INSTITUT DE RECHERCHE EN INFORMATIQUE DE NANTES

Intégration de données formelles dans les diagrammes d'états d'UML

Christian ATTIOGBÉ[*], Pascal POIZAT[**], Gwen SALAÜN[*]

[*] IRIN, Université de Nantes
2 rue de la Houssinière, B.P. 92208, 44322 Nantes Cedex 3, France
email: {attiogbe,salaun}@irin.univ-nantes.fr

[**] LaMI - UMR 8042, Université d'Évry Val d'Essonne
Tour Évry 2, 523 Place des terrasses de l'Agora, 91000 Évry, France
email: poizat@lami.univ-evry.fr

— Méthodes et Spécifications Formelles —

RAPPORT DE RECHERCHE

N° 02.3

Juillet 2002

Christian ATTIOGBÉ*, Pascal POIZAT**, Gwen SALAÜN*

Intégration de données formelles dans les diagrammes d'états d'UML

26 p.

Les rapports de recherche de l'Institut de Recherche en Informatique de Nantes sont disponibles aux formats PostScript® et PDF® à l'URL :

http://www.sciences.univ-nantes.fr/irin/Vie/RR/

Research reports from the Institut de Recherche en Informatique de Nantes are available in PostScript® and PDF® formats at the URL:

http://www.sciences.univ-nantes.fr/irin/Vie/RR/indexGB.html

Intégration de données formelles dans les diagrammes d'états d'UML

Christian ATTIOGBÉ*, Pascal POIZAT**, Gwen SALAÜN*

Résumé

Dans ce rapport, nous présentons une approche générique pour intégrer des données exprimées dans des langages de spécification formelle à l'intérieur de diagrammes d'états d'UML. Les motivations principales sont d'une part de pouvoir modéliser les aspects dynamiques des systèmes complexes avec un langage convivial et graphique tel que les diagrammes d'états d'UML ; d'autre part de pouvoir spécifier de façon formelle et potentiellement à un haut niveau d'abstraction les données mises en jeu dans ces systèmes à l'aide de spécifications algébriques ou de spécifications orientées état (Z, B). Cette approche introduit une utilisation flexible et générique des données. Elle permet aussi de prendre en compte différentes sémantiques associées aux diagrammes d'états. Nous présentons d'abord les fondements formels de notre approche puis une étude de cas vient illustrer le pragmatisme de la proposition.

Catégories et descripteurs de sujets : D.2.1 [**Software Engineering**]: Requirements/Specifications; D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.3 [**Programming Languages**]: Language Constructs and Features

Termes généraux : Design, Languages, Theory

Mots-clés additionnels et phrases : Intégration de spécifications formelles et semi-formelles, diagrammes d'états, UML, spécifications algébriques, Z, B

Table des matières

1 Introduction

Depuis quelques années, la notation UML [48] est devenue omniprésente en conception orientée-objet et ce, principalement grâce à sa convivialité (notations graphiques) et sa relative souplesse d'emploi. Elle permet de décrire les différents aspects (statiques, dynamiques, fonctionnels) des systèmes complexes grâce à ses différents types de diagrammes. Cependant, UML souffre d'une critique incessante concernant son manque de sémantique formelle [52, 9], ce qui pose problème pour s'assurer de la cohérence des systèmes décrits à l'aide de plusieurs diagrammes et plus généralement pour les étapes de vérification qui suivent la modélisation de ces systèmes. La formalisation d'UML fait d'ailleurs aujourd'hui l'objet de nombreux travaux dont les principaux sont issus du Precise UML Group [28].

D'un autre côté, les méthodes formelles existent maintenant depuis plusieurs décennies et ont pour finalité la description de systèmes logiciels mais de façon plus *mathématique*. Leur principal intérêt est que la sémantique des langages de spécification est bien définie et permet donc de vérifier la correction des systèmes spécifiés. Les spécifications formelles permettent aussi la description des systèmes à un haut niveau d'abstraction. En contrepartie, leur difficulté d'apprentissage et d'utilisation leur a souvent été reprochée.

L'utilisation conjointe des langages de spécifications formelles et semi-formelles semble donc être un domaine prometteur avec pour objectif de profiter des avantages de ces deux types d'approches. Dans cet article, nous proposons une approche qui répond à cette motivation. Elle permet de spécifier à l'aide d'un langage intégré les différents aspects des systèmes complexes. Les aspects statiques et fonctionnels sont spécifiés à l'aide de langages de spécifications formelles (spécifications algébriques [10], Z [57] ou B [5]). Ceci permet de vérifier les spécifications mais aussi potentiellement de décrire les données à un haut niveau d'abstraction. La flexibilité que nous proposons quand au choix du langage de spécification des aspects statiques permet au spécifieur de choisir le langage formel qui lui parait le plus adapté à cette tâche : celui qu'il connaît le mieux, celui qui sera mieux outillé ou bien encore celui qui lui permet de réutiliser des spécifications statiques déjà écrites.

Les aspects dynamiques sont modélisés avec les diagrammes d'états d'UML [48]. Notre travail est cependant partiellement générique. Différentes sémantiques dynamiques peuvent être prises en compte, et notre approche est donc aussi applicable aux Statecharts [29], aux différentes sémantiques des diagrammes d'états d'UML [41], et plus généralement à tout système à base d'états et de transitions. Dans notre approche, la spécification est dirigée par le contrôle : le comportement dynamique est le comportement principal de la spécification et il décrit comment les données de l'aspect statique sont manipulées. Ce travail s'intègre complètement dans la thématique plus générale de l'*intégration* ou *combinaison* de méthodes formelles où nous avons déjà eu plusieurs résultats [8]. Au niveau global de la spécification, notre approche permet donc aussi d'adresser les problèmes de cohérence entre parties statiques et dynamiques de descriptions de systèmes.

Le reste du rapport se décompose comme suit. La section 2 présente les fondements de notre approche : les liens syntaxiques entre les diagrammes d'états et les données ainsi que la sémantique de l'intégration. La section 3 illustre ensuite de façon pragmatique comment notre proposition peut être appliquée pour spécifier un système complet (une station essence). Dans la section 4, nous présentons les travaux connexes à notre approche. Enfin, dans la section 5, nous concluons sur ce travail et discutons des perspectives possibles.

2 Fondements formels de la combinaison

Dans cette section, nous formalisons la combinaison des diagrammes d'états d'UML avec des données formelles. Nous étudions successivement les aspects syntaxiques et sémantiques.

2.1 Aspects syntaxiques

Nous détaillons tout d'abord les liens entre les diagrammes d'états et les données qui s'intègrent dans les diagrammes. Nous suivons une approche dirigée par le contrôle. Le comportement principal de la spécification est par conséquent donné par la modélisation de la partie dynamique. Les données apparaissant dans les diagrammes d'états correspondent à des termes exprimés à partir de définitions formelles de types de données. Ces termes sont soit des termes algébriques, soit des appels d'opérations définies dans des schémas Z ou des machines B. Nous nous

basons sur l'existence d'une fonction d'évaluation par type de langage formel de description de données. Notre approche permet d'utiliser conjointement plusieurs de ces langages. Cependant, un mélange fort de spécifications écrites avec des langages différents (une importation d'un module par un autre par exemple) n'est pas autorisé. Ceci se justifie aisément ; il s'agit d'éviter les incompatibilités sémantiques qui en découleraient. Par ailleurs, notre but ici n'est pas de proposer une telle combinaison complexe ni de résoudre les incohérences et conflits des langages afin de les faire cohabiter.

Les interactions entre les parties dynamiques et statiques se situent à différents niveaux : sur les transitions et à l'intérieur des états. La forme générale d'une transition dans les diagrammes d'états est rappelée dans la figure 1.

FIG. 1 – Transition dans les diagrammes d'états

Concernant les activités au sein d'un état, la notation est augmentée afin de permettre des modifications de données internes au diagramme d'états. Cependant, nous ne considérons pas dans les états la possibilité d'événements avec données car ces transitions internes peuvent être vues comme des cas particuliers de transitions entre états. Les événements (signaux, temporels, de modification) peuvent faire intervenir des données. Par exemple, un événement signal peut recevoir une information dans une variable d'un type précédemment défini dans des définitions abstraites des données (réception). La garde peut également être exprimée avec des données concrètes. Enfin, les actions peuvent induire soit la génération d'événement (émission), soit des modifications des données déclarées en variables locales à l'intérieur du diagramme. Nous résumons ces interactions dans le tableau 1 avec des exemples illustratifs, puis ces liens sont formalisés dans la grammaire du tableau 2. Seules les extensions de la notation classique des diagrammes d'états sont introduites.

ACTIVITE	EVENEMENT	GARDE	ACTION	exemple avec données
modifications locales	réception	garde		/ x_1 :=*opération*$_1$,.., x_n :=*opération*$_n$ *événement*(x_1 : *type*$_1$,.., x_n : *type*$_n$) *prédicat*
			envoi	/ *événement*(d_1,.., d_n)
			modifications locales	/ x_1 :=*opération*$_1$,.., x_n :=*opération*$_n$

TAB. 1 – Liens dans les diagrammes d'états avec les données

```
LIEN       ::=  ACTIVITE | [EVT] [[GARDE]] [/ ACTION]
ACTIVITE   ::=  / AFF-TERME-DONNEE+
EVT        ::=  nom-evt(DECL-VAR+)
GARDE      ::=  PREDICAT
ACTION     ::=  nom-evt(TERME-DONNEE+) | AFF-TERME-DONNEE+
```

TAB. 2 – Grammaire BNF des liens

Les non-terminaux `DECL-VAR`, `PREDICAT`, `TERME-DONNEE` et `AFF-TERME-DONNEE` correspondent aux données introduites dans le tableau 1, et ne sont pas détaillés dans la grammaire car ce sont des structures classiques.

Au niveau des diagrammes d'états, il est possible de définir abstraitement des données locales. Elles sont déclarées en utilisant les *notes* d'UML qui décrivent habituellement des informations de tout type sous forme textuelle. Une notation (`IMPORT`, Figure 2) est introduite à ce niveau pour indiquer quels modules (spécifications algébriques, schémas Z ou machines B) sont importés ainsi que les familles de langages associées. Un module contient des définitions abstraites de types de données utilisées par la fonction d'évaluation servant à interpréter les données apparaissant dans le diagramme d'états. Les variables sont ensuite déclarées et typées. Un module peut

soit contenir plusieurs définitions abstraites, soit une seule (ce qui est souvent le cas avec les langages orientés état où le nom du schéma ou de la machine est un type). En fonction des cas, le type d'une variable est préfixé par le nom du module, ou le nom du module est directement utilisé en tant que type. Pour finir, nous insistons sur le fait que le sens des spécifications statiques doit être cohérent, et que nous posons cette hypothèse comme point de départ à la définition de nos règles sémantiques.

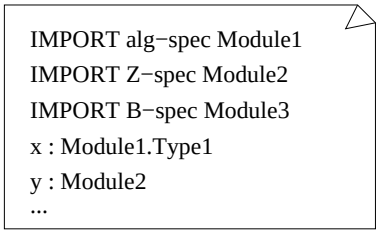

FIG. 2 – Déclaration locale de données dans les diagrammes d'états

2.2 Aspects sémantiques

Notre but est de proposer une sémantique formelle des interactions entre comportements et données au sein des diagrammes d'états. Il ne s'agit donc pas de formaliser complètement les diagrammes d'états d'UML tel que cela a été fait par exemple dans [41]. Nous conservons une description très générale de ces diagrammes ; de cette façon, par extension, tous les diagrammes d'état et leurs différentes sémantiques sous-jacentes (telle que celle décrite dans [41]) pourront être considérées. Par contre, nous définissons précisément le sens de l'intégration des données dans les diagrammes dynamiques par des enrichissements d'environnement et des fonctions d'évaluation.

Pour représenter la sémantique des opérations sur les données, nous utilisons une sémantique opérationnelle inspirée du travail de Bruns [17]. Dans la suite, nous posons $\mathcal{D}$ comme étant l'ensemble des diagrammes d'états avec leurs sémantiques associées. Pour les règles qui suivent, nous considérons un diagramme D qui appartient à $\mathcal{D}$. Nous posons également $\mathcal{T}_D$ comme l'ensemble des transitions de D, $\mathcal{S}_D$ l'ensemble de ses états et $\mathcal{S}_{0D}$ l'ensemble de ses états initiaux. Nous notons les éléments de $\mathcal{T}_D$ sous la forme $S \xrightarrow{label} S'$.

La construction de base est un environnement de liaison de variables E. E est un ensemble fini de paires (x, v) qui indiquent que la variable x est liée à la valeur v dans l'environnement E. La notation $E \vdash e \rhd_X v$ signifie qu'en utilisant l'évaluation définie dans le cadre X (spécifications algébriques, Z ou B) alors e s'évalue en v dans le contexte de E. Nous restons ici volontairement succincts, des détails concernant la sémantique de $\rhd$ seront donnés dans la suite. Les éventuelles variables libres présentes dans e sont éventuellement valuées en utilisant les liaisons définies dans l'environnement E. De plus, si E et E' sont des environnements alors EE' est l'environnement dans lequel les variables de E et E' sont définies et les liaisons de E' écrasent celles de E. Pour définir les règles d'inférence, nous utilisons la notation $E \vdash decl \Rightarrow E'$ qui dénote que la déclaration $decl$ transforme l'environnement E en E'. La notation $TRUE_X$ désigne la valeur de vérité *vrai* dans le cadre X. Pour connaître le cadre X auquel correspond une construction e donnée (*i.e.* une variable, un terme ou un appel d'opération), nous définissons des règles de typage particulières que nous appelons règles de *méta-typage* en ce sens qu'elles ne donnent pas un type mais un méta-type (spécifications algébriques, Z ou B) à une construction. Nous utiliserons T pour les types usuels et X pour les méta-types.

Règles de méta-typage. Ces règles sont données en figure 3. La première permet de méta-typer les variables en se servant des déclarations locales de données. La seconde règle donne par induction le méta-type d'une construction à partir des méta-types des éléments qui la composent. La fonction $DeclData(S)$ dénote les déclarations locales des données dans le diagramme d'états S. La variable op symbolise une opération appliquée à un ensemble de termes t_i. Le module M contient des définitions abstraites de données. La fonction def teste si une déclaration d'un type T ou d'une opération op est présente dans un module M.

$$\frac{\begin{array}{c} D \in \mathcal{D} \\ S \in \mathcal{S}_D \\ \texttt{IMPORT}\ X\text{–}SPEC\ M \in DeclData(S) \\ def(T, M) \\ x : T \in DeclData(S) \end{array}}{x ::_D X} \qquad \frac{\begin{array}{c} D \in \mathcal{D} \\ S \in \mathcal{S}_D \\ \texttt{IMPORT}\ X\text{–}SPEC\ M \in DeclData(S) \\ \forall\, i \in 1..n \quad t_i ::_D X \\ def(op, M) \end{array}}{op(t_i) ::_D X}$$

FIG. 3 – Règles de méta-typage

Les règles de la sémantique se décomposent en trois groupes. Le premier décrit les modifications d'environnement découlant des exécutions des données apparaissant dans les transitions des diagrammes d'états. Le second groupe de règles présente l'évolution des aspects dynamiques et leur influence sur les modifications et/ou mises à jour des environnements. En ce qui concerne ces deux groupes de règles, nous nous plaçons dans un cadre partiel qui nous permet de considérer n'importe quelle sémantique des diagrammes d'états. Ces diversités sémantiques s'expliquent principalement par des divergences de choix dans le modèle de communication. Nous montrons comment ces règles peuvent être complétées (troisième groupe) afin de prendre en compte un type précis de communication. Ici, nous nous intéresserons aux diagrammes d'états d'UML de [48] et compléterons notre sémantique dans ce sens.

Règles d'évolution des environnements. Nous détaillons à présent ce premier groupe de règles en traitant chaque construction de notre grammaire (tableau 2). Avant de donner les règles d'inférence, nous rappelons en encadré la construction concernée. Dans la suite, n et m sont deux variables entières.

`ACTIVITE ou ACTION ::= [/] AFF-TERME-DONNEE+`

$$\frac{\forall\, i \in 1..n \quad y_i :: X\ ,\ t_i :: X\ ,\ E \vdash t_i \rhd_X w_i}{E \vdash [\ /\ ]\ y_1 := t_1, \ldots, y_n := t_n \Rightarrow E\{(y_1, w_1), \ldots, (y_n, w_n)\}}$$

Les assignations ou appels d'opérations mettent à jour les environnements. Concernant les spécifications algébriques, la compréhension de cette règle semble assez naturelle (les t_i se réécrivent en w_i). Pour les langages orientés état, l'interprétation de la règle est quelque peu différente. Les variables y_i désignent l'état du schéma Z ou la machine B concerné par la modification (dans le cas de Z ou B, y_i est donc lié dans E). Les w_i dénotent les nouveaux états obtenus à partir de l'environnement et par évaluation des t_i (*i.e.* en quelque sorte $w_i \equiv y_i'$) . La notation utilisée dans cette règle est propre à notre approche et légèrement distincte des notations usuelles en Z ou B (*i.e.* pointées ou à effet de bord). Elle se justifie par notre volonté de disposer d'une syntaxe commune aux différents langages de données considérés.

`EVT ::= nom-evt(DECL-VAR+)`

$$\frac{\forall\, i \in 1..n \quad \exists\, v_i : T_i}{E \vdash evt(x_1 : T_1, \ldots, x_n : T_n) \Rightarrow E\{(x_1, v_1), \ldots, (x_n, v_n)\}}$$

Une réception lie les variables en entrée avec des valeurs reçues du bon type. Enfin, les gardes et actions (règles ci-dessous) ne modifient pas localement l'environnement.

`GARDE ::= PREDICAT`

$$\frac{}{E \vdash G \Rightarrow E}$$

`ACTION ::= nom-evt(TERME-DONNEE+)`

$$\frac{}{E \vdash evt(e_1, \ldots, e_n) \Rightarrow E}$$

Règles dynamiques. Au premier ensemble de règles viennent s'ajouter des règles dynamiques (figure 4) qui symbolisent l'influence du comportement du diagramme sur les environnements décrits précédemment. Deux règles décrivent respectivement l'initialisation et l'évolution individuelle d'un diagramme. À ce niveau, nous faisons complète abstraction d'une quelconque sémantique particulière de la communication dans les diagrammes d'états.

$$\frac{\begin{array}{c} D \in \mathcal{D} \\ S \in \mathcal{S}_{0D} \\ x_i : T_i \in DeclData(S) \\ \exists\, v_i : T_i \end{array}}{\vdash S \Rightarrow \{(x_i, v_i)\}} \qquad \frac{\begin{array}{c} D \in \mathcal{D} \\ S, S' \in \mathcal{S}_D \\ G ::_D X \\ S \xrightarrow{EVT\ G\ /\ ACT} S' \in \mathcal{T}_D \\ E \vdash EVT \Rightarrow E' \\ E' \vdash G \rhd_X TRUE_X \\ E' \vdash ACT \Rightarrow E'' \end{array}}{E \vdash S \xrightarrow{EVT\ G\ /\ ACT} S' \Rightarrow E''}$$

FIG. 4 – Règles dynamiques

$$\frac{\begin{array}{c} D \in \mathcal{D} \\ concur(S_{emet}, \cup_{i\in 1..n}\{S_{rec_i}\}) \\ S_{rec_i}, S_{emet}, S'_{rec_i}, S'_{emet} \in \mathcal{S}_D \\ S_{rec_i} \xrightarrow{evt(x_1:T_1,\ldots,x_m:T_m)\ G_{rec_i}\ /\ ACT_{rec_i}} S'_{rec_i} \in \mathcal{T}_D \\ S_{emet} \xrightarrow{EVT_{emet}\ G_{emet}\ /\ evt(e_1,\ldots,e_m)} S'_{emet} \in \mathcal{T}_D \\ E_{emet} \vdash EVT_{emet} \Rightarrow E'_{emet} \\ G_{emet} ::_D X_{emet} \\ E'_{emet} \vdash G_{emet} \rhd_{X_{emet}} TRUE_{X_{emet}} \\ \forall\, j \in 1..m \\ e_j ::_D X_j \\ E'_{emet} \vdash e_j \rhd_{X_j} v_j \\ E'_{rec_i} = E_{rec_i}\{(x_1, v_1) \ldots (x_j, v_j) \ldots (x_m, v_m)\} \\ G_{rec_i} ::_D X_{rec_i} \\ E'_{rec_i} \vdash G_{rec_i} \rhd_{X_{rec_i}} TRUE_{X_{rec_i}} \\ E'_{rec_i} \vdash ACT_{rec_i} \Rightarrow E''_{rec_i} \end{array}}{\begin{array}{l} E_{rec_i} \vdash S_{rec_i} \xrightarrow{evt(x_1:T_1,\ldots,x_m:T_m)\ G_{rec_i}\ /\ ACT_{rec_i}} S'_{rec_i} \Rightarrow E''_{rec_i} \\ E_{emet} \vdash S_{emet} \xrightarrow{EVT_{emet}\ G_{emet}\ /\ evt(e_1,\ldots,e_m)} S'_{emet} \Rightarrow E'_{emet} \end{array}}$$

FIG. 5 – Règle de communication

Règles de communication. Nous avons jusqu'ici décrit comment les environnements et les transitions évoluent. Lorsque nous nous intéressons à une sémantique particulière de la communication alors nous ajoutons des règles pour la gérer. Pour illustrer cette dernière étape, nous choisissons le modèle de communication des diagrammes d'état d'UML, et plus particulièrement le cas des états concurrents (communication synchrone en broadcast). Nous rappelons que plusieurs états concurrents se synchronisent (implicitement) sur des événements de même nom. La règle est donnée en figure 5. Nous avons n receveurs et un émetteur, ce que nous notons $concur(S_{emet}, \cup_{i\in 1..n}\{S_{rec_i}\})$.

L'idée inhérente à cette règle est que si un état (S_{emet}) évolue par une émission *evt* et qu'un certain nombre (n) d'états (S_{rec_i}) sont en attente de réception sur ce même événement, alors la synchronisation a lieu entre les $n+1$ états concurrents et les différents environnements sont mis à jour. La présence de gardes et d'actions induit successivement des évaluations de prédicats et des modifications d'environnement.

Nous avons uniquement donné la règle concernant le cas où les étiquettes des transitions sont complètes (*i.e.* ont la forme *événement[garde]/action*). Ce choix se justifie car cette règle est la plus générale, et toutes les autres sont obtenues simplement en supprimant les prémisses qui n'ont plus raison d'être présentes (*e.g.* absence d'évaluation d'une certaine garde à partir d'un environnement donné si la transition ne porte pas de garde).

Un problème demeure encore. En effet, si deux transitions apparaissant dans deux états concurrents différents ont la forme `evt[g1]/act` et `act[g2]/evt`, il n'y a qu'une synchronisation possible (soit sur `evt`, soit sur `act`) alors que la règle ci-dessus permet les deux. Cela s'explique car nous ne considérons pas la séquentialité dans

$$\frac{\begin{array}{c} D \in \mathcal{D} \\ S, S' \in \mathcal{S}_D \\ S \xrightarrow{EVT\ G\ /\ ACT} S' \in \mathcal{T}_D \end{array}}{\begin{array}{c} \exists\, S'' \in \mathcal{S}_D \\ S \xrightarrow{EVT\ G} S'' \in \mathcal{T}_D \\ S'' \xrightarrow{/\ ACT} S' \in \mathcal{T}_D \end{array}}$$

FIG. 6 – Règle de découpage *événement[garde]/action*

l'exécution des étiquettes des transitions. Pour résoudre ce problème, il est possible de transformer le diagramme avec la règle de la figure 6. Elle induit qu'une transition portant `evt[g]/act` est découpée en deux parties `evt[g]` et `/act`.

Sémantique de $\triangleright_X$. Il nous reste à préciser la sémantique de $\triangleright_X$ par rapport aux divers cas de manipulation de données, *i.e.* à quoi correspondent les fonctions permettant l'évaluation. Nous avons à notre disposition plusieurs types de fonction d'évaluation qui dépendent du type de langages de spécification de données : orienté axiomatique ou orienté état. Ici, nous nous focalisons sur la fonction d'évaluation concernant les spécifications algébriques. Dans le cas de Z et B, les fonctions $\triangleright_Z$ er $\triangleright_B$ correspondent à la modification d'état induit par l'opération. De plus amples détails concernant la définition de ces fonctions sont regroupés dans [53].

En ce qui concerne les spécifications algébriques, la fonction d'évaluation correspond à une fonction de réécriture de termes. Le système de réécriture est obtenu à partir des axiomes algébriques par application des algorithmes d'orientation, assurant la terminaison et la confluence, tels que ceux décrits dans [34]. Le choix de la réécriture est justifié puisqu'il est adapté à une sémantique opérationnelle, et par conséquent qu'il nous permet de rester dans un contexte pragmatique et exécutable. Pour appliquer ces algorithmes, nous nous restreignons à la sémantique initiale des types de données car l'interprétation d'un ensemble d'axiomes par un système de réécriture n'a vraiment de sens que dans le cas d'un modèle initial [13]. Nous restreignons les langages de spécifications algébriques avec sémantique *lâche* à cette approche initiale (*e.g.* voir [33] pour CASL [24]).

Exemple. À présent, illustrons l'application de nos règles sémantiques sur un cas précis de spécification. Il nous faut pour cela instancier le (ou les) environnement(s) de liaison de variables, la règle dynamique montrant l'évolution du diagramme d'états, et la fonction d'évaluation $\triangleright_X$ employée (ainsi que la valeur de vérité $TRUE_X$). Pour rester dans un cas simple et compréhensible, nous utilisons un type de données représentant des entiers naturels (figure 7) directement spécifiés avec le langage d'entrée du prouveur de théorèmes LP [27], qui est un sous-ensemble de Larch.

Les règles de réécriture sont obtenues à partir des axiomes de la spécification algébrique par simple orientation de gauche à droite. Nous avons donc l'équivalence suivante pour l'opérateur d'évaluation : $\triangleright_{alg\text{–}spec} \equiv \xrightarrow{*}$, et $TRUE_{alg\text{–}spec} \equiv true$. Considérons le diagramme d'états de la figure 8 qui représente trois comportements concurrents qui interagissent sur l'événement `tick`, ainsi que la règle décrite figure 5 qui décrit la synchronisation entre plusieurs diagrammes concurrents. L'instanciation de cette règle est donnée dans la figure 9.

L'évolution du diagramme (*i.e.* passage de S1 à S1' par exemple) dénote le passage d'un état à un autre par une transition. Certaines prémisses sont omises dûes au fait que certaines transitions ne sont pas complètes (*i.e.* n'ont pas la forme *événement [garde] / transition*) mais ont des formes simplifiées. Nous nous plaçons dans le cas de la première évolution ce qui explique que certains environnements (ceux des états S1 et S2) soient vides au départ. Les variables locales doivent aussi être initialisées lors de l'animation de la spécification, ainsi la variable `n` contient la valeur `v` par application de la première règle dynamique (Figure 4). La dernière premisse de la règle ci-dessus exprime que le terme `n+1` se réécrit en `m` en substituant la variable `n` par la valeur `v`. Les autres prémisses et conclusions expriment une évolution dynamique possible des diagrammes et les modifications des environnements qui en découlent.

```
set name NAT
declare sort Nat
declare op
     0 : -> Nat
     s : Nat -> Nat
     __ + __ : Nat, Nat -> Nat
     ..
assert sort Nat generated by 0, s;
declare var x, y : Nat
assert
     x + 0 = x;
     x + s (y) = s (x + y);
     ..
```

FIG. 7 – Spécification des entiers naturels en LP

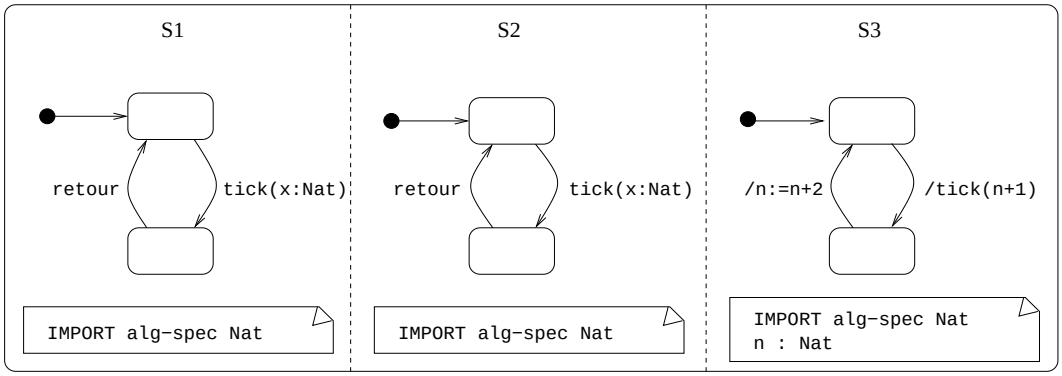

FIG. 8 – Exemple de diagrammes d'états avec synchronisation

3 Une étude de cas : la station essence

Cette partie a pour but d'illustrer l'utilisation de notre approche sur un exemple concret, simplifié mais réaliste. Il s'agit d'une station essence équipée de plusieurs pompes. Nous présentons successivement le cahier des charges de cette étude de cas, son analyse, la modélisation des aspects statiques et la modélisation des aspects dynamiques.

3.1 Cahier des charges et analyse

Cahier des charges. L'étude de cas concerne la spécification du fonctionnement d'une station essence. La station propose uniquement le service avec paiement par carte bancaire à la pompe et ne permet pas de paiement à la caisse. Les transactions nécessaires pour l'authentification de la carte et le paiement effectif (transmission des données au centre bancaire) ne font pas l'objet de l'étude. La station comprend trois pompes à carburant correspondant au super, au sans plomb et au gas-oil. Elle est aussi équipée d'un pupitre muni d'un lecteur de carte pour les paiements, de boutons pour la sélection de diverses options, d'un pavé numérique pour la saisie, d'un écran d'affichage et d'une imprimante de tickets. La station gère tous ces équipements et a pour fonctionnalité principale de fournir aux utilisateurs, à leur demande, le carburant sélectionné en quantité voulue par ces derniers. Pour cela la station permet aux utilisateurs de saisir le numéro de carte via le pavé numérique, de sélectionner le type de carburant et de choisir éventuellement l'impression d'un ticket.

Analyse et hypothèses de travail. A partir de cette description succincte du cahier des charges, nous avons éclairci les fonctionnalités attendues de la station considérée ici comme un système. Nous avons explicité les conditions nécessaires pour assurer ses fonctionnalités et les interactions impliquées avec l'environnement. Cette analyse nous permet de dégager les différents composants de notre système, de les modéliser et de décrire le fonctionnement

$$\frac{\begin{array}{c} \mathit{concur}(S3, \{S1, S2\}) \\ S1 \xrightarrow{\texttt{tick(x:Nat)}} S1' \\ S2 \xrightarrow{\texttt{tick(x:Nat)}} S2' \\ S3 \xrightarrow{\texttt{/tick(n+1)}} S3' \\ \{(\texttt{n}, \texttt{v})\} \vdash \texttt{/tick}(\texttt{n} + 1) \Rightarrow \{(\texttt{n}, \texttt{v})\} \\ \{(\texttt{n}, \texttt{v})\} \vdash \texttt{n} + 1 \xrightarrow{*} \texttt{m} \\ \{\} \vdash \texttt{tick}(\texttt{x} : \texttt{Nat}) \Rightarrow \{(\texttt{x}, \texttt{m})\} \end{array}}{\begin{array}{c} \{\} \vdash S1 \xrightarrow{\texttt{tick(x:Nat)}} S1' \Rightarrow \{(\texttt{x}, \texttt{m})\} \\ \{\} \vdash S2 \xrightarrow{\texttt{tick(x:Nat)}} S2' \Rightarrow \{(\texttt{x}, \texttt{m})\} \\ \{(\texttt{n}, \texttt{v})\} \vdash S3 \xrightarrow{\texttt{/tick(n+1)}} S3' \Rightarrow \{(\texttt{n}, \texttt{v})\} \end{array}}$$

FIG. 9 – Instanciation de la règle de communication

global attendu de la station essence. Nous distinguons deux principales parties dans l'analyse : une partie statique concernant les données intervenant dans le système et une partie dynamique concernant l'évolution du système à travers ses divers constituants.

Partie statique. Pour fournir les carburants aux utilisateurs sans interruption brusque pendant un service, la station doit disposer de quantité suffisante de chaque carburant proposé lorsqu'elle est en service. Nous faisons l'hypothèse que la station dispose de plusieurs cuves comme c'est le cas dans la réalité. Il y a une cuve par type de carburant. Une cuve est associée à chaque pompe. Une pompe est déclarée hors service lorsque le volume de carburant contenu dans la cuve associée est inférieur à un certain seuil à définir. La station ne peut alors plus fournir de carburant de ce type avant l'approvisionnement de la cuve.

Pour assurer son bon fonctionnement, le système doit pouvoir connaitre à tout moment l'état des cuves. De même, cet état doit être tenu à jour par rapport aux opérations effectuées (retrait de carburant, approvisionnement en carburant). Pour pouvoir calculer les sommes à payer (et imprimer le ticket), le système doit disposer du prix par litre et par type de carburant ainsi que du volume servi. Un enregistrement de ces données est donc nécessaire. Pour la modélisation de la partie statique nous devons traiter les cuves, les carburants et les pompes.

Partie dynamique. Le déclenchement du fonctionnement de la station est provoqué par un utilisateur. Il insère sa carte bancaire et s'ensuit une étape d'authentification. Si l'authentification est correcte, il est invité à sélectionner son carburant et à valider ou corriger sa sélection. Cette interaction met en évidence précisément un lecteur *gestionnaire de carte*. En fonction des choix de l'utilisateur la station initialise son écran, affiche le prix au litre du carburant sélectionné et le montant maximal de carburant qu'il est possible de délivrer. L'utilisateur doit décrocher la pompe et se servir jusqu'à satisfaction. Le montant à payer et la quantité sont affichés au fur et à mesure. En fin de service, lorsque l'utilisateur raccroche la pompe, la station imprime le ticket si cette option a été choisie. Toute cette interaction relève d'un sous-système *gestionnaire de pompes*. Celui-ci échange des données avec l'utilisateur. Nous faisons l'hypothèse que lorsqu'une pompe de la station est en service, elle garantit le service du carburant à l'utilisateur. Pour établir cela, le gestionnaire de pompes doit tester après chaque service, la quantité de carburant restant dans la cuve associée à la pompe et, soit la pompe reste en service soit elle passe hors service si la quantité restante est inférieure à un seuil donné. Nous distinguons donc un *gestionnaire de cuves* ayant des interactions avec le gestionnaire de pompes.

La figure 10 donne un aperçu du système global que nous percevons comme étant la composition parallèle des diagrammes des trois gestionnaires : ce sont des états concurrents avec des interactions. Chacun des gestionnaires est décrit plus loin. D'autres sous-systèmes sont considérés comme externes : l'utilisateur et un sous-système qui fournit, continuellement pendant un service, la quantité de carburant servie et le montant correspondant.

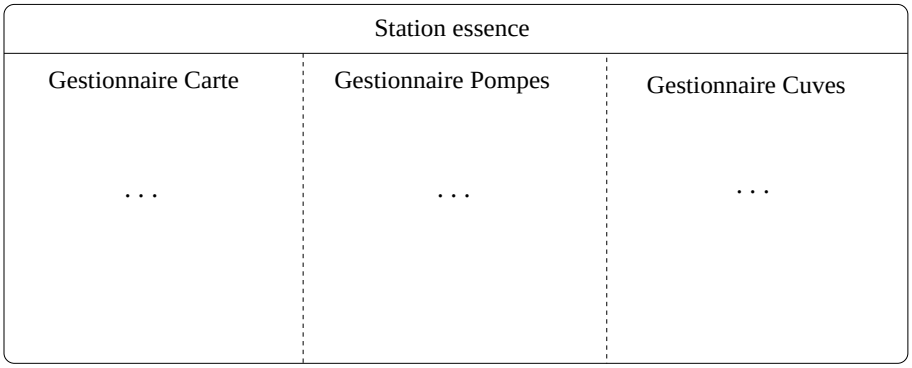

FIG. 10 – Diagramme d'état du système global

3.2 Modélisation des aspects statiques

Dans cette étude de cas, nous avons formalisé les données avec le langage d'entrée du prouveur de théorèmes LP et le langage Z. En LP nous avons décrit les spécifications `Nat` pour les entiers naturels et `Reel` pour les réels. En Z nous avons spécifié les données concernant les pompes et la gestion des cuves. Nous ne présentons ici que les principales spécifications en Z qui sont utilisées dans les diagrammes d'état de la partie dynamique. Toute la spécification est analysée avec Z/EVES pour sa correction syntaxique (analyse syntaxique et contrôle de type) et la sémantique statique (contrôle des domaines).

Modélisation des données pour la gestion des Cuves. Nous modélisons la gestion des cuves en Z. Pour chaque cuve, nous conservons le carburant contenu dans la cuve, ainsi que la capacité minimale et maximale de carburant qu'elle peut contenir. Différentes opérations sont implantées pour spécifier les mises à jour des volumes de carburant contenues dans chaque cuve.

$$POMPE == \mathbb{N}$$

Nous modélisons les pompes par des entiers naturels. Une pompe est identifiée par un entier. Les propriétés d'une pompe sont décrites plus loin en faisant le lien avec les cuves (*GCuves*).

$$Bool ::= vrai \mid faux$$

Les valeurs *vrai* et *faux* caractérisent le type *Bool* (les booléens).

$$CarburantZ ::= super \mid sans\_plomb \mid gas\_oil$$

Il n'y a que trois types de carburant, leur énumération permet de spécifier les carburants manipulés.

Cuve

$$capaciteMax : \mathbb{Z}$$
$$capaciteMin : \mathbb{Z}$$
$$quantite : \mathbb{Z}$$

$$quantite < capaciteMax \land quantite > capaciteMin$$

Une cuve est caractérisée par sa capacité minimale, sa capacité maximale et la quantité courante de carburant.

GCuves

$pompes : \mathbb{F}\,\mathbb{N}$
$cuve : POMPE \pfun Cuve$
$carburant : POMPE \rightarrowtail CarburantZ$
$qtPompe : POMPE \pfun \mathbb{Z}$

$\mathrm{dom}\, cuve = pompes$
$\mathrm{dom}\, carburant = pompes$
$\mathrm{dom}\, qtPompe = pompes$

Ce schéma permet de gérer plusieurs pompes. A chaque pompe nous associons une cuve, un carburant et la quantité de carburant restant dans la pompe (*qtPompe*).

InitGCuves

$GCuves'$

$pompes' = \{1, 2, 3\}$
$cuve' = \emptyset$
$carburant' = \{(1, super), (2, sans\_plomb), (3, gas\_oil)\}$
$qtPompe' = \emptyset$

Dans ce schéma d'initialisation, nous gérons précisément trois pompes identifiées par 1, 2, 3. Les autres variables d'état sont initialisées par d'autres opérations sur le schéma *GCuves*. Nous ne présentons pas ici toutes les opérations mais comme exemple nous donnons ci-après celle qui permet de faire la mise à jour de la quantité de carburant restant dans chaque pompe (*majQt*) et celle qui permet de tester la quantité de carburant (*OkSeuil*).

majQt

$\Delta GCuves$
$pp? : \mathbb{N}$
$qtte? : \mathbb{Z}$

$pp? \in pompes$
$pompes' = pompes$
$cuve' = cuve$
$carburant' = carburant$
$qtPompe'(pp?) = qtPompe(pp?) + qtte?$

Cette opération effectue la mise à jour (ajout et retrait) de la quantité de carburant dans une pompe donnée. La quantité peut être négative pour traiter les diminutions ou positive pour les augmentations. Les variables d'entrée *pp*? et *qtte*? peuvent être remplacées par celles utilisées dans les diagrammes en employant la notation Z `majQt[nouvpp/pp?; nouvqtte/qtte?]`. Le résultat de l'opération est recupérée de la même façon dans une variable d'un diagramme d'états donné.

OkSeuil

$\Xi GCuves$
$pp? : \mathbb{N}$
$seuil? : \mathbb{Z}$
$res! : Bool$

$pp? \in \mathrm{dom}\, qtPompe$
$qtPompe(pp?) < seuil? \lor qtPompe(pp?) = seuil?$
$res! = faux$

Cette opération permet de tester la quantité de carburant dans une cuve par rapport à un seuil indiqué. Le résultat est positif si la quantité restant dans la pompe passée en paramètre d'entrée (*pp*?) est supérieure au seuil qui est

fourni en paramètre d'entrée (*seuil*?).

Ces types de données définies vont nous permettre de décrire et manipuler les données dans la modélisation des comportements de la partie dynamique.

3.3 Modélisation des aspects dynamiques

Nous avons modélisé à l'aide de diagrammes d'états d'UML, le comportement du gestionnaire de carte, le comportement du gestionnaire de pompes et le comportement du gestionnaire de cuves. Les diagrammes d'états associés sont détaillés dans la suite. Dans notre modélisation, nous avons considéré une abstraction des interactions afin de construire les enchaînements des événements et des actions, et d'établir le lien avec les données élaborées précédemment. Les données sont prises en compte et manipulées dans les diagrammes comme décrit dans la section 2. Elles apparaissent en argument des événements comme par exemple `majEtatPompe(etatp : Bool)`, dans les gardes, dans les actions comme par exemple `/activerPompe(c)` et dans les notes comme variables locales. Ces dernières ont des types importés (`IMPORT ...`) à partir des modules spécifiés dans la partie statique. Dans cette étude de cas, les modules définis contiennent chacun un seul type, nous n'avons donc pas eu besoin de préfixer le nom du type par le nom du module qui le contient.

A présent, nous détaillons les composantes de base de notre système en commençant par le gestionnaire de carte (figure 11). Il effectue une suite d'événements qui permet d'insérer la carte, de taper le code et de sélectionner le carburant désiré. Ensuite, il interagit avec le gestionnaire de pompes afin d'effectuer le service du carburant. Plusieurs cas de terminaison sont modélisés : soit l'authentification a echoué (`erreur`), soit le service est fini (`serviceTermine`) et l'impression du ticket est effectuée si l'utilisateur en a fait la demande (`imprimeTicket`), soit le service n'a pas été possible (non décrochage de la pompe avant soixante secondes) et l'exécution termine sur une erreur (`erreurp`).

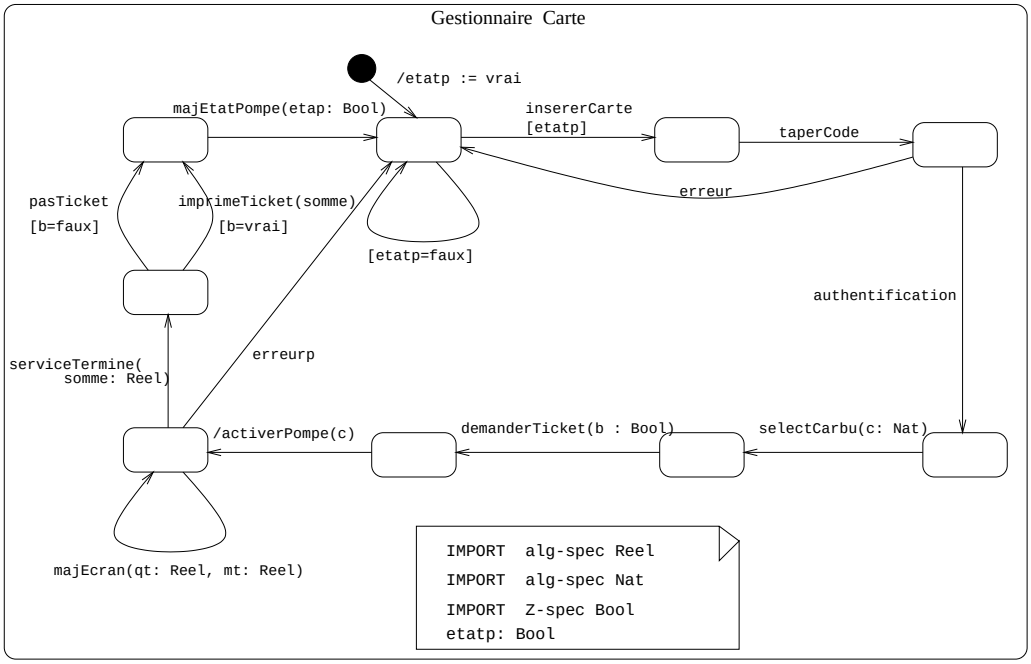

FIG. 11 – Diagramme du gestionnaire de carte

Le diagramme du gestionnaire de pompes (figure 12) utilise les variables locales `montantMax`, `seuilPompes` et `t`. La variable `seuilPompes` représente la quantité minimale de carburant pour que les pompes restent en service ; c'est un seuil défini. Notons la présence d'événements externes qui correspondent aux interactions du système avec son environnement, en particulier avec l'utilisateur qui déclenche le fonctionnement des pompes. Ces événements ne sont pas complètement

explicités ; il y a par exemple `insererCarte` dans le gestionnaire de carte (figure 11) et `service` dans le gestionnaire de pompes (figure 12). Le calcul de la somme à payer (`montant`) est effectuée par un sous-système en dehors du gestionnaire et la valeur est communiquée via l'événement `recevoirMontant`. Il en est de même pour la quantité servie `qtservi`. Il y a des interactions avec ce sous-système. Ce dernier aurait pu être spécifié afin de calculer la somme à payer à partir de la quantité servie à partir d'un type de données conservant les couples <*carburant, prix au litre*> ; ceci n'est pas considéré pour simplifier la présentation de l'étude. Nous gérons néanmoins cet aspect par l'événement `recevoirMontant` dans le diagramme du gestionnaire de pompes (figure 12). Le gestionnaire de pompes (figure 12) assure le service du carburant en interagissant avec le gestionnaire de cuves et de carte. Lorsqu'un service de carburant est entamé, l'utilisateur dispose de soixante secondes pour décrocher la pompe et se servir ; passé ce délai une erreur est signalée. Le gestionnaire de pompes possède la capacité maximale que l'utilisateur peut obtenir, il peut interrompre le service en conséquence (`finService`). Le gestionnaire de pompes interagit avec l'utilisateur pour le service. Par exemple avec les événements `insererCarte` et `demanderTicket`. Pendant le service, le gestionnaire de pompes effectue plusieurs synchonisations avec le gestionnaire de carte pour la mise à jour de l'écran (`majEcran`) avec la quantité de carburant et la somme correspondante. L'un fait `/majEcran(qtservi, montant)` et l'autre `majEcran(...)`. Cela correspond à une synchronisation telle que décrite dans la section 2. L'interaction se poursuit avec l'utilisateur pour la fin de service (`finService`) et le raccrochage de la pompe (`raccrochage`). Le gestionnaire de pompes bloque alors la pompe puis se synchronise avec le gestionnaire de cuves pour la mise à jour de la quantité de carburant (`diminuerQt`), et enfin il communique de nouveau avec le gestionnaire de carte pour finir le service (`serviceTermine`). Notons la mise à jour, dans le gestionnaire de pompes, de l'état de la pompe (`/majEtatPompe(etatp)`) après le test de la quantité disponible (`/testSeuil(...)`) après chaque service.

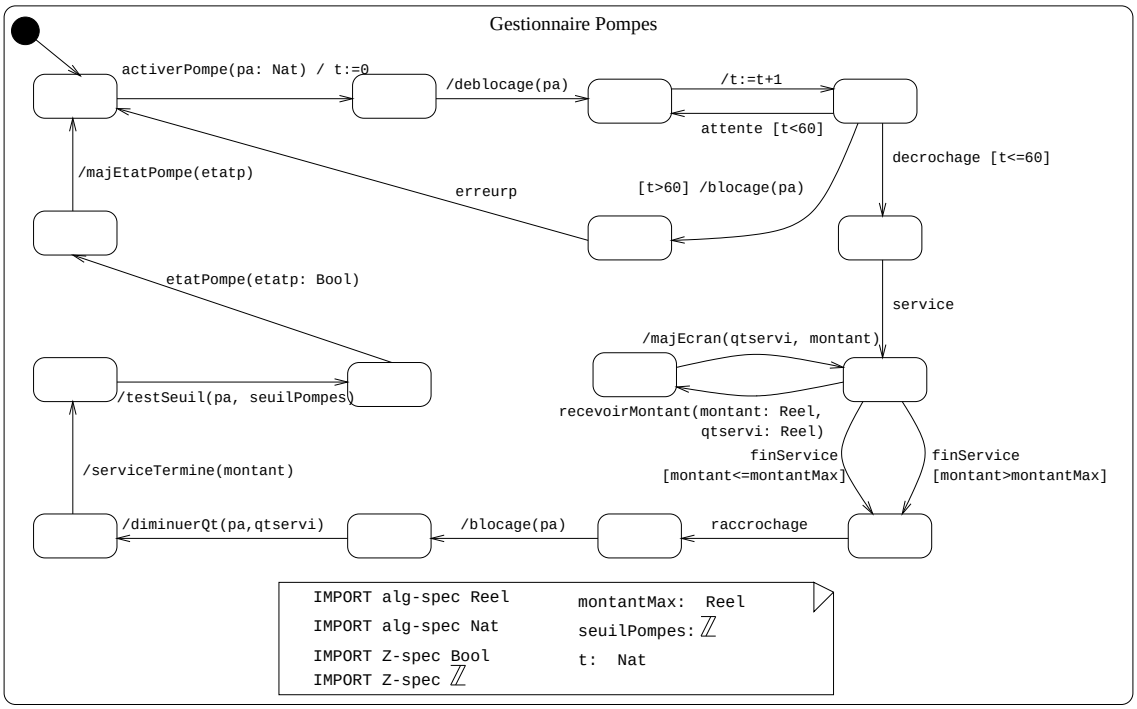

FIG. 12 – Diagramme du gestionnaire de pompes

Le gestionnaire de cuves utilise comme données locales la variable `etatp` et le schéma d'état *GCuves* qui permet la gestion des données concernant les cuves. Il possède aussi par ce biais les données représentant le niveau de remplissage des trois cuves (*qtPompe*). Le gestionnaire de cuves interagit avec le gestionnaire de pompes (`diminuerQt`) et avec l'environnement pour l'approvisionnement en carburant (`augmenterQt`). Il réalise les opérations de mise à jour de données effectuées sur les cuves (`majQt`) et le test des quantités avec `testSeuil`.

Nous avons montré dans cette étude la façon d'utiliser concrètement l'intégration des données formelles dans les diagrammes d'états sur la base des fondements théoriques présentés dans la section 2. Les données sont formalisées et analysées au préalable en Z et en LP. Nos diagrammes d'états interragissent d'une part avec la façon habituelle à travers les événements partagés et d'autre part en échangeant les données introduites, via un mécanisme de communication (émission/réception) que

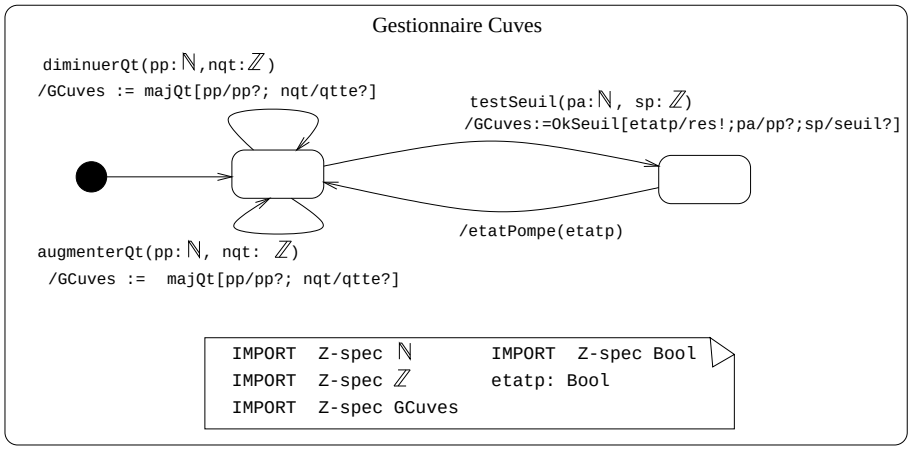

FIG. 13 – Diagramme pour le gestionnaire de cuves

nous avons formalisé dans la section 2. Nous obtenons aini un modèle intégrant les diagrammes d'états et des données formelles.

4 Travaux connexes

Dans cette section, nous comparons notre approche avec les travaux existants qui visent à intégrer ou combiner les diagrammes d'états d'UML (ou plus généralement les Statecharts) avec des langages de spécification formelle concernant la spécification des aspects statiques. Comme nous l'avons précisé en introduction, les travaux existants imposent l'utilisation d'un langage de définition de type de donnée unique et sont donc moins flexibles que notre approche. Le niveau de réutilisabilité des composants de spécification est donc plus grand dans notre approche.

Diagrammes d'états et OCL. En complément de la description des données dans les diagrammes de classes (qui présente, malgré les formalisations existantes des diagrammes de classe [28], l'inconvénient majeur de poser le problème de la cohérence multi-diagrammes en UML), l'extension la plus usuelle des diagrammes d'états en UML consiste à leur associer des contraintes exprimées en OCL (Object Constraint Language). OCL n'est toutefois essentiellement fait que pour exprimer des contraintes sur les données internes ou associées aux classes et objets. OCL n'est donc pas à proprement parler un moyen de décrire des données abstraites ou formelles. Des travaux toutefois s'intéressent à la formalisation d'OCL [20, 22].

Toute une partie de la recherche actuelle s'intéresse aussi à l'utilisation des diagrammes d'états dans le cadre de la vérification de systèmes réactifs et/ou temps réel (extension avec des données de base, entières, représentant des horloges). Ces travaux, comme par exemple [25, 42], procèdent habituellement, après formalisation des diagrammes d'états éventuellement restreints, par traduction vers les formalismes d'entrée d'outils de vérification de modèle ou de vérification de systèmes hybrides comme Uppaal [39]. S'il s'agit de travaux intéressants quand à la prise en compte d'aspects temporels, ils ne nous paraissent pas satisfaisants quand à la prise en compte de vraies données abstraites (*i.e.* autres données que les horloges) qui ne peuvent pas être traduites efficacement vers les outils en question.

Diagrammes d'états et Z. Bussow et Weber ont travaillé sur une méthodologie pratique pour la spécification de systèmes de contrôle temps-réel [60, 18]. Leur modèle, μSZ, est composé de trois vues afin de gérer la complexité et favoriser la séparation des aspects : (1) la vue architecturale spécifiée avec des diagrammes d'objets et de classes ; (2) la vue réactive spécifiée avec les Statecharts ; (3) la vue fonctionnelle spécifiée avec Z. Les liens systématiques entre les vues réactives et fonctionnelles entraînent des obligations de preuves pour garantir la compatibilité sémantique.

D'autres travaux se sont intéressés aux liens entre diagrammes d'UML et Z. On peut citer RoZ [26], ou FuZed [16] initialement basé sur Fusion qui a été appliqué à UML. Ils ne prennent pas en compte les diagrammes d'états d'UML. Toutes ces approches procèdent cependant par traduction dans Z. Récemment, l'approche de RoZ a été combinée avec des diagrammes d'états pour adresser le problème de la cohérence entre statique et dynamique [43]. Il s'agit d'une approche complémentaire à la notre au sens où nous nous préoccupons aussi ces problèmes de cohérence. Notre approche propose cependant une démarche plus dynamique (sémantique opérationnelle) et cherche à généraliser au niveau de la statique (*i.e.* Z mais aussi les spécifications algébriques par exemple).

Diagrammes d'états et méthode B. Les premiers travaux sont ceux de Sekerinski [54] qui visent à concevoir graphiquement des systèmes réactifs en utilisant les Statecharts. Son approche présente à cet effet une traduction des Statecharts dans la notation des machines abstraites (AMN) de la méthode B. La principale contribution est de spécifier un système à partir du formalisme convivial que constituent les Statecharts, et de traduire ensuite la spécification en machines abstraites B. Ces dernières constituent un cadre homogène pour les étapes suivantes de développement formel (*e.g.* analyse des propriétés de sûreté) et particulièrement le raffinement. Par cette traduction, l'auteur donne aux Statecharts une sémantique formelle en terme d'AMN. L'outil *iState* implémente ces mécanismes de traduction vers B, mais aussi vers différents langages de programmation [55, 56].

Lano *et al* [37, 38] décrivent une méthode pour supporter graphiquement la conception de systèmes en utilisant les AMN de B, ainsi que des lignes directrices pour exprimer la structuration d'aspects réactifs en B. Seule une version simplifiée des Statecharts (RSDS) est prise en compte afin d'éviter certaines complications découlant des divergences sémantiques bien connues pour ce formalisme (spécialement à propos de l'enchaînement des événements). La finalité est d'utiliser les supports de développement offerts par la méthode B.

Ledang et Souquières [40] proposent des contributions à la modélisation des diagrammes d'état d'UML en B. Les auteurs se focalisent sur la traduction en B de la partie concernant les événements (et non celle liée aux activités). Ils proposent une nouvelle approche pour modéliser les événements différés ou non différés et prendre en compte la sémantique particulière de ces événements. De plus, une modélisation de la communication à l'intérieur des diagrammes d'états d'UML est présentée. Nous pouvons également citer les thèses de Meyer [44] et Nguyen [46] qui ont travaillé sur la traduction des diagrammes d'état dans la méthode B.

Comme dans les approches combinant les diagrammes d'états avec Z, les travaux procèdent ici généralement par traduction vers B qui offre ensuite un cadre de preuve. Laleau et Polack se sont intéressées récemment au problème d'avoir un système de traduction fonctionnant dans les deux sens [36].

Dans le cas des approches utilisant Z, comme dans celles utilisant B, la différence principale par rapport à nos travaux est que nous cherchons à utiliser directement les sémantiques des parties dynamiques et statiques sans traduction de l'une d'entre elle. Nous pensons que cette approche permet plus facilement une réutilisation des preuves. En effet, cela permet au spécifieur de réutiliser une preuve concernant l'un des aspects (et donc faite dans le cadre qu'il connaît et avec les outils dédiés à cet aspect) dans le cadre d'une preuve concernant le système global. Nous espérons qu'à terme cela permette une forme de compositionnalité multi-langages de spécification. Dans le cas d'une intégration complète dans l'un des formalismes (*i.e.* concrètement en Z ou en B), le spécifieur devra entièrement faire sa preuve dans ce cadre.

Diagrammes d'états et spécifications algébriques. Reggio et Repetto ont proposé Casl-Chart, un langage de spécification formelle visuel pour la modélisation des systèmes réactifs [51]. Ce langage combine les Statecharts (suivant la sémantique données par STATEMATE [29]) avec le langage de spécifications algébriques CASL [24]. Une spécification écrite en Casl-Chart est composée de types de données décrits en CASL ainsi que de Statecharts pouvant utiliser ces types de données (événements, gardes et actions). La sémantique du langage combiné est définie à partir des sémantiques originales des deux langages de base.

Nos travaux sont donc très proches de Casl-Chart. Toutefois, notre approche est plus flexible au niveau des données puisqu'elle donne la possibilité de les spécifier en utilisant différents formalismes. Notre approche est aussi plus flexible au niveau de la dynamique puisque nous proposons un cadre sémantique réutilisable à base d'environnements pour l'intégration de données formelles dans d'autres formalismes dynamiques.

5 Conclusion et perspectives

Le but de notre travail est de proposer un langage convivial, et plus généralement une méthode d'intégration, adaptés à la spécification de systèmes complexes. Nous avons choisi de combiner les diagrammes d'états d'UML avec des techniques de spécification formelle de types de données abstraits (spécifications algébriques, Z, B). Cette utilisation conjointe d'une notation semi-formelle pour la modélisation visuelle des aspects dynamiques avec des langages formels pour spécifier les aspects statiques permet de profiter des avantages des deux approches. Contrairement aux approches existantes, notre approche se place à un haut niveau d'abstraction. Elle permet en effet l'utilisation des différentes sémantiques des diagrammes d'états d'UML et plus généralement de celles des Statecharts ou d'autres langages basés sur des états et des transitions comme les travaux récents autour des systèmes de transitions symboliques [30, 19, 21]. Notre approche est aussi plus souple et plus flexible puisqu'elle permet l'utilisation de différents langages de description des données, améliorant par là-même le niveau de réutilisabilité des spécifications.

Les spécifications ainsi obtenues peuvent de plus être vérifiées partiellement soit en ce qui concerne la partie dynamique par vérification de modèle ou animation des spécifications (utilisation directe de STATEMATE en oubliant les données, traduction

de la sémantique dynamique dans le format d'entrée d'outils tels que CWB), soit en ce qui concerne la partie données par utilisation des prouveurs de théorèmes ou des environnements dédiés aux langages utilisés dans l'intégration (CATS [4] pour CASL, Larch Prover [27] pour Larch, AtelierB [1] ou B-Toolkit [2] pour B, Z/EVES [3] pour Z).

Une première perspective concerne l'approfondissement des aspects de vérification. En effet, s'il est déjà possible de pratiquer des vérifications sur les aspects pris séparément, il est important de pouvoir vérifier la spécification dans sa globalité. Une piste en cours est la traduction de notre langage et de sa sémantique dans le format d'entrée d'outils puissants basés sur les logiques d'ordre supérieur tel que PVS [32] ou Isabelle [47] en s'inspirant de la logique pour systèmes mixtes présentée dans [6], des travaux de [7, 59, 12, 58] pour la dynamique et ceux de [14, 45, 35] pour la statique. Pour ce faire, il faut respectivement exprimer en PVS les composants de base modélisant les aspects statiques et dynamiques ainsi que les interactions entre les deux, puis proposer des techniques d'aide à la preuve tenant compte de l'aspect mixte des spécifications.

Nous pensons que les approches procédant par traduction de l'ensemble de la spécification dans le langage de spécification des aspects statiques [55, 56, 40, 50] seraient aussi utilisables mais elles présentent l'inconvénient majeur de ne pas permettre une forme de rétro ingénierie de la preuve (*i.e.* après traduction l'aspect dynamique se retrouve noyé dans une spécification statique ce qui nuit à la lisibilité de l'ensemble au moment de la preuve). Les travaux récents de Laleau et Polack [36] constituent un premier pas permettant de répondre à ce problème. Plus généralement c'est à une *intégration de la preuve* qu'il convient de réfléchir, c'est-à-dire donner les moyens au spécifieur de réutiliser dans le cadre d'une preuve sur le système global des preuves faites sur les différents aspects ; et ce bien sur en tenant compte du fait que les spécifications utilisent potentiellement différents formalismes.

Une autre perspective de ces travaux concerne la généralisation de l'approche dans le but de combiner différents formalismes basés sur l'intégration de données formelles dans des systèmes de transitions (LOTOS [31], SDL [49], Statecharts, Korrigan [21]). En effet, notre approche est relativement générale et flexible en ce qui concerne la dynamique et la statique pour prendre en compte une famille relativement grande de langages mixtes.

Nous disposons déjà d'un outil permettant d'animer des spécifications écrites en CCS avec passage de valeur. Cet outil, ISA [11], permet de gérer des données potentiellement écrites dans n'importe quel langage de spécification et pour lequel il existe une fonction d'évaluation. Cette fonction est alors codée en tant que module externe auquel ISA fait appel. Actuellement seules les données en python son gérées. Nous prévoyons de gérer des spécifications algébriques (avec initialité) par appel à un moteur externe de réécriture de termes (ex. ELAN [15]). Une autre perspective est d'étendre cet outil de façon modulaire et générique pour prendre en compte différents langages de spécification concernant la partie dynamique. Dans un second temps, sa version actuelle (CCS et données) mais aussi les diagrammes d'états d'UML (en suivant les principes décrits ici) pourraient donner lieu à des instanciations particulières de ce framework.

Références

[1] http ://www.atelierb.societe.com/index_uk.html.

[2] http ://www.b-core.com/btoolkit.html.

[3] http ://www.ora.on.ca/z-eves/.

[4] http ://www.tzi.de/cofi/Tools/CATS.html.

[5] J. R. ABRIAL. *The B-Book.* Cambridge University Press, 1996.

[6] M. AIGUIER, F. BARBIER, et P. POIZAT. A Logic for Mixed Specifications. Rapport technique 73-2002, LaMI, 2002. Available at ftp ://ftp.lami.univ-evry.fr/pub/specif/poizat/documents/RR-ABP02.ps.gz.

[7] M. ALLEMAND. Verification of properties involving logical and physical timing features. In *Génie Logiciel & Ingénierie de Systèmes & leurs Applications (ICSSEA'00)*, 2000.

[8] M. ALLEMAND, C. ATTIOGBE, P. POIZAT, J.-C. ROYER, et G. SALAÜN. SHE'S Project : a Report of Joint Works on the Integration of Formal Specification Techniques. In *Workshop on Integration of Specification Techniques with Applications in Engineering (INT'02)*, 2002.

[9] E. ASTESIANO. UML as Heterogeneous Multiview Notation. Strategies for a Formal Foundation. In L. ANDRADE, A. MOREIRA, A. DESHPANDE, et S. KENT, réds., *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'98) – Workshop on Formalizing UML. Why ? How ?*, 1998.

[10] E. ASTESIANO, H.-J. KREOWSKI, et B. KRIEG-BRÜCKNER, réds.. *Algebraic Foundations of System Specification*. Springer-Verlag, 1999.

[11] C. ATTIOGBÉ, A. FRANCHETEAU, J. LIMOUSIN, et G. SALAÜN. ISA, a tool for Integrated Specifications Animation. Available at http ://www.sciences.univ-nantes.fr/info/perso/permanents/salaun/ISA/isa.html.

[12] T. BASTEN et J. HOOMAN. Process Algebra in PVS. In W. R. CLEAVELAND, réd., *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99)*, volume 1579 of *Lecture Notes in Computer Science*, pages 270–284, The Netherlands, 1999. Springer Verlag.

[13] M. BIDOIT. *Pluss, un langage pour le développement de spécifications algébriques modulaires*. Thèse d'état, Université de Paris-Sud – Centre d'Orsay, 1989.

[14] J.-P. BODEVEIX, M. FILALI, et C. MUNOZ. A Formalization of the B Method in Coq and PVS. In *B Users Group Meeting – Applying B in an industrial context: Tools, Lessons and Techniques (FM'99)*, pages 32–48. Springer Verlag, 1999.

[15] P. BOROVANSKÝ, C. KIRCHNER, H. KIRCHNER, P.-E. MOREAU, et C. RINGEISSEN. An Overview of ELAN. In C. KIRCHNER et H. KIRCHNER, réds., *International Workshop on Rewriting Logic and its Applications*, volume 15 of *Electronic Notes in Theoretical Computer Science*, France, 1998. Elsevier Science.

[16] J.-M. BRUEL et R. B. FRANCE. Transforming UML to Formal Specifications. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'98)*, 1998.

[17] G. BRUNS. A Language for Value-Passing CCS. LFCS Report Series ECS-LFCS-91-175, University of Edinburgh, 1995.

[18] R. BÜSSOW et M. WEBER. A Steam-Boiler Control Specification with Statecharts and Z. In J.-R. ABRIAL, E. BÖRGER, et H. LANGMAACK, réds., *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler*, volume 1165 of *Lecture Notes in Computer Science*, pages 109–128. Springer Verlag, 1996.

[19] M. CALDER, S. MAHARAJ, et C. SHANKLAND. A Modal Logic for Full LOTOS Based on Symbolic Transition Systems. *The Computer Journal*, 45(1) :55–61, 2002.

[20] M. V. CENGARLE et A. KNAP. A Formal Semantics for OCL 1.4. In M. GOGOLLA et C. KOBRYN, réds., *The Unified Modeling Language. Modeling Languages, Concepts, and Tools (UML'01)*, volume 2185 of *Lecture Notes in Computer Science*, pages 118–133. Springer-Verlag, 2001.

[21] C. CHOPPY, P. POIZAT, et J.-C. ROYER. A Global Semantics for Views. In T. RUS, réd., *International Conference on Algebraic Methodology And Software Technology (AMAST'00)*, volume 1816 of *Lecture Notes in Computer Science*, pages 165–180. Springer-Verlag, 2000.

[22] T. CLARK et J. WARMER, réds.. *Object Modeling with the OCL, The Rationale behind the Object Constraint Language*, volume 2263 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[23] COFI. The Common Framework Initiative for Algebraic Specification and Development, electronic archives. Notes and Documents accessible by WWW[1] and FTP[2].

[1] http ://www.brics.dk/Projects/CoFI

[2] ftp ://ftp.brics.dk/Projects/CoFI

[24] CoFI Language Design Task Group. CASL – The CoFI Algebraic Specification Language – Summary (version 1.0.1). `Documents/CASL/Summary` in [23], March 2001.

[25] A. David, M. O. Möller, et W. Yi. Formal Verification of UML Statecharts with Real-Time Extensions. In R.-D. Kutsche et H. Weber, réds., *International Conference on Fundamental Approaches to Software Engineering (FASE'02)*, Lecture Notes in Computer Science, pages 218–232. Springer-Verlag, 2002.

[26] S. Dupuy, Y. Ledru, et M. Chabre-Peccoud. An Overview of RoZ : A Tool for Integrating UML and Z Specifications. In B. Wangler et L. Bergman, réds., *Advanced Information Systems Engineering Conference (CAiSE'00)*, volume 1789 of *Lecture Notes in Computer Science*, pages 417–430. Springer-Verlag, 2000.

[27] S. J. Garland et J. V. Guttag. A Guide to LP, the Larch Prover. Technical Report, Palo Alto, California, 1991.

[28] The Precise UML Group. `http ://www.cs.york.ac.uk/puml/`.

[29] D. Harel et A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4) :293–333, 1996.

[30] M. Hennessy et H. Lin. Symbolic Bisimulations. *Theoretical Computer Science*, 138(2) :353–389, 1995.

[31] ISO/IEC. LOTOS : A Formal Description Technique based on the Temporal Ordering of Observational Behaviour. ISO/IEC 8807, International Organization for Standardization, 1989.

[32] J. Crow, S. Owre, J. Rushby, N. Shankar, et M. Srivas. A Tutorial Introduction to PVS. In *Workshop on Industrial-Strength Formal Specification Techniques (WIFT'95)*, USA, 1995. Computer Science Laboratory, SRI International.

[33] H. Kirchner et C. Ringeissen. Executing CASL Equational Specifications with the ELAN Rewrite Engine. Note T-9 in [23], November 2000.

[34] J. W. Klop. Term Rewriting Systems. In S. Abramsky, D. M. Gabbay, et T. S. E. Maibaum, réds., *Handbook of Logic in Computer Science*, volume 2, chapitre 1, pages 1–117. Oxford University Press, Oxford, 1992.

[35] S. Kolyang et T. Wolff. A Structure Preserving Encoding of Z in Isabelle/HOL. In *International Conference on Theorem Proving in Higher Order Logic (TPHOL'96)*, volume 1125 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[36] R. Laleau et F. Polack. Coming and Going from UML to B : A Proposal to Support Traceability in Rigorous IS Development. In D. Bert, J. P. Bowen, M. C. Henson, et K. Robinson, réds., *International Z and B Conference (ZB'02)*, volume 2272 of *Lecture Notes in Computer Science*, pages 517–534. Springer-Verlag, 2002.

[37] K. Lano, K. Androutsopoulos, et D. Clark. Structuring and Design of Reactive Systems Using RSDS and B. In T. S. E. Maibaum, réd., *International Conference on Fundamental Approaches to Software Engineering (FASE'00)*, volume 1783 of *Lecture Notes in Computer Science*, pages 376–390, Germany, 2000. Springer Verlag.

[38] K. Lano, K. Androutsopoulos, et P. Kan. Structuring Reactive Systems in B AMN. In *3rd IEEE International Conference on Formal Engineering Methods (ICFEM'00)*, pages 25–34, UK, 2000. IEEE Computer Society Press.

[39] K. G. Larsen, P. Pettersson, et W. Yi. Uppaal in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1–2) :134–152, 1997.

[40] H. Ledang et J. Souquières. Contributions for Modelling UML State-Charts in B. In M. Butler, L. Petre, et K. Sere, réds., *3rd International Conference on Integrated Formal Methods (IFM'02)*, volume 2335 of *Lecture Notes in Computer Science*, pages 109–127, Finland, 2002. Springer Verlag.

[41] J. Lilius et I. Porres Paltor. The Semantics of UML State Machines. Rapport technique 273, Turku Centre for Computer Science, 1999. Available at `http ://www.tucs.fi/Publications/techreports/TR273.php`.

[42] J. Lilius et I. Porres Paltor. UML : A Tool for Verifying UML Models. In *International Conference on Automated Software Engineering (ASE'99)*, pages 255–258. IEEE Computer Society, 1999. Extended version available at `http ://www.tucs.fi/Publications/techreports/TR272.php`.

[43] O. Maury, C. Oriat, et Y. Ledru. Invariants de liaison pour la cohérence de vues statiques et dynamiques en UML. In *Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'01)*, pages 23–37, 2001.

[44] E. Meyer. *Développements formels par objets : utilisation conjointe de B et d'UML*. Thèse de doctorat, LORIA – Université Nancy 2, 2001.

[45] T. Mossakowski et B. Krieg-Brückner. Static semantic analysis and theorem proving for CASL. In Francesco Parisi-Presicce, réd., *Recent Trends in Algebraic Development Techniques, 12th Workshop on Algebraic Development Techniques (WADT'97)*, volume 1376, pages 333–348. Springer-Verlag, 1998.

[46] H. P. NGUYEN. *Dérivation de spécifications formelles B à partir de spécifications semi-formelles*. Thèse de doctorat, CEDRIC – CNAM, 1998.

[47] T. NIPKOW et L. C. PAULSON. Isabelle-91. In D. KAPUR, réd., *11th International Conference on Automated Deduction (CADE'92)*, Lecture Notes in Artificial Intelligence, pages 673–676. Springer-Verlag, 1992.

[48] Object Modeling Group. *Unified Modelling Language Specification, version 1.4*, Septembre 2001.

[49] P. POIZAT. *Software Specification Methods : An Overview Using a Case Study*, chapitreSDL : a Specification and Description Language based on an Extended Finite State Machine Model with Abstract Data Types. Formal Approaches to Computing and Information Technology (FACIT). Springer-Verlag, 2000.

[50] G. REGGIO, E. ASTESIANO, et C. CHOPPY.. CASL-LTL : A CASL Extension for Dynamic Reactive Systems - Summary. Rapport technique DISI-TR-99-34, DISI - Università di Genova, 1999. Revised February 2000.

[51] G. REGGIO et L. REPETTO. Casl-Chart: A Combination of Statecharts and of the Algebraic Specification Language Casl. In T. RUS, réd., *8th International Conference on Algebraic Methodology and Software Technology (AMAST'00)*, volume 1816 of *Lecture Notes in Computer Science*, pages 243–257, USA, 2000. Springer Verlag.

[52] G. REGGIO et R. WIERINGA. Thirty one Problems in the Semantics of UML 1.3 Dynamics. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99) – Workshop "Rigorous Modelling and Analysis of the UML : Challenges and Limitations"*, 1999.

[53] G. SALAÜN, M. ALLEMAND, et C. ATTIOGBÉ. An Approach to Combine Heterogeneous Specification Components. In *7th International Workshop on Formal Methods and Component Interaction (FMCI'02)*, Electronic Notes in Theoretical Computer Science (ENTCS) series, Spain, July 2002.

[54] E. SEKERINSKI. Graphical Design of Reactive Systems. In D. BERT, réd., *2nd International B Conference (B'98)*, volume 1393 of *Lecture Notes in Computer Science*, pages 182–197, France, 1998. Springer Verlag.

[55] E. SEKERINSKI et R. ZUROB. iState: A Statechart Translator. In M. GOGOLLA et C. KOBRYN, réds., *4th International Conference on the Unified Modeling Language (UML'01)*, volume 2185 of *Lecture Notes in Computer Science*, pages 376–390, Canada, 2001. Springer Verlag.

[56] E. SEKERINSKI et R. ZUROB. Translating Statecharts to B. In M. BUTLER, L. PETRE, et K. SERE, réds., *3rd International Conference on Integrated Formal Methods (IFM'02)*, volume 2335 of *Lecture Notes in Computer Science*, pages 128–144, Finland, 2002. Springer Verlag.

[57] J. M. SPIVEY. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd édition, 1992.

[58] H. TEJ et B. WOLFF. A Corrected Failure-Divergence Model for CSP in Isabelle/HOL. In C. JONES et J. FITZGERALD, réds., *International Formal Methods Europe Symposium (FME'97)*, 1997.

[59] I. TRAORÉ. An Outline of PVS Semantics for UML Statecharts. *Journal of Universal Computer Science (JUCS)*, 6(11) :1088–1108, 2000.

[60] M. WEBER. Combining Statecharts and Z for the Design of Safety-Critical Control Systems. In M. GAUDEL et J. WOODCOCK, réds., *3rd International Symposium of Formal Methods Europe (FME'96)*, volume 1051 of *Lecture Notes in Computer Science*, pages 307–326, UK, 1996. Springer Verlag.

Intégration de données formelles dans les diagrammes d'états d'UML

Christian ATTIOGBÉ[*], Pascal POIZAT[**], Gwen SALAÜN[*]

Résumé

Dans ce rapport, nous présentons une approche générique pour intégrer des données exprimées dans des langages de spécification formelle à l'intérieur de diagrammes d'états d'UML. Les motivations principales sont d'une part de pouvoir modéliser les aspects dynamiques des systèmes complexes avec un langage convivial et graphique tel que les diagrammes d'états d'UML ; d'autre part de pouvoir spécifier de façon formelle et potentiellement à un haut niveau d'abstraction les données mises en jeu dans ces systèmes à l'aide de spécifications algébriques ou de spécifications orientées état (Z, B). Cette approche introduit une utilisation flexible et générique des données. Elle permet aussi de prendre en compte différentes sémantiques associées aux diagrammes d'états. Nous présentons d'abord les fondements formels de notre approche puis une étude de cas vient illustrer le pragmatisme de la proposition.

Catégories et descripteurs de sujets : D.2.1 [**Software Engineering**]: Requirements/Specifications; D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.3 [**Programming Languages**]: Language Constructs and Features

Termes généraux : Design, Languages, Theory

Mots-clés additionnels et phrases : Intégration de spécifications formelles et semi-formelles, diagrammes d'états, UML, spécifications algébriques, Z, B

IRIN, Université de Nantes
2, rue de la Houssinière
B.P. 92208 — F-44322 NANTES CEDEX 3