GÉNÉRATION AUTOMATIQUE DE MESSAGES D'ERREUR POUR L'EXÉCUTION SYMBOLIQUE D'EXPRESSIONS DE PROCESSUS EB^3

par

Jérémy Milhau

Mémoire présenté au Département d'informatique
en vue de l'obtention du grade de maître ès sciences (M.Sc.)

FACULTÉ DES SCIENCES

UNIVERSITÉ DE SHERBROOKE

Sherbrooke, Québec, Canada, 26 novembre 2008

Sommaire

Dans le cadre des systèmes d'information générés automatiquement à partir de spécifications formelles EB^3 (selon la méthode APIS), les concepteurs de systèmes se heurtent à plusieurs difficultés. Il n'est notamment pas possible de proposer à l'avance un message d'erreur spécifique pour chacune des exécutions invalides pouvant survenir pendant la durée de vie du système. En effet, une description exhaustive de toutes les erreurs possibles d'un système d'information, et l'écriture d'un message d'erreur adapté pour chacune d'elles est un travail long et exigeant, souvent mis de côté dans les méthodes de développement traditionnelles. Or, les messages d'erreur sont une source d'aide non négligeable pour les utilisateurs d'un système d'information ; les retirer serait préjudiciable à la prise en main et à la compréhension du système par ses utilisateurs. De ce fait, une méthode de génération automatique de messages d'erreur à partir de spécifications formelles EB^3 permettant d'intégrer des informations sur l'état du système ainsi que les causes de l'erreur semble adaptée tant à l'esprit de la méthode APIS qu'à la nécessité de proposer une réponse à l'utilisateur pour ses requêtes invalides.

Ce mémoire présente une méthode de génération automatique de messages d'erreur à partir de spécifications formelles EB^3. Afin de produire un message d'erreur, une analyse de l'état du système est effectuée pour déterminer quelle est l'origine de l'erreur ; une étude de la source de l'erreur est menée pour en déduire une structure de message d'erreur ; l'intégration de données relatives au contexte d'exécution est ensuite faite sur cette structure et enfin un message de diagnostic est proposé en complément du message d'erreur.

Remerciements

Je tiens tout d'abord à remercier mes deux co-directeurs de recherche. Je remercie le Professeur Marc Frappier, qui m'a proposé d'intégrer le GRIL (Groupe de Recherche en Ingénierie du Logiciel) de l'Université de Sherbrooke dans le cadre d'une maîtrise de recherche en informatique. Il a su être disponible et pédagogue au cours de ma recherche. Je remercie également le Docteur Benoît Fraikin qui m'a orienté dans les différentes étapes du processus de recherche et de rédaction. Sa collaboration m'a beaucoup apporté dans les domaines des connaissances et du fonctionnement de la recherche. De plus je souhaite remercier le Professeur Richard St-Denis pour son aide lors des séminaires et ses remarques allant toujours dans le sens de l'amélioration de nos productions et écrits.

J'exprime également ma reconnaissance à Élodie Antoine, Pierre Konopacki, Michel Embe-Jiague et aux autres membres du GRIL pour m'avoir accompagné dans mes démarches administratives et de recherche au sein du laboratoire, ainsi que pour leur accueil.

Je souhaite également remercier l'ENSIIE (École Nationale Supérieure d'Informatique pour l'Industrie et l'Entreprise, Évry, France) qui m'a permis d'effectuer ma troisième année de formation d'ingénieur à l'Université de Sherbrooke, la Professeure Sandrine Blazy responsable de l'accord ainsi que les Professeurs Catherine Dubois, Olivier Pons et Frédéric Roupin pour avoir soutenu ma candidature à cette maîtrise.

À Marie-Laure.

Abréviations

APIS production automatisée de systèmes d'information, « *Automated Production of Information Systems* »

AST arbre de syntaxe abstraite, « *Abstract Syntax Tree* »

EB^3 boîtes noires basées sur les entités, « *Entity-Based Black-Box* »

EB^3GG générateur de gardes EB^3, « EB^3 *Guards Generator* »

EB^3PAI interpréteur de l'algèbre de processus EB^3, « EB^3 *Process Algebra Interpretor* »

EB^3TG générateur de transactions EB^3, « EB^3 *Transactions Generator* »

ENSIIE École Nationale Supérieure d'Informatique pour l'Industrie et l'Entreprise

GRIL Groupe de Recherche en Ingénierie du Logiciel

IS système d'information, « *Information System* »

PE expression de processus, « *Process Expression* »

SI système d'information

UML langage de modélisation unifié, « *Unified Modeling language* »

Table des matières

Sommaire iii

Remerciements v

Abréviations vii

Table des matières ix

Liste des figures xiii

Liste des tableaux xv

Introduction 1

1 La méthode EB^3 et le projet APIS 5

1.1 De la méthode EB^3 5

1.1.1 Du diagramme entité-relation 5

1.1.2 De la spécification de l'interface graphique 6

1.1.3 De l'évaluation des attributs en fonction de la trace 6

1.1.4 Des règles d'entrées-sorties 6

1.1.5 De l'expression de processus 6

1.2 Des fonctionnalités actuelles de APIS 7

1.2.1 De DCI-WEB 7

1.2.2 De EB^{3}TG 8

1.2.3 De EB^{3}GG 8

1.2.4 De l'interpréteur EB^{3}PAI 9

2 Génération automatique de messages d'erreur **11**
2.1 Introduction 13
2.2 Background 14
2.2.1 Symbolic Execution of EB^3 Process Expression 14
2.2.2 A Specification 14
2.2.3 Error Types 16
2.3 Identifying the Cause of an Execution Error 16
2.3.1 Victims 18
2.3.2 Witnesses 19
2.3.3 Suspects 20
2.3.4 Culprits 23
2.4 Generating Messages for an Execution Error 24
2.4.1 Messages from Culprits 24
2.4.2 Required Action 25
2.4.3 Information System Patterns 27
2.5 Case Study 28
2.5.1 Current State 29
2.5.2 Generated Messages 29
2.6 Conclusion 31

3 Cas d'étude **33**
3.1 Entrelacement et environnement 33
3.1.1 Description du cas 33
3.1.2 Comportement espéré 34
3.1.3 Résultat obtenu 34
3.2 Gardes 36
3.2.1 Description du cas 36
3.2.2 Comportement espéré 36
3.2.3 Résultat obtenu 37
3.3 Parenthésage et séquence 37
3.3.1 Associativité de la séquence 38
3.3.2 Description du cas 38

3.3.3 Comportement espéré 38
3.3.4 Résultat obtenu 39
3.4 Influence de la spécification sur le message d'erreur 39
3.4.1 Description du cas 39
3.4.2 Comportement espéré 40
3.4.3 Résultat obtenu 40

Conclusion **41**

Bibliographie **45**

Liste des figures

1 Comparaison des coûts des méthodes traditionnelles et APIS 2

1.1 Structure du projet APIS 8

2.1 EB^3 PE of a library 15
2.2 Simplified AST of **member** entity 19

3.1 Entrelacement à l'état initial 34
3.2 Entrelacement après l'exécution de Aa(4) 34
3.3 Témoins dans l'investigation sur l'entrelacement 35
3.4 Suspects et coupables dans l'investigation sur l'entrelacement 35
3.5 Expressions gardées 36
3.6 Témoins dans l'investigation sur les gardes 37
3.7 Suspects et coupables dans l'investigation sur les gardes 38

Liste des tableaux

2.1 Suspects relatively to the victim of DisplayNumberOfLoans($John$) 23

2.2 Investigation over Lend($11, 0$) 30

2.3 Investigation over Take($12, 1$) 30

3.1 Investigation sur Bb(1) pour E_1 et E_2 39

Introduction

Contexte

Les systèmes d'information (SI) sont très répandus dans les milieux professionnels et grand public. Plus ils jouent un rôle important, plus leur développement devient crucial, coûteux et long. Cependant, les techniques de spécification formelle comme celle du projet APIS [12, 13] permettent de réduire les coûts et le temps (figure 1) nécessaires pour leur développement. En effet, en spécifiant le comportement autorisé du système via la méthode EB^3 [11, 14], le système sera généré tel qu'il a été conçu, sans qu'une phase d'implémentation ne soit nécessaire.

Au sein des SI surviennent des erreurs qui peuvent avoir plusieurs origines. Si une erreur survient consécutivement à l'envoi d'une requête au SI par l'utilisateur, il faut en informer l'utilisateur pour qu'il puisse modifier sa requête dans le but de la rendre acceptable. Un message d'erreur doit alors être produit dans ce but. Dans un cycle de développement conventionnel, c'est au développeur d'inclure dans son code les messages d'erreur en essayant d'être le plus exhaustif possible. Si les messages d'erreur peuvent ainsi être très pertinents et adaptés, des cas non traités peuvent persister. Parfois, un code d'erreur sera retourné à l'utilisateur, l'invitant à consulter une ressource d'assistance. D'autres fois, alors qu'aucun message d'erreur n'était prévu, le système pourra ne pas indiquer d'erreur à l'utilisateur.

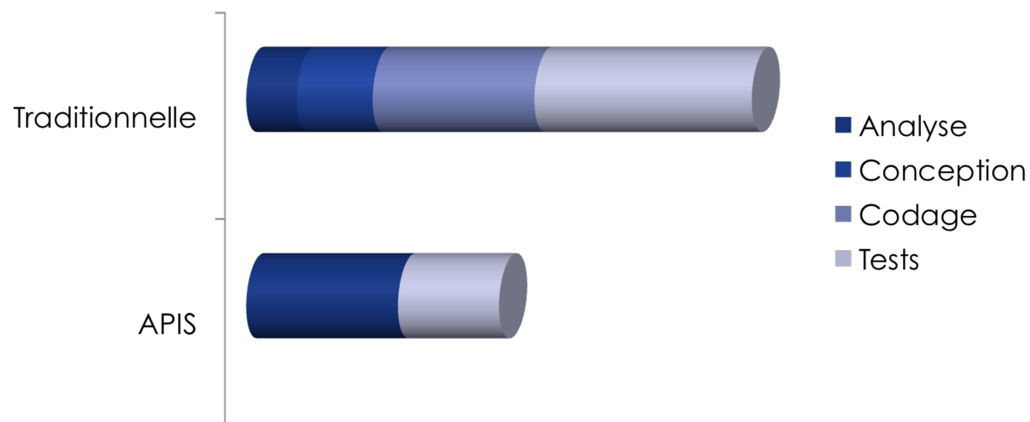

figure 1 – Comparaison des coûts des méthodes traditionnelles et APIS

Problématique

Dans le cadre des systèmes générés par la méthode EB^3, la gestion des messages d'erreur par le concepteur du système n'a pas été prévue. Et pour les raisons exposées, il semble pertinent de rechercher une alternative aux messages d'erreur issus de cycles de développement conventionnels. C'est pourquoi une approche de génération de messages d'erreur basée sur les spécificités de la méthode EB^3 a été envisagée. Elle doit répondre à plusieurs critères : être automatique ; générer un message pertinent et compréhensible ; être la plus exhaustive possible ; s'adapter aux différentes façons d'écrire une même spécification ; contribuer à ce que les erreurs ne restent pas « la part négligée de l'interface personne/machine » [5].

Résultats

Un algorithme de génération automatique de messages d'erreur pour l'exécution symbolique d'expressions de processus EB^3 a été produit et permet d'obtenir des messages d'erreur et un diagnostic pour résoudre le problème rencontré. Cet algorithme a été implémenté en OCaml afin de procéder à des tests. Il est également possible d'y adjoindre de nouveaux patrons afin d'augmenter son efficacité dans les cas où l'utilisateur le trouvera faible.

Méthodologie

Afin de produire cet algorithme, une approche basée sur la comparaison avec les messages des méthodes de conception conventionnelles a été suivie. Pour cela, et à travers l'étude de plusieurs cas, s'est posée la question « Quel message d'erreur aimerais-je avoir, en tant qu'utilisateur, dans ce cas de figure ? ». Puis un rapprochement entre les données disponibles via l'expression de processus et les unités sémantiques du message d'erreur a été réalisé. Enfin, des similitudes entre les messages d'erreur et les opérateurs non respectés sont apparues. De ce fait, il semblait logique de s'orienter vers une approche basée sur les opérateurs de l'algèbre de processus.

Structure du mémoire

Ce mémoire présente une méthode de génération automatique de messages d'erreur pour l'exécution symbolique d'expressions de processus EB^3 dans un système d'information créé en utilisant le projet APIS [12]. Dans le chapitre 1 sont introduits les éléments issus de la méthode EB^3 [11, 14] et du projet APIS utilisés dans la méthode de génération de messages d'erreur. Le chapitre 2 présente un article intitulé « Automatic Generation of Error Messages for the Symbolic Execution of EB^3 Process Expressions ». Cet article décrit la théorie associée à la génération automatique de messages d'erreur. Enfin, un cas d'étude (chapitre 3) complète les exemples de l'article.

Chapitre 1

La méthode EB^3 et le projet APIS

Ce chapitre présente la méthode EB^3 et le projet APIS. La méthode EB^3 (section 1.1) décrit la façon dont doivent être spécifiés les éléments nécessaires à la génération d'un SI, comme les opérateurs de l'algèbre de processus. Le projet APIS (section 1.2), quant à lui, regroupe les outils permettant la traduction d'une spécification formelle en un système d'information et comprend notamment l'interpréteur d'expressions de processus EB^{3}PAI [7].

1.1 De la méthode EB^3

La méthode EB^3 est une méthode de spécification de système d'information basée sur cinq éléments qui serviront chacun dans une ou plusieurs des étapes de la production du système final. La figure 1.1 présente ces éléments et les différents modules les utilisant. On considérera que le système est modélisé par une expression de processus et une trace (ou séquence) d'actions qui ont été exécutées depuis la mise en production du système.

1.1.1 Du diagramme entité-relation

Le diagramme entité-relation de la méthode EB^3 s'exprime en *UML* (« *Unified Modeling language* », langage de modélisation unifié) [20]. Il permet de définir le schéma de la base de données à générer et est également utilisé pour la génération des sorties en fonction des entrées.

1.1.2 De la spécification de l'interface graphique

La spécification de l'interface graphique permet de déterminer comment seront agencés les éléments issus du SI ainsi que leur intégration dans une interface de type Internet. Cela comprend les formulaires nécessaires à l'entrée de données et de paramètres pour les actions, les zones de retour pour afficher les résultats ainsi que la possibilité de signaler une erreur à l'utilisateur.

1.1.3 De l'évaluation des attributs en fonction de la trace

Sous la forme de fonctions récursives sur la trace, sont définies les valeurs que peuvent prendre les attributs. Chaque fonction est associée à un attribut et réciproquement. Ces fonctions détaillent les conséquences qu'a chaque action du système sur l'attribut auquel elle se rapporte. Ces fonctions sont ensuite associées, à un niveau moins abstrait, à des requêtes de mise à jour des attributs dans la base de données.

1.1.4 Des règles d'entrées-sorties

La signature des actions permet de définir le nombre et le type des arguments d'une action, si elle en a. Ces signatures sont utilisées pour vérifier que le format d'entrée d'une action est respecté par l'utilisateur lors de ses requêtes, mais aussi pour s'assurer que l'expression de processus est correcte dans le sens où elle doit respecter ces signatures. Une sortie est également associée à chaque entrée valide du système afin de relier les modifications des attributs ou l'affichage de données à des actions valides de l'utilisateur.

1.1.5 De l'expression de processus

L'algèbre de processus de la méthode EB^3 permet d'écrire des expressions de processus qui serviront d'entrées pour la génération automatique d'un SI. Ces spécifications se basent sur des notations orientées évènement comme les langages de spécification CSP [15] et CCS [19]. Ces expressions de processus sont créées à l'aide d'expressions de processus élémentaires et des opérateurs de l'algèbre de processus EB^3.

Une expression de processus élémentaire peut être λ, une action interne au système, semblable à ϵ dans la théorie des automates ; ou $\mathsf{action}(x_1 \ldots x_n)$ une action élémentaire avec ses n paramètres x_1 à x_n.

Les opérateurs permettant de composer les expressions de processus sont : la séquence de deux expressions de processus[1] notée $E_1 \centerdot E_2$; le choix entre deux expressions de processus noté $E_1 \mid E_2$; la fermeture de Kleene, au sens conventionnel du terme dans la théorie des langages, notée ${E_1}^*$; la synchronisation sur un ensemble Δ notée $E_1|[\Delta]|\,E_2$; et la garde d'une expression de processus par un prédicat, notée $p \implies E$. Des versions quantifiées du choix et de la synchronisation peuvent également être utilisées. Du fait de la fréquence de leur apparition dans les expressions de processus, les opérateurs suivant sont également définis : l'entrelacement noté $E_1 \mathbin{|||} E_2$ est défini par$E_1|[\emptyset]|\,E_2$; l'exécution parallèle notée $E_1 \parallel E_2$ et définie par $E_1|[\Delta]|\,E_2$, où Δ représente l'ensemble des actions communes à E_1 et E_2. Enfin, consécutivement à l'exécution d'actions dans une expression de processus, peuvent apparaître des environnements notés $[\![x_1 := y_1, \ldots x_n := y_n]\!]$. Les environnements représentent des substitutions à venir, dans le sens de l'évaluation paresseuse, de chaque terme x_i par le terme y_i.

1.2 Des fonctionnalités actuelles de APIS

Dans son implémentation actuelle, le projet APIS consiste en plusieurs outils plus ou moins indépendants. La figure 1.1 présente l'organisation entre les différents modules ainsi que les relations entre eux et les éléments de la méthode EB^3. Les modules plus récents comme EB^{3}GG ne sont pas intégrés dans cette figure.

1.2.1 De DCI-WEB

Ce module qui peut éventuellement être indépendant de APIS permet de générer automatiquement une interface et un squelette de site Internet à partir des spécifications formelles de la GUI (« *Graphic User Interface* », interface graphique utilisateur). DCI-WEB établit ensuite le lien entre les actions effectuées sur le site Internet et les requêtes au sys-

[1] appellée aussi concaténation

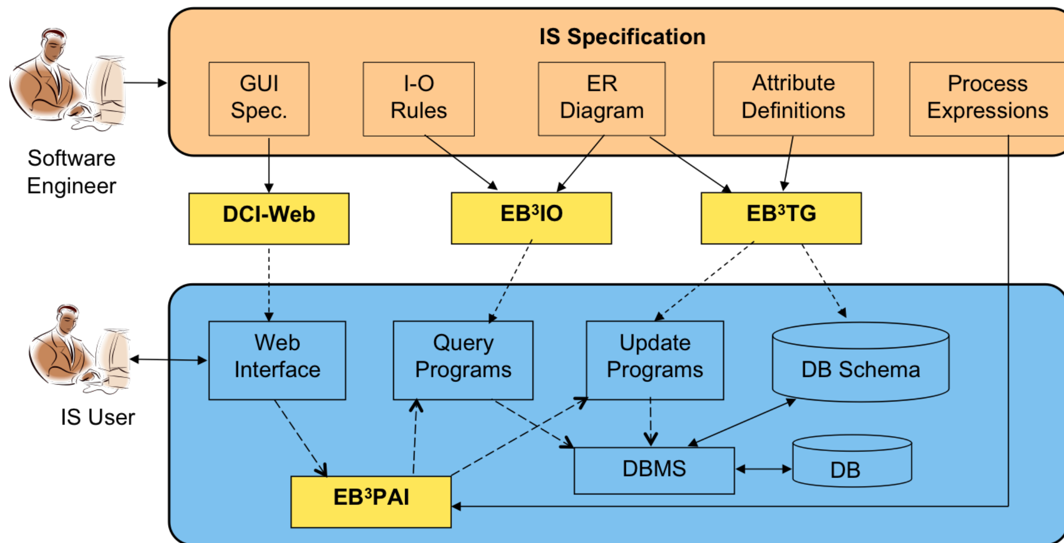

figure 1.1 – Structure du projet APIS

tème d'information. La spécification de l'interface graphique a été définie par Élodie Antoine [3] dans le cadre de sa maîtrise en génie logiciel à l'Université de Sherbrooke.

1.2.2 De EB³TG

Ce module permet de générer l'ensemble des requêtes aux bases de données du système en se basant sur les règles de définition des attributs et du schéma entité-relation. Les requêtes ainsi produites seront ensuite appelées par les actions exécutées par l'utilisateur. Ce module a été développé par Panawé Batanado [4] dans le cadre de sa maîtrise en informatique à l'Université de Sherbrooke.

1.2.3 De EB³GG

Ce module est entièrement dédié aux gardes dans EB³. Le module EB³GG va tout d'abord en faire l'analyse syntaxique puis les optimiser et enfin générer toutes les requêtes nécessaires à leur vérification. Il est interfacé avec EB³PAI pour vérifier l'exécution des processus gardés. Ce module a été mis en place par Pierre Konopacki [16] lors de sa maîtrise en informatique à l'Université de Sherbrooke.

1.2.4 De l'interpréteur EB^{3}PAI

L'interpréteur d'expressions de processus EB^{3}PAI développé par Benoît Fraikin [6] dans le cadre d'un doctorat en informatique à l'Université de Sherbrooke interprète de manière efficace [8, 9, 10] les expressions de processus EB^3. Il accepte les requêtes de l'utilisateur sous la forme d'actions (étiquette et paramètres) qu'il tente d'exécuter dans l'état actuel du système. Si l'action est exécutable, une nouvelle expression de processus est calculée à partir de l'état courant et de l'action entrée et les requêtes correspondantes à l'action sont exécutées pour modifier la base de données. Si l'action n'est pas exécutable, elle est refusée et l'état courant du système est conservé inchangé.

Dans son implémentation Java, EB^{3}PAI utilise une représentation de l'état courant sous la forme d'un AST (*Abstract Syntax Tree*, arbre de syntaxe abstraite). Les feuilles de l'arbre représentent les actions élémentaires de l'expression de processus ; les noeuds représentent les opérateurs. Chaque opérateur est associé à des méthodes comme $alphabet$ qui calcule l'alphabet, c'est-à-dire l'ensemble des étiquettes des actions présentes dans les opérandes. La méthode $canBox(E)$ permet de déterminer si une expression de processus peut terminer en exécutant uniquement des actions internes λ, c'est-à-dire se réduire à box, noté $\boxdot$. Par exemple, $canBox((\mathsf{a}\,.\,\mathsf{b})^*)$ retournera TRUE alors que $canBox(\mathsf{a}\,.\,\mathsf{b})$ retournera FALSE. La méthode $kappaIndice(E, label)$ retourne l'indice de position d'un terme apparaissant dans une quantification (E doit donc correspondre à une quantification) pour une action de $label$ donné. Par exemple $kappaIndice(|||\, y \in \mathrm{Y} \,:\, \mathsf{a}(\,x, y, z\,), \mathsf{a})$ retourne 2.

Chapitre 2

Génération automatique de messages d’erreur

Résumé

Cet article développe la théorie associée à la détection des erreurs d’exécution et à la génération automatique de messages d’erreur associés à ces erreurs. Il présente les méthodes de recherche et de détermination d’une ou des sources de l’erreur, c’est-à-dire un ou plusieurs opérateurs dont les conditions d’exécution associées ne sont pas respectées, et détaille les éléments participant à la génération du message d’erreur. Enfin, il illustre le tout via deux exemples d’un système d’information relatif à une bibliothèque.

Commentaires

Ma contribution au sein de cet article consiste dans le développement de la théorie exposée, la réalisation du cas d’étude et de l’exemple ainsi que la rédaction majoritaire du contenu. Cet article a été soumis et accepté à la conférence iFM 2009 ayant lieu à Dusseldorf, Allemagne.

Automatic Generation of Error Messages for the Symbolic Execution of EB^3 Process Expressions

Jérémy Milhau, Benoît Fraikin and Marc Frappier
Département d'informatique, Université de Sherbrooke,
Sherbrooke, Québec, Canada J1K 2R1
{Jeremy.Milhau,Benoit.Fraikin,Marc.Frappier}@usherbrooke.ca

Abstract

This paper describes an algorithm to automatically generate error messages for events refused by a process expression. It can be used in the context of an information system specified with the EB^3 method. In this method, a process expression is used to describe the valid traces of events that the information system must accept. If a user submits an event which is rejected by this process expression, our algorithm produces an error message explaining why the event has been rejected ; it also suggests which event should be submitted in order to correct the error.

2.1 Introduction

Information systems (IS) are commonly used nowadays in organizations, but their development is expensive and requires a long-term vision for their integration. With the aim of reducing cost and time needed to develop such systems, the Automated Production of Information Systems (APIS) project [12] was launched to provide an efficient way to generate ISs from formal specifications. As part of the APIS project, the Entity-Based Black-Box method (EB^3) [11, 14] provides mechanisms to write formal specifications that describe the input-output behavior of the IS. A specification includes a process expression (PE) that describes the valid traces of input events that the IS must accept.

Fraikin and Frappier have shown that a PE can be efficiently executed by the EB^3 Process Algebra Interpreter (EB^3PAI) [6], the core of the generated IS. If, for one reason or another, the user input, which is called the query henceforth, does not comply with the specification, EB^3PAI rejects it and preserves the state of the IS.

In such a case, an error message [5] must be issued to report to the user that his query is not valid, and explain why and how he can modify his query to comply with the IS specification. In a traditional implementation of an IS, error messages are determined and implemented by a programmer. Since EB^3PAI uses symbolic execution based on the operational semantics of the EB^3 process algebra, the state of the system is represented by the abstract syntax tree (AST) of the PE. Hence, we can not manually determine the error message to produce when an event is rejected. Consequently, we have derived an algorithm that produces an error message through an analysis of the AST of the PE. This problem is somewhat similar to the generation of error messages within a compiler (*e.g.* a Java compiler). The PE of the IS specification corresponds to the programming language addressed by the compiler (*e.g.*, Java for a Java compiler). The user inputs correspond to the program text (*i.e.* the Java program submitted to the Java compiler). An invalid event corresponds to a syntax error in the program text. However, there are major differences : a compiler is built for a specific programming language (*e.g.* Java) ; EB^3PAI must be able to execute any PE. Since a compiler is built for a single language, the compiler designer can exploit the syntax of the language to determine the error message. Error message generation in a compiler is an art that the compiler designer must master [2]. In our case, we must devise a generic algorithm, which works for any PE.

2.2 Background

The EB^3 process algebra is similar to several other process algebra like CSP [15], CCS, ACP and Lotos. We shall briefly introduce it here ; the reader is referred to [14] for a complete description. An elementary PE is either an internal action λ, which plays the same role as ϵ in regular expressions, or an action $\mathsf{a}(t_1, \ldots, t_n)$, where a is an action label and t_i denotes a constant or a variable. Compound PEs are built using the following operators : $E_1 \,.\, E_2$ (sequence, also called concatenation), $E_1 \mid E_2$ (choice), $E_1{}^*$ (Kleene closure of E_1), $E_1|[\Delta]|\,E_2$ (synchronization over the set Δ) and $p \implies E$ (guard p over the PE E). A quantified version of the choice | and the synchronization $|[\Delta]|$ can also be used. Other operators are built from these operators : $E_1 \,|||\, E_2$ (interleaving operator) which is defined as $E_1|[\emptyset]|\,E_2$, and $E_1 \parallel E_2$ (parallel operator) defined as $E_1|[\Delta]|\,E_2$ where Δ is the intersection of the alphabets of the operands. The PE ΓE denotes the application of environment Γ to E ; Γ is a substitution $[\![x_1 := y_1, \ldots x_n := y_n]\!]$ which denotes the values of variables x_i in E.

2.2.1 Symbolic Execution of EB^3 Process Expression

EB^{3}PAI uses symbolic execution based on the operational semantics of the EB^3 process algebra. For instance, the PE $\mathsf{a} \,.\, (\mathsf{b} \mid \mathsf{c}) \,.\, \mathsf{d}$ can execute a and transform into PE $(\mathsf{b} \mid \mathsf{c}) \,.\, \mathsf{d}$, which we note $\mathsf{a} \,.\, (\mathsf{b} \mid \mathsf{c}) \,.\, \mathsf{d} \xrightarrow{\mathsf{a}} (\mathsf{b} \mid \mathsf{c}) \,.\, \mathsf{d}$; this is called a transition and it is computed by EB^{3}PAI from the inference rules of the operational semantics. PE are represented by their AST in EB^{3}PAI.

2.2.2 A Specification

A simple example of IS specification is the library. The library can acquire and discard books, and members can join or leave membership of the library. If a member wants to borrow a book, he must first reserve it if the book is already borrowed. A member can then renew its loans and return a borrowed book. The first person on the reservation list can borrow the book when the previous borrower returned it and anyone can cancel its reservation at anytime. Some statistics such as the current borrower of a book, the number of loans for a member and a list of all borrowers by category can be generated.

The specification of figure 2.1 describes the behavior of an IS which aims to monitor a library for both members and books. This specification follows commonly used patterns [14] for entities (book and member) and relationships (loan and reservation).

main = ($|||\, bId \in \text{BOOKID} : $ **book** $(bId)^*$)
$||$ ($|||\, mId \in \text{MEMBERID} : $ **member** $(mId)^*$)
$||$ DisplayBorrowerByCategory($)^*$

book $(bId : \text{BOOKID})$ = Acquire($bId, _$)
. ($(|\, mId \in \text{MEMBERID} : $ **loan** $(mId, bId))^*$
$||$ ($|||\, mId \in \text{MEMBERID} : $ **reservation** $(mId, bId)^*$)
$||$ DisplayCurrentBorrower($bId)^*$
)
. Discard(bId)

member $(mId : \text{MEMBERID})$ = Join(mId)
. (($|||\, bId \in \text{BOOKID} : $ **loan** $(mId, bId)^*$)
$||$ ($|||\, bId \in \text{BOOKID} : $ **reservation** $(mId, bId)^*$)
$||$ DisplayNumberOfLoans($mId)^*$
)
. Leave(mId)

loan $(mId : \text{MEMBERID}, bId : \text{BOOKID})$ =
(Lend(mId, bId) | Take(mId, bId)) . Renew($mId, bId)^*$. Return(mId, bId)

reservation $(mId : \text{MEMBERID}, bId : \text{BOOKID})$ =
Reserve(mId, bId) . ($isFirst(trace, mId, bId) \Longrightarrow$ Take(mId, bId)
| Cancel(mId, bId))

figure 2.1 – EB^3 PE of a library

This specification describes the life cycle of members and books that are then put in parallel execution. The process **member** describes a member whose key is mId. By using $|||\, mId \in \text{MEMBERID} : $ **member** $(mId)^*$, the specification allows multiple **member** processes with different mId keys to run simultaneously. The same technique is used for the **book** process that allows multiple instances of **book** with different bId to run simultaneously.

Both sets of processes run in parallel execution and synchronize over the shared actions of the **loan** and the **reservation** processes. Since one book can not be borrowed by more than one member at once, the quantification for **loan** in the **book** process is over a choice.

2.2.3 Error Types

Two categories of errors can be detected : syntax errors and execution errors.

Syntax Errors.

Syntax errors are the simplest to detect and manage. They correspond to a violation of the signature of actions defined in the EB^3 specification. For instance, they include query parameters with improper types, an invalid number of parameters (*i.e.* missing or extra parameters), and an invalid action label. The production of error messages for these cases is trivial and not described in this paper.

Execution Errors.

Execution errors denote syntactically correct queries which can not be accepted by the current PE. This paper focuses on the generation of error messages for this kind of errors.

2.3 Identifying the Cause of an Execution Error

This section describes the algorithm used to identify the cause of an error. Note that the PE is supposed to be correct ; it is the user query which is incorrect. However, we will speak about the "causes" of an error in the PE, in order to identify why the query is rejected. The cause of an error is essentially one or several operators in the PE which can not accept the user query.

For the sake of illustration[1], the terminology of police investigation is used to describe the process of finding of the cause of an execution error. The causes of an error are referred as *guilty* operators that are designated among *suspect* operators. Some operators may also

[1] and also for the fun of it.

have an *alibi* that removes them from the suspect list. There are two classes of alibi : acquired and inherent. An acquired alibi depends upon the context of execution. It is called an *environmental alibi*. An inherent alibi is always valid whatever the crime is, *i.e.* whatever the context is. Operators with inherent alibis are referred below as *permissive operators*. The operators that belong to this class will be safe from the investigation due to their semantics that prove they can not be designated as guilty.

The next algorithm, called $main$, computes information from the current state of the IS and the query from the user to deduce a list of guilty operators. It describes the needed steps of the investigation that lead to a *verdict*, *i.e.* a complete error message. The functions called by $main$ are described in the rest of section 2.3.

Algorithm 2.3.1 $main(E, \sigma)$

Description : *The main algorithm for the generation and display of error messages.*

input E **:** *An* EB^3 *PE that describes the current state of an IS.*

input σ **:** *A query from the user refused by* E*.*

$main(E, \sigma)$
begin
let V *be the list of victims,* $v = Victims(E, \sigma)$*,*
let V' *be the list of victims with no environmental alibi,*
$V' = filter\Big(V, \lambda x.noEnvironmentalAlibi(x, E, \sigma)\Big)$*,*
let W *be the list of list of witnesses,* $W = map\Big(\lambda x.Witnesses(x, E, \sigma), V'\Big)$*,*
let S *be the list of list of suspects,* $S = map\Big(\lambda x.Suspects(x), W\Big)$
let C *be the list of culprits,* $C = map\Big(\lambda x.Culprit(x), S\Big)$
foreach c ***in*** C ***do***
$display(getVerdictFrom(c, \sigma, E))$
done
end

where map *and* $filter$ *are the classical list operators in functional programming and* $getVerdictFrom$ *creates the error message ; its algorithm is described in section 2.4.*

The reader is invited to consult [18] for complete details about the implementation of this algorithm in Caml.

2.3.1 Victims

Considering the query entered by the user of the IS, a list of all the potential victims of the error is made. these are the actions of the PE that share with the query the same action label. The intuition is that if a PE can execute the query, there must exist a leaf in the AST that matches this query. If there is no leaf, that matches this query, then it will never be possible to execute the query. When such a leaf exists, we can start analysing why it can be executed. From this list are removed actions that do not comply with the restrictions imposed by the parameters of the query. Indeed, the parameters of the query may have values that are not compatible with the values of corresponding parameters in the PE. That is why these actions with environmental alibis are removed from the list of victims.

Definition 2.3.1 ***Environmental alibi of an action relative to a query***
Let σ = $\mathsf{a}(v_1, ..., v_n)$ be a query. An action ω = $\mathsf{a}(u_1, ..., u_n)$ is said to have an environmental alibi if, after applying the substitutions of the enclosing environments, the predicate $\bigvee_{i=1}^{n}(\neg isVariable(u_i) \wedge (u_i \neq v_i))$ is evaluated to TRUE*, where $isVariable(x)$ returns* TRUE *if x is a variable.*

Alibi example.

Let the following definitions : $\sigma = \mathsf{a}(1, 2)$, $E = |||\, x \in 1..5 : \mathsf{a}(x, 2)$, $F = [\![y := 2]\!]\big(|||\, x \in 1..5 : \mathsf{a}(x, y)\big)$ and $G = [\![y := 3]\!]\big(|||\, x \in 1..5 : \mathsf{a}(x, y)\big)$. If environments are applied as substitutions, then we can rewrite $F = |||\, x \in 1..5 : \mathsf{a}(x, 2)$ and $G = |||\, x \in 1..5 : \mathsf{a}(x, 3)$. Then E and F are similar. Now the predicate is applied to the action $\mathsf{a}(_, _)$ of each PE. For E and F, x is bounded by the $|||\, x \in 1..5$ operator and the second parameter of a equals the second parameter of σ so there is no environmental alibi for both actions. For G, x is bounded, $2 \neq 3$; hence the action in G has an environmental alibi.

Definition 2.3.2 ***Victims relative to*** σ
Let E be a PE and σ a query refused by E. Victims relative to σ are defined as the set of leaves of E whose label is the same as σ and does not have environmental alibi.

Example.

If the user tries to execute the action DisplayNumberOfLoans($John$) in the library IS described in figure 2.1 where $John$ is a valid id for a member (*i.e.* $John \in$ MEMBERID) but $John$ is not a registered member (*i.e.* Join($John$) was never executed) then the system can not execute the query of the user.

Victims relative to DisplayNumberOfLoans($John$) is the list reduced to one leaf : DisplayNumberOfLoans($John$) in the **member** process of the initial specification. The figure 2.2 show the position of this leaf in the **member** process. In this simplified AST of the **member** entity, some operators are numbered to help further references. Operators that will not be referenced are not numbered.

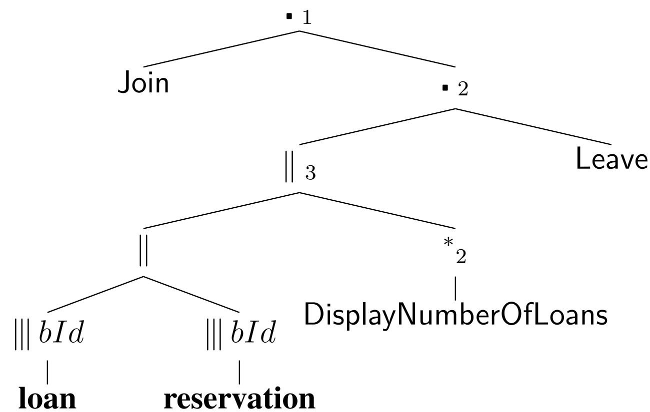

figure 2.2 – Simplified AST of **member** entity in the EB^3 specification of the library

2.3.2 Witnesses

Since the query of the user is refused by the PE, the causes of the error are one or more operators related to the action. Based on the victims, a list of witness operators is established. This list contains the list of all operators on the path from a leaf in the list of the victims to the root of the PE.

Definition 2.3.3 ***Witnesses relative to a victim*** v
Let E be a PE, σ a query from the user refused by E and v a victim of E relative to σ.

Witnesses relative to a victim v are defined as the list of operators which are parents of v in E.

Example.

The calculus of the witnesses relative of a victim is easy to do with the AST representation of the PE. Since only one victim was found relative to the query of the user, only one list of witnesses will be generated by this step.

The list of witnesses relative to the victim of DisplayNumberOfLoans($John$) is : $\|_1$, $\|_2$, $|||$ mId, $^*{}_1$, $\blacksquare_1$, $\blacksquare_2$, $\|_3$, $^*{}_2$. The first four operators come from the PE of the **main** process ; they are omitted from figure 2.2 for the sake of concision. The last four come from the PE of the **member** process.

2.3.3 Suspects

To reduce the field of investigation, some operators, which we call *permissive*, are removed from witnesses. Permissive operators can execute the query if their operands can. In other words, if the query fails, it is not their fault, but the fault of the operators occurring in their operands. This is the crux of the permissiveness evaluation described below. The predicate $isPermissive$ is applied to each witness to evaluate if it can be removed (TRUE) or considered as suspect (FALSE). The following algorithm is written using the functional language Caml [17], and uses pattern matching in order to evaluate the predicate $isPermissive$.

Algorithm 2.3.2 $isPermissive(F, \sigma, \Gamma)$

Description : *Evaluates if an operator must not be considered as suspect.*

input F **:** *An AST whose root is the witness operator for which permissiveness is evaluated. At least one victim must occur in F.*

input σ **:** *A query from the user.*

input Γ **:** *The enclosing environment of F.*

let $isPermissive(F, \sigma, \Gamma)$ $=$ ***match*** F ***with***

$E_1 \mid E_2 \quad \rightarrow$ TRUE

$$
\begin{array}{ll}
{E_1}^* & \rightarrow \text{TRUE} \\
E_1 \centerdot E_2 & \rightarrow label(\sigma) \in alphabet(E_1) \vee \lambda\text{-}terminate(E_1, \Gamma) \\
|\, x \in \mathrm{X} \,:\, E_1 & \rightarrow param(\sigma)[kappaIndice(F, label(\sigma))] \in \mathrm{X} \\
E_1 |[\Delta]| \, E_2 & \rightarrow label(\sigma) \notin \Delta \,\vee \\
 & \quad (label(\sigma) \in alphabet(E_1) \wedge label(\sigma) \in alphabet(E_2)) \\
|[\Delta]| \; x \in \mathrm{X} \,:\, E_1 & \rightarrow param(\sigma)[kappaIndice(F, label(\sigma))] \in \mathrm{X} \\
(\![\omega]\!) E_1 & \rightarrow \text{TRUE} \\
P \Longrightarrow E_1 & \rightarrow evaluate(P, \Gamma)
\end{array}
$$

where $alphabet(E)$ returns the set of all action labels used in E, $label(\sigma)$ returns the label of σ, $param(\sigma)$ returns the list of the parameters of σ, $kappaIndice(E, l)$ returns the position of the quantified variable in E for the action whose label is l, and $evaluate(P, \Gamma)$ returns the evaluation of the predicate P under the environment Γ. λ-$terminate$ is described below in the paragraph of sequence operator.

In the sequel, we illustrate the algorithm for each operator.

Choice. The choice operator is permissive since if one of its operands can execute the query, then the choice can also execute it. For instance, when $\sigma = \mathsf{a}$, the choice operator is permissive in $(\mathsf{b} \centerdot \mathsf{a}) \mid \mathsf{b}$.

Kleene closure. This operator is always permissive by definition. Indeed, if E can execute sigma, then E^* can execute sigma. That is why an error is never due to the Kleene closure. When $\sigma = \mathsf{a}$, the Kleene closure is permissive in $(\mathsf{b} \centerdot \mathsf{a})^*$.

Sequence. The sequence operator may be permissive depending on the location of the label of σ in its operands. Semantically, there are two cases. In the first case, the action occurs in the left operand ; then the sequence operator is permissive, since the left operand should be able to execute the action ; hence, the sequence is not responsible for the error. For instance, let $\sigma = \mathsf{a}$ in $(\mathsf{b} \centerdot_1 \mathsf{a}) \centerdot_2 \mathsf{b}$. The first occurrence $\centerdot_1$ is not permissive, since a doesn't occur in the left operand ; the second occurrence $\centerdot_2$ is permissive, since a occurs in its left operand.

In the second case of permissiveness, the action occurs in the right operand and the left operand can terminate successfully by executing a sequence of the internal action λ, a condition which we denote by λ-$terminate$. For instance, let $\sigma = \mathsf{a}$ in $\mathsf{b}^* \centerdot_1 (\mathsf{b} \centerdot_2 \mathsf{a})$. The first occurrence $\centerdot_1$ is permissive, since b^* can λ-$terminate$, while occurrence $\centerdot_2$ is not permissive, since b can not λ-$terminate$. For the sake of concision, the definition of λ-termination is omitted ; the reader is referred to [6] for a complete definition. It can be computed in $O(n)$, where n is the size of the AST. λ-termination is similar to ϵ-transitions in regular expressions and non-deterministic automata.

Quantified Choice. The quantified choice operator is permissive if the set of the quantification contains the value of the corresponding parameter in the query. Let $\sigma = \mathsf{a}(4)$ in the following. In $|\, x \in 1..3 \,:\, \mathsf{a}(x)$, the quantified choice operator is not permissive. Whereas in $|\, x \in 1..4 \,:\, \mathsf{b}(x) \centerdot \mathsf{a}(x)$ the quantified choice operator is permissive.

Synchronization. Synchronization operators are permissive on non-synchronized actions. If an action is synchronized then all the operands of the synchronization must contain the action. Otherwise the synchronization is declared not permissive. Let $\sigma = \mathsf{a}$ in the following. For $(\mathsf{b} \centerdot \mathsf{a})|[\emptyset]|\,\mathsf{b}$, $|[\emptyset]|$ is permissive. For $(\mathsf{b} \centerdot \mathsf{c})|[\mathsf{c}]|\,(\mathsf{c} \centerdot \mathsf{a})$, $|[\mathsf{c}]|$ is permissive. For $(\mathsf{b} \centerdot \mathsf{a})|[\mathsf{a}]|\,(\mathsf{a} \centerdot \mathsf{b})$, $|[\mathsf{a}]|$ is permissive. For $\mathsf{b}|[\mathsf{a}]|\,\mathsf{a}$, $|[\mathsf{a}]|$ is not permissive because a is not in both sides of the synchronization despite it is in the synchronization set.

Quantified Synchronization. As for quantified choice, quantified synchronization operator is permissive if the set of the quantification contains the value of the corresponding parameter in the query.

Guard. A guard allows executing the guarded expression if, and only if the guard is evaluated to TRUE. In this case, the guard is permissive. In the other case, the guard is declared as not permissive. Let $\sigma = \mathsf{a}$ in the following. For $\text{TRUE} \Longrightarrow (\mathsf{b} \centerdot \mathsf{a})$, the guard is permissive. For $x \geq 0 \Longrightarrow (\mathsf{b} \centerdot \mathsf{a})$ under the environment $([x := 1])$, the guard is permissive. For $\text{FALSE} \Longrightarrow (\mathsf{b} \centerdot \mathsf{a})$, the guard is not permissive. For $x \geq 10 \Longrightarrow (\mathsf{b} \centerdot \mathsf{a})$ under the environment $([x := 1])$, the guard is not permissive. Note that the environment is always permissive ; its value determines the permissiveness of enclosed operators.

Establishing Suspects List.

After the evaluation of the predicate $isPermissive$ for all the witnesses, the suspects list can be determined.

Definition 2.3.4 ***Suspects relative to*** σ
Let E *be a PE,* σ *a query from the user that generates an execution error,* v *a victim of* E *relative to* σ *and* w *its witnesses. Suspects are defined as the list of operators which are in* w *and that are not permissive.*

Example.

The predicate $isPermissive$ is applied on each member of the witnesses list. The operator $\centerdot_1$ is not permissive because σ is located in the right whereas for the operator $\centerdot_2$, σ is on the left as it can be seen at figure 2.2. All others operators are permissive as shown in tableau 2.1.

tableau 2.1 – Suspects relatively to the victim of DisplayNumberOfLoans($John$)

Witness	$isPermissive$	Why ?	Suspect
$\|\|_1$	TRUE	Synchronization ok	no
$\|\|_2$	TRUE	Synchronization ok	no
$\|\|\| mId \in$ MEMBERID :	TRUE	$John \in$ MEMBERID	no
$*_1$	TRUE	Always permissive	no
$\centerdot_1$	FALSE	Right operand and no λ-termination	yes
$\centerdot_2$	TRUE	Left operand	no
$\|\|_3$	TRUE	Synchronization ok	no
$*_2$	TRUE	Always permissive	no

2.3.4 Culprits

For each list of suspects (*i.e.* one list per witness), the head (the nearest operator to the root) is declared guilty. In case this list is empty, no operator is declared guilty, and the algorithm can not provide an error message for the request of the user. In the other case, a

list of culprit operators is established and this concludes the investigations to find the cause of an execution error.

Definition 2.3.5 ***Culprit relative to a victim***
Let E be a PE, σ a query from the user refused by E, v a victim of E for σ and s its suspects. The culprit relative to v is defined as the operator which is in s and that is the closest to the root.

Example.

Since only one suspect was found for DisplayNumberOfLoans($John$), this operator is automatically declared guilty.

2.4 Generating Messages for an Execution Error

After the localization of the cause of an error, the IS must build its error messages in order to explain to the user the cause of the failure to execute his query. This process is divided in several steps, as the message is built for several goals.

2.4.1 Messages from Culprits

For each culprit found, a message is generated according to the type of operator. This message is built using a skeleton that must be completed by information from several origins, like the query, parameters of the query, and elements from the culprit operator (if any). This message aims to report the cause of the error and its context to the user with related information.

Message Skeletons by Operators.

This algorithm uses pattern matching to link operators to message skeletons. Operators which are always permissive are not considered in the algorithm, since they can never be culprits.

Algorithm 2.4.1 $skeletonFromCulprit(\sigma, culprit)$

Description : *Returns the message skeleton for a culprit found for σ.*

input σ **:** *A query from the user.*

input $culprit$ **:** *An operator that was previously declared as culprit.*

let $skeletonFromCulprit(\sigma, culprit) =$ ***match*** $culprit$ ***with***

$E_1 \centerdot E_2$	$\rightarrow$ *"Execution order of actions is not respected."*
$\mid x \in \mathrm{X} \,:\, E_1$	$\rightarrow$ *"The x variable of the query σ is not in* X*."*
$E_1 \vert[\Delta]\vert\, E_2$	$\rightarrow$ *"The system will not be able to execute σ in this process."*
$\vert[\Delta]\vert\; x \in \mathrm{X} \,:\, E_1$	$\rightarrow$ *"The x variable of the query σ is not in* X*."*
$P \Longrightarrow E_1$	$\rightarrow$ *"The constraint P is not verified."*

Example.

For the victim DisplayNumberOfLoans($John$), the culprit operator is $\centerdot_1$. Then the skeleton message associated to this operator is "*Execution order of actions is not respected.*".

2.4.2 Required Action

In order to help the user after the display of the error message, the system must invite him to do something. That is why the algorithm computes *required actions*. These actions are submitted as advice to help the user to execute his initial query.

Definition 2.4.1 ***Required action***
Let E be a PE, σ a query from the user refused by E, v a victim of E for σ and c the culprit associated with v. A required action associated to c to execute σ in E is an action that must be executed before considering to execute σ.

After executing a required action, the user can not assume that σ will be executed without any error. Other attempts to execute σ may fail again, but these attempts may generate other error messages.

Algorithm.

Since there can be several required actions for a single culprit, the algorithm stocks them into a list. The algorithm also takes into account the fact that some actions may not be always executable due to guards or other conditions. That is why the algorithm uses a pair (action, Boolean) to represent a required action. In the case that the Boolean is FALSE, the action can not be executed in the current state of the IS, but is still required in order to execute σ.

Algorithm 2.4.2 $requiredA(exp, \sigma, \Gamma, \phi)$

Description : *Returns a list of required actions for a culprit found for σ.*

input exp **:** *An operator (PE) that is initially a culprit of σ.*

input σ **:** *A query from the user.*

input Γ **:** *An environment of the PE exp.*

input ϕ **:** *A condition associated with the required action, initially* TRUE.

let rec $requiredA(exp, \sigma, \Gamma, \phi)$ =***match*** exp ***with***

$$
\begin{array}{ll}
Action(label, _) & \rightarrow [(exp, \phi)] \\
E_1 \mid E_2 & \rightarrow requiredA(E_1, \sigma, \Gamma, \phi) \oplus requiredA(E_2, \sigma, \Gamma, \phi) \\
E^* & \rightarrow requiredA(E, \sigma, \Gamma, \phi) \\
E_1 \,.\, E_2 & \rightarrow requiredA(E_1, \sigma, \Gamma, \phi) \\
\mid x \in \mathbf{X} \,:\, E_1 & \rightarrow requiredA(E_1, \sigma, \Gamma, \phi) \\
E_1 \mid[\Delta]\mid E_2 & \rightarrow requiredA(E_1, \sigma, \Gamma, \phi) \oplus requiredA(E_2, \sigma, \Gamma, \phi) \\
\mid[\Delta]\mid x \in \mathbf{X} \,:\, E_1 & \rightarrow requiredA(E_1, \sigma, \Gamma, \phi) \\
(\![\omega]\!) E_1 & \rightarrow requiredA(E_1, \sigma, (\Gamma \lhd \omega), \phi) \\
P \Longrightarrow E_1 & \rightarrow requiredA(E_1, \sigma, \Gamma, \phi \wedge P)
\end{array}
$$

where $\oplus$ is the classical concatenation operator for lists and $\lhd$ denotes the environment composition.

Translation to a Recommendation.

Since the user can only execute one action at the same time, only one required action will be executed just after the broadcast of the error message, or none if the user does not take this advice into account. When a required action $(\omega\ , \phi)$ is computed, then the recommendation addressed to the user is : "*In order to execute σ, you should try executing ω under the condition ϕ.*"

2.4.3 Information System Patterns

An IS generated using the EB^3 method is produced from (among other things) an entity-relationship diagram. Several PE patterns based on this diagram are defined in [14]. These patterns may be used to deduce an improved version of the error message.

Definition 2.4.2 ***Entity pattern***
Let $\mathsf{P}(\,x\,)$, $\mathsf{M}(\,x\,)$ and $\mathsf{C}(\,x\,)$ be PEs and let E be a PE that matches the following pattern, where $\mathsf{P}(\,x\,)$ denotes a choice of producers, $\mathsf{M}(\,x\,)$ denotes a choice of modifiers, $\mathsf{C}(\,x\,)$ denotes a choice of consumers and x a key for an entity with x in XID *:*

$$|||\, x \in \text{XID} \,:\, \mathsf{P}(\,x\,) \,.\, \mathsf{M}(\,x\,)^{*} \,.\, \mathsf{C}(\,x\,)$$

In the case that the entity pattern is a sub-expression of the current state of an IS and σ, the query of the user, is a modifier or a consumer relative to this entity pattern, if the system can not execute σ, then the error message skeleton associated to the . operator can be upgraded to : "*The entity x of type E does not exist.*" The translation to a recommendation of the required action can be more precise : "*In order to create it, you should try executing ω under the condition ϕ.*" with $(\omega\ , \phi)$ as computed by the $requiredA$ algorithm.

Definition 2.4.3 1-n ***relationship pattern***

Let $\mathsf{P}_1(\,x\,)$ and $\mathsf{C}_1(\,x\,)$ be PEs and let E_1 be a PE that matches the following pattern :

$$E_1 = |||\, x \in \text{XID} \,:\, \mathsf{P}_1(\,x\,) \,.\, (|||\, y \in \text{YID} \,:\, \mathsf{A}(\,x, y\,))^{*} \,.\, \mathsf{C}_1(\,x\,)$$

Let $\mathsf{P}_2(\,y\,)$ *and* $\mathsf{C}_2(\,y\,)$ *be PEs and let* E_2 *be a PE that matches the following pattern :*

$$E_2 = |||\, y \in \textsc{yId} \,:\, \mathsf{P}_2(\,y\,) \,\centerdot\, (|\, x \in \textsc{xId} \,:\, \mathsf{A}(\,x, y\,))^{*} \,\centerdot\, \mathsf{C}_2(\,y\,)$$

Let $F = E_1 \,||\, E_2$*, then* F *is said to follow the* 1-n *relationship pattern.*

In the case that the 1-n relationship pattern is a sub-expression of the current state of an IS and σ, the query of the user, is an action of A, if the system can not execute σ, then the error message skeleton associated to one of the $\centerdot$ operators of E_1 or E_2 can be upgraded to : "*The relationship* A *between the entity* x *of type* E_1 *and the entity* y *of type* E_2 *does not exist. You can create it by executing* ω*.*" with $(\omega\ , \phi)$ as computed by the $requiredA$ algorithm.

This message may even become more pertinent with a label name associated to both entities and the 1-n relationship. As a good practice, relationship names should be used as process names in the formal specification.

The n-n relationship pattern is similar to the 1-n relationship pattern except that the E_2 PE matches the following pattern :

$$|||\, y \in yId \,:\, \mathsf{P}_2(\,y\,) \,\centerdot\, (|||\, x \in xId \,:\, \mathsf{F}(\,x, y\,))^{*} \,\centerdot\, \mathsf{C}_2(\,y\,)$$

Example.

In our case, the culprit operator is $\centerdot_1$ and above it there is a quantified interleaving in the list of witnesses. This is the entity pattern, so the skeleton message of $\centerdot_1$ can be replaced. The required action algorithm applied to our culprit generate only one element in the list : ($\mathsf{Join}(\,mId\,)$, TRUE). Then we can build the full error message that will be displayed : "*The '*$member$*' entity '*$John$*' does not exist. In order to create it, you should try executing* $\mathsf{Join}(\,mId\,)$*.*"

2.5 Case Study

This section will apply the process described above to other errors.

2.5.1 Current State

Since a system evolves from its initial state, and in order to illustrate some more complex cases of execution error, a state, different from the initial state (see figure 2.1) is used in the following cases.

First, BOOKID is defined as the set of naturals between 0 and 9, MEMBERID is defined as the set of naturals between 10 and 19. Then, the library is populated with 2 books (with bId 0 and 1) and 3 members (with mId 10, 11 and 12). To finish, member 10 has borrowed book 0, just returned book 1 and members 11 and 12 have made reservation for book 1 in this order.

In short, the current state used below is the state resulting from the execution of the actions Acquire(0), Acquire(1), Join(10), Join(11), Join(12), Lend($10, 0$), Lend($10, 1$), Reserve($11, 1$), Reserve($12, 1$) and Return($10, 1$) in this order upon the initial specification.

2.5.2 Generated Messages

Trying to Borrow a Currently Borrowed Book.

If a member wants to borrow a book that is already borrowed by another member, the system should refuse the query and warn the user about this error. The user could expect from a system with "human-generated" error messages a message that explains to him that the book is not available.

Applying the "police investigation" method on this state while attempting to execute Lend($11, 0$) produces a list of seven victims (with six environmental alibis found). Since several victims can share the same culprit, fewer messages will be generated. Only one victim is matching the exact query from the user for both environments and quantification sets. Here is the complete execution of the algorithm for this victim.

From this culprit, the algorithm provides a skeleton message that is used to generate the first part of the message : "*Execution order of actions is not respected.*"

Then, in addition to this first part are generated required actions for this culprit. The execution of the algorithm returns the action Return($10, 0$) with condition TRUE that we can translate into : "*In order to execute* Lend($11, 0$)*, you should try executing* Return($10, 0$)*.*"

tableau 2.2 – Investigation over Lend($11, 0$)

Witnesses	Suspect	Why ?	Culprit	Pattern
$\vert\vert\vert$	no	Always permissive	-	-
$([bId := 0])$	no	Always permissive	-	-
$\cdot$	yes	Right operand and no λ-termination	yes	-
$\vert\, mId \in [10, 19]\,$:	no	$mId \in$ [10,19]	-	-

Since no known pattern is detected in this investigation, the final message is generated : " *Execution order of actions is not respected. In order to execute* Lend($11, 0$), *you should try executing* Return($10, 0$). "

Trying to Bypass the Reservation List.

In case that the user of the system tries to execute an action protected by a FALSE-evaluated guard, the system must block the execution of the query. In the library, this can happen when a user wants to bypass the reservation process by taking a book while he is not the first on the reservation list.

Trying to execute Take($12, 1$) in the current state where member 11 is first in the reservation list produces the investigation shown in the tableau 2.3.

tableau 2.3 – Investigation over Take($12, 1$)

Witnesses	Suspect	Why ?	Culprit	Pattern
$\vert\vert$	no	Synchronization ok	-	-
$\vert\vert\vert$	no	Always permissive	-	-
$([mId := 12])$	no	Always permissive	-	-
$\vert\vert$	no	Synchronization ok	-	-
$\vert\vert\vert$	no	Always permissive	-	-
$([bId := 1])$	no	Always permissive	-	-
$\cdot$	no	Left operand	-	-
$isFirst(trace, mId, bId) \Longrightarrow$	yes	FALSE-evaluated	yes	-

In this case, there are no required action before executing Take($12, 1$), the guard is the only thing that prevents the user to execute the query : "*In order to execute* Take($12, 1$)*, the predicate* $isFirst(trace, mId, bId)$ *must be evaluated to* TRUE*.*"

It could be interesting to point to the user of the IS exactly what makes the guard evaluated to FALSE. In the library specification, the predicate $isFirst(trace, mId, bId)$ is specified as a recursive function over the trace of the IS. The trace of the system consists of a sequence of all the actions executed since the launch of the system. In our example, we could provide to the user the name of the member that is the first in the reservation list. On the other hand, this raises the issue of confidentiality ; hence, there is a need for configurability when selecting messages to display. Another analysis that could be done is to determine, from the definition of the function $isFirst$, which actions can make the guard true. This means solving a predicate, which is a hard problem in the general case, but simple heuristics could probably solve a large number of simple cases.

2.6 Conclusion

We have presented an algorithm that can generate an error message for an event refused by a process expression. The algorithm is generic, in that it can be used on any process expression. The algorithm identifies the causes of an error and suggests corrective actions.

Currently, nothing can ensure that a required action will be executable. This limitation may be annoying for the user of the system since he can not receive useful advice. But if he tries to execute the required action, he can receive another error message that will help him to execute his initial request.

In the case of non-fulfilled guards, the user may not be able to know what to do in order to comply with a guard. Indeed, nothing indicates how to modify the state of the system or the value of attributes that are involved in the guard. By finding which attributes are involved in a guard and providing a list of actions that modify these attributes, the error message could help the user in order to satisfy a guard.

Another limitation could be the names of the variables and actions of the PE. Indeed, without a pertinent name, error message will integrate impertinent information that will not be easy to understand by the user. That is why this method requires an active involvement of the designer of the system by using pertinent and understandable names in his specification.

The message skeletons we have provided are rather simple. They can be enhanced ; for instance, we can compute the query parameter associated to the quantification variable mentioned in a skeleton. We could use this parameter name instead (each action has a signature with parameter names). One can also imagine that annotations could be added to the specification which could automatically be included in the skeletons to provide more information. Guard predicates could also be translated into a natural language representation. They could also be further analysed to target specific subpredicates which make the overall predicate false.

To improve readability, a selection of one among all the generated messages according to a heuristic could be interesting. This heuristic could vary depending on the target and the goal of the message. However, this feature has not been developed for the moment.

The approach of automatic deduction of pertinent error messages from EB^3 specification could be adapted to other specification languages such as CSP [15]. Indeed, the method to find the cause of an error could be transposed to another process algebra, with some modifications to the "permissive" and the "message skeleton" parts.

CSP operators for sequence "$\rightarrow$" and ";" are similar to EB^3 operator "**.**". Hence, the permissiveness and the message skeleton of these operators would not change. CSP external choice operator "$\Box$" is also similar to EB^3 "$|$" whereas CSP's internal choice "$\sqcap$" has no equivalent in EB^3. This operator would need a new predicate to evaluate its permissiveness and a new message skeleton.

Chapitre 3

Cas d'étude

Dans ce chapitre sont présentés différents cas d'études illustrant les résultats obtenus après l'exécution de l'implémentation en OCaml des algorithmes décrits dans le chapitre 2. Des exemples appliqués à un système concret ayant été présentés à la section 2.5, sont détaillés ici des cas plus abstraits.

Pour plus de détails sur l'implémentation OCaml des algorithmes de génération automatique de messages d'erreur, le lecteur est invité à se référer à [18]

3.1 Entrelacement et environnement

Ce cas d'étude présente le comportement des algorithmes de génération de messages d'erreur sur un entrelacement et un environnement issu de cet entrelacement.

3.1.1 Description du cas

Soit l'état initial d'un système d'information décrit par l'expression de processus de la figure 3.1 consistant en l'entrelacement de quatre processus se décomposant en deux actions à exécuter en séquence. Après l'exécution de l'action Aa(4), le système se retrouvera dans un état décrit par la figure 3.2, d'après les règles de transition de l'algèbre de processus EB^3.

$$||| \; xId \in [1,4] \; : \; \mathsf{Aa}(\, xId \,) \,.\, \mathsf{Bb}(\, xId \,)$$

figure 3.1 – Entrelacement à l'état initial

$$(\![xId := 4]\!) \; \mathsf{Bb}(\, xId \,)$$
$$|||$$
$$||| \; xId \in [1,3] \; : \; \mathsf{Aa}(\, xId \,) \,.\, \mathsf{Bb}(\, xId \,)$$

figure 3.2 – Entrelacement après l'exécution de $\mathsf{Aa}(\,4\,)$

À partir de l'état du système de la figure 3.2, l'utilisateur tente d'exécuter l'action $\mathsf{Bb}(\,1\,)$.

3.1.2 Comportement espéré

L'utilisateur veut exécuter $\mathsf{Bb}(\,1\,)$ mais le système ne le permet pas dans son état actuel. Deux cas sont possibles dans l'interprétation de cette tentative d'exécution. Soit l'utilisateur voulait exécuter Bb mais avec un autre paramètre valide (4 dans cet exemple), soit il voulait exécuter $\mathsf{Bb}(\,1\,)$ mais ne savait pas que le système n'était pas disposé à l'accepter.

Ainsi le développeur devrait choisir parmi deux messages d'erreur différents dans le cadre d'une méthode conventionnelle de développement. Le premier afficherait le message "*Le paramètre entré n'est pas correct*" et indiquerait que l'utilisateur peut reformuler sa requête en $\mathsf{Bb}(\,4\,)$. Le second message attendu serait de dire que "*Pour pouvoir exécuter* $\mathsf{Bb}(\,1\,)$ *il faut d'abord exécuter* $\mathsf{Aa}(\,1\,)$".

3.1.3 Résultat obtenu

La méthode de génération automatique fait un choix. Elle privilégie la solution assumant que l'utilisateur veut exécuter l'action entrée telle quelle. De ce fait, le message d'erreur obtenu sera à comparer à "*Pour pouvoir exécuter* $\mathsf{Bb}(\,1\,)$ *il faut d'abord exécuter* $\mathsf{Aa}(\,1\,)$".

L'*investigation* sur cet exemple calcule les listes des *témoins*, présenté à la figure 3.3, et les listes des *suspects* et *coupables*, indiquées à la figure 3.4. Deux listes d'opérateurs témoins sont générées, une pour chaque occurrence de l'action Bb dans l'expression de

```
******* Opérateurs temoins *******

([xId:=4])  (1)
|[{}]|      (2)

*****

.                 (3)
|[{}]| xId:{1,2,3}  (4)
|[{}]|            (5)

*****
```

figure 3.3 – Témoins dans l'investigation sur l'entrelacement pour l'action Bb(1)

```
******* Opérateurs suspects *******
Aucun suspect pour cette occurence
*****

.   (3)
*****

******* Coupables *******

.   (3)
```

figure 3.4 – Suspects et coupables dans l'investigation sur l'entrelacement pour l'action Bb(1)

processus. Elles sont séparées dans la figure 3.3 par cinq étoiles. Si deux occurrences de l'action Bb sont détectées, la première est retirée de l'investigation du fait de l'environnement affectant la valeur 4 à la variable xId. Suite au choix fait de privilégier l'entrée de l'utilisateur dans son entier, on qualifie d'*alibi* cet environnement et l'occurrence de l'action est retirée de l'investigation. De ce fait, aucun opérateur n'est suspect pour cette victime et il n'y a donc pas de coupable. L'opérateur . (3) est le seul suspect pour la seconde occurrence, il est donc désigné coupable.

Le message d'erreur résultant de l'opérateur . (3) est alors retourné "*L'ordre d'exécution des actions n'est pas respecté. Avant d'envisager l'exécution de* Bb(1) *vous devez exécuter* Aa(1)". Cependant, l'algorithme détecte la présence du patron « entité » puisque le coupable (la séquence .) possède un entrelacement quantifié ($|||\ xId \in [1,3]\ :\ $) comme

ascendant. De ce fait le message retourné est "*L'entité 1 n'existe pas. Vous pouvez la créer en exécutant* Aa(1)".

3.2 Gardes

Ce cas d'étude présente le comportement des algorithmes de génération de messages d'erreur sur une expression de processus composée de deux gardes évaluées simultanément à deux valeurs différentes (l'une est vraie alors que l'autre est fausse).

3.2.1 Description du cas

Cet exemple illustre le comportement de l'algorithme de génération des messages d'erreur avec les gardes. Dans ce cas présenté à la figure 3.5, les gardes sont triviales et ne dépendent pas de paramètres. Ce cas particulier est présenté à des fins d'illustration. L'utilisation de gardes non triviales ne modifie pas le comportement de l'algorithme dans le cas général.

$$\begin{array}{ll} & \mathsf{Cc}(1) \; ||| \; 1 > 6 \Longrightarrow (\mathsf{Dd}(1) \; ||| \; (\mathsf{Cc}(1) \,.\, \mathsf{Bb}(1))) \\ \| & \\ & \mathsf{Cc}(1) \; ||| \; 6 > 1 \Longrightarrow (\mathsf{Dd}(1) \,.\, (\mathsf{Cc}(1) \,.\, \mathsf{Bb}(1))) \end{array}$$

figure 3.5 – Expressions gardées

3.2.2 Comportement espéré

Dans le cadre de cette expression de processus, les deux occurrences de Bb(1) ne sont pas exécutables. La première est gardée par une garde évaluée à FALSE, la seconde doit être précédée de l'action Cc(1). Deux messages d'erreur pourraient donc être rédigés mais l'opérateur || complique la tâche. Il faut en effet exécuter Bb(1) de manière synchronisée. De ce fait, le développeur doit dire à l'utilisateur que les actions Dd(1) et Cc(1) doivent être exécutées dans cet ordre.

3.2.3 Résultat obtenu

L'investigation sur l'expression de processus retourne les résultats indiqués à la figure 3.6 pour les témoins et à la figure 3.7 pour les suspects et les coupables.

Deux listes d'opérateurs témoins sont générées, une pour chaque occurrence de l'action Bb(1) dans l'expression de processus. Pour chacune, plusieurs suspects en sont extraits. Les coupables sont alors les suspects de chaque liste les plus proches de la racine de l'expression de processus, ici la garde $isGreater(1,6)$ (3) et **.** (7).

```
******* Opérateurs temoins *******

.                    (1)
|[{}]|               (2)
isGreater(1,6)=>     (3)
|[{}]|               (4)
|[{Bb,Cc,Dd}]|       (5)

*****

.                    (6)
.                    (7)
isGreater(6,1)=>     (8)
|[{}]|               (9)
|[{Bb,Cc,Dd}]|       (10)

*****
```

figure 3.6 – Témoins dans l'investigation sur les gardes pour l'action Bb(1)

Cette investigation conduit à la génération de deux messages d'erreur. Chacun est relatif à l'un des coupables détectés.

Le premier basé sur la garde $isGreater(1,6)$ (3) déclarée coupable est "*Afin de pouvoir exécuter* Bb*(1), le prédicat* $1 > 6$ *doit être évalué à vrai*". Le second message basé sur l'opérateur **.** (7) indique "*L'ordre d'exécution des actions n'est pas respecté. Avant d'envisager l'exécution de* Bb*(1), vous devez d'abord exécuter* Dd*(1)*". Ainsi, le message d'erreur n'indique pas la nécessité de l'exécution de Cc(1) avant celle de Bb(1).

3.3 Parenthésage et séquence

Ce cas d'étude a pour but de montrer que le parenthésage d'une suite de séquence de l'expression de processus n'influe pas les résultats de l'algorithme.

```
 ******* Opérateurs suspects *******

.                  (1)
isGreater(1,6)=>   (3)

*****

.                  (6)
.                  (7)

*****


 ******* Coupables *******

isGreater(1,6)=>   (3)
.                  (7)
```

figure 3.7 – Suspects et coupables dans l'investigation sur les gardes pour l'action Bb(1)

3.3.1 Associativité de la séquence

L'opérateur de séquence de l'algèbre de processus EB[3] est associatif. En effet, soient E_1, E_2 et E_3 trois expressions de processus, alors si $(E_1 \,.\, E_2) \,.\, E_3$ peut exécuter la séquence d'action $\mathfrak{s}$, ce que l'on note $(E_1 \,.\, E_2) \,.\, E_3 \overset{\mathfrak{s}}{\rightsquigarrow} \boxdot$, alors $E_1 \,.\, (E_2 \,.\, E_3) \overset{\mathfrak{s}}{\rightsquigarrow} \boxdot$ et réciproquement si $E_1 \,.\, (E_2 \,.\, E_3) \overset{\mathfrak{s}}{\rightsquigarrow} \boxdot$ alors $(E_1 \,.\, E_2) \,.\, E_3 \overset{\mathfrak{s}}{\rightsquigarrow} \boxdot$. Cela se démontre aisément en décomposant $\mathfrak{s}$ de sorte que $\mathfrak{s} = \mathfrak{s}1 \frown \mathfrak{s}2 \frown \mathfrak{s}3$ où $E_1 \overset{\mathfrak{s}1}{\rightsquigarrow} \boxdot$, $E_2 \overset{\mathfrak{s}2}{\rightsquigarrow} \boxdot$ et $E_3 \overset{\mathfrak{s}3}{\rightsquigarrow} \boxdot$.

3.3.2 Description du cas

Soient les expressions de processus (Aa(1).Bb(1)).Cc(1) notée E_1, et Aa(1).(Bb(1). Cc(1)), notée E_2 par la suite. Ces deux expressions de processus sont équivalentes par associativité.

3.3.3 Comportement espéré

Aucune de ces deux expressions de processus ne peut exécuter l'action Bb(1). Cependant, un message d'erreur identique devrait être proposé dans les deux cas. L'utilisateur du SI s'attend donc à recevoir un message d'erreur lui indiquant d'exécuter Aa(1) préalablement à Bb(1).

3.3.4 Résultat obtenu

Le tableau 3.1 présente les résultats de l'algorithme d'investigation sur Bb(1) pour E_1 et E_2 respectivement. Dans les deux cas l'opérateur $\cdot_1$ est désigné coupable, mais il n'apparaît pas à la même profondeur dans l'AST de E_1 et E_2.

tableau 3.1 – Investigation sur Bb(1) pour E_1 et E_2

(Aa(1) $\cdot_1$ Bb(1)) $\cdot_2$ Cc(1)			Aa(1) $\cdot_1$ (Bb(1) $\cdot_2$ Cc(1))		
Témoin	Suspect	Coupable	Témoin	Suspect	Coupable
$\cdot_2$	non	non	$\cdot_1$	oui	oui
$\cdot_1$	oui	oui	$\cdot_2$	non	non

Or l'algorithme de désignation de l'action nécessaire, présenté en 2.4.2, se base sur l'opérateur coupable et ses opérandes, différents opérandes peuvent donc conduire à des résultats différents. Cependant, l'algorithme est appliqué sur le coupable, qui est l'opérateur suspect le plus proche de la racine de l'AST, ce qui conduit à désigner la même action requise dans les deux cas.

Ainsi, l'action nécessaire retournée sera Aa(1), ce qui conduit à la génération d'un message d'erreur identique dans les deux cas. Le message "*L'ordre d'exécution des actions n'est pas respecté. Avant d'envisager l'exécution de* Bb(1)*, vous devez d'abord exécuter* Aa(1)" sera retourné à l'utilisateur.

3.4 Influence de la spécification sur le message d'erreur

Se basant sur l'expression de processus courante du SI, l'algorithme de génération automatique de messages d'erreur est influencé par la façon dont la spécification est rédigée. Ce cas d'étude a pour objet de montrer cette influence.

3.4.1 Description du cas

Soient $zero?$, $one?$ et $two?$ les prédicats sur la trace d'un SI qui sont évalués à FALSE sauf lorsque la trace du système est de longueur 0, 1 et 2 respectivement. Soient les expres-

sions de processus $(\mathsf{Aa}(1) \,.\, \mathsf{Bb}(1)) \,.\, \mathsf{Cc}(1)$, notée E_1 et $(zero? \implies \mathsf{Aa}(1) \mid one? \implies \mathsf{Bb}(1) \mid two? \implies \mathsf{Cc}(1))^*$, notée E_2. Les deux expressions de processus E_1 et E_2 acceptent la même trace $\mathfrak{s} = \mathsf{Aa}(1)^\frown\mathsf{Bb}(1)^\frown\mathsf{Cc}(1)$.

3.4.2 Comportement espéré

Dans le cadre de la tentative d'exécution de $\mathsf{Bb}(1)$ qui s'avère impossible, les deux expressions de processus devraient retourner le même message d'erreur. En effet, dans les deux cas l'utilisateur doit en premier lieu exécuter $\mathsf{Aa}(1)$.

3.4.3 Résultat obtenu

Le message d'erreur généré pour E_1, déjà généré à la sous-section 3.3.4, est "*L'ordre d'exécution des actions n'est pas respecté. Avant d'envisager l'exécution de* $\mathsf{Bb}(1)$*, vous devez d'abord exécuter* $\mathsf{Aa}(1)$". Concernant E_2, le coupable désigné est la garde $one? \implies$ qui prévient l'exécution de $\mathsf{Bb}(1)$, mais il n'est pas possible de déterminer une unique action requise. En effet, les trois actions $\mathsf{Aa}(1)$, $\mathsf{Bb}(1)$ et $\mathsf{Cc}(1)$ associées respectivement aux prédicats $zero?$, $one?$ et $two?$ seront retournées en tant qu'action nécessaire. L'utilisateur devra faire un choix parmi ces trois actions. Une analyse des prédicats et de leur sens serait alors requise pour proposer à l'utilisateur un message complet et lui indiquer quelle action, si elle existe, inversera la valeur du prédicat coupable. Ainsi, le message d'erreur généré pour E_2 est "*Afin de pouvoir exécuter* $\mathsf{Bb}(1)$*, le prédicat* $one?$ *doit être évalué à vrai*", message moins pertinent et utile que celui généré pour E_1.

Conclusion

Contributions

Cette méthode de génération automatique de messages d'erreur automatise et formalise une des étapes souvent négligées lors du développement de SI. Elle permet de garantir à l'utilisateur qu'un message d'erreur sera généré du fait de son approche systématique. De plus, elle ne nécessite pas de travail complémentaire de la part du concepteur de SI pour fonctionner ; aucune réécriture, adaptation ou modification de spécification n'est nécessaire.

Critique du travail

On peut reprocher aux messages d'erreur générés par cette méthode d'être moins pertinents que s'ils avaient été développés de manière conventionnelle puisqu'ils ne peuvent pas intégrer autant de détails qu'un message pré-établi et adapté à une situation précise. La qualité du message peut également être fortement réduite si le concepteur du système n'a pas intégré dans sa spécification des noms pertinents pour ses actions et ses entités vu que ces noms sont repris dans les messages générés. De même, l'utilisation des gardes limite grandement les possibilités offertes par l'algorithme proposé. En effet, il est difficile de proposer un message d'erreur plus pertinent quand le sens d'une garde n'est pas connu.

Une autre limitation importante vient du fait que le système ne peut garantir à l'utilisateur que l'action proposée dans le cadre du diagnostic soit exécutable. Dans un système à utilisation concurrente, une action requise peut ne plus être exécutable suite à une requête d'un autre utilisateur, ou d'une autre erreur. Les gardes introduisent également un

problème d'indécidabilité induit par le calcul de prédicats du premier ordre. L'affichage d'une action requise ne doit donc pas être interprété comme une solution mais comme une étape nécessaire pour aboutir à l'exécution souhaitée.

Travaux futurs de recherche

Des évolutions du système sont possibles notamment pour combler les limitations pointées. Il serait possible de donner des recommandations d'ordre général au concepteur de spécifications pour que celles-ci soient optimisées dans le cadre de la production de messages d'erreur ; le choix d'utiliser le moins de gardes possibles par exemple, ou de privilégier les patrons de conception classiques d'entité-relation.

En analysant les gardes, l'algorithme pourrait également calculer quels sont les attributs impliqués et quelles seraient les actions permettant de les faire varier pour remplir les conditions demandées. De plus, les algorithmes de traduction des propositions logiques en langage naturel pourraient aider à la compréhension de l'erreur.

Une autre piste serait celle de l'évaluation de la qualité d'un message d'erreur. Il serait alors possible de n'afficher que le message d'erreur que l'algorithme considérera comme le plus pertinent selon divers critères : présence d'un diagnostic ; évaluation selon le type d'opérateur mis en cause ; présence d'un patron... Cette évaluation pourrait être adaptée selon les résultats produits et le comportement de l'utilisateur : le message d'erreur pourrait alors être différent si jamais la même erreur se reproduit, afin d'essayer une autre piste de résolution en cas de difficultés.

Enfin, l'algorithme pourrait être adapté à d'autres langages de spécification formelle, en adaptant les squelettes de messages associés aux opérateurs et aux patrons. Un lien avec la méthode B [1] par le biais de l'animateur B pourrait également être développé. Il a été constaté que l'animateur B indiquait à tout instant quelles actions pouvaient être exécutées ou non. Toutes les actions non exécutables pourraient alors se voir associées à un message expliquant la raison pour laquelle l'action ne peut être exécutée.

Bibliographie

[1] Jean Raymond ABRIAL. The B-Book : Assigning Programs to Meanings. Cambridge University Press, 1996.

[2] Alfred V. AHO, Ravi SETHI et Jeffrey D. ULLMAN. Compilers : Principles, Techniques, and Tools. Addison Wesley, 1986.

[3] Élodie ANTOINE. « Génération automatique d'interfaces Web à partir de spécifications : l'outil DCI-WEB ». Mémoire de maîtrise, Université de Sherbrooke, Sherbrooke, Québec, Canada, février 2008.

[4] Panawé BATANADO. « Synthèse de transactions de base de données relationnelle à partir de définitions d'attributs EB^3 ». Mémoire de maîtrise, Université de Sherbrooke, Sherbrooke, Québec, Canada, juin 2005.

[5] Philip J. BROWN. « Error messages : the neglected area of the man/machine interface ». Communications of the ACM, 26(4) :246–249, 1983.

[6] Benoît FRAIKIN. « Interprétation efficace d'expression de processus EB^3 ». Thèse de doctorat, Département d'informatique, Université de Sherbrooke, Sherbrooke, Québec, Canada, avril 2006.
http ://www.dmi.usherb.ca/∼gril/doc/eb3pai/pdf/06-thesis.pdf.

[7] Benoît FRAIKIN et Marc FRAPPIER. « EB^{3}PAI : an Interpreter for the EB^3 Specification Language ». Dans D. HANEBERG, G. SCHELLHORN et W. REIF, éditeurs, 5th Workshop on Tools for System Design and Verification (FM-TOOLS 2002), proceedings, Reisensburg Castle, Günzburg, Germany, juin 2002.

[8] Benoît FRAIKIN et Marc FRAPPIER. « Efficient Execution of Process Expressions Using Symbolic Interpretation ». Rapport Technique 8, Université de Sherbrooke,

Département d'informatique, Sherbrooke, Québec, Canada, janvier 2005. http ://www.dmi.usherb.ca/~gril/doc/eb3pai/pdf/05-TR8.pdf.

[9] Benoît FRAIKIN et Marc FRAPPIER. « Efficient Interpretation of Large Quantifications in a Process Algebra ». Dans 4th International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems (MSVVEIS-2006), proceedings. INSTICC Press, may 2006.

[10] Benoît FRAIKIN et Marc FRAPPIER. « Efficient Symbolic Execution of Large Quantifications in a Process Algebra ». Dans Jim WOODCOCK et Jin Song DONG, éditeurs, 9th International Conference on Formal Engineering Methods (ICFEM 2007), volume 4789 de LNCS, pages 327–344. Springer Berlin/Heidelberg, novembre 2007.

[11] Benoît FRAIKIN, Marc FRAPPIER et Régine LALEAU. « State-Based versus Event-Based Specifications for Information System Specification : a comparison of B and EB3 ». Software and System Modeling, 4(3) :236–257, juillet 2005.

[12] Benoît FRAIKIN, Frédéric GERVAIS, Marc FRAPPIER, Régine LALEAU et Mario RICHARD. « Synthesizing Information Systems : the APIS Project ». Dans Colette ROLLAND, Oscar PASTOR et Jean-Louis CAVARERO, éditeurs, First International Conference on Research Challenges in Information Science (RCIS), page 12, Ouarzazate, Morocco, avril 2007.

[13] Marc FRAPPIER, Benoît FRAIKIN, Régine LALEAU et Mario RICHARD. « Automatic Production of Information Systems ». Dans AAAI Symposium on Logic-Based Program Synthesis, page 7, Stanford University, Stanford, CA, mars 2002.

[14] Marc FRAPPIER et Richard ST-DENIS. « EB^3 : an entity-based black-box specification method for information systems ». Software and Systems Modeling, 2(2) :134–149, 2003.

[15] Charles Antony Richard HOARE. CSP–Communicating Sequential Processes. Prenctice Hall, 1985.

[16] Pierre KONOPACKI. « Synthèse automatique de gardes EB^3 ». Mémoire de maîtrise, Université de Sherbrooke, Sherbrooke, Québec, Canada, février 2008.

[17] Xavier LEROY et Pierre WEIS. Manuel de référence du langage Caml. InterEditions, 1993.

[18] Jérémy MILHAU et Benoît FRAIKIN. « Technical report 25, An algorithm for automatic generation of error messages for EB^3 ». Rapport Technique, Université de Sherbrooke, Département d'informatique, Sherbrooke, Québec, Canada, septembre 2008.

[19] Robin MILNER. Communication and concurrency. Prentice-Hall, Inc., 1989.

[20] James RUMBAUGH, Ivar JACOBSON et Grady BOOCH. The unified modeling language. University Video Communications, 1996.