B-Testing-Tool : génération de tests aux limites à partir de spécifications B

Bruno Legeard — **Fabien Peureux**

Laboratoire d'Informatique de Besançon
Université de Franche-comté
16, route de Gray
25 030 Besançon cedex
{legeard, peureux}@lifc.univ-fcomte.fr

RÉSUMÉ. *Nous présentons dans cet article un environnement pour la génération de tests fonctionnels aux limites à partir de spécifications B. Cette méthode repose sur la réécriture de spécifications B en systèmes de contraintes équivalents et la partition du domaine des variables d'état de la spécification. Cette partition est basée sur un calcul de bornes effectué par un solveur spécifique utilisant des techniques de Programmation Logique avec Contraintes ensemblistes. Ce solveur permet ensuite la constitution d'états limites du système et leur intégration dans des séquences de test. La vérification des résultats des jeux de test et la déclaration du verdict s'effectuent au moyen d'une observation indirecte : l'observation du comportement d'opérations activables. Les résultats d'une étude de cas industriel, concernant l'application de la génération de jeux de test à la norme GSM 11-11 (carte à puces), sont présentés et discutés.*

ABSTRACT. *This paper presents a test case generation method based on an original approach using formal methods. This method constists in translating the formal specifications of the system to be tested into an equivalent constraint system. A domain partition of the state variables of the specifications is performed through a computation of limit values by a specific solver using Constraint Logic Programming with sets. This specific solver is next used to generate paths to sequence the test patterns. Finally, The formal specifications are used as an oracle by using it to determinate the expected output for a given input. The results of an industrial case-study, about GSM standard (smart card), is presented and discussed.*

MOTS-CLÉS : *génération de tests à partir de spécifications formelles, test aux limites, notation B, Programmation Logique à Contraintes, contraintes ensemblistes, norme GSM*

KEYWORDS: *specification-based testing, boundary testing, B notation, Constraint Logic Programming, set constraint solving, GSM standard*

2e soumission à *Technique et science informatiques – Test de logiciel*, le 31-03-02.

1. Introduction

Dans les domaines applicatifs du logiciel critique – i.e. transports, carte à puce, aéronautique... – plus de 50 % de l'effort de développement est consacré à la validation du produit logiciel. L'équipe de validation est clairement séparée de l'équipe de développement. La conception des tests fonctionnels est, pour l'essentiel, réalisée «à la main», par analyse du document informel de spécifications techniques des besoins. L'approche présentée ici vise à fournir aux ingénieurs validation un outil d'aide à la conception des tests fonctionnels. L'objectif est, à la fois, de diminuer l'effort de conception des tests et d'améliorer la couverture et la justification de ceux-ci. En effet, la conception manuelle des tests fonctionnels, telle que pratiquée dans l'industrie, consiste en une démarche empirique d'analyse du cahier des charges de l'application, qui laisse une très grande place à la subjectivité de l'ingénieur validation. La génération de jeux de test à partir de modèles formels permet ainsi de rationaliser la stratégie de test fonctionnel. La méthode B est une méthode formelle qui couvre toutes les étapes du cycle de développement du logiciel, de la spécification jusqu'à la génération de code exécutable. Elle s'effectue par une succession d'étapes de raffinement dont la correction est garantie par la génération d'obligations de preuves. L'idée principale des travaux présentés dans cet article est d'utiliser le premier niveau de modélisation B – appelé modèle abstrait dans la suite du texte – pour générer des séquences de test aux limites destinées à la validation fonctionnelle du logiciel correspondant au modèle abstrait (le logiciel n´étant pas le fruit du cycle traditionnel de raffinement de la méthode B).

L'approche présentée, intégrée dans un environnement appelé B-TESTING-TOOL, s'effectue en trois étapes principales : réaliser la spécification de l'application à tester, extraire les valeurs limites des variables d'état de la spécification, à partir desquelles des séquences de test sont calculées. Cette méthode repose sur la réécriture des spécifications B en systèmes de contraintes dont l'évaluation est assurée par un solveur spécifique utilisant des techniques de Programmation Logique avec Contraintes ensemblistes (CLPS-B) [LEG 00]. Ce solveur, dédié à la notation logico-ensembliste, constitue la base de la méthode de génération de jeux de test. D'une part, il est utilisé pour partitionner le domaine de chaque variable d'état de la spécification pour en extraire les valeurs limites et construire les états limites du système. D'autre part, il permet de générer des séquences de test en intégrant ces états limites dans des séquences d'opérations. Cette intégration est effectuée par parcours du graphe d'atteignabilité des états de la spécification, évitant au préalable la construction exhaustive de ce graphe. Finalement, la spécification est utilisée comme oracle pour déterminer les valeurs de sortie en fonction des valeurs d'entrée : le verdict est prononcé par comparaison des données de sortie des opérations de la spécification et celles de l'implantation. Cette observation indirecte des états à travers leurs paramètres de sortie permet ainsi d'éviter les problèmes relatifs à l'inobservabilité de certaines variables internes au système. Nous présentons les résultats obtenus sur une application de logiciel Carte à Puces : la norme GSM 11-11 [EUR99]. L'étude réalisée montre une couverture supérieure à 80 % des tests générés en comparaison d'un ensemble de tests

éprouvés préexistants, et une réduction de 30 % de l'effort de conception des tests, modélisation formelle incluse [LEG 01].

Notre présentation s'articule de la manière suivante. La définition d'un cadre théorique pour le test à partir de modèles abstraits B est présentée en section 2. La section 3 décrit la méthode de calcul des états limites et leur utilisation dans la construction de séquences de test. Le solveur CLPS-B, permettant d'animer la spécification B pour générer les séquences de test, est alors présenté. En section 4, le fragment de la norme GSM 11-11, sur laquelle l'étude de cas a porté, est présenté. La spécification correspondante (annexe A), formalisée à l'aide de la notation B, est alors introduite. La méthode de génération de tests est appliquée au fragment de la norme GSM 11-11 en section 5. La section 6 rappelle les résultats globaux et énonce les principaux enseignements de l'étude de cas industriel. La situation de notre méthode par rapport aux travaux connexes est décrite en section 7. La section 8 clôt la présentation en récapitulant les contributions et en présentant les futures orientations de nos travaux.

2. Cadre formel du test à partir d'un modèle B

Après un bref rappel concernant la méthode B, nous présentons les restrictions faites sur cette méthode et sa notation pour introduire un cadre formel pour le test à partir de machines B. En effet, dans notre approche, seule le niveau machine abstraite est utilisé, sans composition de machines, ni raffinement.

2.1. *La méthode B*

La méthode B est une méthode formelle qui couvre toutes les étapes du cycle de développement du logiciel, de la spécification jusqu'à la génération de code exécutable, par une succession d'étapes de raffinement dont la correction est garantie par la génération et la vérification d'obligations de preuves. La théorie des ensembles et le langage des substitutions généralisées constituent les fondements théoriques du langage de la méthode B. Une spécification B est décrite par un ensemble de machines abstraites. Une machine abstraite se décompose en spécifications de données, propriétés (description statique) et spécifications d'opérations (description dynamique). Plus précisément, les données sont spécifiées avec des objets mathématiques (ensembles, fonctions, relations...) tandis que les variables d'état de la machine sont contraintes par une conjonction de propriétés statiques appelées invariant. Les spécifications des opérations, exprimées dans un pseudo-code non-exécutable, sont décrites par des substitutions généralisées qui sont une extension du langage des commandes gardées [DIJ 75]. Les opérations permettent de faire évoluer les variables dans différents états, chacun devant nécessairement vérifier l'invariant. Une opération est ainsi constituée par une signature (avec éventuellement des paramètres d'entrée et de sortie), des pré-conditions que les paramètres et variables d'état doivent vérifier pour que l'opération soit activable, et des substitutions généralisées qui définissent les relations pré-conditions/post-conditions sur les variables d'état (elles peuvent être

non-déterministes et parallèles). Dans cette partie, faute de place, nous considérons que le lecteur possède une connaissance de la méthode B et nous ne présentons pas davantage ses concepts et sa notation. Si ce n'est pas le cas, toutes ces notions sont développées dans le B-BOOK [ABR 96].

Avant d'introduire, dans la prochaine section, un cadre formel pour le test à partir de modèle formalisé à l'aide de la notation B, certaines restrictions concernant cette méthode et notation doivent être posées. En effet, ce cadre formel s'appuie sur la notation B et le concept de machine abstraite, mais se situe en marge du processus traditionnel de la méthode B. En effet, notre approche consiste à formaliser les spécifications techniques des besoins en utilisant le niveau le plus abstrait de la modélisation B - i.e. sans raffinement d'implantation -, afin d'utiliser ce modèle abstrait pour générer des tests fonctionnels pour la validation d'une implantation obtenue en dehors du processus de développement formel de la méthode B. Il est à noter que le modèle abstrait, pris en entrée de la génération, décrit de façon précise et détaillée les éléments de l'expression des besoins. La construction de ce modèle fait l'objet d'un développement itératif avec l'équipe des ingénieurs validation sur la base des spécifications techniques de besoins, et d'une validation par animation du modèle [PY 00], mais sans utiliser le raffinement «à la B», simplement par écriture de différentes versions jusqu'à stabilisation. Dans les études de cas que nous avons réalisées (gestion de processus [LEG 00], protocole *T=1* [JUL 98] ou l'application GSM discutée dans ce papier), cette approche de formalisation de spécifications conduit à la production d'une machine B unique modélisant en elle-même la totalité des éléments d'expression des besoins. C'est une pratique assez différente des cas d'utilisation complets de la méthode B, tel le projet METEOR [BEH 98], dans lesquels un nombre important de machines B est produit. Cependant, cette pratique convient bien à une approche de modélisation en vue de génération de tests.

Finalement, afin d'assurer la testabilité du programme, des hypothèse minimales $\mathfrak{H}_{min}$ sont posées sur le modèle abstrait et le programme implanté du système à tester :

- la machine abstraite B, spécifiant le programme à tester, satisfait les obligations de preuves générées par la méthode B (cette hypothèse permet d'éliminer toute anomalie comportementale lors de l'animation de la spécification),

- on est capable de placer le programme dans un état équivalent à l'état initial de la spécification B (en effet, tous les tests générés s'exécutent depuis cet état),

- à chaque opération du programme, on sait faire correspondre une opération de la machine abstraite,

- à chaque type du programme, on sait faire correspondre, au moyen d'une fonction d'abstraction, un type simple de la machine abstraite.

2.2. *Construction du test*

Le comportement d'une machine abstraite B peut être décrit en terme de suite d'opérations dont la première est activée depuis l'état initial de la machine. Nous appellerons une telle suite d'opérations une trace. Nous définissons, de manière symbolique, l'activation d'une opération de la machine B et la suite d'activations qui constitue une trace. Nous ne considérons dans les définitions suivantes que la spécification B du système et non son implantation en langage de programmation. Ces définitions s'appuient sur les travaux de S.Alnet [ALN 96] et S.Behnia [BEH 00].

Une opération de la machine B est caractérisée par son nom et éventuellement des paramètres d'entrée et des paramètres de sortie. L'invocation d'une opération peut donc s'écrire sous la forme d'une substitution $\mathfrak{I}_i$ telle que :

$$\mathfrak{I}_i = [u_{i,1}...u_{i,k_i} := e_{i,1}...e_{i,k_i}]; [op_i]; [s_{i,1}...s_{i,k'_i} := w_{i,1}...w_{i,k'_i}]$$

où

- op_i désigne l'opération invoquée de la machine B,

- les $u_{i,j}$ sont les éventuels paramètres formels d'entrée de l'opération op_i, i.e. les noms des variables d'entrée de op_i,

- les $w_{i,j}$ sont les éventuels paramètres formels de sortie de l'opération op_i,

- les $e_{i,j}$ sont les variables représentant les éventuelles entrées soumises à l'opération op_i,

- les $s_{i,j}$ sont les variables correspondant aux éventuelles valeurs retournées par l'opération op_i.

On peut maintenant définir une trace en utilisant intégralement le langage des substitutions.

Définition 1 *Une trace* $\mathfrak{t}$*, consistant en la mise en séquence de plusieurs invocations d'opérations, peut s'écrire :*

$$\mathfrak{t} = [\mathfrak{I}_0; \mathfrak{I}_1; \mathfrak{I}_2; ...; \mathfrak{I}_n]$$

où $\mathfrak{I}_0$ *désigne l'initialisation de la machine* B.

A partir de la définition 1, on introduit l'ensemble des traces finies d'une machine B donnée, c'est à dire l'ensemble des séquences activables à partir de l'état initial, que l'on note $\mathfrak{T}$.

Définition 2 *Etant donnée une machine* B *appelée* SP*, l'ensemble des traces finies sur* SP *se note :*

$$\mathfrak{T}_{SP} = \{\mathfrak{t} \,|\, \exists n > 0, \mathfrak{t} = [\mathfrak{I}_0; ...; \mathfrak{I}_n]\}$$

Si la pré-condition d'une opération n'est pas vérifiée, la substitution ne peut établir aucune post-condition. Les traces comportant de tel cas ne sont d'aucun intérêt puisqu'il est impossible de déterminer l'état de la machine. Pour éviter de telles

traces, nous restreignons l'ensemble des traces finies sur une machine SP à son sous-ensemble composé uniquement de traces licites, c'est à dire de traces dont les invocations d'opérations se terminent ($trm(S)$ traduit la terminaison de la substitution généralisée S).

Définition 3 *Etant donnée une machine B appelée SP, l'ensemble des traces licites sur SP se note :*

$$\mathfrak{TL}_{SP} = \{\mathfrak{t} \in \mathfrak{T}_{SP} \mid trm(\mathfrak{t})\}$$

Les valeurs modifiées par une trace $\mathfrak{t}$ sont données, pour chaque variable x, sous la forme d'un prédicat $prd_x(\mathfrak{t})$.

Définition 4 *Etant données une substitution S et une variable x, prd_x définit le transformateur de prédicat où x' est une nouvelle variable correspondant à l'effet de la substitution S sur x :*

$$prd_x(S) \widehat{=} \neg[S](x' \neq x)$$

Il est ainsi possible, au niveau de la spécification, de superviser le comportement de chacune des variables d'état de la machine B au cours d'une trace, et par conséquent de calculer leur valeur après l'activation de chaque opération de la trace.

Définition 5 *Etant donnés une machine B appelée SP, V l'ensemble des variables d'état de cette machine, une trace licite sur SP notée $\mathfrak{t} = [\mathfrak{I}_0; ...; \mathfrak{I}_n]$ $(n > 0)$, l'état E_i composé des valeurs des variables de V après l'invocation $\mathfrak{I}_i$ $(0 \leq i \leq n)$ est défini par :*

$$E_i = prd_V([\mathfrak{I}_0; ...; \mathfrak{I}_i])$$

D'après cette dernière définition, l'état E_0 est égal aux substitutions effectuées lors de l'initialisation des variables d'état de la machine B concernée (c'est-à-dire aux substitutions déclarées dans la clause INITIALISATION de la spécification). Pour une machine donnée, il est le même quelque soit la trace considérée. On peut dès lors construire l'ensemble des états traversés par la machine B au cours d'une trace.

Définition 6 *Etant donnés une machine B appelée SP, V l'ensemble des variables d'état de cette machine, une trace licite sur SP notée $\mathfrak{t} = [\mathfrak{I}_0; ...; \mathfrak{I}_n]$ $(n > 0)$, l'ensemble $\mathfrak{E}_\mathfrak{t}$ des états présentant la valeur des variables de V au cours de la trace $\mathfrak{t}$ est défini par :*

$$\mathfrak{E}_\mathfrak{t} = \{E_0, E_1, ..., E_n\}$$

Nous avons donc défini un test comme étant une trace licite $\mathfrak{t}$ à laquelle correspond un ensemble d'états $\mathfrak{E}_\mathfrak{t}$ présentant la valeur de chaque variable d'état à la suite de chaque invocation $\mathfrak{I}_i$ de $\mathfrak{t}$. Le jeu de tests exhaustif pour une machine donnée est constitué par l'ensemble de toutes les traces licites que l'on peut construire à l'aide des opérations définies dans cette machine. Le prochain paragraphe a pour but de présenter la méthode de soumission du test et d'aborder, par conséquent, la procédure (appelée *oracle*) permettant la déclaration d'un verdict.

2.3. *Oracle*

Le cadre de travail étant établi, il convient de mettre en œuvre une technique permettant d'affirmer le succès ou l'échec du jeu de tests soumis au programme. Pour ce faire, nous disposons de trois types d'objet : un programme écrit dans un langage de programmation donné, un modèle formel abstrait de ce programme écrit avec la notation B et un ensemble de séquences de test (traces licites). Sous les hypothèses $\mathfrak{H}_{min}$, il est possible de soumettre au programme implanté une séquence d'opérations, correspondant à une trace $\mathfrak{t}$ générée à partir de la spécification B. Mais avant de pouvoir exploiter les résultats retournés par le programme implanté lors de la soumission, il faut considérer le cas où la soumission elle-même échoue.

Proposition 1 *Le pilote de test, chargé de soumettre la séquence d'opérations au programme à tester, est capable de détecter l'arrêt définitif du programme, autrement dit un «crash» (par l'inactivité ou l'arrêt de processus), et une boucle infinie de l'exécution du programme (par une estimation du temps de calcul par exemple). Il s'agit dans ces deux cas d'un échec de la soumission et naturellement du test.*

Ainsi, si la soumission d'une trace $\mathfrak{t} = [\mathfrak{I}_0; ...; \mathfrak{I}_n]$ termine, nous disposons pour chacune de ses invocations $\mathfrak{I}_i$ d'une suite $S_{i,1}...S_{i,k'_i}$ de valeurs retournées par le programme. Nous disons qu'une telle soumission est en succès si et seulement si :

$$\forall\, i = 0, ..., n \text{ et } \forall\, j = 1, ..., k'_i$$
$$[s_{i,j} := Abstr(S_{i,j})]\, prd_{s_{i,j}}([\mathfrak{I}_0; ...; \mathfrak{I}_i])$$

où $Abstr$ est la fonction d'abstraction liée à cette soumission

Ainsi, la soumission du jeu de tests exhaustif à un programme est en succès si toutes les traces licites, sur la machine abstraite associée, sont en succès. Dans le cas contraire, elle est en échec.

2.4. *Validité et non-biais*

Définition 7 *Un contexte de test est défini par un ensemble d'hypothèses $\mathfrak{H}$ sur le programme P à tester, un ensemble de tests $\mathfrak{T}$ pour ce programme, et un oracle permettant la déclaration d'un verdict lors de la soumission de $\mathfrak{T}$ à P.*

Définition 8 *Un contexte de test est valide si, sous ses hypothèses $\mathfrak{H}$, il n'accepte aucun programme incorrect.*

Définition 9 *Un contexte de test est non-biaisé si, sous ses hypothèses $\mathfrak{H}$, il ne rejette aucun programme correct.*

Du fait des spécificités de la méthode B, nous pouvons affirmer que le contexte de test, constitué par les hypothèses $\mathfrak{H}_{min}$, le jeu de test exhaustif d'une machine abstraite et la procédure de soumission, est valide et non-biaisé [ALN 96].

3. Sélection et génération des traces de test

La sélection des traces de test est basée sur un calcul d'états limites de la spécification, c'est-à-dire d'états dans lesquels la valeur d'au moins une variable d'état est un extremum d'un de ses sous-domaines d'expression. Le calcul de ces extremums (appelés valeurs limites) est effectué par partitionnement des domaines des variables d'état en sous-domaines, et par le calcul des extremums de ces sous-domaines. Le test consiste à activer chaque comportement des opérations après avoir mis le système dans un état limite (section 3.2).

La méthode de génération des tests s'effectue ainsi en deux étapes. Dans un premier temps, les valeurs limites des variables d'état sont extraites de la spécification et permettent la construction d'états limites du système (section 3.3). Dans un second temps, sur la base des états limites, les traces de test sont générées par parcours du graphe d'atteignabilité des états du système (section 3.4). L'ensemble du processus est assuré par un solveur de contraintes ensemblistes dédié à la notation B appelé CLPS-B [AMB 96] [BOU 02] que nous présentons en section 3.1.

3.1. *Le solveur CLPS-B*

L'évaluation d'expressions B et le parcours du graphe d'états constituent une problématique spécifique pour la résolution de contraintes ensemblistes. En effet, les variables sont construites incrémentalement par les substitutions depuis l'état initial. Les approches de résolution de contraintes ensemblistes fondées sur une réduction d'intervalles ensemblistes n'effectuent pas une propagation de contraintes suffisamment forte. Cela nous a donc amené à développer le solveur CLPS-B fondé sur une représentation explicite des domaines par union d'ensembles clos de variables et constantes [LEG 00]. Dans ce paradigme, les états obtenus en évaluant la spécification correspondent à un système de contraintes et non plus à un ensemble de valeurs. Ce système de contraintes définit la notion d'état contraint qui se substitue aux états valués qui sont classiquement manipulés. L'utilisation de la PLC [BOU 00] dans ce contexte a permis de conserver l'indéterminisme de la spécification et de réduire le nombre d'états générés. Ainsi, le non déterminisme exprimé par l'expression B : $ANY\ \ xx\ \ WHERE\ \ xx \in Y$ est préservé par la contrainte ensembliste $xx \in Y$. La substitution n'est plus calculée pour une valeur particulière de Y mais pour une variable xx ayant comme domaine Y.

Un animateur, basé sur l'évaluation contrainte et le solveur CLPS-B, a été mis en œuvre [PY 00]. Cette application manipule donc des états contraints. Lors de l'animation, à l'activation d'opérations, un nouvel état contraint est calculé par l'évaluation des pré-conditions et des substitutions. Enfin, la vérification de l'invariant est effectuée. Plus de 80 % des opérateurs ensemblistes du langage B sont traités dans CLPS-B. Les primitives sur les entiers ont également été implémentées pour permettre d'exprimer des propriétés sur les cardinaux des ensembles.

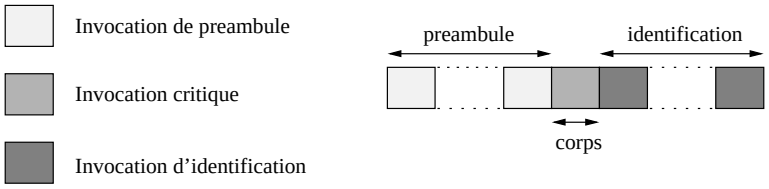

Figure 1. *Constitution d'une trace*

Cet outil est successivement utilisé pour transformer la spécification en un système de contraintes équivalent, calculer les états limites de la spécification, et générer, par animation de la spécification, les traces à soumettre.

3.2. *Critères de sélection et constitution des traces*

Afin de sélectionner les traces à soumettre, notre approche consiste à tester les opérations décrites dans la spécification lorsque le système se trouve dans un *état limite* du système. Nous déterminons les états limites par extraction des valeurs limites des variables du modèle, puis par animation pendant laquelle la valeur des variables d'état de la machine est connue à travers l'ensemble $\mathfrak{E}_t$ (définition 6).

Soient un test représenté par une trace licite $[\mathfrak{I}_0; ...; \mathfrak{I}_l; ...; \mathfrak{I}_n]$ et E_l un état limite de la spécification tels que la sous-trace $[\mathfrak{I}_0; ...; \mathfrak{I}_l]$ permette d'amener le système dans l'état limite E_l. Dans la suite de l'article, cette sous-trace, permettant de positionner le système à l'état limite, est appelé *préambule du test*. L'opération invoquée par $\mathfrak{I}_{l+1}$ devient l'opération testée. Elle est appelée *invocation critique* et constitue le *corps du test*. Finalement, la sous-trace $[\mathfrak{I}_{l+2}; ...; \mathfrak{I}_n]$ est appelée *identification du test* et ses opérations *opérations d'observation*. Conjointement aux valeurs des paramètres de sortie de l'opération du corps du test, les valeurs de sortie des opérations d'observation permettent d'établir le verdict : les résultats obtenus par animation de la spécification sont ceux attendus lors de l'exécution de la même séquence d'opérations sur le système implanté. Une trace de test est ainsi constituée d'un préambule, d'un corps et d'une identification[1] (figure 1).

Deux problématiques se posent alors : d'une part, quels critères utiliser pour déterminer les états limites et comment les calculer, et d'autre part, comment, à partir d'un état limite, générer la séquence d'invocations constituant le test ?

3.3. *Calcul des états limites*

Le calcul des traces de test repose sur la caractérisation de la notion d'état limite. Dans un premier temps, les informations concernant le domaine et les valeurs parti-

1. ce vocabulaire correspond à la norme ISO9646 [ISO]

culières de chacune des variables d'état de la spécification sont extraites du modèle abstrait B. Elles permettent, par propagation de contraintes, de partitionner leur domaine en une union de sous-domaines. Dans un second temps, les valeurs limites, en fait les extremums de chaque sous-domaine, sont calculées et utilisées pour construire les états limites.

Définition 10 *Une valeur limite d'une variable d'état est une valeur extremum d'un des sous-domaines d'expression de cette même variable.*

Définition 11 *Un état limite est un état dont au moins une variable est assignée à une de ses valeurs limites.*

3.3.1. *Partitionnement du domaine des variables d'état*

NOTE. – Les exemples, utilisés dans cette section pour illustrer la méthode de calcul des états limites, sont directement inspirés de la spécification du fragment de la norme GSM 11-11 présentée en prochaine section et disponible en annexe A. Ces exemples sont néanmoins assez généraux pour être compréhensibles sans avoir connaissance de la spécification dans son intégralité.

Dans une spécification B, les expressions de propriété des variables apparaissent dans les clauses PROPERTIES, DEFINITION, INVARIANT, INITIALISATION, dans les pré-conditions et dans le corps des opérations (notamment par l'intermédiaire de structures telles que «PRE...THEN», «IF...THEN...ELSE», «ANY...WHERE»...). L'exploitation de ces prédicats permet de décomposer le domaine de chaque variable en plusieurs sous-domaines disjoints. Par exemple, si une opération contient une substitution conditionnelle «IF $cond_x$ THEN S_1 ELSE S_2» où une variable x apparaît dans la condition $cond_x$, deux sous-domaines de x sont créés : le premier contient toutes les valeurs de x vérifiant $cond_x$ et le second toutes les valeurs de x ne vérifiant pas $cond_x$. Chaque proposition conditionnelle de ce type donne lieu à un tel partitionnement. Le domaine de chaque variable d'état de la spécification est ainsi ré-écrit en une union généralisée de plusieurs sous-domaines. On appelle un tel domaine *P-Domaine*.

Définition 12 *Soient x une variable d'état de la spécification, dont le domaine est D^x, et F l'ensemble des formules de la spécification, dans lesquelles x est la seule variable d'état à apparaître. La résolution de toutes les formules de F entraîne, par propagation de contraintes, une partition du domaine D_x en une union de n sous-domaines disjoints d_i^x (i variant de 0 à n) telle que :*

$$P\text{-}Dom(x) = \bigcup_{i=0}^{n} d_i^x$$

Exemple 1 *Etant donnés la variable $counter\_chv$ de la spécification présentée en annexe A, et son domaine respectif (défini dans la clause* INVARIANT*) $\{0, 1, 2, 3\}$, la*

formule «IF $counter\_chv = 1$ THEN ...», présente dans l'opération VERIFY_CHV, entraîne la partition suivante :

$$\left.\begin{array}{l} counter\_chv \in \{0,1,2,3\} \\ counter\_chv = 1 \end{array}\right\} \quad counter\_chv \in \{1\} \cup \{0,2,3\}$$

Exemple 2 *Etant donnés la variable $current\_file$ de la spécification, et son domaine respectif $\{\emptyset, \{ef\_iccid\}, \{ef\_lp\}, \{ef\_imsi\}, \{ef\_ad\}\}$, la formule «IF $current\_file = \emptyset$ THEN ...», présente dans l'opération READ_BINARY, entraîne la partition : $current\_file \in \{\emptyset\} \cup \{\{ef\_iccid\}, \{ef\_lp\}, \{ef\_imsi\}, \{ef\_ad\}\}$.*

Les exemples 1 et 2 présentent la constitution d'un P-Domaine à partir de formules ne faisant intervenir qu'une seule variable. La prise en compte de toutes les formules, constituées d'une seule variable d'état, engendre la création d'un P-Domaine propre par variable d'état. Parallèlement, chaque formule, construite à l'aide de plusieurs variables différentes liées entre elles, permet de créer, pour les variables correspondantes, des P-Domaines spécifiques n'ayant aucun sens s'ils sont considérés séparément (en effet, on entend par "liées entre elles" le fait que la valeur d'une variable influe sur les valeurs possibles des autres variables). Chaque formule de ce type engendre la constitution d'un ensemble de P-Domaines spécifiques (en fait un P-Domaine par variable) qu'on nomme P-Domaines associés. Une variable a donc un P-Domaine propre et éventuellement un ou plusieurs P-Domaines associés.

Définition 13 *Soient les variables d'état $x_0, x_1, ..., x_m$ liées dans une formule issue de la spécification. La résolution de cette formule fait apparaître des sous-domaines particuliers pour ces variables. Ce calcul, analogue à celui permettant de construire le P-Domaine propre (sinon qu'il porte sur plusieurs variables d'état en même temps), entraîne alors la formation d'un P-Domaine associé telle que*

$$P\text{-}Dom(x_0, x_1, ..., x_m) = \bigcup_{i=0}^{n} (\bigwedge_{j=0}^{m} x_j \in d_i^{x_j})$$

Exemple 3 *Une formule conditionnelle de l'opération READ_BINARY, de la spécification présentée en annexe A, engendre la création de P-Domaines associés : la formule «$permission\_session[PERMISSION\_READ[current\_file]] = \{true\}$» lie la valeur des deux variables $permission\_session$ et $current\_file$. Cette formule est ré-écrite en deux sous-formules qui sont traitées de manière analogue aux exemples 1 et 2 : les sous-formules $PERMISSION\_READ[current\_file] = X \wedge permission\_session[X] = \{true\}$ entraînent la création de P-Domaines associés. Chacun des sous-domaines disjoints est déterminé en fonction de la valeur de la variable X (tableau 1).*

3.3.2. *Calcul des valeurs limites*

On distingue deux types de valeurs limites : les valeurs limites simples, issues du P-Domaine propre de chaque variable, et les valeurs limites multiples, issues des éventuels P-Domaines associés.

Valeur de X	Sous-domaines du P-Domaine associé
ALWays	$current\_file \in \{\{ef\_lp\}\} \wedge (always, true) \in permission\_session$
	$current\_file \in \{\{ef\_lp\}\} \wedge (always, true) \notin permission\_session$
CHV	$current\_file \in \{\{ef\_imsi\}\} \wedge (chv, true) \in permission\_session$
	$current\_file \in \{\{ef\_imsi\}\} \wedge (chv, true) \notin permission\_session$
NEVer	$current\_file \in \{\{ef\_iccid\}\} \wedge (never, true) \in permission\_session$
	$current\_file \in \{\{ef\_iccid\}\} \wedge (never, true) \notin permission\_session$
ADM	$current\_file \in \{\{ef\_ad\}\} \wedge (adm, true) \in permission\_session$
	$current\_file \in \{\{ef\_ad\}\} \wedge (adm, true) \notin permission\_session$

Tableau 1. *Sous-domaines du P-Domaine associé issu du traitement de la formule «*$permission\_session[PERMISSION\_READ[current\_file]] = \{true\}$*» de l'opération READ_BINARY*

Le calcul des valeurs limites simples s'appuie sur le P-Domaine propre de chaque variable d'état ayant un domaine énuméré. Ainsi, pour une variable x donnée, l'ensemble de ses valeurs limites $\mathfrak{L}_x$ correspond à l'ensemble des extremums de chacun des sous-domaines numériques ou à l'ensemble composé d'une valeur, choisie aléatoirement, de chaque sous-domaine non numérique de son P-Domaine propre.

Définition 14 *Soit x une variable d'état de la spécification dont le P-Domaine est égal à $\bigcup_{i=0}^{n} d_i^x$, l'ensemble de ses valeurs limites $\mathcal{L}_x$ est défini par :*

$$\mathcal{L}_x = \bigcup_{i=0}^{n} (min(d_i^x) \cup max(d_i^x)) \quad \textit{si la variable } x \textit{ est numérique}$$
$$\mathcal{L}_x = \{V_0, V_1, ..., V_n\} \quad \textit{avec } V_i \in d_i^x \textit{ sinon}$$

Exemple 4 *En considérant les P-Domaines des exemples 1 et 2, les ensembles des valeurs limites de chacune des variables d'état $counter\_chv$ et $current\_file$ sont respectivement définis par les ensembles $\mathfrak{L}_{counter\_chv} = \{0, 1, 3\}$ et $\mathfrak{L}_{current\_file} = \{\emptyset, V_1\}$ où $V_1 \in \{\{ef\_iccid\}, \{ef\_lp\}, \{ef\_imsi\}, \{ef\_ad\}\}$.*

Les P-domaines associés, par un calcul de borne analogue à celui présenté dans l'exemple 4, donnent lieu à la création de valeurs limites spécifiques appelées multiples. Elles apparaissent sous la forme d'une conjonction de plusieurs valeurs limites.

Définition 15 *Soient les variables d'état $x_0, x_1, ..., x_m$ de la spécification présentes dans le P-Domaine associé $\bigcup_{i=0}^{n} (\bigwedge_{j=0}^{m} x_j \in d_i^{x_j})$, l'ensemble des valeurs limites multiples $\mathcal{L}_{x_0,..,x_m}$ est défini par :*

$$\mathcal{L}_{x_0,..,x_m} = \bigcup_{i=0}^{n} (\bigwedge_{j=0}^{m} (x_j = min(d_i^{x_j}) \vee x_j = max(d_i^{x_j})))$$

si les variables $x_0, x_1, ..., x_m$ *sont numériques*

$$\mathcal{L}_{x_0,..,x_m} = \bigcup_{i=0}^{n} (\bigwedge_{j=0}^{m} (x_j = V_{j,i}))$$

avec $V_{j,i} \in d_i^{x_j}$ *sinon*

Exemple 5 *Considérons le P-Domaine associé dont les sous-domaines sont présentés dans le tableau 1. Chaque sous-domaine donne lieu au calcul d'une valeur limite multiple. Par exemple, le sous-domaine* $current\_file \in \{\{ef\_lp\}\} \wedge (always, true) \notin permission\_session$ *génère la valeur limite multiple* $current\_file = \{ef\_lp\} \wedge (always, false) \in permission\_session$.

NOTE. – Lors du calcul d'un P-Domaine associé, les contraintes de certaines formules ne permettent pas de faire apparaître des sous-domaines particuliers pour les variables. Ces contraintes sont alors retardées et réveillées à l'instanciation d'une des deux variables lors du calcul des valeurs limites multiples. Ainsi, si le domaine des variables x_0 et x_1 est $\{0, 1, 2, 3\}$, la contrainte $x_0 < x_1$ engendre les valeurs limites $\mathcal{L}_{x_0,x_1} = \{x_0 = 0 \wedge x_1 = 1,\ x_0 = 0 \wedge x_1 = 3,\ x_0 = 2 \wedge x_1 = 3\}$

3.4. *Génération des traces*

Nous présentons maintenant comment, à partir du graphe contraint d'atteignabilité issu du modèle abstrait B et de l'ensemble des valeurs limites, les traces de test sont construites. Une trace est toujours représentée par une séquence d'invocations. Chaque invocation est définie par l'activation d'une opération de la spécification. La valeur des arguments d'entrée et de sortie de l'opération, et la valeur des variables d'états suite à son activation sont toujours renseignées. La technique utilisée pour construire les traces de test peut être vue comme un parcours du graphe d'atteignabilité dont les nœuds représentent les états contraints construits pendant la résolution, et les transitions symbolisent l'activation d'une opération. Ce parcours, réalisé par animation du modèle, s'effectue depuis l'état initial jusqu'à un état satisfaisant la ou les valeurs limites recherchées, depuis lequel le corps et l'identification du test sont réalisés.

Cette technique a pour avantage d'éviter au préalable la construction du graphe d'atteignabilité dans sa totalité : les invocations sont calculées à la volée lors de la recherche de l'état limite dans le graphe (afin d'améliorer la convergence de la procédure de résolution, une heuristique classique de type Best-First [PRE 01] est mise en œuvre), et lors du calcul du corps et de l'identification du test. Pour certaines invocations, des variables d'état et valeurs d'entrée d'opérations peuvent être contraintes et non instanciées. Ces valeurs peuvent donc rester indéterminées en fin de résolution. Une phase d'instanciation aléatoire (satisfaisant les contraintes) est alors effectuée afin de générer une trace de test composée d'invocations closes.

Certaines valeurs limites peuvent ne pas être atteignables. Dans ce cadre, un test partiel d'atteignabilité est utilisé. Il est basé sur la vérification des propriétés invariantes de la spécification et une génération de chemin bornée lors du calcul du préambule du test.

3.4.1. *Calcul du préambule du test*

Le préambule est donc construit par animation de la spécification depuis l'état initial de la machine jusqu'à un état vérifiant la ou les valeurs limites recherchées. L'utilisation de cette technique ne permet de considérer qu'une instance close. Une hypothèse d'uniformité sur le domaine du préambule est donc posée. Nous verrons, lors de l'étude de cas, que cette hypothèse s'avère parfois être trop forte et nécessite d'être révisée dans les futurs travaux.

3.4.2. *Calcul du corps du test*

L'animateur contraint est également mis à contribution pour l'invocation de l'opération à tester. Il permet d'en explorer tous les points de choix (en fonction de ses éventuels paramètres d'entrée) qui proviennent en fait d'une décomposition du domaine des paramètres d'entrée en sous-domaines, et d'un calcul de leurs bornes. Cette technique est similaire à celle effectuée sur les variables d'état pour générer leurs valeurs limites. Ainsi, un corps de test est généré pour chaque valeur limite des paramètres d'entrée. A partir d'un même préambule, nous pouvons donc obtenir plusieurs traces de test différentes avec une même opération en invocation critique mais avec des paramètres d'entrée différents.

Exemple 6 *Etant données la variable* $code\_chv$ *et l'opération VERIFY_CHV de la spécification présentée en annexe A, dont le paramètre d'entrée* $code$ *a pour domaine (défini dans la pré-condition de l'opération)* $\{a_1, a_2, a_3, a_4\}$*, la formule «IF* $code\_chv = code$ *THEN ...», présente dans cette opération, entraîne la génération de deux valeurs limites : la première valeur est égale à la valeur de la variable* $code\_chv$*, et la seconde appartient à l'ensemble des valeurs différentes de* $code\_chv$*. Deux invocations critiques sont donc définies, correspondant à l'utilisation de chaque valeur limite comme paramètre d'entrée.*

3.4.3. *Calcul de l'identification du test*

Le pilote de test permet de choisir d'éventuelles opérations d'observation à invoquer pour l'identification. Cette sous-trace peut être ainsi constituée d'une seule opération ou d'une séquence d'opérations. Le principe consistant à calculer autant d'invocations avec la même opération, que cette dernière a de comportements différents, a également été adopté.

4. Présentation de la norme GSM 11-11 et de la machine abstraite associée

Dans le cadre d'une collaboration industrielle avec la société SCHLUMBERGER, nous avons formalisé une partie de la norme GSM 11-11 [EUR99] avec la notation B, afin de générer des traces de test avec l'environnement B-TESTING-TOOL. Ce projet avait pour but d'évaluer, dans un contexte industriel, la contribution de la méthode sur une application de taille réelle. Nous présentons, dans cette partie, le fragment de la norme étudié et introduisons sa modélisation avec la notation B.

4.1. *Présentation de la norme GSM 11-11*

La norme GSM 11-11 définit, en téléphonie mobile GSM, l'interface entre la carte SIM [2] et le ME [3]. Cette norme décrit notamment la structure logique de la SIM, les fonctionnalités et la sécurité durant l'accès aux données. La SIM possède un ensemble de fonctionnalités permettant l'accès sécurisé aux données. Ces primitives sont utilisées par les applications du ME. Les applications contenues dans le ME accèdent et modifient les fichiers de la SIM par l'intermédiaire de fonctions prédéfinies et en respectant les droits d'accès associés à chaque fichier.

Le fragment de la norme, retranscrit avec la notation B, concerne essentiellement la couche logicielle gérant les fonctionnalités d'accès sécurisé aux données. Ainsi, ni le protocole de communication entre la SIM et le ME, ni la couche applicative s'appuyant sur ces fonctions n'est formalisé. Une version simplifiée des fichiers et fonctionnalités de la SIM, choisis pour être testés, est maintenant décrite. La modélisation de ce fragment, présentée en Annexe A, est employée dans la prochaine partie pour illustrer le procédé de génération de tests.

4.2. *Structure logique de la carte SIM*

Les différents fichiers qui définissent la SIM se décomposent en une en-tête et éventuellement un corps. Les informations liées à la structure et aux attributs du fichier sont stockées dans l'en-tête. Le corps contient les données du fichier. Plus précisément, nous distinguons deux types de fichiers : les fichiers DF (Dedicated files) et EF (Elementary files). Les DF sont des dossiers pouvant contenir plusieurs autres fichiers EF ou DF. Ils définissent l'arborescence des fichiers. Un fichier DF particulier appelé MF (Master File) représente la racine de l'arbre. Au contraire, les EF sont des fichiers élémentaires contenant ou non des données. Il existe, en pratique, plusieurs types de fichiers EF (Transparent, Linear fixed et Cyclic) mais nous avons limité notre champ d'étude aux EF Transparent dont le corps est constitué par une séquence d'octets. L'arborescence et les fichiers traités dans le cadre de cet article sont limités à ceux présentés dans la figure 2.

2. Subscriber Identification Module
3. Mobile Equipment

La sélection des fichiers dans l'arbre s'effectue selon des règles précises. Sélectionner un fichier DF ou MF définit le répertoire courant. La sélection d'un EF définit celui-ci comme EF courant. Le répertoire courant est alors le DF ou MF parent de ce fichier EF. En revanche, il n'existe pas, après la sélection d'un DF ou du MF, de fichier EF courant. Les règles de sélection sont les suivantes :

- Tout fichier qui est un fils du répertoire courant peut être sélectionné.

- Tout DF qui est un fils du parent du répertoire courant (les fichiers DF «frères») peut être sélectionné.

- Le parent du répertoire courant peut être sélectionné.

- Le répertoire courant peut être sélectionné.

- Le fichier MF peut être sélectionné.

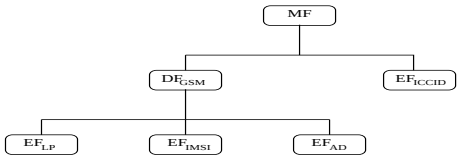

Figure 2. *Arborescence des fichiers de la SIM traités*

4.3. *Aspects sécuritaires*

Nous nous intéressons dans cette partie aux aspects sécuritaires liés aux conditions d'accès des fichiers. Les fichiers possèdent des conditions d'accès particulières pour chaque commande. Ainsi, les conditions d'accès appropriées doivent être vérifiées avant que l'action de la commande ne soit effectuée. Chaque fichier EF possède des droits d'accès en lecture (aucun droit d'accès n'est assigné actuellement par la norme GSM aux répertoires DF). On peut trouver quatre types de droits d'accès différents :

- ALWays : l'action peut être effectuée sans aucune restriction.

- CHV : l'action est effectuée si une des deux conditions suivantes est remplie : soit un code CHV correct a déjà été présenté à la SIM durant la session courante, soit l'opération UNBLOCK CHV a été réalisée avec succès durant la session courante. Cette opération, qui consiste à débloquer le code CHV, est décrite dans la section suivante.

- ADM : l'allocation de ce niveau et les conditions respectives pour la réalisation de ses actions sont la responsabilité de l'organisme administratif approprié. Dans le cadre de notre étude, l'accès d'ADMinistrateur est toujours refusé.

- NEVer : l'action ne peut pas être effectuée par l'intermédiaire de l'interface ME/SIM. La SIM peut effectuer cette action en interne.

Les conditions d'accès associées aux fichiers traités sont décrites dans le tableau 2.

Fichiers	Conditions d'accès en lecture
ef_iccid	NEVer
ef_lp	ALWays
ef_imsi	CHV
ef_ad	ADM

Tableau 2. *Conditions d'accès aux fichiers en lecture*

4.4. *Commandes sélectionnées*

Six opérations sont sélectionnées parmi la vingtaine définie dans la norme GSM 11-11. Ce choix est motivé par la décision de générer des tests concernant uniquement les aspects sécuritaires de lecture des données. Ces opérations sont les suivantes :

- $RESET$: l'activation de cette opération permet de sélectionner le fichier MF comme fichier courant, et de supprimer la condition d'accès CHV accordés éventuellement depuis le début de la session. Cette fonction permet en fait de terminer une session, et permet de simuler plusieurs sessions consécutives.

- $SELECT\_FILE$: cette opération permet de sélectionner un fichier en accord avec les règles décrites en section 4.2.

- $READ\_BINARY$: cette opération permet de lire les données présentes dans le fichier courant. Elle n'est effective que si les droits d'accès en lecture, pour le fichier EF sélectionné, sont satisfaits.

- $VERIFY\_CHV$: Cette fonction vérifie le code CHV présenté par le ME en le comparant à l'approprié enregistré sur la SIM. Les droits CHV ne doivent pas être bloqués pour effectuer cette vérification. Si le code CHV présenté est correct, le nombre d'essais restant pour valider le CHV est réinitialisé à sa valeur initiale (trois) et la session acquiert les droits sur ce CHV. Si le code CHV présenté est faux, le nombre d'essais est décrémenté de un. Après trois présentations fausses consécutives, pas nécessairement au cours de la même session, les droits CHV deviennent bloqués et la condition d'accès ne peut jamais être remplie jusqu'à ce que la fonction $UNBLOCK\_CHV$ ait été exécutée avec succès.

- $UNBLOCK\_CHV$: cette fonction permet de débloquer le CHV. Cette fonction utilise un code de déblocage associé au CHV. Si cette fonction s'exécute correctement, le CHV est débloqué, les droits sont accordés, le nombre restant de déblocage est remis à sa valeur initiale (dix), le nombre restant de présentation du code est remis à sa valeur initiale (trois), et la valeur du code CHV est fixée à une nouvelle valeur donnée en paramètre de la fonction. Si le code de déblocage présenté est faux, le nombre restant de tentatives de déblocage est décrémenté de un. Après la présentation de dix codes de déblocage faux consécutifs, pas nécessairement au cours de la même session, le code de déblocage est lui-même bloqué (état destructif).

- $STATUS$: cette fonction permet le renvoi de la valeur du fichier et répertoire courant et de la valeur actuelle de tous les compteurs de tentatives restantes.

Chaque fonction, excepté l'opération $STATUS$, renvoie toujours une réponse sous forme de code. Les codes-réponses potentiels sont décrits dans le tableau 3.

Codes	Description
9000	- Fin normal de la commande
$90FF$	- Fin normal de la commande (FF correspond à la longueur des données renvoyées)
9400	- Aucun EF n'est sélectionné
9404	- Fichier non trouvé
9804	- les conditions d'accès ne sont pas remplies - code CHV incorrect, il reste au moins encore une tentative - code de déblocage CHV incorrect, il reste au moins encore une tentative
9840	- le code CHV est bloqué - le code de déblocage CHV est bloqué

Tableau 3. *Les codes-réponses potentiels des différentes commandes*

Les fichiers et fonctions de la SIM décrits dans cette section ont été utilisés pour produire des séquences de test. L'application de la procédure de génération sur le modèle formel B de la norme GSM 11-11 est maintenant présentée.

5. Génération de tests pour la norme GSM 11-11

La prise en compte de la totalité des expressions des propriétés des variables de la spécification, présentée en annexe A, permet la génération de 28 valeurs limites, utilisées ensuite pour générer des états limites atteignables. Ces valeurs limites sont présentées dans le tableau 4. On trouve ainsi 23 valeurs limites propres et 5 valeurs limites multiples (les valeurs limites multiples proviennent de la formule conditionnelle «$permission\_session[PERMISSION\_READ[current\_file]] = \{true\}$» de l'opération $READ\_BINARY$).

Limites simples	
Variables d'état	**Valeurs**
$current\_file$	$\emptyset$ ef_ad ef_iccid ef_imsi ef_lp
$current\_directory$	mf df_gsm
$counter\_chv$	3 1 0
$counter\_unblock\_chv$	10 1 0
$code\_chv$	a_1 a_2 a_3 a_4
$blocked\_chv\_status$	$blocked$ $unblocked$

Limites simples	
Variables d'état	**Valeurs**
$blocked\_status$	$blocked$ $unblocked$
$permission\_session$	$\{(always, true), (chv, false),$ $(adm, false), (never, false)\}$ $\{(always, true), (chv, true),$ $(adm, false), (never, false)\}$

Limites multiples	
Variables d'état	**Valeurs**
$current\_file$ $permission\_session$	$current\_file = ef\_lp \wedge$ $(always, true) \in permission\_session$
	$current\_file = ef\_imsi \wedge$ $(chv, true) \in permission\_session$
	$current\_file = ef\_imsi \wedge$ $(chv, false) \in permission\_session$
	$current\_file = ef\_iccid \wedge$ $(never, false) \in permission\_session$
	$current\_file = ef\_ad \wedge$ $(adm, false) \in permission\_session$

Tableau 4. *Valeurs limites des variables d'état de la spécification*

Dans la suite, nous illustrons la méthode de génération des séquences de test. L'exemple utilisé concerne la valeur limite multiple $current\_file = ef\_imsi \wedge (chv, true) \in permission\_session$.

NOTE. – Afin de conserver une certaine lisibilité, après l'activation de chaque opération, nous présentons les valeurs des variables d'état modifiées par l'activation de l'opération par rapport à l'état précédent (la valeur initiale des variables d'état étant présentée dans la clause INITIALISATION de la spécification). Néanmoins, seules les variables de sortie des opérations sont prises en compte lors de la déclaration du verdict.

5.1. *Génération du préambule du test*

La première étape de la procédure consiste à générer un chemin menant de l'état initial à un état de la spécification vérifiant la valeur limite choisie. Les invocations générées par le solveur sont présentées dans le tableau 5. Trois opérations sont activées depuis l'état initial de la machine. Il s'agit successivement de l'activation des opérations $VERIFY\_CHV$ avec en paramètre le code correct, $SELECT\_FILE$ avec en paramètre le dossier df_gsm, et finalement $SELECT\_FILE$ avec en paramètre le fichier ef_imsi. L'état obtenu présente alors, pour les variables $current\_file$ et $permission\_session$, les valeurs attendues.

Opérations	**Variables d'entrée**	**Variables de sortie**	**Variables d'état modifiées par l'opération**
$VERIFY\_CHV$	a_1	9000	$permission\_session = \{..., (chv, true)\}$
$SELECT\_FILE$	df_gsm	$90FF$	$current\_directory = df\_gsm$
$SELECT\_FILE$	ef_imsi	$90FF$	$current\_file = ef\_imsi$

Tableau 5. *Invocations générées depuis l'état initial pour atteindre un état vérifiant la valeur limite multiple* $current\_file = ef\_imsi \wedge (chv, true) \in permission\_session$

5.2. *Génération du corps du test*

Pour tester les réactions du système à l'état limite, chaque opération est activée depuis l'état limite (éventuellement plusieurs fois avec comme données d'entrée les valeurs limites des paramètres d'entrée). Par exemple, le tableau 6 présente l'opération critique $VERIFY\_CHV$ activée, depuis l'état limite considéré, avec un code faux (a_2).

Opération	Variable d'entrée	Variable de sortie	Variable d'état modifiée par l'opération
$VERIFY\_CHV$	a_2	9804	$counter\_chv = 2$

Tableau 6. *Activation de l'opération VERIFY_CHV ,avec un code erroné, depuis l'état limite défini par la valeur limite multiple* $current\_file = ef\_imsi \wedge (chv, true) \in permission\_session$

5.3. *Génération de l'identification du test*

Dans le cadre de la spécification du fragment de la norme GSM 11-11, les phases d'identification sont effectuées par activation successive des opérations $STATUS$ (observation passive) et $READ\_BINARY$ (observation active). Le tableau 7 présente les deux opérations d'observation, activées l'une après l'autre, depuis l'état défini par le tableau 6.

Opération STATUS	
Variables de sortie	**Valeurs**
$current\_directory$	df_gsm
$current\_file$	ef_imsi
$counter\_chv$	2
$counter\_unblock\_chv$	10

(a) Observation passive

Opération READ_BINARY	
Variables de sortie	**Valeurs**
sw	9000
dd	$data(ef\_imsi)$

(b) Observation active

Tableau 7. *Identification effectuée après l'invocation critique du tableau 6*

Dans cet exemple, l'activation de l'opération $STATUS$ montre que le compteur de tentative de vérification des droits CHV a été décrémenté correctement, tandis que l'activation de l'opération $READ\_BINARY$ met en évidence le fait que les droits CHV sont toujours valides.

6. Evaluation et contribution de la méthode

Le fragment de la norme GSM 11-11, pris en considération lors de l'étude de cas, est en fait plus vaste que celle présentée dans cet article (arborescence de fichiers plus importante, plusieurs codes de type CHV, introduction de droits en écriture...). La spécification B présentée en annexe A est donc une version simplifiée de celle utilisée lors de l'étude de cas. Le modèle abstrait utilisé compte ainsi dix variables d'état et dix opérations. La technique, proposée dans cet article, a donné lieu à la génération de 58 valeurs limites et 42 états limites (certaines valeurs limites étant inatteignables). A partir de chaque état limite, 24 invocations critiques différentes (portant sur les 10 opérations) ont été effectuées. Les invocations d'observation ne présentant

aucun point de choix, 1008 traces de test ont été générées [BER 01]. Une comparaison approfondie des tests générés à l'aide de notre méthode avec les patterns de tests (création manuelle) utilisés par l'équipe d'ingénieur validation a permis de valider de manière significative notre approche. Les résultats obtenus sont maintenant discutés en terme de nombre de tests générés, de comparaison avec les tests existants pour cette application, et de temps de conception.

6.1. *Nombre de traces de test générées*

L'utilisation de l'environnement B-TESTING-TOOL a permis de générer automatiquement 1008 traces de test. La comparaison de ce nombre de séquences avec le nombre de séquences utilisé par l'entreprise n'est pas significative. En effet, chacune des séquences générées effectue un seul test tandis que chacune des séquences utilisées par l'entreprise comporte plusieurs tests. L'objet de la comparaison s'est donc porté sur la présence ou non des tests générés dans la batterie de tests de l'entreprise et inversement.

6.2. *Comparaison des jeux de test*

La comparaison des tests montre que plus de 80 % des tests manuels se retrouvent dans ceux générés à l'aide de notre méthode (en fait à une séquence de test de l'entreprise correspond plusieurs séquences générées), tandis que près de 50 % de séquences générées complètent leurs tests.

Les tests non générés automatiquement par B-TESTING-TOOL proviennent de trois phénomènes différents :

– 5% : les valeurs limites nécessaires à la construction de ce type de tests n'ont pas été générées (il ne s'agit pas de valeurs extremums des sous-domaines des variables correspondantes),

– 35% : le test porte sur l'activation de plusieurs opérations depuis un état limite généré (or seule une opération est activée par l'environnement B-TESTING-TOOL),

– 60% : le test permet de vérifier des besoins du cahier des charges qui n'apparaissent pas explicitement dans le modèle B. Par exemple, l'apparition de l'opération RESET dans certaines séquences permet la génération de tests s'étalant sur plusieurs sessions. Or, la spécification ne permet pas de détecter ce genre d'information de manière automatique.

6.3. *Evaluation du temps de conception*

Le tableau 8 présente une estimation hommes/jour du temps de conception des tests pour la norme GSM 11-11 et une comparaison entre notre méthode et une conception *à la main* sur cette même application. Il apparaît que le temps de conception n'est

pas pénalisé par l'introduction de notre méthode, et fait même apparaître un gain de temps.

Conception à la main		Conception avec la méthode présentée	
Conception des tests	20 h/j	Modélisation en B	15 h/j
Rédaction du Plan de Test et de Validation (PTV)	5 h/j	Génération des tests	automatique
Passage des tests	5 h/j	Passage des tests	5 h/j
Total	30 h/j	Total	20 h/j

Tableau 8. *Comparaison concernant le temps de conception*

6.4. *Amélioration de la procédure*

Ainsi, cette méthode de génération permet de rationaliser la conception des tests et d'automatiser le calcul des séquences. Pour autant, l'ingénieur validation n'est pas exclu du processus de conception : d'une part, il réalise la modélisation formelle en B, et d'autre part, il guide le processus de génération de tests, par la combinaison de bornes et par guidage du calcul des séquences. L'approche présentée apparaît donc comme une assistance au test fonctionnel plutôt que son automatisation. Bien que les résultats obtenus soient globalement satisfaisants, il est possible d'augmenter encore la couverture de nos jeux de test, et ainsi d'améliorer notre méthode. Les évolutions à considérer sont les suivantes :

- Manipuler les valeurs limites : il faudrait pouvoir éliminer de la génération de séquences certaines valeurs limites, qui ne présentent en fait que peu d'intérêt. L'inverse est également requis : avoir la possibilité d'ajouter des valeurs limites à celles générées automatiquement.

- Combiner des valeurs limites : cette évolution permettrait de prendre en compte des états limites comportant plusieurs valeurs limites (autres que les valeurs limites multiples).

- Proposer plusieurs préambule : le choix du chemin menant de l'état initial à l'état limite est pour l'instant le premier chemin que découvre CLPS-B au moyen de l'heuristique. Or, suivant le chemin emprunté, le sens du test peut être différent. L'hypothèse d'uniformité sur le domaine du prédicat de chemin s'avère donc trop forte. D'une part, le préambule, calculé par Best-First dans le graphe d'atteignabilité, n'assure pas l'activabilité de tous les comportements observables des opérations exécutées en invocation critique. Pour assurer cette couverture, des travaux portant sur l'étude des variables d'entrée des opérations de la spécification B sont actuellement en cours [LEG 02]. Ils s'appuient notamment sur les travaux de D.Carrington et P.Stocks concernant la méthode TTF : Test Template Framework [CAR 94] [STO 96] (cette méthode est initialement appliquée aux spécifications Z). D'autre part, certains tests s'avèrent être difficilement calculables sans intervention humaine. En effet, le

sens de ces tests n'est pas détectable par l'étude automatique de la spécification. Par exemple, l'apparition de l'opération RESET dans certains préambules permettrait la génération de tests comprenant plusieurs sessions. Ainsi, cette évolution entraîne la mise en place d'objectifs de test d'après une étude préliminaire de la spécification formelle, mais aussi d'après des besoins particuliers énoncés dans le cahier des charges.

- Tester deux opérations activées à la suite : une fois l'état limite atteint, le corps du test n'est constitué que d'une seule opération. Or, certains tests utilisés par l'entreprise portent sur l'activation consécutive de deux opérations.

7. Situation par rapport aux travaux existant

L'utilisation de spécifications formelles comme base pour la génération de jeux de test fonctionnel constitue depuis quelques années un thème de recherche actif, qui ont parfois donné lieu à la création d'outils. Nous nous proposons maintenant de situer notre méthode par rapport à certains de ces travaux.

G.Bernot, M.-C.Gaudel et B.Marre [BER 91] ont étudié la construction de jeux de test à partir de spécifications algébrique en utilisant la notion de contexte de test (un contexte de test étant défini par des hypothèses de test, un ensemble de jeux de test et un oracle). La méthode est implantée dans un outil appelé LOFT [MAR 91] qui permet la production d'ensemble de tests à partir de spécifications algébriques. L'avantage le plus marquant de notre méthode par rapport à LOFT consiste à prendre en compte la possibilité que certaines variables ne puissent pas être fixées directement depuis l'extérieur de l'application testée et requièrent ainsi l'exécution d'opérations pour être positionnées. Le préambule permet en effet d'amener le système dans l'état désiré même si les variables du système ne sont pas visibles depuis l'extérieur. En ce qui concerne la vérification des résultats de test, notre approche est similaire à celle des contextes d'observation [GAU 95]. La sous-trace d'identification constitue en effet un contexte d'observation pour la sous-trace constituant le corps du test.

La démarche décrite dans les travaux de J.Dick et A.Faivre [DIC 93] permet la génération automatique de tests à partir de spécifications écrites en VDM. Elle consiste à partitionner le domaine d'entrée des opérations de la spécification en utilisant une réduction en forme normale disjonctive des spécifications des opérations. Cette technique de partitionnement est similaire à celle implantée dans l'environnement B-TESTING-TOOL utilisant un solveur de contraintes. La seule différence réside dans le fait que notre méthode s'exécute en deux phases : la première phase concerne le partitionnement du domaine des variables d'état de la spécification tandis que la seconde porte sur la valeur des paramètres d'entrée de l'opération à tester (cette seconde phase provient directement de la nature du solveur CLPS-B, qui nativement effectue cette partition). La partition sert ensuite à la construction d'un automate à états finis qui modélise l'ensemble des séquences autorisées. Cette automate est utilisé pour sélectionner des suites de tests. Un avantage majeur de notre méthode par rapport à celle de J.Dick et A.Faivre est de pouvoir parcourir le graphe d'atteignabilité de la

machine B (grâce au solveur-animateur CLPS-B) sans devoir le construire de manière exhaustive, et ainsi évite une partie de l'explosion combinatoire.

Notre approche possède ce même avantage sur la méthode proposée par M.Hierons [HIE 97], qui utilise également un automate à états finis construit à partir de la partition des domaines d'entrée des opérations. La technique qu'il propose permet de générer des cas de test à partir de spécifications écrites avec la notation Z.

Le même principe d'approche a également déjà été appliqué dans le cadre de la méthode B. C'est le cas de l'approche adoptée par L.Van Aertryck, M.Benveniste et D.Le Metayer [AER 97], qui a donné lieu à la construction d'un outil nommé B-CASTING. La principale amélioration apportée à cette approche concerne l'observabilité : les suites de test sont générées en considérant que l'oracle a accès à toutes les variables de la spécifications, ce qui est en fait rarement le cas. Notre méthode évite ce problème grâce à une observation indirecte des états à travers les résultats retournés par les invocations du test (et plus précisément de son identification).

L'approche proposée par S.Behnia [BEH 00] et H.Waeselynck [BEH 99] vise à compléter le processus de raffinement B par une phase de test pour valider le modèle B. En cela, le problème traité concerne la génération de test à partir d'une hiérarchie de machines et de raffinements. Notre approche s'en tient à prendre en compte un modèle abstrait sans composition de machines B et ne cherche pas à s'inscrire dans un processus B traditionnel.

8. Conclusions et futurs travaux

Nous proposons dans cet article un environnement pour la génération de séquences de test fonctionnel aux bornes à partir de spécifications B. La technique de génération mise en place, en marge du processus traditionnel de la méthode B, s'appuie sur une extraction de valeurs limites des variables d'état du modèle abstrait B, par partitionnement de leur domaine respectif. Une séquence de test, définie par la notion de trace licite, se compose de trois parties :

1) un préambule permettant depuis l'état initial d'atteindre un état limite (un état dont une des variables est assignée à une de ses valeurs limites),

2) un corps constitué de l'invocation de l'opération à tester,

3) une identification utilisée pour vérifier la correction de l'opération invoquée dans le corps.

Un solveur spécifique de contraintes, CLPS-B, permet la réécriture de la spécification B en contraintes, le calcul des valeurs puis des états limites et la construction des séquences de test par animation contrainte. Les principaux points forts et l'originalité de notre approche résident dans la construction des traces au moyen de l'animateur contraint B (ne nécessitant pas la construction d'un graphe d'atteignabilité ou automate à états finis de la spécification), et dans la technique d'observation indirecte des états à travers les résultats retournés par les opérations invoquées dans la sous-trace d'identification.

Les études de cas réalisées avec B-TESTING-TOOL ont montré l'intérêt du pilotage du processus de génération de tests par l'ingénieur validation, notamment pour :

- ôter ou ajouter de nouvelles valeurs limites,

- choisir des valeurs limites propres à combiner pour générer des états limites constitués de valeurs limites associées,

- générer au préalable et sélectionner des objectifs de test via une étude statique de la spécification (méthode TTF),

- avoir la possibilité de choisir plusieurs préambules,

- décider du nombre et de la nature des opérations à activer lors du corps du test.

Un projet actuellement en cours, portant sur la génération de séquences de test fonctionnel pour la Java Card Virtual Machine, devrait nous permettre de mettre en place les évolutions évoquées et d'évaluer leurs impacts. Cette nouvelle expérience doit contribuer à la consolidation d'une version diffusée de l'environnement B-TESTING-TOOL.

9. Bibliographie

[ABR 96] ABRIAL J.-R., *The B-BOOK : Assigning Programs to Meaning*, Cambridge University Press, 1996.

[AER 97] VAN AERTRYCK L., BENVENISTE M., LE METAYER D., « CASTING : A formally based software test generation method », *In 1st Int. Conf. Formal Engineering Methods. IEEE*, 1997, p. 99-112.

[ALN 96] ALNET S., « Test de programme à partir de spécifications B : les problèmes », Rapport de DEA, Laboratoire de Recherche en Informatique - Université de Paris Sud, 1996, http ://citeseer.nj.nec.com/54798.html.

[AMB 96] AMBERT F., LEGEARD B., LEGROS E., « Programmation en logique avec contraintes sur ensembles et multi-ensembles héréditairement finis », *Technique et Science Informatiques*, vol. 15, n° 3, 1996, p. 297–328.

[BEH 98] BEHM P., DESFORGES P., MEYNADIER J., « METEOR : An Industrial Success in Formal Development », *Second conference on the B method*, vol. LNCS 1393, Montpellier, 1998, Springer-Verlag, Invited Lecture.

[BEH 99] BEHNIA S., WAESELYNCK H., « Test criteria definition for B models », *World Congress on Formal Method*, vol. LNCS 1708, 1999, p. 509-529, Springer-Verlag.

[BEH 00] BEHNIA S., « Test de modèles formels en B : cadre théorique et critères de couverture », Thèse de doctorat, LAAS – CNRS de Toulouse, 2000.

[BER 91] BERNOT G., GAUDEL M.-C., MARRE B., « Software testing based on formal specifications : A theory and a tool », *Software Engineering Journal*, vol. 6, n° 6, 1991, p. 387-405.

[BER 01] BERNARD E., LEGEARD B., LUCK X., PEUREUX F., « Generation of functional test sequences from B formal specifications using Constraint Logic Programming with sets - industrial case-study », rapport n° TFC01-01, 2001, Laboratoire d'Informatique de l'Université de Franche-Comté.

[BOU 00] BOUQUET F., LEGEARD B., PEUREUX F., PY L., « Un système de résolution de contraintes ensemblistes pour l'évaluation de spécifications B », HERMÈS, Ed., *JFPLC'00*, Marseille, France, june 2000, p. 125-144.

[BOU 02] BOUQUET F., LEGEARD B., PEUREUX F., « CLPS-B – A constraint solver for B », *Proceedings of the conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, Grenoble, April 2002.

[CAR 94] CARRINGTON D. A., STOCKS P. A., « A tale of two paradigms : Formal methods and software testing », *In Proceedings of the* 8^{th} *Z User Meeting*, Cambridge, June 1994, p. 51-68.

[DIC 93] DICK J., FAIVRE A., « Automating the generation and sequencing of test cases from model-based specifications », *FME'93 :Industrial Strength Formal Methods, Europe*, vol. LNCS 670, 1993, p. 268-284, Springer-Verlag.

[DIJ 75] DIJKSTRA E. W., « Guarded commands, nondeterminancy and formal derivation of programs », *CACM*, vol. 18, n° 8, 1975, p. 453-457.

[EUR99] European Telecommunications Standards Institute, F-06921 Sophia Antipolis cedex - France, « GSM 11.11 V7.2.0 Technical Specification », 1999.

[GAU 95] GAUDEL M.-C., « Testing can be formal too », *TAPSOFT'95 : Theory and Practice of Software Development*, vol. LNCS 915, 1995, p. 82-96, Springer-Verlag.

[HIE 97] HIERONS R., « Testing from a Z specification », *The Journal of Software Testing, Verification and Reliability*, vol. 7, 1997, p. 19-33.

[ISO] ISO. Information Processing Systems, Open Systems Interconnection, « OSI Conformance Testing Methodology and Framework – ISO 9646 ».

[JUL 98] JULLIAND J., LEGEARD B., MACHICOANE T. et al., « Specification of an Integrated Circuit Card Protocol Application Using the B Method and Linear Temporal Logic », *Second conference on the B method*, vol. LNCS 1393, Montpellier, 1998, Springer-Verlag.

[LEG 00] LEGEARD B., PY L., TATIBOUET B., « Évaluation de spécifications formelles en programmation logique avec contraintes ensemblistes – Application à l'animation de spécifications B », *AFADL'2000*, 2000, p. 21–35.

[LEG 01] LEGEARD B., PEUREUX F., « Generation of functional test sequences from B formal specifications – Presentation and industrial case-study », 16^{th} *IEEE International Conference on Automated Software Engineering (ASE'01)*, San Diego, USA, November 2001, IEEE Computer Society Press, p. 377-381.

[LEG 02] LEGEARD B., PEUREUX F., UTTING M., « A comparison of the LIFC/B and TTF/Z test-generation methods », *The* 2^{nd} *International Conference of B and Z Users (ZB'02)*, vol. LNCS 2272, Grenoble, January 2002, Springer-Verlag, p. 309-329.

[MAR 91] MARRE B., « Toward Automatic Test Data Set Selection Using Algebraic Specifications and Logic Programming », FURUKAWA K., Ed., *Proceedings of the Eighth International Conference on Logic Programming*, Paris, France, 1991, The MIT Press, p. 202–219.

[PRE 01] PRETSCHNER A., « Classical search strategies for test case generation with Constraint Logic Programming », *Proceedings of the Workshop on Formal Approaches to Testing of Software (FATES'01)*, Aalborg, Denmark, August 2001, BRICS, p. 47-60.

[PY 00] PY L., « Evaluation de spécifications formelles B en Programmation Logique avec Contraintes Ensemblistes - Application à l'animation et au model-checking », Thèse de doctorat, Université de Franche-Comté Besançon, 2000.

[STO 96] STOCKS P. A., CARRINGTON D. A., « A framework for specification-based testing », *IEEE Transactions in Software Engineering*, vol. 22, n° 11, 1996, p. 777-793.

A. Spécification B d'un fragment de la norme GSM 11.11

MACHINE GSM

SETS

$FILES = \{mf, df\_gsm, ef\_iccid, ef\_lp, ef\_imsi, ef\_ad\}$;
$PERMISSION = \{always, chv, never, adm\}$;
$VALUE = \{true, false\}$;
$BLOCKED\_STATUS = \{blocked, unblocked\}$;
$CODE = \{a_1, a_2, a_3, a_4\}$;
$DATA$

CONSTANTS

$FILES\_CHILDREN$,
$PERMISSION\_READ$,
MAX_CHV,
$MAX\_UNBLOCK$,
$CODE\_UNBLOCK\_CHV$

DEFINITIONS

$MF == \{mf\}$;
$DF == \{df\_gsm\}$;
$EF == \{ef\_iccid, ef\_lp, ef\_imsi, ef\_ad\}$;
$COUNTER\_CHV == 0..MAX\_CHV$;
$COUNTER\_UNBLOCK\_CHV == 0..MAX\_UNBLOCK$

PROPERTIES

$FILES\_CHILDREN \in FILES \longleftrightarrow FILES \wedge$
$FILES\_CHILDREN = \{(mf, df\_gsm), (mf, ef\_iccid), (df\_gsm, ef\_lp), (df\_gsm, ef\_imsi), (df\_gsm, ef\_ad)\} \wedge$
$PERMISSION\_READ \in EF \longrightarrow PERMISSION \wedge$
$PERMISSION\_READ = \{(ef\_iccid, never), (ef\_lp, always), (ef\_imsi, chv), (ef\_ad, adm)\}$
$MAX\_CHV = 3 \wedge$
$MAX\_UNBLOCK = 10 \wedge$
$CODE\_UNBLOCK\_CHV \in CODE \wedge$
$CODE\_UNBLOCK\_CHV = a_3$

VARIABLES

$current\_file$,
$current\_directory$,
$counter\_chv$,
$counter\_unblock\_chv$,
$blocked\_chv\_status$,
$blocked\_status$,
$permission\_session$,
$code\_chv$,
$data$

INVARIANT

$current\_file \subseteq EF \wedge$
$card(current\_file) \leq 1 \wedge$
$current\_directory \in DF \cup MF \wedge$
$counter\_chv \in COUNTER\_CHV \wedge$
$counter\_unblock\_chv \in COUNTER\_UNBLOCK\_CHV \wedge$
$code\_chv \in CODE \wedge$
$permission\_session \in PERMISSION \longrightarrow VALUE \wedge$
$blocked\_chv\_status \in BLOCKED\_STATUS \wedge$
$blocked\_status \in BLOCKED\_STATUS \wedge$
$data \in EF \longrightarrow DATA \wedge$
$(counter\_chv = 0) \Longleftrightarrow (blocked\_chv\_status = blocked) \wedge$
$(counter\_unblock\_chv = 0) \Longleftrightarrow (blocked\_status = blocked) \wedge$
$(current\_file = \emptyset \vee (dom(FILES\_CHILDREN \rhd current\_file) = \{current\_directory\}))$

INITIALIZATION

$current\_file := \emptyset \parallel$
$current\_directory := mf \parallel$
$counter\_chv := MAX\_CHV \parallel$
$counter\_unblock\_chv := MAX\_UNBLOCK \parallel$
$blocked\_chv\_status := unblocked \parallel$
$blocked\_status := unblocked \parallel$
$permission\_session := \{(always, true), (chv, false), (adm, false), (never, false)\} \parallel$
$code\_chv := a_1 \parallel$
$data :\in EF \longrightarrow DATA$

OPERATIONS

$sw \longleftarrow$ **SELECT_FILE**$(ff) \mathrel{\hat{=}}$
PRE
$ff \in FILES$
THEN
$IF\ (((current\_directory, ff) \notin FILES\_CHILDREN)$
$\wedge\ (\exists(dp) \cdot (dp \in FILES\ \wedge\ ((dp, current\_directory) \notin FILES\_CHILDREN \vee (dp, ff) \notin FILES\_CHILDREN)))$
$\wedge\ ((ff, current\_directory) \notin FILES\_CHILDREN)$
$\wedge\ (ff \neq mf)$
$\wedge\ (ff \neq current\_directory))$
$THEN$
$sw := 9404$
$ELSE$
$sw := 90FF \parallel$
$IF\ (ff \in DF \cup MF)$
$THEN$
$current\_directory := ff \parallel$
$current\_file := \emptyset$
$ELSE$
$current\_file := \{ff\}$
END
END
END;

$sw, dd \longleftarrow$ **READ_BINARY** $\mathrel{\hat{=}}$
PRE
$current\_file \subseteq EF$
THEN
$IF\ (current\_file = \emptyset)$
$THEN$
$sw := 9400$
$ELSE$
$IF\ (permission\_session[PERMISSION\_READ[current\_file]] = \{true\})$
$THEN$
$sw := 9000 \parallel$
$ANY\ ff\ WHERE\ (ff \in current\_file)$
$THEN$
$dd := data(ff)$
END
$ELSE$
$sw := 9804$
END
END
END;

$cd, cf, cc, cf \longleftarrow$ **STATUS** $\mathrel{\hat{=}}$
BEGIN
$cd := current\_directory \parallel$
$cf := current\_file \parallel$
$cc := counter\_chv \parallel$
$cuc := counter\_unblock\_chv$
END;

$sw \longleftarrow$ **VERIFY_CHV**$(code) \triangleq$
PRE
$code \in CODE$
THEN
$IF\ (blocked\_chv\_status = blocked)$
$THEN$
$sw := 9840$
$ELSE$
$IF\ (code\_chv = code)$
$THEN$
$counter\_chv := MAX\_CHV \parallel$
$permission\_session(chv) := true \parallel$
$sw := 9000$
$ELSE$
$IF\ (counter\_chv = 1)$
$THEN$
$counter\_chv := 0 \parallel$
$blocked\_chv\_status := blocked \parallel$
$permission\_session(chv) := false \parallel$
$sw := 9840$
$ELSE$
$counter\_chv := counter\_chv - 1 \parallel$
$sw := 9804$
END
END
END
END;

$sw \longleftarrow$ **UNBLOCK_CHV**$(code\_unblock, new\_code) \triangleq$
PRE
$code\_unblock \in CODE \wedge$
$new\_code \in CODE$
THEN
$IF\ (blocked\_status = blocked)$
$THEN$
$sw := 9840$
$ELSE$
$IF\ (CODE\_UNBLOCK\_CHV = code\_unblock)$
$THEN$
$code\_chv := new\_code \parallel$
$blocked\_chv\_status := unblocked \parallel$
$counter\_chv := MAX\_CHV \parallel$
$counter\_unblock\_chv := MAX\_UNBLOCK \parallel$
$permission\_session(chv) := true \parallel$
$sw := 9000 \parallel$
$ELSE$
$IF\ (counter\_unblock\_chv = 1)$
$THEN$
$counter\_unblock\_chv := 0 \parallel$
$blocked\_status := blocked \parallel$
$sw := 9840$
$ELSE$
$counter\_unblock\_chv := counter\_unblock\_chv - 1 \parallel$
$sw := 9804$
END
END
END
END;

RESET $\triangleq$
BEGIN
$current\_file := \emptyset \parallel$
$current\_directory := mf \parallel$
$permission\_session := \{(always, true), (chv, false), (adm, false), (never, false)\}$
END

END