Stimulus: un langage synchrone à contraintes

Bertrand Jeannet, Étienne Closse, Fabien Gaucher, Daniel Weil

Stimulus: un langage synchrone à contraintes

B. Jeannet[1], E. Closse[1], F. Gaucher[1] et D. Weil[1]

[1]Dassault Systèmes, 18, chemin de Malacher, 38240 Grenoble

Stimulus est un langage synchrone à contraintes, dans lequel les équations flots de données définissant la valeur de signaux sont généralisées par des contraintes liant plusieurs signaux. Stimulus se révèle facile à utiliser et s'applique à la mise au point des exigences fonctionnelles de systèmes de contrôle-commande et au test de ces systèmes. Il est utilisé dans l'industrie des transports et de l'énergie pour les exigences d'applications critiques.

1 Introduction

La vogue des langages à contrainte tels Prolog et ses descendants Mercury et Oz [SHC96, RBD+03] est quelque peu retombée, peut-être parce que les applications dans lesquelles ils excellent ont été surestimées. Stimulus est un langage synchrone à contrainte plus récent qui se distingue de ces langages par les aspects suivants : (i) il s'agit d'un langage *synchrone* et non *logique*, dont les contraintes se réduisent toujours à des contraintes sur des variables de type scalaire (booléens, entiers, etc...), plus intuitives que des contraintes d'unification sur des termes *à la* Prolog ; (ii) il s'agit d'un langage spécifique à un domaine, dont les restrictions autorisent en contre-partie des analyses précises pour garantir la cohérence d'un programme et le compiler efficacement.

La motivation pour ajouter des contraintes à un langage synchrone est la modélisation de propriétés fonctionnelles et temporelles de programmes synchrones. En effet, tandis qu'un programme synchrone émet des sorties déterministes en fonction de ses entrées et de son état interne, une propriété sur de tels programme peut se ramener à des contraintes liant ses sorties à ses entrées et à un état interne propre à la propriété [HLR93].

La conception de Stimulus s'est faite en ayant à l'esprit deux écueils à éviter :

— un langage trop déclaratif est difficile à déboguer et souvent coûteux à exécuter, comme le montre l'exemple de Prolog ;

— le mécanisme de *backtrack*, plus ou moins inhérent à tout langage à contrainte, est à exploiter avec la plus grande parcimonie pour les mêmes raisons.

Deux critères ont également guidé nos choix : (i) s'appliquer au mieux à la mise au point d'exigences et au test de systèmes de contrôle-commande, et (ii) éviter à l'utilisateur toute annotation dont le but est de guider l'exécution, comme le `cut` de Prolog.

2 Stimulus en quelques mots

Stimulus est un langage synchrone flots de données dans la lignée des langages Lustre et Lucid Synchrone [CPHP87, CPP05] : les variables sont des signaux discrets, qui possèdent une valeur unique par instant logique, *cf.* Fig. 3, les valeurs des signaux sont définies par des

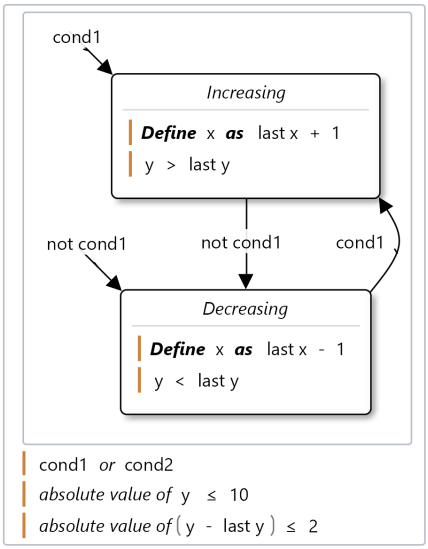

Figure 1. Automate, équations et contraintes

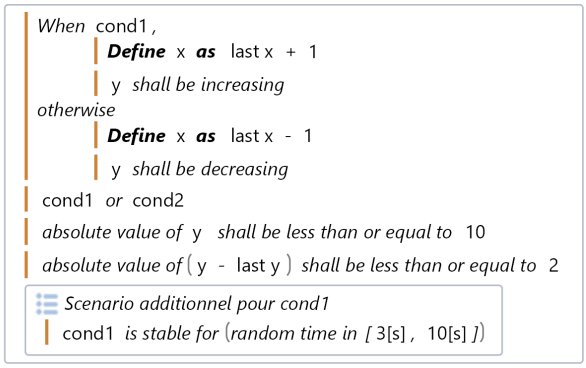

Figure 2. Programme équivalent à celui de la Fig. 1 mais utilisant les composants de la librairie standard de STIMULUS, affichés sous forme de phrases à trous, et auxquel on a rajouté un scénario pour le signal *cond*1

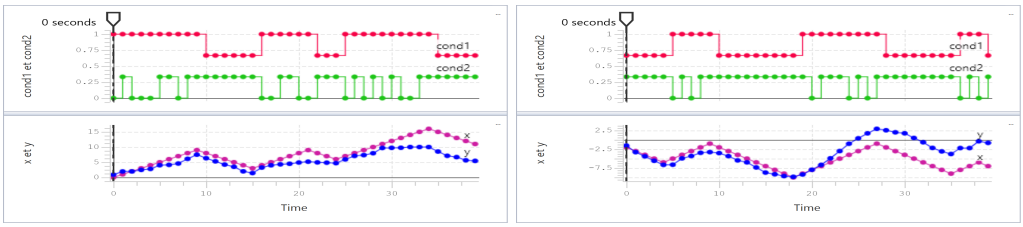

Figure 3. Deux exécutions du programme de la Fig. 2

équations avec *affectation unique*, et le contrôle est spécifié à l'aide d'automates synchrones hiérarchiques [CPP05], *cf.* les équations sur x introduites par le mot-clé `define` dans chaque état de l'automate de la Fig. 1, où `last x` désigne la valeur de x à l'instant précédent.

Le qualificatif *synchrone* signifie que du point de vue du programmeur, pour tout composant, la lecture des entrées, le calcul et l'émission des sorties se fait dans le même instant logique, et qu'il en est de même pour la propagation des entrées/sorties entre composants en parallèle ; ceci rend la composition parallèle synchrone déterministe et commutative. C'est au compilateur que revient la tâche de trouver un ordre d'exécution séquentiel qui donne l'illusion de l'instantanéité des calculs décrite ci-dessus.

Dans ce contexte STIMULUS ajoute les concepts suivants provenant de LUTIN [RRJ08] :

— le concept de *contrainte*, qui introduit du non-déterminisme sur les données, illustré par les contraintes sur *cond*1, *cond*2, et y de la Fig. 1 et les simulations de la Fig. 3 ;
— le choix non-déterministe dans les automates, qui introduit du non-déterminisme sur le contrôle et du backtrack, en cas de contraintes insatisfaisables ;
— la construction *observateur* qui transforme en reconnaisseur de signaux les instructions qu'elle contient, normalement interprétées comme des générateurs de signaux.

STIMULUS possède aussi un système de macros typées, illustrées par le **When** de la Fig. 2, qui rend les programmes plus lisibles en remplaçant les automates par des phrases à trous.

3 Compilation et exécution

La tâche du compilateur est de générer un bytecode dans lequel les instructions sont ordonnées, comme pour tout compilateur synchrone, et où les variables à résoudre ensemble

et les appels aux solveurs sont explicites. À l'exécution, l'interprète enchaine calculs explicites, recueil des contraintes actives dépendant des états des automates, et appels au solveur pour tirer des solutions. Le tirage de solutions est aléatoire et paramétré par une graine, ce qui permet d'explorer de nombreuses exécutions possibles et de reproduire l'une d'entre elles.

Un choix sémantique essentiel du langage STIMULUS est que les automates *lisent* les conditions de leurs transitions plutôt qu'ils ne les contraignent, ce qui implique que les automates sont exécutés *après* le calcul de ces conditions. Par exemple, sur la Fig. 1, les signaux *cond*1 et *cond*2 sont résolus avant y et non en même temps. Cette sémantique répond à plusieurs préoccupations mentionnées dans l'introduction :

— elle permet au compilateur de partitionner les contraintes en petits paquets résolus séquentiellement, ce qui rend l'exécution à la fois plus facile à comprendre par l'utilisateur et plus efficace ; les *modes* de MERCURY ont un effet similaire [SHC96] ;
— les transitions conditionnelles restent déterministes et n'introduisent pas de backtrack ;
— tout ceci se fait sans annotation explicite de l'utilisateur.

Cette sémantique pourrait s'avérer trop contraignante en pratique, mais ce n'est pas du tout le cas, d'après notre expérience et le retour de nos utilisateurs.

4 Application aux exigences fonctionnelles

L'application initialement visée par STIMULUS est le test automatisé de systèmes de contrôle-commande. Un banc de test de tels systèmes est constitué (i) d'un environnement qui stimule le système avec des entrées réalistes, (ii) du système sous test, et (iii) un observateur des propriétés attendues.

L'expérience montre que (1) il est très fastidieux de spécifier « à la main » les entrées réalistes d'un tel système, et que (2) lorsque l'observateur est violé, l'analyse de l'erreur mène plus souvent à la correction des propriétés attendues qu'à la correction du système sous test.

Le point (1) est la motivation initiale des langages LUTIN [RRJ08] et STIMULUS, avec l'idée de modéliser les entrées réalistes par des contraintes résolues à l'exécution.

L'observation (2) faite par [JHR13] a mené au cas d'usage de la *mise au point interactive d'exigences*, traitées par STIMULUS comme des générateurs de signaux. Cet usage répond à un besoin bien identifié par nos clients et permet de détecter :

— par inspection visuelle des simulations, les exigences manquantes ou erronées (« ce n'est pas ce que j'ai voulu dire ! ») ;
— et de manière automatique, les exigences contradictoires.

L'objectif est de détecter les erreurs du cahier des charges au moment où on les introduit, et non beaucoup plus tard lors de la confrontation avec implantation. Ce cas d'usage est détaillé dans [JG16, Gau22].

Une fois que les exigences sont mises au point, le second cas d'usage est celui du test automatisé : il consiste à placer les exigences dans le bloc *observateur* du banc de test, qui les transforme en oracle, et de tester si du code ou des exigences de plus bas niveau, constituant le système sous test, les satisfont.

Pour conclure, voici un ordre d'idée des métriques d'un modèle un peu conséquent d'un client dans l'industrie automobile : 1000 automates, exprimés au travers de composants de la librairie standard, et essentiellement composés en parallèle, 2000 variables utilisateurs, et 5000 variables au total générées par le compilateur. A l'exécution, les nombres moyens de contraintes et de variables à résoudre ensemble, par appel au solveur, sont tous deux inférieurx à… 2 ! Ceci s'explique d'une part par les choix sémantiques décrits au §3, d'autre part par la constatation que les exigences écrites par les utilisateurs se ramènent généralement à des contraintes élémentaires simples, mais qui dépendent de conditions d'activation compliquées. Enfin, le nombre de cycles synchrones exécutés par seconde se situe entre 100 et 500, ce qui permet l'obtention de simulations pertinentes en quelques secondes.

Références

[CPHP87] P. CASPI, D. PILAUD, N. HALBWACHS et J. PLAICE : Lustre : a declarative language for programming synchronous systems. *In Symp. on Principles of Programming Languages, POPL'87*. ACM, 1987.

[CPP05] J-L. COLAÇO, B. PAGANO et M. POUZET : A conservative extension of synchronous data-flow with state machines. *In EMSOFT'05*. ACM, 2005.

[Gau22] F. GAUCHER : Extending Model-Based Systems Engineering with Requirements-In-the-Loop Simulation : The Landing Gears Case-Study. `https://cesam.community/wp-content/uploads/2023/01/14h30-15_1430_CSDM2022_Gaucher.pdf`, 2022.

[HLR93] N. HALBWACHS, F. LAGNIER et P. RAYMOND : Synchronous observers and the verification of reactive systems. *In Algebraic Methodology and Software Technology, AMAST'93*. Springer, 1993.

[JG16] B. JEANNET et F. GAUCHER : Debugging Embedded Systems Requirements with STIMULUS : an Automotive Case-Study. *In European Congress on Embedded Real Time Software and Systems, ERTS'16*, 2016.

[JHR13] E. JAHIER, N. HALBWACHS et P. RAYMOND : Engineering functional requirements of reactive systems using synchronous languages. *In Int. Symp. on Industrial Embedded Systems, SIES'13*. IEEE, 2013.

[RBD+03] P. Van ROY, P. BRAND, D. DUCHIER, S. HARIDI, M. HENZ et C. SCHULTE : Logic programming in the context of multiparadigm programming : the oz experience. *Theory and Practice of Logic Programming*, 3(6), 2003.

[RRJ08] P. RAYMOND, Y. ROUX et E. JAHIER : Lutin : A language for specifying and executing reactive scenarios. *EURASIP J. of Embedded Systems*, 2008.

[SHC96] Z. SOMOGYI, F. HENDERSON et T. C. CONWAY : The execution algorithm of mercury, an efficient purely declarative logic programming language. *J. of Logic Programming*, 29(1-3), 1996.