniteur temps-réel, ou une partie de système d'exploitation, ou enfin une base de données implantée sur un SGBD). Il suffit pour cela qu'une certaine "rétro-conception" ait eu lieu, qui a pour but de formaliser les services rendus par les noyaux en question. Cette formalisation prend la forme d'un ou plusieurs modèles abstraits qui vont pouvoir être invoquées directement depuis les derniers modèles obtenus à l'issue des raffinements.

7.5 Généricité

Les deux mécanismes de raffinement et de décomposition que nous venons de présenter avaient pour but d'organiser la construction de modèles de façon à élaborer progressivement la *solution* d'un certain problème. Il existe un troisième mécanisme de structuration qui n'a plus pour but de s'intéresser directement à la découverte de la solution d'un problème donné. Il s'agit plutôt de déterminer que le problème auquel on s'intéresse a déjà reçu une certaine solution qui se situe dans le cadre d'un *problème plus général*. Problème que l'on peut maintenant particulariser (on dit *instancier*) pour le cas qui nous intéresse. Cette particularisation nécéssite bien entendu de faire les preuves d'adéquation des paramètres de l'instanciation.

L'intérêt réside ici dans la réutilisation de la solution déjà élaborée dans ce cas général. Les raffinements et les décompositions déjà effectués sont en effet eux aussi instanciés. Mais le plus intéressant dans cette affaire c'est que cette solution particulière (dont le raffinement peut d'ailleurs être poursuivie) ne nécéssite pas de refaire les preuves déjà effectuées dans le cas général. En effet, ces preuves, qui sont *implicitement instanciées elles aussi* avec les mêmes paramètres que les modèles, ces preuves donc *restent valables dans le cas particulier*. C'est là l'intérêt essentiel de cette approche *générique* qui fait maintenant l'objet d'une recherche très active (avec le LORIA de Nancy) et qui va bientôt s'intégrer dans B.

8 Deux visions d'un modèle

Nous venons de voir quels étaient les éléments constitutifs d'un modèle et aussi ce qu'on entendait sous les concepts de raffinement, de décomposition et de généricité. Mais il n'est peut-être pas encore tout à fait évident de bien comprendre ce qu'un tel modèle représente au juste. C'est ce que nous allons expliciter dans ce qui suit. En fait, on peut considérer un modèle de deux points de vue relativement distincts suivant le type de système qu'il est censée décrire : soit un modèle local, soit un modèle global.

8.1 Modèle local : le B "classique"

Dans ce premier cas, le modèle représente essentiellement un futur logiciel informatique qu'on pourra ensuite utiliser. Les "transitions" que nous avons envisagées plus haut décrivent alors (de façon abstraite dans les premiers modèles et de plus en plus concrète dans les derniers raffinements) les *services* que ce futur logiciel est censé offrir à ses "utilisateurs".

Chaque service, éventuellement paramétré, est en général *pré-conditionné* par une contrainte qui doit obligatoirement être vérifiée avant toute utilisation sous peine de non garantie du bon

fonctionnement de l'ensemble (en particulier de la préservation des propriétés invariantes). A noter que cette vérification n'est pas effectuée à l'exécution comme c'est le cas dans la plupart des systèmes développés de façon classique (vérification qui, en cas de non-conformité, déclenche, comme on le sait, une "exception", qui est un des mécanismes les plus mauvais de la programmation). En fait, elle est validée au moyen d'une *preuve* qui montre que la condition en question est bien vraie, dans tous les cas, sur le "lieu" de chaque appel du service en question.

Au cours des divers raffinements du modèle initial, les services offerts restent identiques quant à leur forme extérieure : mêmes noms, mêmes paramètres éventuels. Le contenu de ces services ainsi que les données du modèle pourront par contre changer de nature de façon parfois très substantielle. A noter que l'utilisateur ne "voit" pas les raffinements, il voit seulement le modèle initial (il a l'illusion que "l'exécution" a lieu à ce niveau-là).

8.2 Modèle global : le B "événementiel"

Dans ce deuxième cas, le modèle représente une entité beaucoup plus importante qu'un logiciel informatique comme c'était le cas précédemment. Il s'agit plutôt de la modélisation d'un système dynamique pris dans son ensemble : le modèle représente alors en fait la *simulation* d'une certaine *réalité observable*.

Modèle de simulation Un tel système peut éventuellement (mais pas nécessairement) comprendre des composants informatiques de même que des composants matériels et éventuellement des éléments de communication entre les deux. Nous nous intéressons donc ici à la modélisation complète d'un ensemble dynamique dans lequel de nombreux "agents" (matériels, logiciels, voire humains) sont à l'oeuvre simultanément. A noter que certains de ces agents peuvent être hostiles : on peut par exemple de cette façon représenter les pannes réelles survenant de façon aléatoire.

Vision événementielle Les "transitions" ne représentent plus du tout comme précédemment les services d'un futur module informatique mais plutôt les *événements* qui rythment la "vie" du système global qu'on veut modéliser. Bien entendu, il n'y a plus de pré-conditions comme précédemment puisqu'il n'est plus question maintenant d'invoquer un service sous certaines conditions seulement. Par contre, chaque événement est la plupart du temps *gardé* par une contrainte qui représente la condition sous laquelle il peut se déclencher. Autrement dit, lorsque cette condition n'est pas remplie, l'événement correspondant ne peut pas arriver, il est suspendu jusqu'à ce que sa garde redevienne vraie. Mais il n'est pas pour autant certain qu'il se déclenchera alors : la garde ne représente en effet qu'une condition nécessaire.

Dans le modèle (et non dans les faits réels, bien entendu), un seul événement au plus peut se produire à un moment donné et l'exécution de l'événement en question est supposée être instantanée. Cette apparente simplification n'en est pas une en réalité car il s'agit seulement d'un problème de granularité du temps, granularité qu'on pourra toujours affiner à volonté au cours des raffinements successifs comme nous allons le voir.

Les gardes de plusieurs événements pouvant être vraies simultanément, il en résulte un certain non-déterminisme, dit externe (entre les événements), par opposition au non-déterminisme, dit interne, qui est, lui, inhérent à l'exécution d'un événement donné.

Systèmes vivaces Lorsque toutes les gardes des événements sont simultanément fausses, le système est évidemment bloqué, il ne peut plus évoluer en aucune façon. Cette situation n'est pas souhaitable en général : nous nous intéressons en effet la plupart du temps à des systèmes qui ne s'arrêtent jamais (on devra alors prouver qu'il en est bien ainsi).

Raffinement de la granularité temporelle d'observation Au cours des raffinements, de nouvelles variables d'état sont introduites mais aussi de nouveaux événements. Ces divers éléments précisent ainsi l'observation statique aussi bien que dynamique qu'on peut être amené à effectuer sur le système. C'est par l'intermédiaire de ces nouveaux événements, que le temps est, pour ainsi dire, lui aussi affiné comme nous l'avons déjà souligné ci-dessus. On peut faire ici une analogie avec ce qui se passe lorsque l'on observe soudain un bouillon de culture au travers d'un microscope : l'observation y est beaucoup plus fine qu'auparavent non seulement spatialement mais aussi dynamiquement : on voit en effet beaucoup plus de "choses" bouger par rapport à ce que l'on pouvait observer à l'oeil nu. Raffiner le modèle d'un système, c'est en quelque sorte traduire ce qu'on peut voir quand on l'observe au travers d'un microscope.

Un événement déjà présent dans l'abstraction peut, quant à lui, voir sa garde renforcée au cours d'un raffinement (rappelons qu'un condition P est dite plus forte qu'un condition Q lorsque P implique Q). On pourrait donc croire à tord que l'événement concret qui est ainsi représenté arrive moins souvent que sa contrepartie plus abstraite. En fait, il n'en est rien : la réalité est toujours la même bien entendu, c'est seulement l'observation qu'on en fait qui est plus fine. L'événement abstrait qui était modélisé représentait seulement une vision grossière de cette réalité. Au cours du raffinement, cette observation devient plus fine, c'est-à-dire que la même réalité est maintenant modélisée à l'aide de plusieurs événements (l'ancien événement abstrait et certains des événements nouvellement introduits). On doit bien vérifier (par la preuve) que ces renforcements des gardes des événements raffinés n'induisent pas davantage de bloquages que dans l'abstraction.

Décomposition d'un modèle global en modèles locaux Au cours des différents raffinements on décompose petit à petit le modèle global en ses différents constituants nettement séparés, par exemple matériel, logiciel et communication. En fait, on construit ainsi l'architecture globale du futur système. A la fin du processus de développement, seule la partie logicielle, s'il y en a une, est effectivement éventuellement poussée jusqu'au bout. A noter que la construction de ces parties logicielles se poursuivent dès lors sous la forme d'une où plusieurs modèles locaux comme il a été décrit ci-dessus. Autrement dit, nous avons effectivement "recollé" une partie du modèle global (celle correspondant au futur logiciel) sur un modèle local, c'est-à-dire en définitive sur un (ou plusieurs) module informatique assurant un certain nombre de services.

La modélisation des parties purement matérielles et communiquantes restantes n'est pas poursuivie plus avant. Elle est cependant très importante en l'état car elle rend explicite les critères sur lesquels vont pouvoir être basés les essais à effectuer sur les divers équipements matériels et de communication qui vont constituer la partie "physique" du futur système global. Ces essais permettent en effet de contrôler que les équipements en question respectent bien les hypothèses qu'on avait supposées être vérifiées pour assurer le fontionnement prouvé et correct du logiciel de contrôle qu'on a effectivement construit. Autrement dit, on va vérifier sur le terrain que la réalité matérielle est bien fidèle à la modélisation mathématique qui en avait été faite a priori.

Propriétés dynamiques des systèmes événementiels A côté des propriétés dites statiques du système (les invariants déjà envisagés plus haut) on peut être amené à considérer d'autres propriétés intéressantes caractérisant la dynamique de tels systèmes événementiels. Il s'agit essentiellement de propriétés, dites temporelles, qui explicitent certaines contraintes portant sur la façon dont le système peut évoluer. Par exemple, on voudra exprimer que telle condition portant sur les données pourra être vraie, sinon en permanence, du moins de temps en temps. Autrement dit, on stipule qu'elle ne devra jamais être toujours fausse. Dans le même ordre d'idées, on voudra exprimer qu'une propriété des données reste vraie lorsqu'elle le devient pour la première fois (à la suite de l'arrivée d'un événement).

8.3 Un petit exemple

Nous allons illustrer succinctement l'exposé précédent du "B-événementiel" afin que le lecteur à qui cette technique serait présentée pour la première fois ici puisse l'appréhender de manière

un peu plus tangible. Pour cela, nous allons reprendre le cas du poste d'aiguillage que nous avons évoqué plus haut (§ 5.4).

On s'attache d'abord à modéliser petit à petit le cadre dans lequel est plongé notre futur système. Par exemple, les grandes catégories d'objets à notre disposition sont clairement les suivantes : les portions de voies, les aiguilles, les itinéraires, les trains.

On définit ensuite les relations statiques qui relient ces objets entre eux. Ces liens définissent essentiellement quelle est la "géométrie" de l'espace ferroviaire placé sous le contrôle d'un poste d'aiguillage. Par exemple, la continuité des voies nous amène à préciser les propriétés de la géométrie des portions correspondantes (bifurcation, croisement, succession simple, etc.). Les liens entre aiguilles et portions de voies sont évidemment liés aux bifurcations. Enfin les itinéraires sont définis par les différentes portions de voies qu'ils contiennent (ainsi que le sens de parcours correspondant), portions qui doivent être compatibles avec la géométrie des voies et respecter certaines contraintes de contiguïté évidentes. Pour chaque itinéraire est aussi précisé le positionnement de chacune des aiguilles concernées, positionnement qui doit, bien sûr, être compatible avec le tracé de l'itinéraire en question.

Puis viennent les relations dynamiques entre objets. Le positionnement effectif des aiguilles (gauche, droite, entrebaillé). Le lien dynamique de réservation entre train et itinéraire avec la contrainte spatio-temporelle évidente : un train au plus par itinéraire à un instant donné. Enfin le lien entre un train et les portions de voie qu'il occupe avec, là aussi, une contrainte de spatio-temporelle évidente : un train au plus par portion de voies à un instant donné.

La sécurité exige bien entendu d'étendre les contraintes que nous venons d'évoquer aux deux invariants de base de notre système, à savoir que deux trains distincts ne peuvent pas réserver deux itinéraires ayant des portions de voies communes (des nuances liées à la disponibilité peuvent cependant être introduites ici), et que les aiguilles d'un itinéraire réservé pour un train sont effectivement positionnées convenablement.

On voit ainsi comment nous pouvons petit à petit établir un lien entre les énoncés de propriétés de sécurité évoquées ci-dessus (§5.4) et leur traductions formalisées. Cette traçabilité est tout à fait fondamentale.

On envisage enfin les définitions d'événements. A ce premier niveau, on n'introduit que quelques événements : la réservation d'un itinéraire pour un train, le repositionnement d'une aiguille, l'avancement d'un train, la libération d'un itinéraire (on peut envisager deux événements ici, par exemple une libération "automatique" liée à la disparition d'un train à sa sortie d'un itinéraire, ou au contraire une libération "manuelle" liée à un train annulé, qui n'est donc pas encore rentré sur son itinéraire). Tous les événements précédents doivent être gardés comme nous l'avons vu. Par exemple, la réservation d'un itinéraire ne peut avoir lieu que si cet itinéraire obéit à certaines contraintes.

Les preuves que l'on est amenées à faire à ce niveau consiste précisément à démontrer que tout événement qui se déclenche sous sa garde ne remet pas en cause les contraintes et les invariants du système. En cas d'impossibilité, on doit soit affaiblir les invariants, soit plutôt renforcer certaines gardes. Ces corrections sont la plupart du temps extrêmement enrichissantes car elles impliquent souvent un certain nombre de réflexions approfondies sur la nature du système.

On peut ensuite raffiner ce premier modèle par toute une série d'étapes en détaillant par exemple les modalités de réservation d'un itinéraire, en introduisant les circuits de voies détecteurs de la présence des trains, les feux, les moteurs d'aiguille, et aussi les communications entre ces équipements et la partie centrale (informatisée) de ce système (partie qui va s'introduire, elle aussi, le moment venu comme un des composants parmi d'autres du futur système).

Au cours de chaque raffinement, on est amené à définir de nouveaux invariants concernant les nouveaux objets envisagés et leurs liens avec les objets déjà formalisés. On est aussi amené à définir de nouveaux événements correspondants à ces nouveaux objets (par exemple le démarrage ou l'arrêt d'un moteur d'aiguille). On introduit aussi des événements "hostiles" correspondant par exemple à l'arrivée de pannes intempestives sur les capteurs, les signaux, les moteurs d'aiguille, etc. Les preuves associées doivent montrer la préservation de ces nouvelles contraintes par les événements et aussi le raffinement correct des événements qui existaient déjà.

Ce travail de modélisation s'achève enfin par la décomposition entre les parties qui resteront matérielles et celles qui deviendront le futur logiciel de contrôle de poste d'aiguillage. A noter que cette approche globale nous conduit naturellement à construire un contrôleur générique, c'est-à-dire un ensemble qui peut s'adapter à la réalisation de *familles de systèmes* plutôt qu'à un seul en particulier.

La description informelle que nous venons d'entreprendre n'a bien entendu aucune prétention d'exhaustivité, de complétude, ni même de cohérence. Elle avait seulement pour but, rappelons-le, de donner un peu de vie à certains concepts abstraits.

8.4 Autres applications du B événementiel

Dans l'exemple précédent, nous avons illustré l'usage que l'on pouvait faire du B événementiel pour le développement complet d'un système réactif. En fait, cette approche peut aussi servir dans de nombreux cas qui n'avaient pas été du tout envisagés initialement. A noter que ces nouvelles ouvertures en sont à l'heure actuelle au niveau des études de cas.

Par exemple, on utilise maintenant le B événementiel pour faire la synthèse d'*algorithmes séquentiels* "compliqués", c'est à dire contenant, par exemple, plusieurs boucles imbriqées ainsi que de nombreux séquencements. En fait, le développement progressif de ce type d'algorithmes à partir d'*événements indépendants* s'avère beaucoup plus simple à maitriser et à prouver que par une approche classique. Toutes les preuves sont en effet menées sur les événements, ce qui est considérablement plus léger que sur l'algorithme lui-même. Les événements ne sont en effet assemblés (de façon purement syntaxique) pour former l'algorithme final qu'à la fin du processus.

Cette usage du B événementiel a été étendu sans difficulté au développement d'algorithmes dont certaines composantes sont *distribuées* sur plusieurs sites alors que d'autres restent sequentielles sur un même site. C'est typiquement ce que l'on trouve dans le cas des protocoles dont l'exécution est répartie sur un réseau. Nous avons pu ainsi prouver totalement certaines parties de protocoles complexes IEEE 1394 en collaboration avec le LORIA de Nancy.

Nous avons aussi utilisé le B événementiel pour développer des *circuits*. D'après les expériences que nous avons tentées, nous pensons que, là aussi, l'approche est prometteuse.

Un autre domaine, qui semble aussi intéressant et prometteur, est celui de l'analyse formelle prouvée des *risques de défaillance*.

9 La preuve

Au cours des sections précédentes, nous avons souvent parlé de preuves. Il est temps maintenant de préciser ce que l'on entend exactement par là. Nous allons comme précédemment tenter de répondre à un certain nombre de questions relatives, cette fois-ci, à cette problématique de la preuve. *On pourra sauter cette partie en première lecture*.

9.1 Que prouve-t-on ?

La première question est évidemment la suivante : que prouve-t-on exactement quand on effectue un développement B ? La réponse à cette question est très simple. Nous avons mis en avant tout au long de ce texte la recherche et la formalisation des propriétés du système qu'on désirait construire. A noter que ces propriétés sont de nature très diverse : elles peuvent être fort générales et abstraites au début du développement puis devenir de plus en plus précises au fur et à mesure qu'on se rapproche de la fin du processus. Ce qu'on doit prouver est donc parfaitement bien défini, à savoir que le système qu'on a finalement construit possède bien les propriétés qu'on attendait de lui.

9.2 Quand prouve-t-on ?

La deuxième question qui vient alors à l'esprit est la suivante : quand ces preuves doivent-elles être effectuées ? En première approximation, on serait tenté de répondre à cette question en proposant que cela soit fait sur le système final (puisque c'est lui qui compte en définitive) et non sur ses ébauches intermédiaires qui sont encore instables (pas entièrement déterminées). C'est essentiellement l'approche qui avait été retenue en son temps pour la "preuve de programmes". On sait ce qu'il en est advenu. L'expérience a montré qu'elle était tout simplement impraticable. En effet le programme final s'avère être un "objet" déjà beaucoup trop compliqué pour être appréhendé d'un seul coup. En conséquence, sa preuve est, elle aussi, intrinsèquement compliquée et donc très difficile à réaliser (même à la main).

Petit à petit, l'idée a donc fait son chemin que la preuve d'un programme (et plus généralement d'un système) devait s'effectuer *à l'intérieur même du processus de sa construction*.

En fait, nous avons déjà signalé plusieurs fois ici que la collecte des propriétés d'un système était effectuée tout au long du développement. Il en résulte donc, à la différence des tests qui ne sont nécessairement conduits qu'à la fin du développement (c'est-à-dire trop tard), que la preuve est pratiquée, elle aussi, tout au long du développement ; elle accompagne celui-ci.

9.3 Preuves de conservation d'invariant

Cette approche a immédiatement induit une certaine pratique de la preuve. Dès qu'une propriété apparaît dans le développement, essentiellement sous la forme d'un invariant, la preuve qu'il en est bien ainsi (c'est-à-dire que la propriété en question est bien conservée par les "transitions" du modèle ou du raffinement), est effectuée immédiatement. Il s'agit donc d'une preuve de conservation d'invariant.

9.4 Preuves de raffinement correct

Une fois prouvées, il est fondamental que les propriétés que nous avons retenues à un certain niveau de développement ne soient pas, par la suite, invalidées accidentellement dans les niveaux plus concrets. Pour s'assurer que ce danger est écarté, on pratique une autre sorte de preuves consistant à montrer que chaque raffinement est bien "correct" par rapport à l'abstraction qui le précède. Sans rentrer dans aucun détail par trop technique, on peut dire que si un raffinement est prouvé correct alors toutes les propriétés de l'abstraction qui le précède sont effectivement conservées. Il s'agit là d'une caractéristique à la fois du genre de propriétés qu'on prouve (la conservation d'invariant) et du mode de raffinement qu'on pratique. Noter que parmi les conditions à prouver il en est une qui concerne le fait que le modèle raffiné ne se bloque pas plus souvent que sa version abstraite.

9.5 Autres preuves

On doit aussi prouver un certain nombre de conditions particulières qui assurent qu'une décomposition est correcte. De meme l'énoncé des propriétés dynamiques (temporelles) des modèles événementiels requierent-elles des preuves très spécifiques.

9.6 Fabrication des énoncés de preuve

Il est important de noter ici que *le praticien n'a pas à écrire lui-même le détail de ce qui doit être prouvé*, ce qui serait extrêmement fastidieux et, bien entendu, sujet à erreurs. Un outil spécifique, *le générateur d'obligations de preuves* (nous y reviendrons dans la section suivante consacrée à l'Atelier B) analyse les modèles concernés et, en application de la "Théorie B" exposée dans le B-Book et maintenant étendue, génère les différentes conditions qu'il est nécessaire de prouver pour assurer qu'ils sont corrects.

Ensuite, entre en jeu un autre outil, le prouveur, dont le rôle est d'effectuer plus ou moins automatiquement les preuves des énoncés générés par l'outil précédent (nous y reviendrons aussi dans la section suivante).

10 L'Atelier B

La mise en oeuvre pratique d'une approche formelle comme B sur des projets industriels d'envergure est impensable sans l'aide d'un outil performant. Cet outil existe, c'est l'Atelier B. Dans cette section, notre intention est d'en présenter les caractéristiques les plus marquantes. *Cette partie peut aussi être sautée en première lecture.*

L'Atelier B a été développé de façon continue depuis maintenant presque dix ans dans un contexte industriel nettement séparé, il faut bien le souligner, de celui de la recherche dont les origines pour B remontent au début des années 80. Divers industriels ont été impliqués dans cette réalisation. Il s'est agi d'Alstom Transport pour commencer, de Steria par la suite avec un financement substantiel venant de la RATP et de la SNCF, puis maintenant de Clearsy, filiale commune de Steria et de Teamlog. La société Matra Transport International (maintenant Siemens Transportation System) a, de son côté, beaucoup participé à la validation de l'Atelier B et a fourni certains développements propres tout à fait significatifs (dans le domaine du prouveur en particulier). L'ensemble a finalement trouvé toute sa mesure industrielle lors de son utilisation pour la réalisation de la partie sécuritaire des différents logiciels du métro sans conducteur METEOR, projet mené à bien par la RATP et réalisé par Matra Transport International. Il a été aussi utilisé avec succès dans plusieurs réalisations significatives chez Alstom Transport.

L'originalité de l'Atelier B réside essentiellement dans les trois composants principaux auxquels nous avons déjà fait allusion plus haut, à savoir le générateur d'obligations de preuves, le prouveur, et finalement les divers traducteurs. Nous allons maintenant passer rapidement en revue ces différents outils.

Le générateur d'obligations de preuve est l'outil capable de générer à partir d'une ou plusieurs modèles abstraits les diverses conditions de validation qui les rendent "corrects" au sens que nous avons très succinctement présenté ci-dessus. Le B-Book contient le corpus théorique qui constitue en quelque sorte la spécification de cet outil.

Le prouveur est l'outil indispensable capable d'effectuer, la plupart du temps de façon automatique, les preuves des énoncés générés par l'outil précédent. Il permet aussi à un utilisateur d'intervenir de façon interactive sur la preuve automatique pour lui donner quelquefois un solide "coup de pouce". Cet utilisateur peut enfin augmenter la "puissance de preuve" du prouveur automatique en définissant lui-même, si besoin est, un certain nombre d'extensions spécifiques (prouvées elles aussi par l'outil).

Les traducteurs enfin, plus classiques, ont pour rôle de transformer le dernier niveau de raffinements de modèles B, qui sont très voisins de programmes impératifs habituels, en des codes classiques.

A noter que même s'ils ont déjà clairement atteints aujourd'hui le stade adulte, ces trois composants restent l'objet d'améliorations possibles. Le générateur d'obligations de preuves pourrait accompagner chacun des énoncés qu'il génère de références explicites sur les parties de textes B qui lui ont donné naissance. Ceci faciliterait le travail du développeur dans sa recherche éventuelle des failles correspondantes. Le prouveur interactif possède à l'heure actuelle une panoplie extrêmement riche de commandes explicites : elles pourraient cependant être plus facilement mises en oeuvre au moyen d'une interface réellement ouverte. Enfin des traducteurs nouveaux pourraient être développés, traducteurs produisant par exemple soit du code spécialisé destiné à tourner sur des cartes à puces, soit même du code VHDL destiné à être implanté ensuite directement sur des "chips". Noter que tous ces dévelopements sont déjà pour partie engagés dans le cadre de divers contrats de recherche avec la participation de diverses équipes universitaires (Grenoble, Nancy).

En dehors des trois outils dont nous venons de parler l'Atelier B comprend des éléments plus classiques qu'on trouve habituellement dans les phases amont d'un compilateur (analyseur lexical, analyseur syntaxique, vérificateur de type). Il contient aussi des éléments bien répertoriés qu'on trouve ailleurs, à savoir un outil de gestion de projets, un système d'accès multiple sophistiqué nécessité par le travail en équipe sur de grands projets, la gestion harmonieuse de nombreuses formes de fichiers, un moniteur de session capable de relancer les tâches qu'il faut au moment où il faut, enfin divers outils de navigation et d'édition.

Pour fixer quelques ordres de grandeur, la partie du logiciel du métro METEOR réalisée grâce à l'Atelier B, comprend grosso modo 100 000 lignes de code ADA (générées de façon entièrement automatique). Ce développement a engendré de l'ordre de 30 000 obligations de preuves dont plus de 90% ont été prouvées automatiquement par le prouveur (après, il est vrai, une certaine extension ad-hoc de ce dernier). Les 2 500 preuves restantes ont nécessité un travail de l'ordre de quelques hommes-mois avec le prouveur interactif. Le bilan global établi par Matra Transport International est positif puisque, en contre-partie de ce coût de preuves interactives, le coût intégral des tests de bas niveau (toujours très chers pour ce genre de programmes pour lesquels on désire une large couverture) a pu être économisé. En effet, en accord avec la RATP, le constructeur a été autorisé à ne pas effectuer ces tests-là, la conviction étant acquise que la preuve peut les remplacer avec une couverture qui est ici, bien sûr, totale par définition. Jusqu'à présent, d'après nos connaissances, le logiciel développé avec B a donné entière satisfaction au constructeur et à son client.

11 Intervenir dans un projet industriel avec B

Après avoir parcouru à grands traits dans les sections précédentes certains aspects techniques du développement B, on va se poser maintenant la question de savoir dans quelle mesure cette approche est effectivement applicable à des projets industriels opérationnels. Comme précédemment, nous allons d'abord étudier cette intégration dans le cadre de la mission première de B (logiciels). Puis nous analyserons aussi la possibilité d'utiliser B dans un contexte plus large que celui qui était prévu initialement (systèmes).

Posons tout d'abord un premier principe : il n'y a pas de projets qui soient mieux adaptés que d'autres à l'utilisation d'une méthode telle que B. Dire le contraire dans un autre contexte (de génie civil, par exemple), reviendrait à prétendre qu'il y a des projets de ponts qui se prêtent mieux que d'autres à l'utilisation de la résistance des matériaux.

Comme nous l'avons déjà signalé au début de ce texte, B peut être introduit à différentes étapes de l'avancement d'un projet et pas forcément sur la totalité de celui-ci. Initialement, B était plutôt destiné à être utilisé sur la partie proprement informatique (plutôt même sur une portion de cette partie dans laquelle la sûreté de fonctionnement était considérée comme vitale), et ce une fois que la définition et même l'architecture du système étaient relativement figées :

c'est effectivement ce qui s'est passé dans le cas du projet METEOR.

Nous pensons que l'industriel qui voudrait aujourd'hui "faire du B" doit au préalable engager une réflexion approfondie sur l'influence et l'intégration des conséquences de cette décision sur le *processus de développement* qu'il a mis en place jusque là. S'il est satisfait de ce processus de développement, il n'a aucune raison de se lancer dans B. Par contre, s'il est amené à le mettre en cause (faiblesse des études préliminaires, poids financier excessif des tests, mauvais fonctionnement de certains projets antérieurs), alors il peut être adéquat de se lancer dans l'utilisation de B. Nous pensons en définitive que le coût des dépenses initiale que l'on doit supporter pour se lancer dans B réside beaucoup plus dans l'effort d'adaptation du processus de développement que dans l'indispensable formation du personnel.

Ainsi, le pilotage d'un projet utilisant B, de même que son profil de financement, doivent être revus par rapport à ce qu'on a l'habitude de constater dans les développements classiques. En effet, avec B, les coûts sont clairement déplacés de la fin du cycle vers son début. Nous pensons qu'il s'agit là d'une tendance extrêmement positive car nous sommes persuadés qu'une des causes principales de la mauvaise qualité des systèmes complexes réside principalement dans le manque de réflexion initiale. Avec B, les dépenses de tests, de codage proprement dit et même de maintenance sont très largement diminuées. Par contre, celles relatives à la rédaction du cahier des charges, à l'architecture générale et à la conception sont, elles, nettement augmentées. Enfin un nouveau chapitre apparaît, celui relatif à la preuve. Ces raisons font qu'il n'est pas évident de se lancer dans l'utilisation de B pour le développement d'un projet industriel car cela implique un certain nombre de décisions qui vont à l'encontre de la pratique habituelle. Utiliser B dans un projet industriel est donc en fin de compte essentiellement une *décision stratégique beaucoup plus qu'une décision purement technique*. C'est pourquoi cette décision doit nécessairement être prise à un niveau assez élevé dans la hiérarchie.

Comme nous l'avons déjà dit plusieurs fois au cours de ce texte, nous nous sommes rendus compte que B pouvait être utilisé avec profit beaucoup plus en amont dans le cycle de développement. Cela signifierait donc qu'on pourrait par exemple mettre en oeuvre cette approche dès la phase de rédaction (en français) du cahier des charges d'un système.

Une question qui peut alors venir à l'esprit à la suite de cette éventualité d'utiliser B lors des phases initiales du processus de développement est la suivante. Il semble exister une incompatibilité flagrante, de nature irréductible, entre, d'une part, "faire du B", c'est-à- dire utiliser un formalisme et une technique de preuve, et, d'autre part, écrire "en français" un cahier des charges qui, en tout état de cause, devrait normalement exister avant toute tentative d'approfondissement formel. Comment résoudre cette contradiction ? Quelle sorte de B faisons-nous donc à ce niveau-là ?

La réponse à ces questions réside dans une sorte de dialectique : "faire du B" pour préparer l'écriture du cahier des charges et partir du cahier des charges pour "faire du B". En fait, l'idée consiste à utiliser l'approche B en quelque sorte derrière le rideau pour modéliser ce qui peut l'être à un moment donné (on peut toujours partir de quelques propriétés qui vont de soi), de façon à pouvoir poser ensuite quelques bonnes questions dont les réponses par les gens de métier vont permettre de préciser petit à petit les propriétés du système. Ces propriétés vont enfin être rédigées très soigneusement en français et inscrite dans le cahier des charges.

Les gens de métier ne voient pas nécessairement directement les modèles B qu'on a ainsi développés ; ils ne les appréhendent en fait qu'indirectement au travers de la traduction en français qu'on a été amené à en faire et des questions qu'on a pu leur poser au préalable. C'est donc un peu le monde à l'envers : une certaine formalisation précède la rédaction du cahier des charges en français, lequel se trouve ainsi d'emblée extrêmement structuré et précis car la formalisation en B et même certaines preuves sont situées tout juste derrière.

La culture B commence à se répandre à un rythme régulier au sein de l'enseignement supérieur (départements informatiques des universités, IUT, écoles d'ingénieurs) : témoins les jeunes diplômés ayant appris B et qui se présentent maintenant comme tels sur le marché du travail. Les formations supérieures de qualité qui se multiplient dans ce domaine en sont évidemment la cause. Nous pensons que l'industriel qui voudrait se lancer dans cette approche peut maintenant compter sur ces jeunes. Mais, en tout état de cause, il lui faudra mettre en place une équipe permanente très solide basée sur quelques éléments (3 ou 4 personnes) de grande valeur, le noyau B, secondée au gré de la charge par un second niveau de moindre qualification. Ce type de formule a déjà donné de très bons résultats dans la pratique.

12 Conclusion

Comme on a pu le constater à la lecture de ce texte, B n'est pas une technologie figée. Il a bien évolué au cours de son existence depuis plus de dix ans et il évoluera certainement encore dans l'avenir. Que son appréhension par le monde industriel prenne un certain temps est en définitive plutôt rassurant. Cela signifie peut-être que B n'est pas, après tout, une de ces technologie qui flambent pour finir par s'éteindre un beau jour, brusquement remplacée par la mode suivante.