DEPS : UN LANGAGE POUR LA SPECIFICATION DE PROBLEMES DE CONCEPTION DE SYSTEMES

P.A. YVARS

Institut Supérieur de Mécanique de Paris (SupMeca) - LISMMA
3 rue Fernand Hainaut - 93407 St Ouen Cedex – France
pierre-alain.yvars@supmeca.fr

L. ZIMMER

Dassault Aviation - Direction de la Prospective
78 quai Marcel Dassault - 92252 St Cloud – France
laurent.zimmer@dassault-aviation.com

RESUME : *Nous présentons DEPS (Design Problem Specification) un nouveau langage de modélisation conçu pour poser et résoudre des problèmes de conception de systèmes. Ce langage combine des traits de modélisation structurelle propres aux langages orientés objet avec des traits de spécification de problème propres aux langages de programmation mathématique. Il permet d'encapsuler la formulation mathématique des problèmes dans des modèles que l'on peut organiser en fonction de l'architecture du système étudié. La représentation de problème à base de modèles qui est proposée à des fins de résolution ou d'optimisation est plus adaptée à une utilisation en ingénierie de systèmes que celle, purement équationnelle, des optimiseurs habituellement utilisés.*
Pour présenter les éléments du langage DEPS et illustrer son intérêt en conception de système, nous modélisons un problème complet de conception d'un petit système robotique. Nous ferons ensuite un point sur l'état actuel du développement du langage et de son compilateur puis nous donnerons des pistes de travaux futurs en modélisation comme en résolution.

MOTS-CLES : *langage, spécification, conception, résolution, synthèse de système.*

1 INTRODUCTION

L'objectif du langage DEPS est de permettre d'abord de poser formellement un problème de conception donné et ensuite, après compilation, soit le résoudre c'est-à-dire trouver une ou plusieurs solutions si elles existent soit l'optimiser c'est-à-dire trouver la ou les meilleures solutions par rapport à des critères donnés.

Dans le cadre de cet article nous définirons informellement les problèmes de conception de système en les opposant aux problèmes d'évaluation de performance de systèmes. L'évaluation de performance nécessite de disposer d'une description complète du système étudié tandis que la conception vise à compléter une description plus ou moins partielle du système à étudier. Une description peut-être partielle parce que : des paramètres de conception ne sont pas (encore) fixés, le choix de certains composants n'est pas déterminé, l'allocation des ressources n'est pas terminée voire l'organisation du système et de ses sous-systèmes n'est pas figée. Rentrent donc dans cette définition et sans prétention d'exhaustivité les problèmes de dimensionnement, de configuration, d'allocation, de génération d'architecture ou une combinaison de ceux-ci.

Comme les descriptions partielles de système sont naturellement plus fréquentes dans les phases amont de définition des systèmes, DEPS est un candidat naturel pour la conception préliminaire même si son usage n'est pas limité à cette étape. En conception préliminaire, DEPS peut être considéré comme un outil de synthèse qui propose des systèmes admissibles par opposition à des outils d'analyse utilisés à des étapes ultérieures pour vérifier ou valider un système.

Le langage DEPS est conçu comme une combinaison entre langage orienté objet et langage de programmation mathématique. Aux premiers ont été empruntés les traits de structuration et d'abstraction qui facilitent la modélisation des problèmes de conception parce qu'ils permettent, notamment, de capturer l'architecture du système étudié. Aux seconds ont été empruntés les concepts mathématiques nécessaires à la résolution du problème : inconnues, équations, et inéquations.

L'idée de combiner approche orientée objet et méthodes de calcul pour proposer des formalismes de résolution de problèmes déclaratifs et de haut niveau dans les sciences de l'ingénieur n'est pas nouvelle. Nous décrivons dans la suite les travaux par rapport auxquels nous nous positionnons en distinguant d'une part des travaux précurseurs que l'on peut rattacher à la problématique générale de la Représentation des Connaissances en Intelligence Artificielle et d'autre part des travaux plus récents entrepris en ingénierie des systèmes à base de modèle et qui sont consécutifs au développement et à la diffusion des langages de modélisation de l'ingénierie système.

2 TRAVAUX PRECURSEURS

Dans le logiciel précurseur ThingLab (Borning, 1977) des contraintes stockées dans la classe d'un objet permettent d'exprimer des relations constamment vraies. On peut exprimer, par exemple, que toute résistance obéit à la loi d'Ohm. Mais ThingLab a été conçu pour produire des simulations animées. Les méthodes de résolution utilisées, suffisantes dans ce domaine, sont inefficaces pour résoudre les systèmes d'équations et d'inéquations des problèmes de dimensionnement en Ingénierie. Par ailleurs, dans ThingLab, une contrainte est déclarée dans un objet et son comportement est programmé dans des méthodes de cet objet. Ce caractère procédural de la représentation des contraintes ne permet pas de considérer ThingLab comme un langage déclaratif de modélisation.

Le langage NEMO-TEC (Shvetsov et al., 1997) est décrit comme une extension orientée objet du solveur de problème mathématique Unicalc basé sur la technologie des modèles sous-définis. Cette technologie russe est comparable aux méthodes de programmation par contraintes d'intervalle. Un problème de positionnement des éléments (volant, pédales, siège) d'un poste de conduite automobile est donné comme exemple d'application. Ces éléments et un conducteur type sont modélisés à l'aide d'objets (points, segments, angles, triangles) dépendant géométriquement les uns des autres. Le modèle résultant montre l'intérêt de l'approche objet pour spécifier le problème. Par contre, l'utilisation du calcul sous-défini permet seulement de limiter le domaine admissible des positions et des angles des éléments mais ne résout pas à proprement parler de système. En effet, NEMO-TEC a été essentiellement conçu pour générer un système de calcul autonome, composé d'un modèle compilé du réseau de dépendances entre les inconnues du problème et d'une interface utilisateur. Ce système doit guider un utilisateur dans ses choix successifs de conception et les valider par propagation. Le caractère interactif de la conception est privilégié au détriment de la capacité de résolution dans ce travail.

L'ONERA dans le cadre de travaux de recherche en conception préliminaire d'avion, a proposé de combiner programmation logique avec contraintes (PLC) et concepts objet pour représenter et résoudre des problèmes de configuration (Bensana et Mulyanto, 2000). La programmation logique apporte les capacités de recherche arborescente et d'écriture de règles logiques ; sa généralisation au cadre PLC apporte les capacités de formulation et de résolution de contraintes mathématiques sur des variables discrètes ou continues. Les concepts de l'Orienté Objet (Classes, Attributs, Instances) permettent, eux, de capturer la structure du produit (i.e. sa décomposition en classes) et de l'utiliser pour proposer des configurations (i.e. différentes instanciations du produit). Un prototype de logiciel de configuration a été développé. Il s'appuie sur des langages impératifs standards de l'Internet pour le développement de l'interface graphique et sur un langage de programmation logique avec contraintes pour d'une part le développement d'un prédicat de résolution du problème de configuration, appelé « fonction de synthèse », et d'autre part pour le développement d'une modélisation objet du produit étudié. Ces deux derniers développements ont été portés en Prolog IV (Bouvier et al., 1996) et en ECLiPSe (Apt et Wallace, 2006), deux langages de programmation logique avec contraintes. Le prototype a été appliqué au problème de configuration d'un avion sans pilote qui peut se composer de plusieurs sortes de fuselages, voilures, propulsions et capteurs.
Dans sa thèse, Mulyanto (Mulyanto, 2002) décrit l'implémentation de la représentation orientée objet de ce prototype. Le produit à configurer est décrit par sa *nomenclature générique* c'est-à-dire l'ensemble des classes dont il est composé et par *sa base de composants*, l'ensemble des composants techniques autorisés. Chaque objet, que ce soit une classe ou un composant est spécifié dans un fichier Prolog qui lui est dédié. Chaque fichier est constitué d'une clause qui décrit la structure de l'objet et de prédicats qui définissent son comportement. Au chargement du fichier la clause et les prédicats seront ajoutés aux clauses et aux prédicats du programme principal de résolution du problème de configuration. Cette implémentation permet de séparer clairement dans les fichiers la définition du problème de configuration (*nomenclature+base de composants*) de sa résolution. Par contre, elle mélange dans la représentation objet les traits procéduraux des prédicats avec les traits déclaratifs des clauses et les équations doivent être décomposées en prédicats binaires et ternaires. Au final si l'implémentation proposée permet de simuler des traits objets dans un programme Prolog, le cadre reste fondamentalement celui de la programmation logique avec contraintes. En outre, la portée reste limitée à l'expression de problèmes de configuration et la représentation des concepts mathématiques reste procédurale. On n'a donc affaire ni à un langage objet à part entière ni à un véritable langage déclaratif de modélisation mathématique.

Le langage COB (Jayaraman et Tambay, 2002) développé à peu près à la même époque à l'Université de New York à Buffalo propose lui aussi de combiner contraintes et objets en s'appuyant sur les capacités de résolution d'un langage de programmation logique avec contraintes, en l'occurrence CLP(R). Mais il diffère notablement du développement précédent par un certain nombre de points exposés ci-dessous qui constituent des avancées notables.

D'abord COB est un vrai langage. Sa syntaxe est formellement spécifiée par une grammaire et sa sémantique opérationnelle est validée par le développement d'un compilateur (Tambay et Jayaraman, 2003). Ce compilateur traduit un programme COB en un programme CLP(R).

Ensuite le modèle théorique des objets de COB est déclaratif. Un *Objet Contraint* de COB est composé d'un ensemble d'*attributs* qui sont les variables caractérisant l'état interne d'un objet. Les méthodes impératives des langages objets sont remplacées par un ensemble de *contraintes* qui gouvernent ces attributs. Dans le cas où celles-ci sont des équations ou des inéquations algébriques la syntaxe de COB est conforme à la notation mathématique usuelle. C'est le compilateur qui prend en charge de façon transparente une transformation des formules à des fins de résolution.

Enfin, s'appuyant sur les capacités de résolution de systèmes d'équations dans les réels de CLP(R), COB aborde la modélisation et la résolution de problèmes réalistes de calcul en ingénierie (civile, chimique, électrique, mécanique …). Les objets sont utilisés pour modéliser la structure de systèmes complexes et les contraintes permettent de poser aussi bien des problèmes d'évaluation de performances comme le calcul de charges dans des treillis de poutre que des problèmes de dimensionnement comme le calcul du nombre d'engrenages et du nombre de dents d'un train d'engrenage.

Malheureusement COB reste trop influencé par CLP(R). En plus des attributs et des contraintes, une classe est aussi composée de prédicats et de constructeurs. Les prédicats permettent d'exprimer des traitements à l'aide de règles « à la Prolog » et les constructeurs permettent de construire les objets du problème. Ces traits procéduraux complexifient la représentation globale des problèmes : les règles manipulent des variables logiques qui ne sont pas de même nature que les variables qui sont des attributs des objets et l'utilisation des constructeurs est obligatoire à la déclaration (*constructor*) comme à l'invocation (*new*). Finalement COB s'apparente plus à une tentative de langage général de programmation multi-paradigme (classes et objets, méthodes, contraintes et règles) qu'à un langage spécifique de modélisation dédié à l'ingénieur.

Il faut remarquer que tous les développements présentés jusqu'ici, outre qu'ils sont relativement anciens, n'ont jamais dépassé le stade du prototype et qu'à notre connaissance ils n'ont pas inspiré de travaux académiques directs ces dix dernières années.

3 TRAVAUX RECENTS

Le développement et la diffusion des langages de modélisation de l'ingénierie logicielle (UML) puis système (SysML) ont relancé des travaux de recherche combinant orientation objet et calcul.

En effet, ces langages orienté objet sont bien adaptés à la description de systèmes. Par ailleurs, les capacités d'extension des ateliers logiciels qui les supportent ont motivé des recherches pour ajouter les capacités de calcul qui manquent naturellement aux environnements de base puisque, comme il est dit dans le tutoriel SysML de l'OMG (OMG, 2009) : *"Computational engine is provided by applicable analysis tool and not by SysML"*. Les travaux qui suivent rentrent dans le cadre des méthodes et outils de l'ingénierie dirigée par les modèles.

Il est nécessaire de mentionner OCL (*Object Constraint Language)* d'abord parce que ce langage est présenté comme un complément d'UML mais surtout parce que son acronyme prête à confusion. OCL est un langage de spécification qui permet typiquement de spécifier des conditions invariantes que doit vérifier un modèle UML pour être correct. Mais même dans sa version la plus récente (OCL, 2014) les contraintes d'OCL restent des spécifications dont l'évaluation à des fins de vérification n'est pas prise en charge par le langage lui-même et dont la résolution à des fins de dimensionnement n'est même pas envisagée. Ce langage est conçu pour vérifier qu'une instance d'un modèle est conforme à son modèle lors de sa saisie; il ne permet pas de dimensionner correctement le système modélisé.

La plate-forme s-COMMA (Soto, 2009) est une tentative d'utilisation d'UML pour produire un environnement combinant modélisation et résolution. L'idée est d'utiliser les capacités d'outils de méta-modélisation et de transformation de modèles associés au standard UML pour bâtir un langage visuel et orienté objet de modélisation de problèmes de satisfaction de contraintes. La plate-forme est composée de deux parties principales, un outil de modélisation et un outil de projection :

- Le langage de modélisation visuel combine les aspects déclaratifs de la programmation par contraintes avec les aspects de structuration de la programmation orientée objet ;
- l'outil de projection est un traducteur qui transforme la modélisation d'un problème décrit dans le langage précédent en un programme de programmation par contraintes dédié à un solveur cible donné.

Les avantages de l'approche sont :

- Une certaine indépendance du langage de modélisation de problèmes par rapport aux langages de résolution ;
- un développement d'outils de projection sur de nouveaux solveurs facilité par l'utilisation d'outils de développement dirigé par les modèles comme KM3 (langage de spécification de méta-modèles) ou ATL (langage de description de règles de transformation de modèles).

Le travail décrit dans ce papier est intéressant mais il cherche avant tout à démontrer l'intérêt d'une approche de développement dirigé par les modèles pour développer des outils de programmation par contraintes orientés objet dont les capacités de résolution dépendent du solveur cible visé. Dans le cas spécifique des problèmes de conception système qui nous intéressent l'intérêt de l'outil obtenu n'est pas établi. Le problème du « mariage stable » qui sert de fil conducteur dans l'article pour les différentes transformations de modèle est un petit pro-

blème combinatoire et pas un problème de conception système.

Les autres benchmarks cités dans l'article sont décrits en détail dans le manuel utilisateur de s-COMMA (Soto et Granvilliers, 2007) et recensés sur le site du langage. Tous les exemples donnés sont des problèmes combinatoires à variables entières y compris l'exemple « Engine ». Cet exemple donné comme un problème de conception de système technique s'avère être un petit problème de configuration qui illustre l'intérêt de la composition d'objets mais ne pose pas de problème de résolution. De plus sur les quatre solveurs supportés par s-COMMA, le seul qui soit théoriquement capable de résoudre des systèmes numériques non linéaires n'est jamais mis en œuvre sur un exemple. La capacité de s-COMMA à modéliser et résoudre ce type de problème n'est donc pas démontrée.

Par ailleurs, l'approche de développement dirigé par les modèles utilisée dans s-COMMA ne présente pas que des avantages. La description de l'architecture de compilation présentée dans le manuel utilisateur nous montre qu'elle est organisée en trois couches (modélisation, projection, résolution) et qu'il faut donc deux transformations pour les traverser : le modèle objet *s-comma* est compilé dans un premier temps en un modèle à plat *flat-s-comma* à la syntaxe plus proche de celle des langages de résolution cible puis ce modèle à plat sera lui-même compilé dans le langage cible d'un solveur donné.
Cette chaîne complexe de compilation découple le modèle de description du problème du modèle de résolution et pose le problème fondamental de la mise au point du modèle du problème. Toute erreur non syntaxique de modélisation sera détectée dans la couche de résolution sous la forme d'une erreur de calcul, d'un temps de résolution inacceptable ou de résultats incohérents. Les outils de mise au point fournis par le logiciel de résolution (messages d'erreur, traces d'exécution …) ne manipulent que les concepts du langage cible, celui du solveur, et sont par conséquent de trop bas niveau pour l'utilisateur qui ne devrait connaitre idéalement que le langage source de modélisation.

D'autres travaux d'ingénierie dirigée par les modèles préfèrent s'adosser directement à SysML puisque ce langage est intrinsèquement orienté Ingénierie des Systèmes. SysML permet notamment de décrire l'organisation structurelle d'un système et sa décomposition en sous-systèmes et composants à l'aide des *diagrammes de blocs* mais surtout il propose, un nouveau type de représentation que l'on ne trouve pas dans UML, les *diagrammes paramétriques*. Ce type de diagramme est constitué de blocs particuliers (avec un stéréotype « contraintes ») qui permettent d'intégrer dans le modèle la description des équations qui régissent le comportement du système. Le diagramme paramétrique peut donc servir de support pour des études d'analyse système ou de vérification d'exigences quantitatives à condition d'injecter les données dans des outils de calcul adéquats.

Pour les problématiques de dimensionnement qui nous intéressent, on peut citer des expérimentations industrielles, par exemple, le travail de la société InterCAX LLC qui commercialise ParaSolver, plug-in pour le logiciel Artisan Studio, un environnement de modélisation SysML. Les exemples proposés dans le tutorial (Parasolver, 2013) montrent l'intérêt de la démarche mais aussi les limites actuelles de l'approche c'est-à-dire une modélisation graphique très lourde avec de nombreuses redondances, même sur les exemples jouets présentés dans le tutoriel, et des capacités de résolution très limitées.

Toutes ces difficultés montrent que le sujet reste un sujet de recherche. A Georgia Tech, (Shah et al., 2012) ont conçu et prototypé dans un outil de modélisation SysML un logiciel de configuration et de dimensionnement de système qui a été évalué sur un problème de sélection et de dimensionnement des composants d'un système hydraulique. Les recherches menées ont porté sur l'amélioration des capacités de modélisation et des capacités de résolution. Il s'agit d'une part d'être capable de formuler le problème d'une façon qui convienne aux ingénieurs et d'autre part de le résoudre efficacement et si possible de manière optimale. Pour ce faire les chercheurs ont proposé de combiner ingénierie à base de modèle en SysML et programmation mathématique avec GAMS. SysML est utilisé pour modéliser le problème et GAMS pour le résoudre.
La conception et l'architecture du prototype sont comparables à celles de la plate-forme s-COMMA présenté ci-avant :

- Utilisation d'outils de développement dirigé par les modèles (MOF pour la méta-modélisation et MOFLON pour la transformation de modèle),
- des transformations successives pour passer d'un modèle SysML, étendu par l'utilisation de profils, orienté objet à un modèle à plat en GAMS.

Par contre, le prototype obtenu est plus crédible que s-COMMA vis-à-vis de la problématique qu'il adresse.
Le sous-système hydraulique d'une machine à débiter des troncs d'arbres qui est étudié présente toutes les caractéristiques d'un système industriel complexe et le problème à résoudre est un problème à l'échelle combinant configuration et dimensionnement du système en fonction d'exigences de performance et de coût.
Dans le papier, la modélisation du système, la formalisation des exigences et la représentation des équations dans les différents diagrammes SysML sont détaillées et les solutions trouvées par le solveur sont présentées.
Les résultats produits mettent en évidence les qualités des éléments de base :

- le langage mathématique GAMS (General Algebraic Modeling System) et son optimiseur de problèmes algébriques non linéaires mixtes BARON est capable de résoudre globalement et efficacement le problème de configuration et de dimensionnement posé ;
- SysML prend en compte des caractéristiques spécifiques aux systèmes notamment des éléments

d'interfaçage et de connections entre composants et sous-systèmes (ports, flux …) qui en facilitent sa description.

En revanche, comme pour s-COMMA, le choix d'une approche transformation de modèle produit les mêmes inconvénients notamment celui, majeur, de la mise au point du modèle du problème. Cette limitation avait été pointée par Shah dans son mémoire de Master (Shah, 2010) mais elle ne semble pas avoir été résolue depuis. Shah suggère que pour progresser véritablement il faudrait disposer d'un vrai langage de modélisation orienté objet pour les problèmes de conception en ingénierie qui serait le pendant dans le domaine de la programmation mathématique de ce qu'est Modelica en simulation.

Le travail que nous proposons s'inscrit complètement dans cette idée. Notre problématique est de concevoir et développer un langage objet natif de modélisation de problème de conception de systèmes. Nous pensons que ce type de langage correspond non seulement à un besoin mais surtout que, à notre connaissance, les autres manières d'envisager la résolution de problèmes de conception portent en elles-mêmes leurs limitations.

Dans les travaux précurseurs que nous avons présentés, les langages développés sont à la fois trop généraux et trop difficiles à utiliser pour des ingénieurs non informaticiens.

Les travaux récents partent du principe que la description du problème se fait en UML ou en SysML et que sa résolution sera délégué à un solveur adéquat après transformation de modèle. Rien n'est moins sûr car ces langages sont faits pour décrire des systèmes et pas modéliser des problèmes ; par ailleurs la mise au point du modèle d'un problème nécessite des phases de mise au point rapide qui ne nous semblent pas compatibles avec la lourdeur de la transformation de modèle.

4 LE LANGAGE DEPS

4.1 Genèse du langage et motivation

DEPS tire son inspiration d'expériences passées auxquelles nous avons participé : les projets Deklare et Komod et le projet RNTL CO2.

Deklare (Vargas et al, 1995) et KoMod (Sellini and Yvars, 1998) étaient axés sur la représentation des structures fonctionnelles et organiques des produits à concevoir pour permettre la pose et la résolution des problèmes de conception associés à l'aide d'une librairie de programmation par contraintes du commerce.

L'objectif du projet RNTL CO2 (Conception par contraintes) était de développer des méthodes et des outils pour résoudre des problèmes de conception. Le projet a produit Constraint Explorer (CE), un environnement logiciel comprenant un langage de modélisation et un solveur numérique à base de méthodes d'intervalles. CE a été utilisé par Dassault Aviation sur des problèmes de conception en avant-projet (Zimmer and Zablit, 2001).

Les besoins actuels exprimés en conception préliminaire notamment dans le champ de la génération d'architecture de systèmes admissibles (Albarello et al, 2012) montrent que les besoins auparavant identifiés par les auteurs du présent article en conception préliminaire de produits sont encore plus pertinents dans le domaine de la conception de système. Ce constat nous a conduits à démarrer ce travail de spécification et de développement d'un langage objet natif adapté à la conception de systèmes.

4.2 Présentation du langage

4.2.1 Cas d'étude

Nous présentons dans ce paragraphe l'exemple du dimensionnement d'un manipulateur RR (Rotoïde, Rotoïde) dans le plan (cf. figure 1). Ce cas concret nous servira de fil conducteur dans la suite de ce papier pour illustrer les différents traits du langage DEPS.

Un manipulateur RR plan (cf. Figure 1) est constitué d'une liaison rotoïde paramétrée par son angle de rotation q_1 suivi d'un bras de longueur l_1 puis d'une liaison rotoïde paramétrée par son angle de rotation q_2 suivie d'un bras de longueur l_2.

Le problème consiste à dimensionner le manipulateur RR de manière à atteindre un point P dans le plan.

Concevoir ce manipulateur implanté en $(0, 0)$ dans le plan, revient à déterminer les valeurs des variables de conception l_1 et l_2 telles qu'il existe une valeur du couple de variables de fonctionnement (q_1, q_2) permettant d'atteindre un référentiel P donné dans le plan par les coordonnées de son origine (x, y) et d'un angle q d'orientation de son axe des x par rapport à l'axe des abscisses du repère absolu $(0, \mathbf{x_0}, \mathbf{y_0})$. La notation employée est celle de Denavit-Hartenberg limitée à la dimension du plan (Khalil and Dombre, 1999).

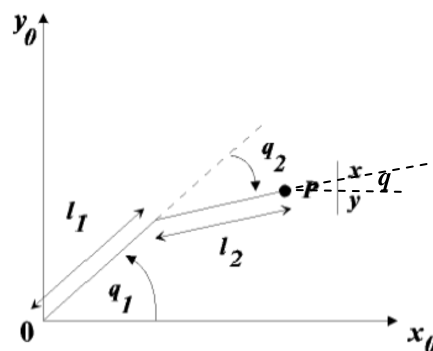


Figure 1 : Géométrie du robot plan

Le référentiel à atteindre par l'extrémité de notre robot plan sera donc représenté par la matrice 3 x 3 homogène P donnée telle que :

$$P = \begin{pmatrix} \cos q & -\sin q & x \\ \sin q & \cos q & y \\ 0 & 0 & 1 \end{pmatrix}$$

et lorsque P est atteint par le robot on a l'égalité :

$$P = Rot(q_1) \times Trans(l_1) \times Rot(q_2) \times Trans(l_2) \quad (1)$$

Avec la matrice de Rotation $Rot(q_i)$ telle que :

$$Rot(q_i) = \begin{pmatrix} \cos q_i & -\sin q_i & 0 \\ \sin q_i & \cos q_i & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

et la matrice de Translation $Trans(l_i)$ telle que :

$$Trans(l_i) = \begin{pmatrix} 1 & 0 & l_i \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Le second membre de l'équation (1) est appelée modèle géométrique direct du manipulateur RR.

Ce cas d'étude est académique. Le problème posé est simple à résoudre mais présente des difficultés de formalisation qui vont nous permettre d'illustrer l'intérêt du langage DEPS.

Dans la suite nous allons détailler des caractéristiques de ce langage en modélisant ce problème en DEPS.

4.2.2 Les modèles DEPS

Le trait fondamental du langage est le Modèle. Tout modèle (cf. figure 2) est défini à l'aide du mot clé *Model* suivi de son nom et de sa liste d'arguments (éventuellement vide). Il comporte dans l'ordre : une zone de déclaration-définition des constantes du modèle (mot clé *Constants*), une zone de déclaration des variables du modèle (mot clé *Variables*), une zone de déclaration-création des éléments du modèle (mot clé *Elements*) et une zone de définition des propriétés du modèle (mot clé *Properties*). La définition d'un modèle DEPS se termine par le mot clé *End.*

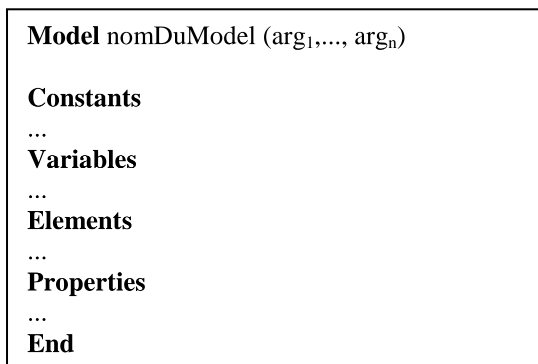


Figure 2 : Structure d'un modèle en DEPS

4.2.3 Les constantes

Une constante est une grandeur numérique dont la valeur ne varie pas durant la durée de vie de n'importe quel exemplaire du Modèle dans laquelle elle est déclarée et définie.

Une constante définie est une constante dont la valeur est donnée.
Une constante locale au modèle doit être déclarée et définie dans la zone *Constants* du modèle.
Une constante passée en argument au modèle doit être déclarée dans la zone *Constants* du modèle et sa définition est donnée par la valeur de l'argument.

Par ailleurs, des constantes universelles sont prédéfinies dans DEPS. Leur nom est réservé et préfixé par le caractère *u*. Ainsi *uPi* représente la constante dont la valeur est celle du nombre transcendant π.

4.2.4 Les variables

Une variable est une inconnue du modèle. Elle peut être vue comme une instance de Quantité (cf. §4.2.9). Une variable doit être déclarée dans la zone *Variables* du modèle. En tant qu'inconnue sa valeur n'est pas définie.

4.2.5 Les éléments

Un élément est une instance de modèle. Il peut être passé comme argument à un modèle pour représenter une agrégation et doit être alors déclaré dans la zone *Elements* du modèle. Il peut être également créé dans un modèle de manière à modéliser une composition et doit être alors déclaré et créé dans la zone *Elements* du modèle.

Lorsqu'un élément est passé en argument à un modèle, il est simplement déclaré dans la zone *Elements* du dit modèle (cf. §4.2.10 figure 6) Il est déclaré en le typant par le nom du modèle auquel il se réfère.

Pour créer un élément, on le construit en faisant appel au nom du modèle auquel il se réfère suivi de la liste de ses arguments effectifs.

4.2.6 Notation pointée

L'ensemble des éléments d'un problème est organisé à l'aide de relations d'agrégation et de composition formant une structure arborescente. L'accès aux éléments de cette structure est autorisé par l'emploi d'une notation pointée.

Une constante, une variable ou un élément à différents niveaux de cette structure arborescente peuvent être désignés et manipulés à l'aide d'un chemin d'accès.
Ainsi, *r.b1.L* désignera la longueur *L*, variable de la barre *b1* du robot *r*. Elle pourra être utilisée dans une propriété ou comme argument d'un modèle.

4.2.7 Les déclarations d'expressions

Une expression déclarée (mot-clé *Expr*) pointe sur une expression algébrique et permet de la référencer.
Ainsi une même expression algébrique peut être utilisée à plusieurs endroits d'un modèle par le biais de son nom sans être réécrite.

Une *Expr* est déclarée dans la zone *Variables* d'un modèle de la même manière qu'une variable en la préfixant par le mot-clé *Expr*.

Une *Expr* peut être initialisée (à l'aide de l'opérateur d'affectation *:=*) en zone *properties* du modèle où elle est déclarée ou bien à l'intérieur de la zone *properties* d'un autre modèle via un chemin d'accès. Initialiser une *Expr* revient à définir l'expression algébrique explicite sur laquelle elle pointera et dont elle prendra la valeur.

Ainsi dans notre problème de robot plan, la notion la plus élémentaire à représenter est la notion de référentiel. Le modèle « Referential » (cf. figure 3) représente la matrice homogène définie au §4.2.1. Dans ce modèle *x*, *y*, *theta*, *a* et *b* sont déclarés en *Expr*. De plus, *a* et *b* sont initialisées.

```
Model Referential ()
Constants

Variables
expr x : real in [-inf, +inf];
expr y : real in [-inf, +inf];
expr theta : Angle;
expr a : real in [-1, 1];
expr b : real in [-1, 1];

Elements

Properties
a := cos(theta);
b := sin(theta);
End
```

Figure 3 : Modèle DEPS d'un référentiel 2D

4.2.8 Les propriétés

Une propriété est une relation nécessairement respectée par toute instance du modèle qui la contient.

Dans l'état actuel de DEPS, les propriétés sont des relations algébriques (égalité et/ou inégalités entre expressions, définition de la valeur d'une expression nommée) portant sur des constantes et des variables du modèle ou d'un ou plusieurs éléments du modèle.
Les équations et inéquations, linéaires ou non linéaires, sont naturellement des propriétés.

Ainsi, le modèle d'une liaison (cf. figure 5) comportera un référentiel d'entrée (*refIn*), un référentiel de sortie (*refOut*) et une matrice de passage (*mat*), elle même référentiel, tels que :

$$refOut = mat * refIn$$

Cette dernière relation est explicitée dans la partie *properties* du modèle de liaison *Link*.

```
Model Link()
Constants

Variables

Elements
refIn : Referential();
refOut : Referential();
mat : Referential();

Properties
refOut.a = mat.a*refIn.a - mat.b*refIn.b;
refOut.b = mat.b*refIn.a+mat.a*refIn.b;
refOut.x := mat.a*refIn.x+mat.a*refIn.y + mat.x*1;
refOut.y := mat.b*refIn.x+mat.a*refIn.y + mat.y*1;

End
```

Figure 4 : Modèle DEPS de Liaison

4.2.9 Les Quantités

Les quantités (mot clé *Quantity*) permettent de représenter des types de grandeurs physiques, technologiques, à valeurs symboliques ou numériques. Elles sont nécessaires en Ingénierie de Systèmes et permettent de typer les constantes et les variables du problème.
Une quantité possède :
- Un type de base réel (*real*), entier (*int*), réel énuméré (*enumreal*) ou bien encore entier énuméré (*enumint*) ;
- Une borne min qui représente la valeur minimale pouvant être prise par toute constante ou variable ayant pour type la quantité définie ;
- Une borne max qui représente la valeur maximale pouvant être prise par toute constante ou variable ayant pour type la quantité définie ;
- Une dimension qui représente la dimension au sens de l'analyse dimensionnelle de la quantité (La valeur *[U]* caractérise les quantités sans dimension) ;
- Une unité de la quantité (Dans l'exemple de la figure 5, la quantité *Angle* a pour unité le radian *Rad*).

```
Quantity Angle

 Type : real ;
 Min  : -uPi ;
 Max  : uPi ;
 Dim  : [U];
 Unit : Rad ;

End
```

Figure 5 : Exemple de représentation d'une Quantité en DEPS

4.2.10 Extension de modèle

En DEPS, tout modèle peut être étendu (ou spécialisé) à l'aide du mot-clé *extends*. Ainsi, si l'on souhaite créer le modèle d'une barre de notre robot 2D (cf. figure 6), on définira un modèle étendu du modèle de liaison qui contiendra par construction tout ce que contient une liaison (*Constantes*, *Variables*, *Elements* et *Properties*) avec en plus les caractéristiques spécifiques à une barre c'est à dire dans notre cas sa longueur L, variable de conception de notre robot. De plus, notre barre étant totalement définie une fois la valeur de la variable *L* fixée, le référentiel correspondant sera donc bloqué en rotation (mat.theta := 0) et selon l'axe des y (mat.y := 0). La translation selon l'axe des x valant *L* (mat.x := *L*).

```
Model Truss() extends Link
Constants

Variables
L : real in [0, +inf];

Elements

Properties
mat.y :=0;
mat.x := L;
mat.theta := 0;

End
```

```
Model Axis() extends Link
Constants

Variables
q : Angle;

Elements

Properties
mat.y := 0;
mat.x := 0;
mat.theta := q;

End
```

Figure 6 : Modèle DEPS de Barre et d'Axe

De la même manière nous pouvons créer un modèle d'axe (cf. figure 6) correspondant à une translation nulle et à une rotation d'un angle *q*.

```
Model Connect(L1, L2)
Constants

Variables

Elements
L1, L2 : Link;

Properties
L1.refOut.x = L2.refIn.x;
L1.refOut.y = L2.refIn.y;
L1.refOut.a = L2.refIn.a;
L1.refOut.b = L2.refIn.b;

End
```

Figure 7 : Modèle DEPS de connexion binaire entre liaisons

4.2.11 Arguments de modèle

Tout modèle peut posséder lors de sa définition un nombre quelconque d'arguments. Les arguments d'un modèle sont soit des constantes (ou des chemins vers des constantes) soit des éléments d'autres modèles (ou des chemins vers des éléments). Pour les constantes, le passage d'argument s'effectue par valeur. Pour les éléments il s'effectue par référence.

```
Model A2DRobot(base, effector)
Constants

Variables

Elements
base: Point;
effector : Point;

b1 : Truss();
b2 : Truss();
j1 : Axis();
j2 : Axis();
c12: Connect(j1, b1);
c12 : Connect(b1, j2);
c23 : Connect(j2, b2);

Properties
j1.refIn.x := base.x;
j1.refIn.y := base.y;
j1.refIn.theta := base.theta;
b2.refIn.x := effector.x;
b2.refIny := effector.y;
b2.refIn.theta := effector.theta;

End
```

Figure 8 : Modèle DEPS du Robot Plan

Ainsi, un robot 2D (cf. figure 8) possède une base fixée représentée par l'argument *base* ainsi qu'une extrémité portant un référentiel représenté par l'argument *effector.*

```
Model Point(px, py, alpha) extends Referential
Constants
px : real in [-inf, +inf];
py : real in [-inf, +inf];
alpha : Angle;

Variables

Elements

Properties
x := px;
y := py;
theta := alpha;

End
```

Figure 9 : Modèle DEPS d'un référentiel fixé

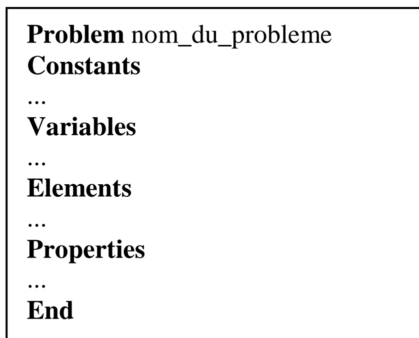
Ainsi l'origine du robot sera initialisé dans le champ *properties* à la valeur *base* et l'extrémité du robot à la valeur *effector*.

Un référentiel à atteindre étant caractérisé par ses coordonnées (*px, py*) et son orientation *alpha* (cf figure 9), les coordonnées x et y de l'origine du référentiel sont initialisées à px et py.

Pour connecter deux liaisons entre elles, nous réalisons un modèle de connexion binaire explicité figure 7.

4.2.12 Définition du problème

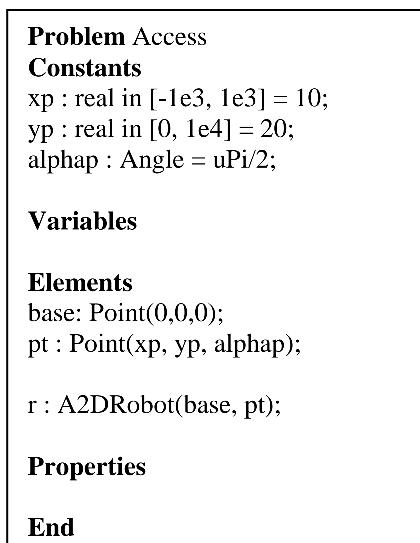
Un problème en DEPS est défini par le mot clé *Problem* et reprend la structure d'un modèle (cf. figure 10). C'est le plus haut niveau dans l'arbre des modèles obtenu par le biais du champ *Elements.*



Problem nom_du_probleme
Constants
...
Variables
...
Elements
...
Properties
...
End

Figure 10 : Structure d'un problème en DEPS

A présent notre problème (cf figure 11) revient à chercher les robots positionnés en (0,0) dont les variables de conception r.b1.L et r.b2.L sont telles qu'il existe une configuration des axes r.j1 et r.j2 (c'est à dire que les angles r.j1.q et r.j2.q existent) pour atteindre le référentiel pt situé en (10, 20, π /2).



Problem Access
Constants
xp : real in [-1e3, 1e3] = 10;
yp : real in [0, 1e4] = 20;
alphap : Angle = uPi/2;

Variables

Elements
base: Point(0,0,0);
pt : Point(xp, yp, alphap);

r : A2DRobot(base, pt);

Properties

End

Figure 11 : Modèle DEPS du Problème de positionnement du robot RR

Pour disposer d'une vision globale et synthétique du problème du robot 2D, un diagramme de classe UML de ce problème est donné en figure 12.

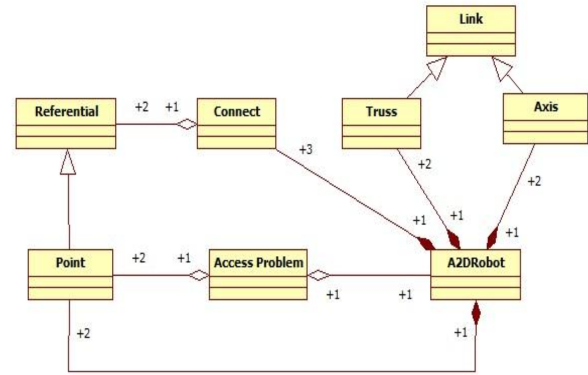


Figure 12 : Modèle de classe UML du problème du robot plan

Au final, nous obtenons une représentation concise de notre problème, constituée d'éléments de modèles réutilisables et extensibles. De plus, les modèles sont exprimés en manipulant les objets usuels de l'ingénieur : quantités, constantes physiques, équations, inéquations etc.

Du point de vue de la résolution, aspect non abordé dans cet article qui met l'accent sur la modélisation, on remarquera que le nombre d'inconnues du problème de dimensionnement du robot 2D dans le modèle DEPS final, obtenu par recensement des variables qui ne sont pas des *Expr,* est bien de quatre :
- les longueurs $l_1$et l_2 respectivement représentées par les variables $r.b_1.L$ et $r.b_2.L$,
- les angles q_1 et q_2 respectivement représentés par les variables $r.j_1.q$ et $r.j_2.q$.

Une analyse du système posé montre qu'en tenant compte des symétries de rotation dans son modèle géométrique direct, le robot RR est régi par quatre équations trigonométriques. Les bornes maxi des domaines de *r.b1.L* et *r.b2.L* n'étant pas contraintes, plusieurs solutions sont possibles et même une infinité.

De tels systèmes sont, à l'heure actuelle, aisés à résoudre à l'aide d'outils de programmation par contraintes d'intervalles. Nous avons utilisé CE pour sa facilité d'utilisation testée dans une étude précédente (Yvars et al, 2009).
Un exemple de solutions du problème du robot RR obtenues avec CE est donné Tableau 1.

q1	l1	q2	l2
1.97	18.42	-0.398	21.7
2.2546	26.3	-0.6838	25.8

Tableau 1 : Exemples de solutions du problème

L'intérêt fondamental de cette approche est qu'aucune inconnue supplémentaire ne viendra parasiter le modèle à résoudre par le solveur. De plus, la formalisation du problème ne nécessite aucune manipulation du système d'équations originel de la part de l'ingénieur. La représentation du problème est naturelle.

5 CONCLUSION ET FUTURS TRAVAUX

Nous avons présenté dans ce papier la première version d'un langage destiné à représenter des problèmes de conception de systèmes en vue de leur résolution.
Dans sa version actuelle, DEPS permet d'exprimer naturellement un problème de dimensionnement comme dans le cas d'étude présenté. Nous travaillons à enrichir le langage pour qu'il puisse exprimer d'autres types de problème de conception dans des versions ultérieures. Un compilateur du langage est en cours de développement. Il génère pour le moment un modèle intermédiaire dans le langage de modélisation de CE. Ce n'est bien entendu qu'une étape provisoire qui nous permet de disposer de capacités de résolution pour mettre au point DEPS.
La prochaine version du compilateur intègrera nativement des méthodes de résolution à base de méthodes d'intervalles sans passer par une quelconque représentation intermédiaire. Nous l'évaluerons alors sur un problème plus conséquent de dimensionnement d'une pile à combustible embarquée.

REFERENCES

Albarello, N., J.B. Welcomme and C. Reyterou, 2012. A formal design synthesis and optimization for systems architectures. *9th International Conference of Modeling, Optimization and Simulation (MOSIM'12),* Bordeaux, France.

Apt, K and M. Wallace, 2006. *Constraint Logic Programming using Eclipse.* Cambridge University Press.

Bensana, E. and T. Mulyanto, 2000. A generic approach for conceptual design based on object oriented and constraint logic programming. *EDA 2000.*

Borning, A., 1977. ThingLab - An Object-Oriented System for Building Simulations Using Constraints. *5th International Joint Conference on Artificial Intelligence (IJCAI 1977)*, Cambridge, MA, USA, vol. 1, p. 497-498.

Bouvier, P., Colmerauer, A., N'Dong, S., Touraivane et al, 1996. *Le manuel de Prolog IV*, PrologIA, Marseille.http://alain.colmerauer.free.fr/alcol/ArchivesPublications/Prolog4Manuel.pdf

Jayaraman B., and P. Tambay, 2002. Modeling Engineering Structures with Constrained Objects *PADL 2002*, LNCS 2257, pp. 28-46.

Khalil W., and E. Dombre, 1999. *Modélisation, identification et commande des robots*. 2ème édition. Hermes, France.

Mulyanto, T., 2002. *Utilisation des techniques de programmation par contraintes pour la conception d'avions.* Thèse de l'Ecole Nationale Supérieure de l'Aéronautique et de l'Espace, France.

OCL., 2014. *OCL 2.4.* http://www.omg.org/spec/OCL/2.4

OMG., 2009. OMG *System Modeling Language Tutorial.* http://www.omgsysml.org/INCOSE-OMGSysML-Tutorial-Final-090901.pdf

Parasolver., 2013. *Artisan Studio Para Solver™ 7.2 R1 Tutorials*. www.InterCax.com.

Shvetsov I., A. Semenov, and V. Telerman, 1997. Application of subdefinite models in engineering. *Artificial Intelligence in Engineering*, 11(1).

Sellini F. and P.A. Yvars, 1998. Modèles objet et représentation déclarative du produit en conception mécanique. *Revue L'Objet, Numéro spécial: les représentations par objet en conception*, 4(2).

Shah A.A., 2010. *Combining mathematical programming and SysMl for component sizing as applied to hydraulic systems*. Master Thesis, Georgia Institute of Technology.

Shah A.A., C.J.J. Paredis, R. Burkhart and D. Schaefer, 2012. Combining Mathematical Programming and SysML for Automated Component Sizing of Hydraulic Systems. *Journal of Computing and Information Science in Engineering*.

Soto, R., 2009. *Langage et transformation de modèles en programmation par contraintes*. Thèse de Doctorat de l'Université de Nantes, France.

Soto, R. and L. Granvilliers, 2007. *s-COMMA User's Manual*. http://www.inf.ucv.cl/~rsoto/s-comma/

Tambay, P. and B. Jayaraman, 2003. *The Cob Programmer's Manual*.http:// www.cse.buffalo.edu/tech-reports/2003-01.pdf

Vargas C., A. Saucier and P.A. Yvars, 1995. Ingénierie d'aide à la conception : Un environnement pour la réalisation d'un système d'aide à la conception d'organes mécaniques. *Revue Internationale de CFAO et d'Infographie*, 10(1-2), p.113-128.

Yvars P.A., P. Lafon and L. Zimmer, 2009. Optimization of Mechanical system : Contribution of Constraint Satisfaction Method. *CIE'39 International Conference on Computers and Industrial Engineering*, Troyes, France.

Zimmer, L. and P. Zablit, 2001. Global aircraft predesign based on constraint propagation and interval analysis. *CEAS Conference on Multidisciplinary Aircraft Design and Optimization*, Köln, Allemagne.