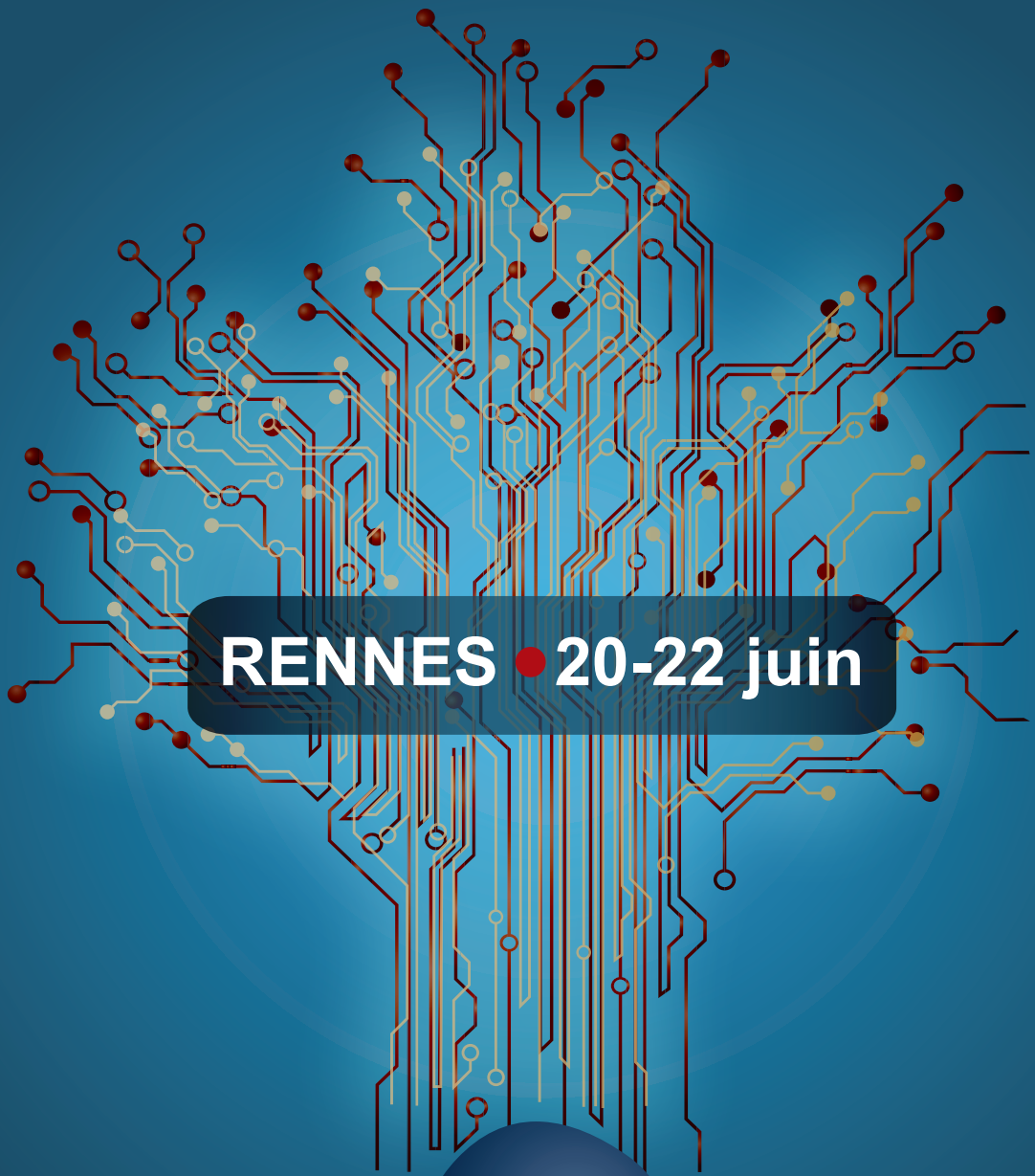
Actes des Quatrièmes journées nationales du

GDR·GPL 2012

Groupement De Recherche CNRS du Génie de la Programmation et du Logiciel



Actes des Quatrièmes journées nationales du

Groupement De Recherche CNRS du Génie de la Programmation et du Logiciel

Université de Rennes I
20 au 22 juin 2012

Editeurs : Laurence Duchien
Olivier Barais

Impression : service de reprographie, Université Rennes I

Table des matières

Préface

C'est avec grand plaisir que je vous accueille aux Quatrièmes Journées Nationales du GDR GPL. Ces journées marquent la première année du second quadriennal du GDR Génie de la Programmation et du Logiciel (GPL), créé en 2008 pour une durée de 4 ans par le CNRS (GDR 3168) et renouvelé en ce début d'année 2012. Les missions principales du GDR GPL sont l'animation scientifique de la communauté et la promotion de nos disciplines, notamment en direction des jeunes chercheurs. Cette animation scientifique est d'abord le fruit des efforts de nos groupes de travail, actions transverses et de l'Ecole des Jeunes Chercheurs en Programmation.

Le GDR GPL est maintenant dans sa cinquième année d'activité et ces journées seront l'occasion de lancer officiellement les actions du second quadriennal. Ces dernières années, les journées nationales se sont affirmées comme un temps fort de l'activité de notre GDR, l'occasion pour toute la communauté de se retrouver et se rencontrer. Plusieurs événements scientifiques sont co-localisés avec ces journées nationales : l'École des Jeunes Chercheurs en Programmation qui passera un temps à Rennes pendant les journées du GDR, la première édition de la conférence CIEL 2012, fusion des journées IDM et de la conférence LMO, ainsi que les journées Compilation organisées par le nouveau groupe Compilation dont c'est la première année de fonctionnement.

Ces journées sont une vitrine où chaque groupe de travail ou action transverse donne un aperçu de ses recherches. Une trentaine de présentations ont ainsi été sélectionnées par les responsables des groupes de travail. Comme les années précédentes, nous avons demandé aux groupes de travail de nous proposer, en règle générale, des présentations qui avaient déjà fait l'objet d'une sélection dans une conférence nationale ou internationale ; ceci nous garantit la qualité du programme. Nous ferons également un retour sur les actions spécifiques 2011 avec une présentation de leurs résultats.

Trois conférenciers invités nous ont fait l'honneur d'accepter notre invitation. Il s'agit de Jean-Bernard Stefani (Inria Grenoble-Rhône-Alpes) dont la présentation sera commune à la conférence CIEL, de Robert France (Colorado State University) et de Claude Jard (ENS de Cachan, Antenne de Bretagne). Une table ronde, animée par Christel Seguin, sera dédiée aux industriels et à leur positionnement vis-à-vis du logiciel. Nous aurons aussi le plaisir d'accueillir Michel Bidoit, représentant la direction de l'INS2I, qui fera le point sur les actions de notre institut.

Comme les années précédentes, ces journées ont aussi pour objectif de préparer l'avenir en favorisant l'intégration des jeunes chercheurs dans la communauté et leur future mobilité. Dans cet esprit, nous les avons encouragés à proposer un poster ou une démonstration de leurs travaux, en leur offrant les frais d'inscription. Trente ont répondu à cet appel. Nous avons également couplé une partie des journées avec l'école Jeunes Chercheurs en Programmation en vue d'un échange plus nourri entre les générations.

Avant de clôturer cette préface, je tiens à remercier tous ceux qui ont contribué à l'organisation de ces journées nationales : les responsables de groupes de travail ou d'actions transverses, les membres du comité de direction du GDR GPL et, tout particulièrement, le comité d'organisation de ces journées nationales présidé par Olivier Barais. Je remercie chaleureusement l'ensemble des collègues rennais qui n'ont pas ménagé leurs efforts pour nous accueillir dans les meilleures conditions.

Laurence Duchien
Directrice du GDR Génie de la Programmation et du Logiciel

Comités

Comité de programme des journées nationales

Le comité de programme des journées nationales 2012 est composé par les membres du comité de direction du GDR GPL et les responsables de groupes de travail.

Laurence Duchien (présidente), LIFL, Université Lille 1

Yamine Ait Ameur,LISI / ENSMA
Mireille Blay-Fornarino, I3S, Université Nice-Sophia-Antipolis
Yohan Boichut, LIFO, Université d'Orléans
Isabelle Borne, IRISA, Université de Bretagne Sud
Frédérice Dadeau, FEMTO-ST, Université de Franche-Comté
Catherine Dubois, CEDRIC, ENSIIE
Lydie Du Bousquet, LIG, Université Joseph Fourier
Frédéric Gava, LACL, Université Paris-Est
Jean-Louis Giavitto (Président du jury des posters), IRCAM, CNRS
Laure Gonnord, LIFL, Université Lille 1
Gatan Hains, LACL, Université Paris-Est
Pierre-Cyrille Heam, FEMTO-ST, Université Franche-Comté
Akram Idani, LIG, ENSIMAG
Claude Jard, IRISA, ENS-Cachan en Bretagne
Thomas Jensen, IRISA, CNRS
Yves Ledru, LIG, Université Joseph Fourier
Pierre-Etienne Moreau, LORIA, INRIA
Pascal Poizat, LRI, Université Evry-Val d'Essone
Marc Pouzet, Ecole Normale Supérieure, Université Pierre et Marie Curie, IUF
Fabrice Rastello, INRIA, ENS Lyon
Romain Rouvoy, LIFL, Université Lille 1
Olivier H. Roux, IRCCyN, Université de Nantes
Salah Sadou, VALORIA, Université Bretagne-Sud
Christel Seguin, ONERA Centre de Toulouse
Chouki Tibermacine, LIRMM, Université Montpellier II
Sarah Tucci, CEA LIST

Comité scientifique du GDR GPL

Franck Barbier(LIUPPA, Pau)
Charles Consel (LABRI, Bordeaux)
Roberto Di Cosmo (PPS, Paris VII)
Christophe Dony (LIRMM, Montpellier)
Stéphane Ducasse (INRIA, Lille)
Jacky Estublier (LIG, Grenoble)
Nicolas Halbwachs (Verimag, Grenoble)
Marie-Claude Gaudel (LRI, Orsay)
Gatan Hains (LACL, Créteil)
Valérie Issarny (INRIA, Rocquencourt)
Jean-Marc Jézéquel (IRISA, Rennes)
Dominique Méry (LORIA, Nancy)
Christine Paulin (LRI, Orsay)

Comité d'organisation

Olivier Barais (président), Université Rennes 1
Sandrine Blazy, Université de Rennes 1
Arnaud Blouin, INSA Rennes
Johann Bourcier, Université de Rennes 1
Benoit Combemale, Université de Rennes 1
Clément Guy, Université de Rennes 1
Aymeric Hervieu, Kereval
Thomas Jensen, INRIA
Jean-Marc Jézéquel, Université de Rennes 1
Alan Schmitt, INRIA
Elisabeth Lebret, INRIA
Loic Lesage, INRIA
Lydie Mabil, INRIA
Jonathan Marchand, ENS Cachan
Gerson Sunye, Université de Nantes

Conférenciers invités

The case for reversible languages and systems*

Jean-Bernard Stefani

INRIA Grenoble-Rhône-Alpes, France
jean-bernard.stefani@inria.fr

Outline

The notion of reversible computation has already a long history. It has its origin in physics with the observation by Landauer that only irreversible computations need to consume energy. It has since attracted interest in diverse fields, including e.g. hardware design, computational biology, program debugging, and quantum computing. Of particular interest is its application to the study of programming abstractions for reliable systems. Indeed, most fault-tolerant schemes exploiting system recovery techniques, including exception handling, checkpoint/rollback and transaction management schemes, rely on some form of *undo* or another. This suggests exploiting reversible computing ideas to support the more systematic design and construction of recoverable systems.

In this talk we report on the main ideas of an ongoing thread of research, carried out in the context of the ANR REVER project [1], that aims at designing a reversible programming model and at revisiting programming abstractions for the construction of dependable distributed systems. Specifically, we discuss:

- Danos and Krivine's causal consistency criterion for undoing concurrent executions and its support in a small paradigmatic concurrent language, the higher-order π-calculus.
- Different ways to control reversibility, and the subtleties involved in devising an imperative form of reversibility control.
- The inherent costs in implementing a fully reversible concurrent model of computation.
- The introduction of compensations in a reversible setting and the understanding of transactions as *ballistic* processes.

We conclude with a discussion of the many avenues open for further research on the subject, including the combination of reversibility with dynamic modularity features in programmaing languages.

References

1. http://www.pps.jussieu.fr/~jkrivine/REVER/ANR_REVER/Welcome.html.

* This work has been partially supported by the French National Research Agency (ANR), project REVER n. ANR 11 INSE 007.

The Fall and Rise of Software Modeling Practices

Auteur : Robert France (Colorado State University)

Résumé :
It may seem to some that the bright light of Model-Driven Software Development (MDD) has dimmed significantly. This perception cannot be completely dismissed. MDD, as with many evolving major software development approaches, has seen its share of overly enthusiastic advocates, overzealous detractors, unrealistic expectations, and inadequate (to put it kindly) practices and technologies marketed as "essential" tools-of-the-MDD-trade. In this talk I will describe some of the real problems that led to a loss in confidence in the ability of MDD practices to deliver. I will also discuss why good software modeling practices are needed to meet the challenges of modern software development, and discuss some of the more promising new research directions that will help define the next generation of software modeling practices and technologies.

QoS-Aware Management of Monotonic Service Orchestrations

Claude Jard

ENS Cachan, Antenne de Bretagne, France
Claude.Jard@bretagne.ens-cachan.fr

Abstract. We study QoS-aware management of service orchestrations, specifically for orchestrations having a data-dependent workflow. Our study supports multi-dimensional QoS. To capture uncertainty in performance and QoS, we provide support for probabilistic QoS. Under the above assumptions, orchestrations may be non-monotonic with respect to QoS, meaning that strictly improving the QoS of a service may strictly decrease the end-to-end QoS of the orchestration, an embarrassing feature for QoSaware management. We study monotonicity and provide sufficient conditions for it. We then propose a comprehensive theory and methodology for monotonic orchestrations. Generic QoS composition rules are developed via a QoS Calculus, also capturing best service binding—service discovery, however, is not within the scope of this work. Monotonicity provides the rationale for a contractbased approach to QoS-aware management. Although function and QoS cannot be separated in the design of complex orchestrations, we show that our framework supports separation of concerns by allowing the development of function and QoS separately and then "weaving" them together to derive the QoS-enhanced orchestration. Our approach is implemented on top of the Orc script language for specifying service orchestrations.
Keywords: Services, Semantic Web, Quality of Service, Petri Nets, Formal languages.

1 An Abstract Algebraic Framework for QoS composition

As QoS composition is the primary building block of QoS-aware management, it is of interest to develop abstract algebraic composition rules. We propose such an abstract algebraic framework encompassing key properties of QoS domains and capturing how the QoS of the orchestration follows from combining QoS contributions by each requested service. This algebraic framework relies on an abstract dioid (D, max, $\leq$), where D is the (possibly multi-dimensional) QoS domain. The abstract addition of the dioid identifies with the "max" operation associated with the partial order of the QoS domain; it captures both the preference among services in competition and the cost of synchronizing the return of several services requested in parallel. The increment in QoS caused by the different service calls is captured by the abstract multiplication of the dioid.

2 A Careful Handling of Monotonicity

We then study monotonicity in this generic QoS context, by proposing conditions ensuring it for both nonprobabilistic and probabilistic QoS frameworks. Guidelines for how to enforce monotonicity are derived and ways are proposed to circumvent a lack of monotonicity. The mathematical justification of the extension required to deal with probabilistic QoS domains that are only partially, not totally ordered is non-trivial. We see our in-depth treatment of monotonicity as a major contribution, as this issue has been infrequently recognized and was not properly handled.

3 Support for Separation of Concerns

QoS-aware management of composite services requires developing a QoS-aware model of a service orchestration, which can be cumbersome. At a minimum, it is a significant increase in difficulty compared to the specification of only the function of the orchestration. It is thus desirable to offer means to develop function and QoS in most possible orthogonal ways. By taking advantage of our abstract algebraic QoS framework, we are able to address this need. More precisely, we have developed an implementation of our approach in which QoS-aware orchestration models are automatically generated, from a specification of the function only, augmented with the declaration of the QoS domains and their algebra. This model can be executed to analyze the orchestration and perform QoS contract composition. We have implemented this technique on top of the Orc language for orchestrations.

4 A Methodology: Managing QoS by Contracts

By building on top of monotonicity, we advocate the use of contract-based QoS-aware management of composite services, in which the considered orchestration establishes QoS contracts with both its users and its requested services. Contract-based design amounts to performing QoS contract composition, which is the activity of estimating the tightest end-to-end QoS contract an orchestration can offer to its customer, from knowing the contract with each requested service. Late service selection or binding is performed on the basis of run-time QoS observations, by simply selecting, among different candidates, the one offering best QoS. Monotonicity ensures that this greedy policy will not lead to a loss in overall QoS performance of the orchestration. Best service binding is a built-in mechanism in our model. To ensure satisfaction of the QoS contract with its users, it is enough to monitor the conformance of each requested service with respect to its contract, since a requested service improving its QoS can only improve the overall QoS of the orchestration. To account for uncertainty in QoS, QoS is considered probabilistic. Such contracts consist of the specification of a probability distribution for the QoS dimensions. Performing this requires formalizing what it means, for a service, to perform better than its contract. We rely

for this on the notion of stochastic ordering for random variables, a concept that is widely used in econometrics. All our results regarding monotonicity extend to the case in which ordering of QoS values is replaced by stochastic ordering.

Retour sur les actions spécifiques 2011

Action Spécifique du GDR GPL
2011
Expert-User Modelling

Participants (animateurs) :

- I3S -, Philippe Renevier-Gonin
- IRISA - Benoît Combemale
- LIFL - Xavier Le Pallec
- LIG - Sophie Dupuy-Chessa, Agnès Front, Dominique Rieu, Nadine Mandran
- LIP6 - Marie-Pierre Gervais, Laurent Wouters
- LIUPPA - Thierry Nodenot
- FUNDP Namur : Nicolas Genon
- CEA List : Hubert Dubois, Morayo Adedjouma, Fadoi Lakhal
- LAMIH : Kathia Oliveira

Activités

Pour la conception ou l'évolution d'un système informatique, il est établi depuis une dizaine d'années qu'il est crucial d'impliquer les utilisateurs finaux dans toutes les phases du développement (analyse, conception, implémentation, test, déploiement) (Standish Group). Ce constat est à l'origine du renouvellement récent de l'intérêt pour le end-user programming (Rosson). Le GDR-GPL se propose, en particulier, d'étudier cette préoccupation sous l'angle de l'Ingénierie Dirigée par les Modèles (AS-MDA 2003/5) au travers de l'action spécifique Expert-User Modelling. Cette AS se focalise sur les experts métiers qui, pour un domaine donné, partage un vocabulaire spécifique qui correspond à un ensemble de concepts sur lequel les concepteurs informaticiens peuvent s'appuyer pour définir un support de description. La différence principale avec le end-user programming classique porte sur la manipulation de modèles par l'utilisateur final (expert ou non) pour la programmation : en IDM, l'application est décrite ou évolue à partir de plusieurs types de modèles qui cohabitent. L'utilisateur final/expert doit donc être conscient de ce travail collaboratif constant et les mécanismes informatiques d'édition de modèles doivent être construits de manière à respecter cette collaboration (validation, cohérence, politique d'accès…). Il ne construit pas seul son application à partir d'un ensemble de briques et d'un langage (de programmation ou de modélisation) simplifié.

Les participants à l'AS sont pour la plupart confrontés à ce problème dans différents projets de recherche : ils collaborent avec les experts métiers qui interviennent dans les domaines du tourisme, de la cartographie, de l'avionique, du management et de la botanique.

L'AS a permis d'établir le constat suivant : les travaux en expert-user modelling nécessitent des phases d'évaluation auprès d'experts métiers alors que nous maîtrisons mal les techniques d'expérimentations. Les réflexions au sein de l'AS se sont alors focalisées sur la manière d'évaluer les modèles et les langages de modélisation du point de vue des experts métier.

Nous avons aussi pu constater la nécessité de la pluridisciplinarité dans ce type de travaux. Par exemple, différents aspects des syntaxes concrètes graphiques sont au coeur d'autres disciplines ou y sont traités abondamment : la représentation visuelle de données/connaissances (géographie), les mécanismes cognitifs humains (psychologie), le contexte professionnel dans lequel se situe l'activité de modélisation (ergonomie cognitive), l'organisation dans laquelle se situe le ou les experts (sociologie). Nos propositions ainsi que leurs évaluations doivent tenir compte de cette pluridisciplinarité.

Les évaluations des modèles et des langages de modélisation sont d'ailleurs un verrou pour l'expert-user modelling, Il existe de nombreux travaux (Rossi et Brinkkemper 1996) (Siau et Cao cité dans (Krogstie 2003)) (Mohagheghi et Aagedal 2007) en Génie Logiciel sur la qualité de la syntaxe abstraite d'un méta-modèle. Il en existe par contre très peu en ce qui concerne la/es syntaxe(s) concrète(s) graphique(s) associée(s). Les caractéristiques à respecter ou critères de qualités d'une syntaxe concrète sont encore trop peu définis. Ce constat peut aussi être fait pour les protocoles d'expérimentation associés : ils sont pour l'instant à l'état embryonnaire (Aranda et al. 2007, Patig 2008).. Aussi les consignes que nous développons dans la section suivante sont un premier pas important pour aider la communauté dans la maîtrise des pratiques expérimentales.

Consignes pour les expérimentations

Dans une science artificielle comme l'informatique, l'objectif est la construction d'objets, de formalismes, d'applications et/ou de logiciels. Les méthodes expérimentales pour construire et évaluer ces objets vont être différentes selon les types d'objets à évaluer et les moments de la construction. Dans le cas de tests de performance d'algorithmes, les méthodes utilisées sont celles préconisées par la biologie. Un plan expérimental est construit. Il permet d'étudier l'influence de différents facteurs sur la performance ou la rapidité d'un algorithme ou d'un programme. Dans le cas de l'évaluation des interfaces Homme-Machine c'est également cette méthode qui est utilisée avec un questionnaire d'utilisabilité rempli en fin d'expérience.

Dans d'autres cas les objets ou applications construits s'inscrivent dans un processus de conception et de développement. L'objet ou l'application à construire doit s'appuyer sur des pratiques existantes et sur les besoins et attentes des sujets. C'est la phase d'exploration et de compréhension. A partir de ces informations une première version d'objets ou d'applications peut être proposée. Les sujets peuvent alors émettre des avis ou des opinions sur cette proposition. Il s'agit de la phase de co-conception où les utilisateurs sont mis face à l'application pour la critiquer et faire des propositions pour la rendre plus proche de leurs usages. Cette phase de co-conception nécessite plusieurs allers-retours entre les concepteurs et les utilisateurs. Il est parfois nécessaire de refaire une phase d'exploration pour comprendre et pour raffiner un point non vu dans la première phase exploratoire. Enfin, quand l'objet ou l'application est dans sa version la plus aboutie, il convient de l'évaluer : d'un point de vue des performances techniques et d'un point de vue de l'utilisateur.

La **démarche expérimentale** doit donc accompagner ces trois phases de construction de la recherche : exploration, co-conception et évaluation.

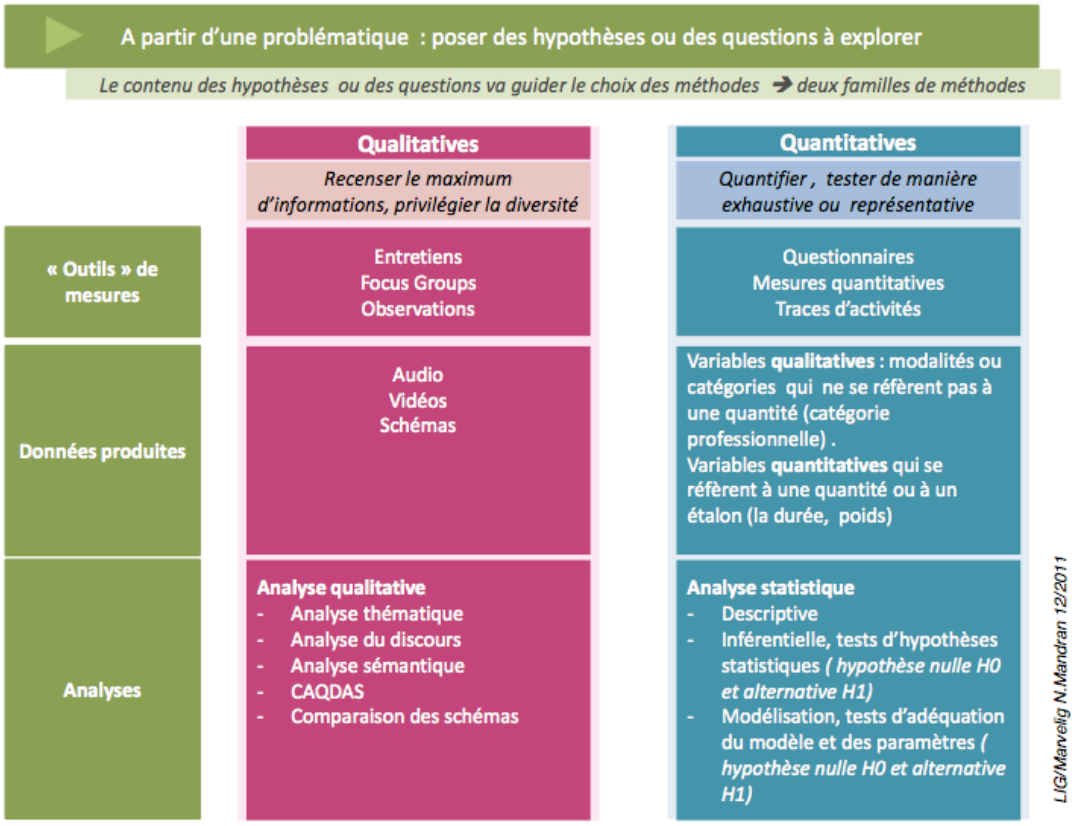
Pour l'**exploration**, les méthodes les plus appropriées sont les méthodes dites qualitatives. Elles permettent de rencontrer l'utilisateur pour comprendre et observer ses pratiques et pour recenser ses besoins et attentes. Pour la phase de **co-conception**, il s'agit de faire évoluer le produit de manière à le rendre le plus utilisable possible pour l'utilisateur en mettant par exemple, un ou des utilisateurs et le concepteur autour du produit pour en arriver à une version/spécification la plus consensuelle possible. On utilise ici des méthodes expérimentales qualitatives issues de la sociologie ou de l'anthropologie. Ce sont essentiellement des méthodes impliquant des séances en petits groupes (comme pour la méthode des focus groups). Les données produites sont des données audio ou vidéo ou bien des schémas.

Dans ces deux phases (exploration et co-conception), les méthodes qualitatives vont avoir un intéret majeur qui est d'extraire un maximum de pratiques et/ou de vocabulaire auxquelles un concepteur/chercheur ne peut penser. Ces méthodes qualitatives ne permettent pas de faire des statistiques et donc d'apporter une quelconque forme de preuve. Mais ce n'est pas l'objectif des méthodes qualitatives. Toutefois, pour pallier ce déficit et répondre aux reproches des sciences expérimentales, on peut leur associer une expérimentation plus quantitative : les utilisateurs pourraient par exemple noter le précédent

produit (ou certains de ses aspects) via un questionnaire. Il est important dans ce cas de disposer d'un questionnaire qui ne souffre pas d'un quelconque parasitage. Dans le cas contraire, l'expérimentation engendrerait une étude dont la portée serait plus réduite que celle prévue au départ (et donc un résultat biaisée).

Quant à la phase d'**évaluation**, elle peut se faire selon deux méthodes : une méthode qualitative où l'on observe les personnes en train d'utiliser l'application et où l'on recueille leur opinions à l'issue de l'utilisation (dans le cas où les experts sont peu nombreux); une méthode quantitative où l'activité des utilisateurs/participants est tracée (sur l'application) ou chronométrée/filmée (sur papier).

Pour illustrer ces trois étapes et la dualité qualitatif/quantitatif, nous pouvons nous placer dans le cadre de la construction d'un langage de conception informatique pour des jardiniers (projet ANR MOANO). L'exploration consiste à étudier le matériel existant (documents, logiciels) et à suivre les jardiniers durant leurs activités. Cette étape permet de recueillir les différents termes qu'ils utilisent dans leur métier de manière à construire un langage de conception qui soit approprié. Nous pouvons ensuite passer à la co-conception et effectuer des discussions de groupes (tels des focus-groups) en leur présentant un langage de conception construit à partir de la précédente étape. De ces discussions pourra émerger un consensus autour du langage de conception. L'étape de validation pourra donc être menée. Elle peut prendre plusieurs formes selon les objectifs du langage : on demande à une population de jardiniers la plus représentative possible de spécifier plusieurs exemples de situations assez différentes (afin d'utiliser tous les mécanismes du langage) et on évalue au choix la rapidité d'écriture, la concision des productions, leur cohérence...



Le schéma ci-dessus liste des outils de mesure, des types de données produites et des types d'analyses qui peuvent être utilisés ou produites dans le cas de démarches qualitatives ou quantitatives.

Conclusion

L'AS expert-user modelling a permis de structurer une communauté autour de cette problématique. Les échanges ont permis de faire émerger le problème commun de l'évaluation. Elle aussi permis de dresser des verrous associés à des axes de travaux futurs :

- **Efficacité cognitive > quels critères.** Les travaux récents de Daniel L. Moody semblent être une base pertinente pour définir des critères de qualité pour la syntaxe concrète graphique d'un langage de modélisation. L'identification de ces critères et leur intégration dans des outils de modélisation destinés aux experts métier nous semblent être un verrou à lever afin d'améliorer l'efficacité cognitive des modèles.
- **Evaluation > les protocoles d'expérimentation.** Les différentes évaluations qu'ont menées les participants de l'AS ont utilisé quelques uns des rouages classiques des protocoles d'expérimentation. Nadine Mandran, ingénieure ergonome au LIG et qui a mené plusieurs expériences relatives aux préoccupations de l'AS a décrit la démarche à suivre pour ce type d'expérimentation. C'est cette démarche qui est plus haut.
- **Pour lever le verrou de l'évaluation,** il est important de renforcer les échanges entre chercheurs et de cristalliser les avancées. Aussi les participants de l'AS souhaitent mettre en commun leurs pratiques en matière d'évaluation. Ils envisagent de les partager en créant une communauté pour l'évaluation au sein d'un outil de capitalisation collaborative de connaissances développé par l'équipe SIGMA du LIG, COpen**.** Au vu de l'intérêt de la communauté informatique pour ce genre de problématique (**AS expert-user m**odelling, AS Evaluation des Systèmes d'Information et point de vue des utilisateurs de l'association Inforsid), la communauté pour l'évaluation pourrait rapidement être étendue.

Bibliographie

Aranda Jorge, Ernst Neil, Horkoff Jennifer, et Easterbrook Steve. A framework for empirical evaluation of model comprehensibility. Dans Proceedings of the International Workshop on Modeling in Software Engineering, MISE '07, pages 7–, Washington, DC, USA, 2007. IEEE Computer Society.

Krogstie John. Evaluating uml using a generic quality framework. Dans UML and the unified process, pages 1–22. IGI Publishing, Hershey, PA, USA, 2003.

Lieberman Henry, Paternò Fabio, Klann Markus and Wulf Volker, End-User Development: An Emerging Paradigm,Human–Computer Interaction Series, 2006, Volume 9, 1-8

Mohagheghi Parastoo et Aagedal Jan. Evaluating quality in model-driven engineering. Dans Proceedings of the International Workshop on Modeling in Software Engineering, MISE '07, pages 6–, Washington, DC, USA, 2007. IEEE Computer Society.

Patig Susanne. A practical guide to testing the understandability of notations. Dans Proceedings of the fifth Asia-Pacific conference on Conceptual Modelling - Volume 79, APCCM '08, pages 49–58, Darlinghurst, Australia, Australia, 2008. Australian Computer Society, Inc.

Rossi Matti et Brinkkemper Sjaak. Complexity metrics for systems development methods and techniques. Information Systems, 21(2) :209–227, 1996.

Standish Group, reports, https://secure.standishgroup.com/reports/reports.php#reports, dernière visite 01/04/2012

Action Spécifique “Modélisation et analyse de systèmes embarqués”

J.-M. Bruel, F. Kordon, J. Hugues, A. Canals et A. Dohet

23 avril 2012

1 Contexte

Les systèmes embarqués sont actuellement un intérêt majeur avec le développement de nouvelles classes d'applications dites “ ubiquitaires ”. Ainsi, d'un “ marché de niche ” avec des contraintes fortes de fiabilités impliquant des techniques de développement particulières (architectures statiques, OS dédiés, techniques de programmation spécifiques, langages de description souvent spécifiques aussi), on doit intégrer de nouvelles contraintes à celles déjà existantes : répartition, mobilité. Dans ce contexte, les techniques de modélisations jouent un rôle important et doivent évoluer pour suivre cette adaptation. Il est intéressant de disposer d'un état de l'art sur les principales approches de conceptions s'appuyant sur des langages de modélisation reconnus. Citons à titre d'exemple : AADL, SysML, MARTE.

2 Présentation de l'action spécifique

L'objectif de cette action spécifique est de soutenir la publication d'une monographie dont le but est de traiter des aspects modélisation. L'idée est de partir d'une étude de cas commune et de présenter différents (deux ou trois) langages de modélisation susceptibles de traiter ce même cas. Un tel ouvrage vise typiquement un lectorat de type ingénieur ou étudiant de Master. L'objectif est de donner des indications à la fois “ théoriques ” sur quelques langages émergents mais aussi une vision “ pratique ” (i.e. comment faire) pour des personnes souhaitant rapidement prendre en main ce type de démarche. Le contenu prévu de l'ouvrage sera de trois chapitres “ chapeaux ” dont l'un présente une l'étude de cas qui sera traitée dans les trois parties. Ensuite, chaque partie (par langage de modélisation choisi) se découpe de la manière suivante :

- un chapitre de présentation synthétique du langage,
- un chapitre pour modéliser le cas d'étude avec le langage,
- un chapitre expliquant l'exploitation possible de ce modèle avec des exemples concrets

Une idée, pour faciliter la diffusion du livre et aussi promouvoir le cas d'étude, sera d'avoir un site compagnon qui contiendra les modèles du cas d'étude (dans les différents langages proposés). Le plan de l'ouvrage est détaillé ci-après. Les détails sur les auteurs sont indiqués plus loin. Chaque partie à un responsable chargé de la cohérence de l'ensemble.

L'échéancier pour cet ouvrage est le suivant :

- mars 2011 : début du projet (l'infrastructure de travail ? subversion, environnement latex et étude de cas en chapitre 3 ? a été mise en place).
- mai 2011 : réunion de travail commune pour faire le point sur l'étude de cas (financée hors GdR).
- juin 2011 : remise d'une version révisée du chapitre 3 (étude de cas).

– juillet 2011 : remise des chapitres de présentation des notation et de modélisation de l'étude de cas (4, 5, 8, 9, 12, 13) et remise des chapitres introductifs (1 et 2).
– septembre 2011 : remise des chapitres d'analyse et de génération de code (6, 7, 10 ,11, 14, 15).
– septembre 2011 : remise des commentaires sur les chapitres 1, 2, 4, 5, 8, 9, 12, 13.
– novembre 2011 : remise des commentaires sur les chapitres 6, 7, 10 ,11, 14, 15. Remise de la version révisée des chapitres 4, 5, 8, 9, 12, 13. Remise du chapitre 16.s
– décembre 2011 : réunion de travail commune pour harmoniser les différents chapitres (financée par le GdR).
– juin 2012 : remise de l'ouvrage à Hermes (finalisation des chapitres "hors parties", indexation, etc.)

3 Contenu

Chapitre 0 Préface (Auteurs : D. Potier). Préface par le directeur Recherche & technologie du pôle System@tic.

Chapitre 1 Introduction Générale (Auteurs : F. Kordon, J. Hugues, A. Canals & A. Dohet). Introduction générale du livre permettant de déterminer les objectifs et d'en brosser le contenu.

Chapitre 2 Concepts Ge ?ne ?raux (Auteurs : F. Kordon, J. Hugues, A. Canals & A. Dohet). Précision des éléments de modélisation ou d'expression des propriétés qui sont communs.

Chapitre 3 Étude de cas : le PaceMaker (Auteurs : F. Kordon, J. Hugues, A. Canals & A. Dohet). La description de l ?e ?tude de cas retenue : le PaceMaker. Elle provient d'un regroupement d'Universités américaines et est donc "neutre" par rapport aux notations que nous souhaitons traiter dans cette monographie.

Partie I SysML

Chapitre 4 Présentation des concepts de SysML (Auteurs : J-M. Bruel & P. Roques). Présentation détaillé des principes et de la notation SysML.

Chapitre 5 Modélisation de l'étude de cas en SysML (Auteurs : L. Fejoz, Ph. Leblanc & A. Canals). Réalisation un modèle de l'étude de cas PaceMaker

Chapitre 6 Analyse des exigences (Auteurs : L. Apvrille & P. de Saqui-Sannes). Utiliser les outils et techniques disponibles dans l'environnement SysML pour vérifier concrètement les exigences en s'appuyant sur le modèle de l'étude de cas.

Chapitre 7 Analyse de sûreté (Auteurs : F. Boniol, Ph. Dhaussy & J-C. Roger). Utiliser les outils et techniques disponibles dans un environnement SysML pour vérifier concrètement des propriétés de sûreté en s'appuyant sur le modèle de l'étude de cas.

Partie II MARTE

Chapitre 8 Présentation des concepts de MARTE (Auteurs : S. Gérard & F. Terrier). Présentation détaillé des principes et de la notation MARTE.

Chapitre 9 Modélisation de l'étude de cas en MARTE (Auteurs : J. Delatour & J. Champeau). Réaliser un modèle de l'étude de cas PaceMaker et identifier dans le chapitre 3 les propriétés caractéristiques (du domaine de l ?embarqué).

Chapitre 10 Analyse à partir du modèle (Auteurs : F. Boniol, Ph. Dhaussy, S. Gérard & J-C. Roger). Utiliser les outils et techniques disponibles dans l'environnement MARTE pour vérifier concrètement des propriétés du modèle de l'étude de cas.

Chapitre 11 Déploiement et génération de code à partir du modèle (Auteurs : C. Mradhai, A. Radermacher & S. Gérard). Utiliser les outils et techniques disponibles dans l'environnement MARTE pour produire du code (ou assister la production de code).

Partie III AADL

Chapitre 12 Présentation des concepts de AADL (Auteurs : J. Hugues). Présentation détaillé des principes et de la notation AADL.

Chapitre 13 Modélisation de l'étude de cas en AADL (Auteurs : E. Borde). Réaliser un modèle de l'étude de cas PaceMaker et identifier dans le chapitre 3 les propriétés caractéristiques (du domaine de l'embarqué) à vérifier.

Chapitre 14 Analyse à partir du modèle (Auteurs : Th. Robert & J. Hugues). Utiliser les outils et techniques disponibles dans l'environnement AADL pour vérifier concrètement des propriétés du modèle de l'étude de cas.

Chapitre 15 Génération de code à partir du modèle (Auteurs : L. Pautet). Utiliser les outils et techniques disponibles dans l'environnement AADL pour produire du code (ou assister la production de code).

Chapitre 16 Synthèse sur la modélisation de systèmes complexes (Auteurs : F. Kordon, J. Hugues, A. Canals & A. Dohet). Bien repositionner les différentes parties dans une démarche coordonnée. Proposer notre vision de l'utilisation de ces trois démarches.

4 Auteurs

Les contributions des auteurs sont indiquées dans le plan de l'ouvrage. Il est à noter qu'ils sont tous des acteurs majeurs du domaine considéré. Ils ont également tous participé, soit à l'élaboration ou à la normalisation d'un des langages étudié dans le livre soit à leur utilisation opérationnelle dans des projets industriels. Les quatre directeurs de l'ouvrage figurent en tête de cette liste :

Prénom	**Nom**	**Institution**	**Email**
Fabrice	Kordon	LIP6	Fabrice.Kordon@lip6.fr
Jérôme	Hugues	ISAE	Jerome.HUGUES@isae.fr
Agusti	Canals	C-S	Agusti.CANALS@c-s.fr
Alain	Dohet	DGA	alain.dohet@dga.defense.gouv.fr
Ludovic	Apvrille	Telecom Paris Tech	ludovic.apvrille@telecom-paristech.fr
Frederic	Boniol	ONERA CdT	boniol@cert.fr
Etienne	Borde	Telecom Paris Tech	etienne.borde@telecom-paristech.fr
Jean-Michel	Bruel	IRIT	bruel@irit.fr
Joel	Champeau	ENSTA-Bretagne	joel.champeau@ensta-bretagne.fr
Pierre	de Saqui-Sannes	ISAE	p.de-saqui-sannes@isae.fr
Jérôme	Delatour	ESEO	jerome.delatour@eseo.fr
Philippe	Dhaussy	ENSTA-Bretagne	philippe.dhaussy@ensta-bretagne.fr
Loïc	Fejoz	RTaW	loic.fejoz@realtimeatwork.com
Sébastien	Gérard	CEA	Sebastien.Gerard@cea.fr
Philippe	Leblanc	IBM	philippe.leblanc@fr.ibm.com
Laurent	Pautet	Telecom Paris Tech	laurent.Pautet@telecom-paristech.fr
Thomas	Robert	Telecom Paris Tech	thomas.robert@telecom-paristech.fr
François	Terrier	CEA Tech	Francois.TERRIER@cea.fr
Jean-Charles	Roger	ENSTA-Bretagne	jean-charles.roger@ensta-bretagne.fr
Pascal	Roques	indépendant	pascal.roques@gmail.com

5 Directeurs de l'ouvrage

Fabrice Kordon

Professeur à l'Université P. & M. Curie, responsable de l'équipe Modélisation et Vérification du LIP6. Son domaine de recherche est à l'intersection des systèmes répartis, du Génie Logiciel et des méthodes formelles. Il est membre de nombreux comités de programmes de conférences internationales sur les activités de vérification logicielle et est membre du comité éditorial de la revue " Science of Computer Programming " (Elsevier). Il a déjà co-dirigé 5 ouvrages dont 2 chez Hermes en Français et 2 à sortir chez Wiley (traduc- tion/augmentation des livres Hermes).

Jérôme Hugues

Enseignant Chercheur à l'Institut Supérieur de l'Aéronautique et de l'Espace (ISAE,Toulouse), membre du comité de standardisation du langage AADL depuis 2006. Ses sujets de recherche couvrent l'ingénierie des systèmes embarqués et la génération de code automatique de ces systèmes depuis des langages de modélisation, intégrant des outils de vérification et d'analyse aux niveaux modèles et code. Il a participé à l'organisation de nombreuses conférences. Il est membre de Ada France, SysML France, l'INCOSE et la SEE ou ? il participe au groupe "Systèmes Complexes". Il a co-rédigé un ouvrage sur Ada et la concurrence, publié chez Cambridge University Press, et plusieurs chapitres d'ouvrages.

Agusti Canals

Ingénieur Génie Logiciel (Université Paul SABATIER, Toulouse -M2 ISI). Il travaille à CS Communication et Systèmes depuis 1981. à ce jour Adjoint au Directeur Technique de CS, Manager de contrat et Expert en génie Logiciel (certifié " UML Professional " et " SysML Builder " par l'OMG). Coordinateur du livre NEPTUNE (2003) et de la revue Génie Logiciel (pour les actes NEPTUNE, 2003-2011). Il est membre de Ada France, SysML France (membre fondateur), l'AFIS et la SEE (membre actif au groupe Génie Logiciel et président du groupe Systèmes Complexes).

Alain Dohet

Ingénieur général de l'armement, en poste à la Direction Générale pour l'Armement (organisme du ministère de la défense assurant la conduite des programmes de systèmes). Responsable de l'orientation des activités, compétences, méthodes et outils dans le domaine des systèmes de systèmes (SdS), de l'ingénierie système, ainsi que dans celui de l'analyse, à des fins de certification, de la sûreté de fonctionnement des systèmes informatiques embarqués et des logiciels critiques. A contribué à un précédent ouvrage de la SEE sur la "Gestion de la complexité et de l'information dans les grands systèmes critiques". A créé et organisé, en liaison avec la SEE, les sessions 2007 à 2010 de l'"Ecole Systèmes de systèmes", qui rassemble chaque année les ingénieurs et chercheurs intéressés par les problématiques SdS pour faire le point sur l'évolution des approches et partager les bonnes pratiques.

Study on Heterogeneous Modeling

Action Spécifique 2011 du GDR GPL

Benoit Baudry[1], Benoit Combemale[1], Julien DeAntoni[2], and Frédéric Mallet[2]

[1] Firstname.Lastname@irisa.fr
Equipe Triskell (UMR IRISA), Rennes
[2] Firstname.Lastname@i3s.unice.fr
Equipe AOSTE (INRIA - UMR I3S - UNS), Sophia Antipolis

Abstract. Heterogeneity becomes a key concern in software engineering, transverse to various communities. In this paper, we summarize a study on heterogeneous modeling realized during an *Action Spécifique 2011*, a short term initiative (from May 2011 to December 2011) funded by the GDR GPL from CNRS, and led by the IRISA and I3S labs.

1 Context of the Study

Heterogeneity can be revealed in very different form depending on the domain and concern. In the context of software engineering, we identify three kind of heterogeneity :

- Heterogeneous systems: orchestration of sub-systems (e.g., ULS)
- Heterogeneous execution platforms: cooperation of execution platforms (e.g., simulation engines)
- Heterogeneous modeling: cooperation of domain specific models

These three kinds of heterogeneity are independent and possibly complementary. Indeed, a uniform system (i.e., a system that could not be decomposed into autonomous sub systems) can be described by separating the concerns through heterogeneous modeling languages (structure, behavior, time...), and inversely an heterogeneous system (*e.g.,* System of Systems) can be uniformly described at a higher level of abstraction. In both cases, and depending of the intrinsic nature of the system (performance, distributed, ..), the system can be heterogeneously executed, both at design time (cooperation of simulation engine) and at runtime (using different execution "units"). Anyway, the three kinds of heterogeneity can be combined and can be independently addressed.

The *Action Spécifique 2011* summarized in this paper aim at studying the current approaches in heterogeneous modeling (see `http://www.combemale.fr/research/gemoc/as2011`).

2 Criteria of the Study

We summarize in this paper a survey whose goal is to classify various approaches for heterogeneous modeling according to specific criteria. In this section we give an unambiguous description of such criteria and motivates the choices of the criteria while providing definitions needed to understand semantics of programming/modeling languages according to the state of the art.

First, a well accepted basis is that software languages (including programming and modeling languages) are built upon two fundamental features: syntax and semantics. Syntax refers to the concepts and relations that specify a well-formed program or model. Semantics refers to the meaning of these programs or models [1]. In the following, we detail the classical understanding of syntax and semantics according to the domain of use. We also use the term *mogram* to describe a program as well as a model [2].

The syntax of a language `L1` can be either concrete or abstract. The concrete syntax is usually the one finally used for someone who wants to write mograms in the `L1` language. This syntax can be textual or graphical. A textual concrete syntax can for instance be specified using a notation like BNF (Backus-Naur Form) as usually done in the context of programming language. BNF and their variant provide an elegant way to specify a concrete syntax. However, one limitation of the concrete syntax is that it specifies the

way the language looks like but does not focus on its structural aspect. The abstract syntax avoids the specification of specific keywords and syntactic conventions to keep only the structure of a language. Since the middle sixties, researchers have shown the importance to consider the abstract syntax of a language to reason about its meaning [3,4]. The use of an abstract syntax is currently promoted by the Model Driven Engineering (MDE) . In an MDE development the abstract syntax is a first-order citizen whose definition is given via a metamodel. The main artifact is then a model, i.e., a manageable artifact that represents the abstract syntax of a specific instance of the metamodel. The abstract syntax can then be linked to a concrete syntax to ease the edition. Because the abstract syntax keeps only the structural aspects of specifications, its definition, i.e., the metamodel, is a set of concepts and relations. While the formal semantics of models are seldom dealt with, the abstract syntax of programs has been largely used to specify the semantics of programs [1,5,6]. The semantics of a program can mainly be given in five different ways:

operational the semantics is defined by an abstract interpreter of the language where rules define how operators modify the system state [6,5];

denotational the semantics is defined by an association of each language construction to a mathematical function [6,5,7];

transformational the semantics is defined by reducing constructs of the language to more elementary ones by means of transformations into a simpler language whose semantics is already given [8].

axiomatic the semantics is defined by a mathematical theory associated with each language elements to enable the proof of some properties [6,5,9].

attribute grammar the semantics is defined by decorations of a context free grammar with attributes you are interested in. Basically attributes can take values from arbitrary domains and arbitrary functions can be specified, written in a language of choice, to describe how attributes values in rules are derived from each other [10].

Among these very common ways to specify formally a language semantics, each has its own pros and cons but an often desired goal is to avoid over specification because it reduces interpreter/compiler possibilities. It is possible to specify the semantics of concurrent program with such techniques (see chapter 14 of [5] for a quick overview). However, such approaches are tedious and each concurrency or synchronization operators must be formalized for each language that deals with concurrency.

Concurrency theory has been heavily inspired by the works of Hoare and Milner on process algebras, and also by the works on general net theory (Place/Transition nets). Milner's work on CCS (Calculus of Communicating System [11]) and Hoare's work on CSP (Communicating Sequential Processes [12]); were originally inspired from an inherently concurrent model called the "Actor Model" [13]. They independently suggested the same novel communication primitives: an atomic action of synchronization, with the possible exchange of values, as the central primitive of communication. An important point of such languages is that they provide explicit operators to capture concurrency and synchronization.

During the same period, the notion of MoC (*Model of Computation*, including time and communication) appears with work on digraphs [14] and the use of Petri nets techniques [15,16]. These models have largely been refined and studied to specify the concurrency within and inter computations [17]. Even though multiple variants exist, the main idea is to represent the concurrent computations as nodes of a labeled directed graph. Each node of the graph corresponds to an operation in the computation and each edge represents a queue of data directed from one node to another. The operation execution takes data/tokens off the incoming edges and places results on outgoing edges. An operation can "fire" only if there are sufficient tokens on incoming edges. By doing so, MoCs focus on the data dependencies of programs. More precisely, they represent causality relationships between computations so that they can extract the potential parallelism from the causalities. They do not focus on the operation(s) performed by a node while it was the case for usual specification of the formal semantics of languages. It is also noticeable that the evolution of the notion of MoC put a large emphasis on the role of time and its impact on the functional specification.

Based on the different approaches used in the state of the art to implement languages, we defined four criteria on languages that enable the comparison between heterogeneous approaches and semantics:

1. syntax: it defines the concepts of the language as well as their relationships. It can either be concrete or abstract, based on the Model driven engineering technology or not.

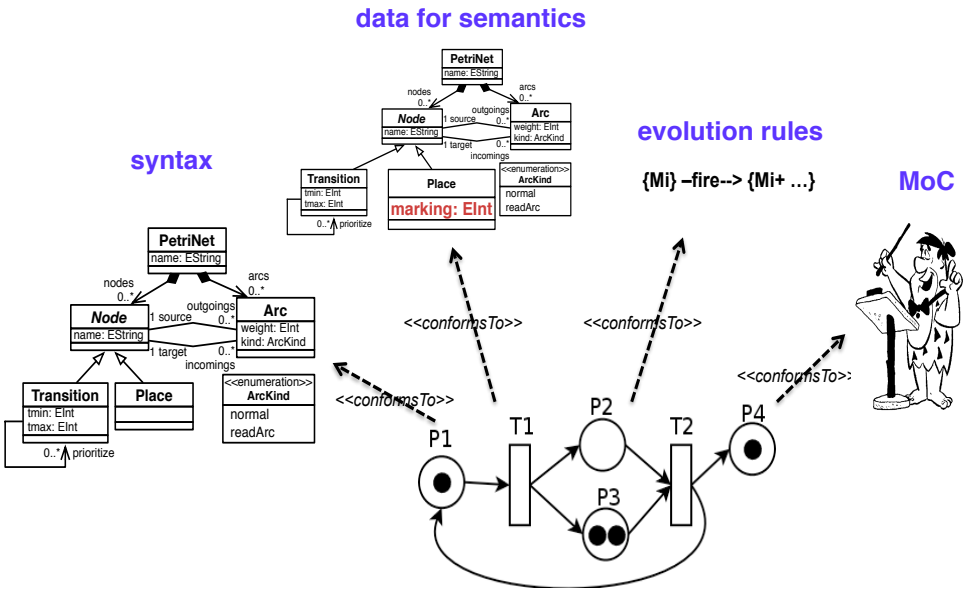


Fig. 1: Illustration of our four criteria on the Petri net language

2. data for semantics: it models the set of data needed to encapsulate the state of a system and on which the semantics is specified. It can be seen as the data part of an attribute grammar or as the data used in the specification of the system state in structural operational semantics rules. It can also be the domain on which the functions of a denotational semantics are applied.
3. evolution rules: It specifies how the data for semantics evolves according to the operator of the language. It is close to the function that can be defined in an attribute grammar or in the denotational semantics or close to the effect part of a structural operational semantics rule. It can also be seen as the set of transformations applied on the data in a transformational approach. We limit these evolution rules to define only the data manipulation and propose to separate all synchronization, concurrency and time, which is the matter of the MoC.
4. MoC: it defines the way the different evolution rules are scheduled. To do so, causalities, concurrency and time are specified and linked to the activation of evolution rules. By splitting evolution rules and MoC, it is easier to point at the changes of the system state from the ordering in which the changes occur.

We illustrate in Figure 1 our four criteria based on the Petri net language.

3 Preliminary Results of the Study

Based on the previous criteria, Figure 2 shown an excerpt of our classification (S = Syntax, E = Evolution rules, and M = Moc). Note that the "data for semantics" does not impact the classification and are not considered in the classification.

In this figure, we offer four main classes of existing approaches, and a positioning of perspectives (called GeMoC in the figure). For each category, we specify whether the approaches provide mechanisms for languages composition ($< s1, e1, d1, m1 > \circ < s2, e2, d2, m2 >$). For each criteria, we also specify whether it is *free* (can be defined by users), *variable* (can be chosen by users) or *fixed* (imposed by the approach).

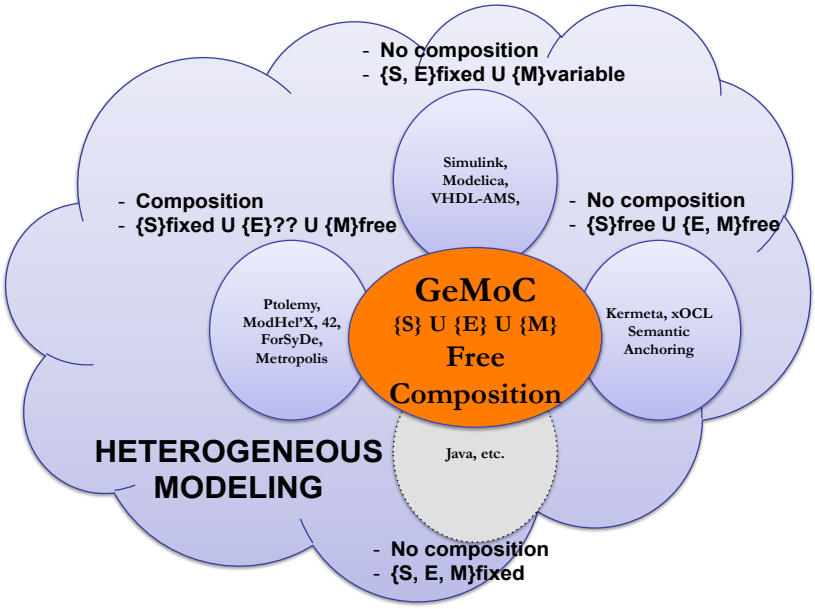


Fig. 2: Preliminary results of our survey on heterogeneous modeling

References

1. Schmidt, D.A.: Programming language semantics. In: In CRC Handbook of Computer Science, CRC Press (1995) 28–1
2. Kleppe, A.: Software Language Engineering: Creating Domain-Specific Languages Using Metamodels. Addison-Wesley Professional (2008)
3. McCarthy, J.: Towards a mathematical science of computation. Information Processing **62** (1962) 21–28
4. McCarthy, J., Painter, J.: Correctness of a compiler for arithmetic expressions. Mathematical Aspects of Computer Science **19** (1967)
5. Winskel, G.: The formal semantics of programming languages: an introduction. MIT Press, Cambridge, MA, USA (1993)
6. Meyer, B.: Introduction to the theory of programming languages. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1990)
7. Stoy, J.E.: Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. MIT Press, Cambridge, MA, USA (1981)
8. Pepper, P.: A study on transformational semantics. In Bauer, F., Broy, M., Dijkstra, E., Gerhart, S., Gries, D., Griffiths, M., Guttag, J., Horning, J., Owicki, S., Pair, C., Partsch, H., Pepper, P., Wirsing, M., Wössner, H., eds.: Program Construction. Volume 69 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (1979) 322–405 10.1007/BFb0014674.
9. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12** (October 1969) 576–580
10. Knuth, D.E.: Semantics of context-free languages. Theory of Computing Systems **2** (1968) 127–145 10.1007/BF01692511.
11. Milner, R.: Communication and concurrency. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1989)
12. Hoare, C.A.R.: Communicating sequential processes. Commun. ACM **21** (August 1978) 666–677
13. Hewitt, C., Bishop, P., Steiger, R.: A universal modular actor formalism for artificial intelligence. In: Proceedings of the 3rd international joint conference on Artificial intelligence, Morgan Kaufmann Publishers Inc. (1973) 235–245
14. Cerf, V.G.: Multiprocessors, semaphores, and a graph model of computation. PhD thesis (1972) AAI7222158.
15. Ramchandani, C.: Analysis of asynchronous concurrent systems by timed petri nets. Technical report, Cambridge, MA, USA (1974)
16. Ramamoorthy, C., Ho, G.: Performance evaluation of asynchronous concurrent systems using petri nets. IEEE Transactions on Software Engineering **6** (1980) 440–449
17. Miller, R.: A comparison of some theoretical models of parallel computation. IEEE Transactions on Computers **100**(8) (1973) 710–717

Campagne de collecte de données et vie privée

Nicolas Haderer[1], Miguel Núñez, del Prado Cortez[2,3], Romain Rouvoy[1], Marc-Olivier Killijian[2], and Matthieu Roy[2,3]

[1] INRIA Lille – Nord Europe, Project-team ADAM
University Lille 1, LIFL – CNRS UMR 8022, France
{nicolas.haderer,romain.rouvoy}@inria.fr
[2] LAAS-CNRS, France
[3] Université de Toulouse ; UPS, INSA, INP, ISAE ; LAAS ; F-31077 Toulouse, France
{mnunezde,mkilliji,mroy}@laas.fr

Résumé Les communautés scientifiques ont souvent recours à la simulation dans le but de valider leurs théories. Cependant, la pertinence des résultats obtenus est fortement dépendante de la qualité des traces générées par les simulateurs. Ce phénomène est particulièrement vrai lorsque l'on considère les traces de mobilité humaine qui sont difficilement prévisibles. Dans ce contexte, la popularité des nouvelles générations de smartphones, équipés d'une grande variété de capteurs (GPS, bluetooth, accéléromètre, etc.), offre de nouvelles perspectives pour la collecte de données réalistes au sein d'une population. Cependant, la nature sensible, du point de vue de la vie privée, des informations collectées représente un des principaux obstacles aux déploiement généralisé d'une application de collecte de données et à son adoption auprès des utilisateurs.

C'est pourquoi nous présentons UBILAB , une nouvelle plate-forme permettant aux scientifiques de mettre en place facilement des campagnes de collecte de données et d'inférer automatiquement différentes attaques sur les données partagées par les utilisateurs mobiles afin de les avertir d'un risque potentiel d'atteinte à leurs informations privées.

1 Introduction

La nouvelle génération de smartphones (Android, iPhone), maintenant équipée d'une grande variété de capteurs (GPS, bluetooth, accéléromètre, etc.), offre de nouvelles perspectives à diverses communautés scientifiques afin de réaliser différentes campagnes de collectes de données massives d'une population et de son environnement. Ces données peuvent ainsi être exploitées pour mieux comprendre les mouvements d'une population, de mettre au point de nouveaux protocoles de communication, d'analyser les interactions sociales des utilisateurs, etc. La nature sensible des données collectées, généralement couplant des informations temporelles et géographiques, peuvent révéler des informations critiques sur la vie privée d'un utilisateur (résidence privée, opinion politique ou réseau social), même si celles-ci ont été préalablement anonymisées. Ce risque potentiel représente un des principaux obstacles aux déploiement généralisé d'une application de collecte de données et à son adoption auprès des utilisateurs.

Dans ce contexte, nous présentons UBILAB , une plate-forme dédié à la gestion de campagnes de collecte de données auprès d'utilisateurs de téléphones mobiles. UBILAB est le résultat de la l'association de deux plate-formes : ANTDROID [5] et GEPETO (GeoPrivacy Enhanced TOolkit)[1]. Ce système profite ainsi de l'architecture de ANTDROID pour rapidement mettre en place une campagne de collecte de données, et

des algorithmes de GEPETO pour inférer automatiquement différentes attaques sur les données partagées par les utilisateurs mobiles afin de les avertir d'un risque potentiel d'atteinte à leurs vies privées.

2 La plateforme ANTDROID

La plate-forme ANTDROID est composée de deux parties — chacune est destinée aux différents acteurs évoluant dans la plate-forme : les scientifiques et les cobayes. Le serveur d'application dédié, destiné aux scientifiques, repose sur le style architectural REST (*REpresentational State Transfer*) fournissant l'ensemble des services pour la définition, la diffusion et l'exploitation d'une expérience de collecte de données. L'application cliente pour smartphone est destinée aux utilisateurs voulant participer à une campagne de collecte de données.

La définition d'une campagne est décrite avec un langage de script dédié ANTDROID SCRIPTING LANGUAGE (cf. Listing 1.1), permettant de spécifier les données qui doivent être collectées par le téléphone mobile (lignes 6-11) et l'événement qui déclenche la collecte (ligne 5). Ce choix permet aux scientifiques de bénéficier d'une grande flexibilité pour la définition du schéma de leurs données n'imposant aucune structure particulière. Les données collectées sont ensuite stockées au format XML et peuvent être facilement manipulées par le langage XQuery pour analyse et extraction. ANTDROID est une plate-forme évoluant dans le *cloud*, proposant une interface web pour créer et gérer une campagne de collecte de données sans requérir l'installation complexe de logiciels.

```
var gsm = new Experiment("GSM_Signal_Strength")                          1
// Update of GPS position every 120 seconds                              2
gsm.configure("Location", { strategy:"fine", period:120 })               3
// Event triggered when location changes                                 4
gsm.onLocationStateChange(event) {                                       5
    return { trace: {                                                    6
        lat:  event.getLatitude(),                                       7
        lon:  event.getLongitude(),                                      8
        time: event.getTime(),                                           9
        ss:   event.getSignalStrengthLevel()                             10
}   } }                                                                  11
// Start collecting activity traces                                      12
gsm.start()                                                              13
```

Listing 1.1. GSM Signal Strength Experiment

L'application cliente est une application Android téléchargeable, utilisée par une communauté d'utilisateurs pour partager leurs traces d'activités. Pour ce faire, l'utilisateur s'abonne à une ou plusieurs campagnes publiées par des scientifiques. Ces campagnes correspondent à des scripts de collecte automatique des informations requises par le scientifique. Ces scripts sont téléchargés via le serveur d'application dédié puis interprétés par un moteur de script intégré dans l'application cliente. Afin de maîtriser toute diffusion d'information, l'application cliente dispose de différents contrôles permettant aux utilisateurs d'autoriser ou non la collecte de certaines informations jugées trop sensibles (par ex., sa position). L'application intègre également un certain nombre d'optimisations permettant de limiter sa consommation énergétique afin de ne pas perturber les usages des utilisateurs.

3 Analyses des traces de mobilité avec GEPETO

Les données collectées par le téléphone mobile peuvent ensuite être envoyées manuellement par l'utilisateur ou automatiquement lorsque le téléphone est alimenté en

courant pour limiter sa consommation énergétique. Avant d'être disponible pour les scientifiques, les données sont stockées dans une base de données temporaire ou un ensemble d'analyses sera effectué par la plate-forme GEPETO (GeoPrivacy Enhanced TOolkit) permettant de détecter si les données partagées peuvent comporter des risques vis-à-vis de la vie privée de l'utilisateur.

3.1 Le processus d'analyse

Le processus d'analyse commence par l'élaboration d'un modèle de mobilité basé sur les chaines de Markov (MMC) d'un individu [4]. Ce modèle repose sur un automate probabiliste où les états représentent des points d'intérêt (*POI*) et les transitions, c'est-à-dire les déplacement d'un *POI* à un autre. Le modèle produit peut être à gros grain (représentation globale du mouvement à travers plusieurs villes) ou à grain fin (représentation plus spécifique dans une seule ville avec plus de sémantique). La Figure 1 montre les modèles de Bob : à gauche, un modèle générique des déplacements et à droite, un modèle plus spécifique pour la ville (Toulouse) où il passe le plus de temps.

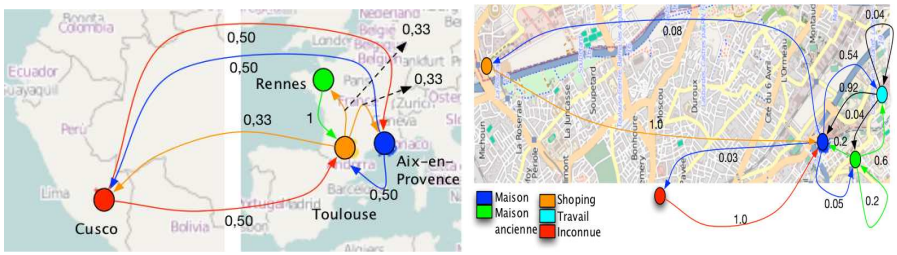


FIGURE 1. Exemple de *MMC* globale et spécifique de *Bob*.

Avec ce modèle (MMC), il est d'abord possible de calculer la prédictibilité potentielle en utilisant l'équation proposée par Gambs et *al.* [3] rappelée dans l'équation 1 ci-dessous. Par exemple, supposons que le vecteur stationnaire pour le MMC spécifique de Bob est $VS = \{0,68, 0,24, 0,06, 0,02, 0,01\}$, alors, selon le résultat sus-citée, la prédictibilité est de 64%. Il est également possible d'améliorer la prédictibilité, c'est-à-dire avoir un modèle plus intrusif, en utilisant un modèle qui mémorise les n derniers *POIs* où l'individu était (tel que le modèle *n*-MMC[3]) ou un modèle qui introduit le temps comme le *t*-MMC [2].

$$Pred = \sum_{k=1}^{l} (\pi(k) \times P_{max\_out}(k, *)) \quad (1)$$

Ensuite, en observant la configuration du MMC, il est possible d'en déduire des sémantiques des POIs basées sur la densité des états et les transitions. On peut alors attacher des labels aux points d'intérêts, tels que maison, travail, loisir, etc., qui sont, en eux-même, des « quasi identificateurs » et permettent à un adversaire de trouver l'identité d'un utilisateur anonyme [1]. Par la suite, il est possible d'associer une adresse physique à un état quelconque en utilisant les cordonnées GPS du medioid, grâce aux services de géocodage inversé. Dans la table 1, nous présentons, comme illustration, le résumé des informations mentionnées ci-dessus concernant les POIs de Bob.

Densité	Latitude	Longitude	Label	Adresse
390	43.57291	1.46875	Maison	4-6 Allée des sciences Appliquées
70	43.563	1.4774	Travail	L.A.A.S. - C.N.R.S.
61	43.56743	1.46617	Ancienne maison	Résidence Universitaire Clément Ader
57	43.57664	1.46806	Inconnue	Chemin des Herbettes
20	43.62799	1.48265	Shopping	Rue Saint-Jean Balma

TABLE 1. Résumé des informations concernant les POIs de Bob.

Le résultat de ces analyses est ensuite renvoyée à l'utilisateur afin qu'il puisse évaluer si les données collectées peuvent compromettre sa vie privée. L'utilisateur peut alors ensuite décider de valider les données pour les rendre directement disponible pour les scientifiques, de les supprimer, ou d'appliquer des algorithmes d'assainissement (distorsion aléatoire, sous échantillonnage, etc..) fourni par GEPETO pour ajouter du bruit sur les traces de mobilité.

4 Conclusion

Nous avons présenté UBILAB , une plate-forme pour la gestion de campagnes de collecte de données. Nous avons montré comment cette plate-forme peut être utilisée par les scientifiques pour diffuser et exploiter une expérience de collecte de données et les différents mécanismes pour avertir les participants si leurs données partagées peuvent comporter des risques vis-à-vis de leurs informations privées. Dans ce cas, les données peuvent être supprimées ou dégradées en appliquant différents algorithmes d'assainissement fournis par la plate-forme. Cependant, une trop grande dégradation des données peuvent les rendre inutiles pour les scientifiques. Nous envisageons donc dans de futur travaux d'élaborer un mécanisme permettant d'avoir un compromis entre la protection des informations privées des utilisateurs et la qualité des données offertes aux scientifiques.

Références

1. S. Gambs, M.-O. Killijian, and M. N. del Prado Cortez. Gepeto : a geoprivacy-enhancing toolkit. *AINA'09 Workshop on Advances in Mobile Computing and Applications : Security, Privacy and Trust, Perth, Australia.*, August 2009.
2. S. Gambs, M.-O. Killijian, and M. N. del Prado Cortez. Towards temporal mobility markov chains. In *DYNAM, an OPODIS workshop*, Toulouse, France., December 2011.
3. S. Gambs, M.-O. Killijian, and M. N. del Prado Cortez. Next place prediction using mobility markov chains. In *Mobility, Privacy and Measurement.*, Bern, Switzerland, April 2012.
4. S. Gambs, M.-O. Killijian, and M. N. n. del Prado Cortez. Show me how you move and i will tell you who you are. In *Transactions on Data Privacy*, volume 2, pages 103–126, Catalonia, Spain, August 2011.
5. N. Haderer, R. Rouvoy, and L. Seinturier. AntDroid : A distributed platform for mobile sensing. Rapport de recherche RR-7885, INRIA, Lille, France., February 2012.

Sessions des groupes de travail

Session de l'action AFSEC

Approches Formelles des Systémes Embarqués Communicants

Une extension de Lustre avec du temps continu

Marc Pouzet[1 2 3]

[1] Université Pierre et Marie Curie
[2] DI, École normale supérieure
[3] Institut Universitaire de France

Abstract. Les modeleurs hybrides tels que Simulink/Stateflow ont évolué pour passer d'environnements de simulation seulement à des environnement de développement logiciel, intégrant compilateurs, outils de test, de vérification formelle et de simulation numérique. Ils rencontrent un énorme succès dans l'embarqué car ils permettent de modéliser, simuler, tester à la fois un controleur (discret) et son environnement physique (continu) jusqu'à obtenir du code embarqué exécutable à partir d'une spécification de haut niveau. Bien que largement utilisés dans de nombreux domaines industriels, ces outils soulèvent un certain nombre de questions qui concernent leur typage, leur sémantique et leur compilation.

L'an dernier, nous avions présenté l'intérêt et les bases d'un langage hybride étendant le langage Lustre avec des équations différentielles ordinaires (ODEs). Il permettait de combiner, au sein d'un même code source, des équations de suites écrites en Lustre et des ODEs. Ce langage repose sur l'utilisation d'un système de types pour cloisonner les signaux discrets et les signaux continus. Plutôt que de définir une technique de compilation ad-hoc pour le langage étendu, nous avions montré comment effectuer cette compilation par traduction vers le sous-ensemble synchrone, lui-même traduit vers du code impératif par un compilateur synchrone quelconque.

Dans cet exposé, je montrerai comment étendre ce langage élémentaire avec des structures de controle telles que les automates hiérarchiques similaires à ceux qui existent dans SCADE 6. Ces automates permettent de représenter directement les automates hybrides introduits par Pnueli et al. On se rapproche ainsi d'un langage proche de Simulink/Stateflow. Je montrerai comment les mêmes techniques de typage et de compilation par traduction peuvent être étendues.

Ces idées sont mises en oeuvre dans un prototype, appelé ZéLus, et une démonstration sera faite durant l'exposé.

Ce travail a été réalisé en collaboration avec Albert Benveniste, Tim Bourke et Benoit Caillaud.

References

1. Benveniste, A., Bourke, T., Caillaud, B., Pouzet, M.: Non-Standard Semantics of Hybrid Systems Modelers. Journal of Computer and System Sciences (JCSS) **Special issue in honor of Amir Pnueli** (2012)

2. Benveniste, A., Bourke, T., Caillaud, B., Pouzet, M.: Divide and recycle: types and compilation for a hybrid synchronous language. In: ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES'11), Chicago, USA (April 2011)
3. Benveniste, A., Bourke, T., Caillaud, B., Pouzet, M.: A Hybrid Synchronous Language with Hierarchical Automata: Static Typing and Translation to Synchronous Code. In: ACM SIGPLAN/SIGBED Conference on Embedded Software (EMSOFT'11), Taipei, Taiwan (October 2011)

Synthèse de protocoles à partir d'exigences - Garantie de correction par contrôle asynchrone.

Loïc Hélouët[1], Claude Jard[2], and Rouwaida Abdallah[2]

[1] INRIA Rennes, IRISA, Rennes, France
[2] ENS Cachan, antenne de Bretagne, France
loic.helouet@irisa.fr, claude.jard@irisa.fr, rouwaida.abdallah@irisa.fr

Abstract. Ce travail étudie le problème de la synthèse de protocoles distribués à partir d'un ensemble d'exigences de haut niveau décrites par des High-level Message Sequence Charts (HMSCs). Bien que la synthèse par projection soit réputée incorrecte, nous proposons une technique de contrôle asynchrone, qui garantit la correction du protocole synthétisé pour un sous-ensemble de HMSCs.

1 Introduction

La synthèse de protocoles est un sujet récurrent en informatique. Le problème consiste à partir d'une description de haut niveau d'un système et d'une description d'une architecture cible, pour obtenir in fine un ensemble de machines indépendantes qui implémentent sur l'architecture la spécification de départ. L'architecture spécifie notamment comment les actions du système se distribuent sur un ensemble de machines physiques, les moyens de communication, etc.

Le problème se décline de manière différente pour chaque langage de spécification, architecture, langage cible, mais aussi pour chaque notion de correction d'implémentation. Dans ce travail, nous proposons une technique de synthèse de machines communicantes à partir de High-level Message Sequence Charts (HMSCs), un langage de scénarios standardisé par l'ITU [7]. Les scénarios sont des langages d'expression d'exigences de systèmes distribués asynchrones. Les HMSCs sont en quelque sorte des automates étiquetés par des ordres partiels (ou bMSCs) qui représentent des interactions asynchrones entres processus. Ils tiennent déjà compte de la répartition des actions sur une architecture. Les HMSCs permettent d'exprimer des choix, des itérations,...Un exemple de HMSC est représenté Figure 1. Le comportement M_1, qui implique les processus A, B, C peut être répété un nombre arbitraire de fois avant que le comportement M_2, impliquant les processus A, B, D ne soit réalisé.

Nous choisissons comme modèle cible les automates communicants, tels que proposés par Brand & Zafiropulo [4]. Ce modèle, tout en restant formel, est très proche d'une réelle implémentation en machine. Nous dirons qu'un ensemble d'automates communicants est une *synthèse correcte* d'une spécification par scénarios si les deux modèles définissent *exactement* les mêmes ensembles de comportement.

Nous proposons dans ce document un résumé de travaux disponibles dans [8]. Ces travaux sont par ailleurs en cours de soumission.

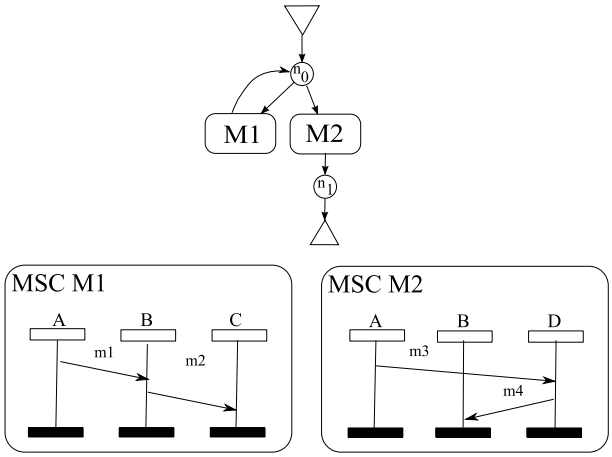


Fig. 1. Un exemple de HMSC

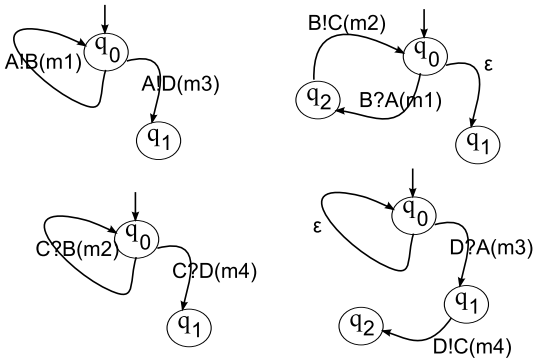


Fig. 2. Des automates communicants

2 Synthèse

De nombreux travaux se sont penchés sur la synthèse à partir de scénarios [5, 6, 2, 9], et l'approche la plus appropriée est de projeter la spécification sur chacun de ses agents (ou processus) pour dériver une machine communicante par agent [1]. Cependant, cette approche n'est correcte que pour un sous-ensemble restreint de spécifications : dans de nombreux cas, les machines synthétisées autorisent plus de comportements que la spécification de départ.

Le plus souvent, ces nouveaux comportements apparaissent parce qu'un contrôle global exprimé dans la spécification est perdu lors de la projection. Parmi les HMSCs qui ne sont pas implémentables, une classe bien connue est celle des HMSCs non-locaux [3]. Un HMSC est dit local si pour tout choix du protocole décrit, un et un seul processus décide de la suite du comportement du protocole. Lorsque deux processus peuvent décider sans communication d'exécuter une branche d'un HMSC plutôt qu'une autre, on considère que ce choix est décrit à un trop haut niveau d'abstraction, et ne peut pas être implémenté.

Cependant, il existe d'autres types de HMSCs qui ne sont pas implémentables par projection. Les automates communicants obtenus par projection de l'exemple de la Figure 1 sont représentés Figure 2. Cette implémentation n'est pas correcte. En effet, nous pouvons voir Figure 3 deux comportements possibles de ces automates communicants. Si le chronogramme de gauche est un comportement attendu du protocole, le diagramme de droite n'est pas spécifié dans le HMSC de départ. Ce dernier comportement correspond à une situation dans laquelle le processus B choisit de recevoir le message $m4$ avant le message $m2$, ce qui n'était pas initialement prévu lorsque le processus A envoie le message $m1$ avant d'envoyer le message $m3$. Dans ce type de HMSCs, une partie de l'ordre de réception des messages est perdu durant la projection en automates communicants.

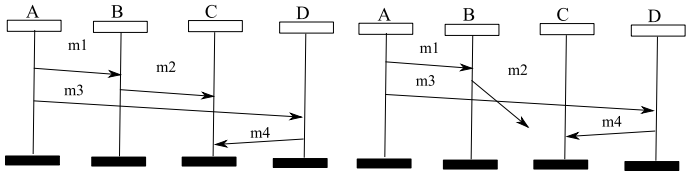


Fig. 3. Deux comportements du protocole synthétisé

Nous avons identifé une sous-classe de scénarios dits "reconstructibles" permettant une synthèse correcte par simple projection [6]. Dans cette classe, les processus disposent de suffisamment d'information pour toujours savoir quel message recevoir. Cependant, cette classe est assez restreinte. Par exemple, le HMSC de la figure 1 n'est pas reconstructible.

3 Architecture de contrôle

Le problème posé par les techniques de synthèse par projection est qu'une information de contrôle globale est perdue. Comme montré dans la Figure 3, certains processus peuvent consommer trop tôt des messages qui leurs sont destinés. Une solution pour assurer la correction de la synthèse est de l'interdire à partir de HMSCs non reconstructibles. Une autre possibilité est de fournir aux processus l'information perdue durant la synthèse par ajout de contrôleurs et instrumentation des messages échangés.

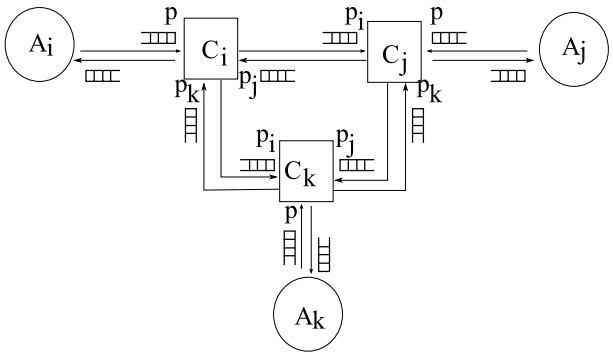


Fig. 4. L'architecture d'implémentation contrôlée

Nous proposons une instrumentation des machines obtenues par projection par des contrôleurs asynchrones. Nous ajoutons à chaque automate synthétisé A_i un contrôleur C_i. Le rôle de chaque contrôleur est d'intercepter et d'instrumenter les messages envoyés par l'automate contrôlé (ou qui lui sont destinés) par une horloge vectorielle. Cette horloge permet aux machines synthétisées de retrouver une partie de l'information de contrôle perdue durant la projection, et de décider si un message doit être retransmis à un automate ou retardé (i.e. conservé dans un buffer du contrôleur). Cette technique permet d'assurer une synthèse correcte pour toute la classe des HMSCs à choix local. La figure 4 présente l'architecture pour une spécification portant sur 3 processus i, j, k. Les automates synthétisés sont A_i, A_j, A_k, et leurs contrôleurs sont C_i, C_j, C_k. Les automates ne communiquent pas directement entre eux, mais via leurs contrôleurs, qui instrumentent les messages échangés par des horloges vectorielles, et retardent leur délivrance aux automates si nécessaire pour se conformer au protocole spécifié par le HMSC d'origine.

En appelant $\mathcal{L}(H)$ l'ensemble des exécutions d'un HMSC H, et $L(\bigcup_{i\in 1\ldots n} A_i \cup \bigcup_{i\in 1\ldots n} C_i)$ le langage des exécutions des machines et contrôleurs synhétisés (en abstrayant les actions des contrôleurs), nous pouvons montrer le résultat suivant:

Theorem 1. *Soit H un HMSC à choix local à n processus, et soient $\{A_i\}_{i\in 1\ldots n}$ et $\{C_i\}_{i\in 1\ldots n}$ les automates et les contrôleurs synthétisés. Alors*

$$\mathcal{L}(H) = L(\bigcup_{i\in 1\ldots n} A_i \cup \bigcup_{i\in 1\ldots n} C_i)$$

4 Conclusion

Nous avons proposé une technique de contrôle asynchrone permettant une implémentation correcte de HMSCs à choix local. Nous pensons que cette technique de contrôle asynchrone s'étend au delà des problèmes de synthèse, et pourrait permettre de garantir des propriétés assez simples (bornes sur les canaux de communication, évitement d'un mauvais état global, ...) de systèmes distribués. Bien sûr, l'ajout de contrôleurs et donc de nouveaux messages dans un système peut faire varier le comportement temporel du système. Un autre aspect intéressant serait donc d'analyser l'impact du contrôle asynchrone sur les performances du système.

References

1. Miguel Abdalla, Ferhat Khendek, and Greg Butler. New results on deriving sdl specifications from mscs. In *SDL Forum*, pages 51–66, 1999.
2. N. Baudru and R. Morin. Synthesis of safe message-passing systems. In *FSTTCS*, pages 277–289, 2007.
3. H. Ben-Abdallah and S. Leue. Syntactic detection of process divergence and non-local choice in message sequence charts. In *Proc. of TACAS'97*, volume 1217 of *LNCS*, pages 259 – 274, April 1997.
4. D. Brand and P. Zafiropoulo. On communicating finite state machines. Technical Report RZ1053, IBM Zurich Research Lab, 1981.
5. B. Genest, A. Muscholl, H. Seidl, and M. Zeitoun. Infinite-state high-level MSCs: Model-checking and realizability. In *ICALP*, volume 2380 of *LNCS*, pages 657–668, 2002.
6. L. Hélouët and C. Jard. Conditions for synthesis of communicating automata from HMSCs. In *5th International Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, 2000.
7. ITU-T. Z.120 : Message sequence charts (MSC). Technical report, International Telecommunication Union, 1998.
8. Claude Jard, Rouwaida Abdallah, and Loïc Hélouët. Realistic Implementation of Message Sequence Charts. Rapport de recherche RR-7597, INRIA, 2011.
9. M. Mukund, K.N. Kumar, and M. Sohoni. Synthesizing distributed finite-state systems from MSCs. In *CONCUR*, pages 521–535, 2000.

Avoiding Shared Clocks in Networks of Timed Automata

Sandie Balaguer and Thomas Chatain*

INRIA & LSV (CNRS & ENS Cachan)

Abstract. Networks of timed automata (NTA) are widely used to model distributed real-time systems. Quite often in the literature, the automata are allowed to share clocks, i.e. the transitions of one automaton may be guarded by a condition on the value of clocks reset by another automaton. This is a problem when one considers implementing such model in a distributed architecture, since reading clocks a priori requires communications which are not explicitly described in the model. We focus on the following question: given a NTA $A_1 \parallel A_2$ where A_2 reads some clocks reset by A_1, does there exist a NTA $A_1' \parallel A_2'$ without shared clocks with the same behavior as the initial NTA? For this, we allow the automata to exchange information during synchronizations only, in particular by copying the value of their neighbor's clocks. We discuss a formalization of the problem and give a criterion using the notion of contextual timed transition system, which represents the behavior of A_2 when in parallel with A_1. Finally, we effectively build $A_1' \parallel A_2'$ when it exists.

Keywords: networks of timed automata, shared clocks, implementation on distributed architecture, contextual timed transition system, behavioral equivalence for distributed systems

1 Introduction

Timed automata [3] are one of the most famous formal models for real-time systems. They have been deeply studied and very mature tools are available, like Uppaal [20], Epsilon [15] and Kronos [12].

Networks of Timed Automata (NTA) are a natural generalization to model real-time distributed systems. In this formalism each automaton has a set of clocks that constrain its real-time behavior. But quite often in the literature, the automata are allowed to share clocks, which provides a special way of making the behavior of one automaton depend on what the others do. Actually shared clocks are relatively well accepted and can be a convenient feature for modeling systems. Moreover, since NTA are almost always given a sequential semantics, shared clocks can be handled very easily even by tools: once the NTA is transformed into a single timed automaton by the classical product construction, the notion of distribution is lost and the notion of shared clock itself becomes meaningless.

Here we are concerned with the expressive power of shared clocks according to the distributed nature of the system. We are not aware of any previous study

* This work is partially supported by the French ANR project ImpRo.

about this aspect. Our purpose is to identify NTA where sharing clocks could be avoided, i.e. NTA which syntactically use shared clocks, but whose semantics could be achieved by another NTA without shared clocks. To simplify, we look at NTA made of two automata A_1 and A_2 where only A_2 reads clocks reset by A_1. The first step is to formalize what aspect of the semantics we want to preserve in this setting. Then the idea is essentially to detect cases where A_2 can avoid reading a clock because its value does not depend on the actions that are local to A_1 and thus unobservable to A_2. To generalize this idea we have to compute the knowledge of A_2 about the state of A_1. We show that this knowledge is maximized if we allow A_1 to communicate its state to A_2 each time they synchronize on a common action.

In order to formalize our problem we need an appropriate notion of behavioral equivalence between two NTA. We explain why classical comparisons based on the sequential semantics, like timed bisimulation, are not sufficient here. We need a notion that takes the distributed nature of the system into account. That is, a component cannot observe the moves and the state of the other and must choose its local actions according to its partial knowledge of the state of the system. We formalize this idea by the notion of contextual timed transition systems (contextual TTS).

Then we express the problem of avoiding shared clocks in terms of contextual TTS and we give a characterization of the NTA for which shared clocks can be avoided. Finally we effectively construct a NTA without shared clocks with the same behavior as the initial one, when this is possible.

Related work. The semantics of time in distributed systems has already been debated. The idea of localizing clocks has already been proposed and some authors [1,5,17] have even suggested to use local-time semantics with independently evolving clocks. Here we stay in the classical setting of perfect clocks evolving at the same speed. This is a key assumption that provides an implicit synchronization and lets us know some clock values without reading them.

Many formalisms exist for real-time distributed systems, among which NTA [3] and time Petri nets [22]. So far, their expressiveness was compared [6, 11, 14, 25] essentially in terms of sequential semantics that forget concurrency. In [4], we defined a concurrency-preserving translation from time Petri nets to networks of timed automata.

While partial-order semantics and unfoldings are well known for untimed systems, they have been very little studied for distributed real-time systems [9, 13]. Partial order reductions for (N)TA were proposed in [5,21,23]. Behavioral equivalence relations for distributed systems, like history-preserving bisimulations were defined for untimed systems only [8, 18].

Finally, our notion of contextual TTS deals with knowledge of agents in distributed systems. This is the aim of epistemic logics [19], which have been extended to real-time in [16, 26]. Our notion of contextual TTS also resembles the powerset construction used in games of imperfect information [7, 24].

Organization of the paper. The paper is organized as follows. Section 2 recalls basic notions about TTS and NTA. Section 3 presents the problem of avoiding shared clocks on examples and rises the problem of comparing NTA component by component. For this, the notion of contextual TTS is developed in Section 4. The problem of avoiding shared clocks is formalized and characterized in terms of contextual TTS. Then Section 5 presents our construction.

The proofs are given in the appendix.

2 Preliminaries

2.1 Timed Transition Systems

The behavior of timed systems is often described as timed transition systems.

Definition 1. *A* timed transition system *(TTS) is a tuple* $(S, s_0, \Sigma, \to)$ *where* S *is a set of states,* $s_0 \in Q$ *is the initial state,* Σ *is a finite set of actions disjoint from* $\mathbb{R}_{\geq 0}$*, and* $\to \subseteq S \times (\Sigma \cup \mathbb{R}_{\geq 0}) \times S$ *is a set of edges.*

For any $a \in \Sigma \cup \mathbb{R}_{\geq 0}$, we write $s \xrightarrow{a} s'$ if $(s, a, s') \in \to$, and $s \xrightarrow{a}$ if for some s', $(s, a, s') \in \to$. A *path* of a TTS is a possibly infinite sequence of transitions $\rho = s \xrightarrow{d_0} s'_0 \xrightarrow{a_0} \cdots s_n \xrightarrow{d_n} s'_n \xrightarrow{a_n} \cdots$, where, for all i, $d_i \in \mathbb{R}_{\geq 0}$ and $a_i \in \Sigma$. A path is *initial* if it starts in s_0. A path $\rho = s \xrightarrow{d_0} s'_0 \xrightarrow{a_0} \cdots s_n \xrightarrow{d_n} s'_n \xrightarrow{a_n} s'_n \cdots$ generates a *timed word* $w = (a_0, t_0)(a_1, t_1) \dots (a_n, t_n) \dots$ where, for all i, $t_i = \sum_{k=0}^{i} d_k$. The duration of w is $\delta(w) = \sup_i t_i$ and the untimed word of w is $\lambda(w) = a_0 a_1 \dots a_n \dots$, and we denote the set of timed words over Σ and of duration d as $\mathrm{TW}(\Sigma, d) = \{w \mid \delta(w) = d \wedge \lambda(w) \in \Sigma^*\}$. Lastly, we write $s \xrightarrow{w} s'$ if there is a path from s to s' that generates the timed word w.

In the definitions of product and timed bisimulation below, we use two TTS $T_1 = (S_1, s_1^0, \Sigma_1, \to_1)$ and $T_2 = (S_2, s_2^0, \Sigma_2, \to_2)$, and $\Sigma_i^{\not\varepsilon}$ denotes $\Sigma_i \setminus \{\varepsilon\}$.

Product of TTS. The *product* of T_1 and T_2, denoted by $T_1 \otimes T_2$, is the TTS $\big(S_1 \times S_2, (s_1^0, s_2^0), \Sigma_1 \cup \Sigma_2, \to\big)$, where $\to$ is defined as:

- $(s_1, s_2) \xrightarrow{a} (s'_1, s_2)$ iff $s_1 \xrightarrow{a}_1 s'_1$, for any $a \in \Sigma_1 \setminus \Sigma_2^{\not\varepsilon}$,
- $(s_1, s_2) \xrightarrow{a} (s_1, s'_2)$ iff $s_2 \xrightarrow{a}_2 s'_2$, for any $a \in \Sigma_2 \setminus \Sigma_1^{\not\varepsilon}$,
- $(s_1, s_2) \xrightarrow{a} (s'_1, s'_2)$ iff $s_1 \xrightarrow{a}_1 s'_1$ and $s_2 \xrightarrow{a}_2 s'_2$, for any $a \in (\Sigma_1^{\not\varepsilon} \cap \Sigma_2^{\not\varepsilon}) \cup \mathbb{R}_{\geq 0}$.

Timed Bisimulations. Let $\approx$ be a binary relation over $S_1 \times S_2$. We write $s_1 \approx s_2$ for $(s_1, s_2) \in \approx$. $\approx$ is a *strong timed bisimulation* relation between T_1 and T_2 if $s_1^0 \approx s_2^0$ and $s_1 \approx s_2$ implies that, for any $a \in \Sigma \cup \mathbb{R}_{\geq 0}$, if $s_1 \xrightarrow{a}_1 s'_1$, then, for some s'_2, $s_2 \xrightarrow{a}_2 s'_2$ and $s'_1 \approx s'_2$; and conversely, if $s_2 \xrightarrow{a}_2 s'_2$, then, for some s'_1, $s_1 \xrightarrow{a}_1 s'_1$ and $s'_1 \approx s'_2$.

Let $\Rightarrow_i$ (for $i \in \{1, 2\}$) be the transition relation defined as:

- $s \overset{\varepsilon}{\Rightarrow}_i s'$ if $s(\xrightarrow{\varepsilon}_i)^* s'$,
- $\forall a \in \Sigma$, $s \overset{a}{\Rightarrow}_i s'$ if $s(\xrightarrow{\varepsilon}_i)^* \xrightarrow{a}_i (\xrightarrow{\varepsilon}_i)^* s'$,
- $\forall d \in \mathbb{R}_{\geq 0}$, $s \overset{d}{\Rightarrow}_i s'$ if $s(\xrightarrow{\varepsilon}_i)^* \overset{d_0}{\Longrightarrow}_i (\xrightarrow{\varepsilon}_i)^* \cdots \overset{d_n}{\Longrightarrow}_i (\xrightarrow{\varepsilon}_i)^* s'$, where $\sum_{k=0}^{n} d_k = d$.

$\approx$ is a *weak timed bisimulation* relation between T_1 and T_2 if $s_1^0 \approx s_2^0$ and $s_1 \approx s_2$ implies that, for any $a \in \Sigma \cup \mathbb{R}_{\geq 0}$, if $s_1 \xrightarrow{a}_1 s_1'$, then, for some s_2', $s_2 \stackrel{a}{\Rightarrow}_2 s_2'$ and $s_1' \approx s_2'$; and conversely. We write $T_1 \approxeq T_2$ (resp. $T_1 \approx T_2$) when there is a strong (resp. weak) timed bisimulation between T_1 and T_2.

2.2 Networks of Timed Automata

The set $\mathcal{B}(X)$ of clock constraints over the set of clocks X is defined by the grammar $g ::= x \bowtie k \mid g \wedge g$, where $x \in X$, $k \in \mathbb{N}$ and $\bowtie \in \{<, \leq, =, \geq, >\}$. Invariants are clock constraints of the form $i ::= x \leq k \mid x < k \mid i \wedge i$.

Definition 2. *A* network of timed automata (NTA) *[3] is a parallel composition of timed automata (TA) denoted as $A_1 \parallel \cdots \parallel A_n$, with $A_i = (L_i, \ell_i^0, X_i, \Sigma_i, E_i, Inv_i)$ where L_i is a finite set of* locations*, $\ell_i^0 \in L_i$ is the* initial *location, X_i is a finite set of* clocks*, Σ_i is a finite set of* actions*, $E_i \subseteq L_i \times \mathcal{B}(X_i) \times \Sigma_i \times 2^{X_i} \times L_i$ is a set of* edges*, and $Inv_i : L_i \to \mathcal{B}(X_i)$ assigns* invariants *to locations.*

If $(\ell, g, a, r, \ell') \in E_i$, we also write $\ell \xrightarrow{g,a,r} \ell'$. For such an edge, g is the *guard*, a the *action* and r the set of clocks to be *reset*. $C_i \subseteq X_i$ is the set of clocks reset by A_i and for $i \neq j$, $C_i \cap C_j$ may not be empty.

Semantics. To simplify, we give the semantics of a network of two TA $A_1 \parallel A_2$. We denote by $((\ell_1, \ell_2), v)$ a *state* of the NTA, where ℓ_1 and ℓ_2 are the current locations, and $v : X \to \mathbb{R}_{\geq 0}$, with $X = X_1 \cup X_2$, is a *clock valuation* that maps each clock to its current value. A state is legal only if its valuation v satisfies the invariants of the current locations, denoted by $v \models Inv_1(\ell_1) \wedge Inv_2(\ell_2)$. The initial state is $s_0 = ((\ell_1^0, \ell_2^0), v_0)$, where v_0 maps each clock to 0. For each set of clocks $r \subseteq X$, the valuation $v[r]$ is defined by $v[r](x) = 0$ if $x \in r$ and $v[r](x) = v(x)$ otherwise. For each $d \in \mathbb{R}_{\geq 0}$, the valuation $v + d$ is defined by $(v + d)(x) = v(x) + d$ for each $x \in X$.

The *TTS generated by* $A_1 \parallel A_2$ is $\mathrm{TTS}(A_1 \parallel A_2) = (S, s_0, \Sigma_1 \cup \Sigma_2, \to)$, where S is the set of legal states, and $\to$ is defined by

- Local action: $((\ell_1, \ell_2), v) \xrightarrow{a} ((\ell_1', \ell_2), v')$ iff $a \in \Sigma_1 \setminus \Sigma_2^{\not\varepsilon}$, $\ell_1 \xrightarrow{g,a,r} \ell_1'$, $v \models g$, $v' = v[r]$ and $v' \models Inv_1(\ell_1')$, and similarly for a local action in $\Sigma_2 \setminus \Sigma_1^{\not\varepsilon}$,
- Synchronization: $((\ell_1, \ell_2), v) \xrightarrow{a} ((\ell_1', \ell_2'), v')$ iff $a \neq \varepsilon$, $\ell_1 \xrightarrow{g_1,a,r_1} \ell_1'$, $\ell_2 \xrightarrow{g_2,a,r_2} \ell_2'$, $v \models g_1 \wedge g_2$, $v' = v[r_1 \cup r_2]$ and $v' \models Inv_1(\ell_1') \wedge Inv_2(\ell_2')$,
- Time delay: $\forall d \in \mathbb{R}_{\geq 0}, ((\ell_1, \ell_2), v) \xrightarrow{d} ((\ell_1, \ell_2), v + d)$ iff $\forall d' \in [0, d], v + d' \models Inv_1(\ell_1) \wedge Inv_2(\ell_2)$.

A *run* of a NTA is an initial path in its TTS. The semantics of a TA A alone can also be given as a TTS denoted by $\mathrm{TTS}(A)$ with only local actions and delay. A TA is *non-Zeno* iff for every infinite timed word w generated by a run, time diverges (i.e. $\delta(w) = \infty$). This is a common assumption for TA. In the sequel, we always assume that the TA we deal with are non-Zeno.

Remark 1. Let $A_1 \parallel A_2$ be such that $X_1 \cap X_2 = \emptyset$. Then $\mathrm{TTS}(A_1) \otimes \mathrm{TTS}(A_2)$ is isomorphic to $\mathrm{TTS}(A_1 \parallel A_2)$. This is not true in general when $X_1 \cap X_2 \neq \emptyset$.

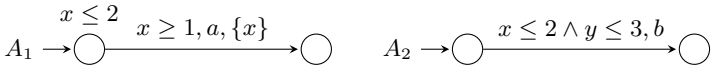


Fig. 1. A_2 could avoid to read clock x which belongs to A_1.

3 Need for Shared Clocks

3.1 Problem Setting

We are interested in detecting the cases where it is possible to avoid sharing clocks, so that the model can be implemented using no other synchronization than those explicitly described by common actions.

To start with, let us focus on the case of a network of two TA, $A_1 \parallel A_2$, such that A_1 does not read the clocks reset by A_2, and A_2 may read the clocks reset by A_1. We want to know whether A_2 really needs to read these clocks, or if another NTA $A_1' \parallel A_2'$ could achieve the same behavior as $A_1 \parallel A_2$ without using shared clocks.

A first remark is that our problem makes sense only if we insist on the distributed nature of the system, made of two separate components. On the other hand, if the composition operator is simply used as a convenient syntax for describing a system that is actually implemented on a single sequential component, then a simple product automaton would perfectly describe the system and every clock becomes local.

So, let us consider the example of Fig. 1, made of two TA, supposed to describe two separate components. Remark that A_2 reads clock x which is reset by A_1. But a simple analysis shows that this reading could be avoided: because of the condition on its clock y, A_2 can only take transition b before time 3; but x cannot reach value 2 before time 3, since it must be reset between time 1 and 2. Thus, forgetting the condition on x in A_2 would not change the behavior of the system.

3.2 Transmitting Information During Synchronizations

Consider now the example of Fig. 2. Here also A_2 reads clock x which is reset by A_1, and here also this reading could be avoided. The idea is that A_1 could transmit the value of x when synchronizing, and afterwards, any reading of x in A_2 can be replaced by the reading of a new clock x' dedicated to storing the value of x which is copied on the synchronization. Therefore A_2 can be replaced by A_2' pictured in Fig. 2, while preserving the behavior of the NTA, but also the behavior of A_2 w.r.t. A_1.

We claim that we cannot avoid reading x without this copy of clock. Indeed, after the synchronization, the maximal delay in the current location depends on the exact value of x, and even if we find a mechanism to allow A_2' to move to different locations according to the value of x at synchronization time, infinitely many locations would be required (for example, if s occurs at time 2, x may have any value in $(1, 2]$).

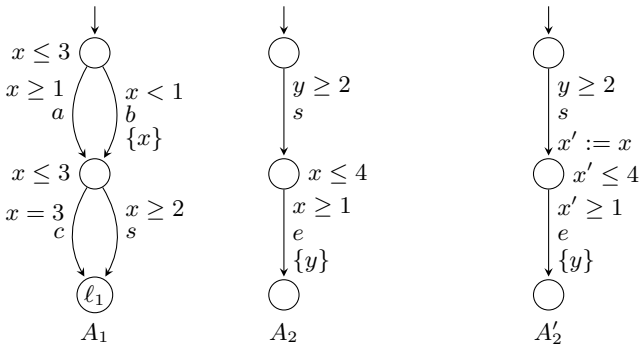


Fig. 2. A_2 reads x which belongs to A_1 and A'_2 does not.

Coding Transmission of Information. In order to model the transmission of information during synchronizations, we allow A'_1 and A'_2 to use a larger synchronization alphabet than A_1 and A_2. This allows A'_1 to transmit discrete information like its current location, to A'_2.

But we saw that A'_1 also needs to transmit the exact value of its clocks. For this we allow an automaton to copy its neighbor's clocks into local clocks during synchronizations. This is denoted as updates of the form $x' := x$ in A'_2 (see Fig. 2). This is a special case of updatable timed automata as defined in [10]. Moreover, as shown in [10], the class we consider, with diagonal-free constraints and updates with equality (they allow other operators) is not more expressive than classical TA for the sequential semantics (any updatable TA of the class is bisimilar to a classical TA), and the emptiness problem is PSPACE-complete.

Semantics. $\mathrm{TTS}(A_1 \parallel A_2)$ can be defined as previously, with the difference that the synchronizations are now defined by: $((\ell_1, \ell_2), v) \xrightarrow{a} ((\ell'_1, \ell'_2), v')$ iff $\ell_1 \xrightarrow{g_1,a,r_1}_1 \ell'_1$, $\ell_2 \xrightarrow{g_2,a,r_2,u}_2 \ell'_2$ where u is a partial function from X_2 to X_1, $v \models g_1 \wedge g_2$, $v' = (v[r_1 \cup r_2])[u]$, and $v' \models Inv(\ell'_1) \wedge Inv(\ell'_2)$. The valuation $v[u]$ is defined by $v[u](x) = v(u(x))$ if $u(x)$ is defined, and $v[u](x) = v(x)$ otherwise.

Here, we choose to apply the reset $r_1 \cup r_2$ before the update u, because we are interested in sharing the state reached in A_1 after the synchronization, and r_1 may reset some clocks in $C_1 \subseteq X_1$.

3.3 Towards a Formalization of the Problem

We want to know whether A_2 really needs to read the clocks reset by A_1, or if another NTA $A'_1 \parallel A'_2$ could achieve the same behavior as $A_1 \parallel A_2$ without using shared clocks. It remains to formalize what we mean by "having the same behavior" in this context.

First, we impose that the locality of actions is preserved, i.e. A'_1 uses the same set of local actions as A_1, and similarly for A'_2 and A_2. For the synchronizations,

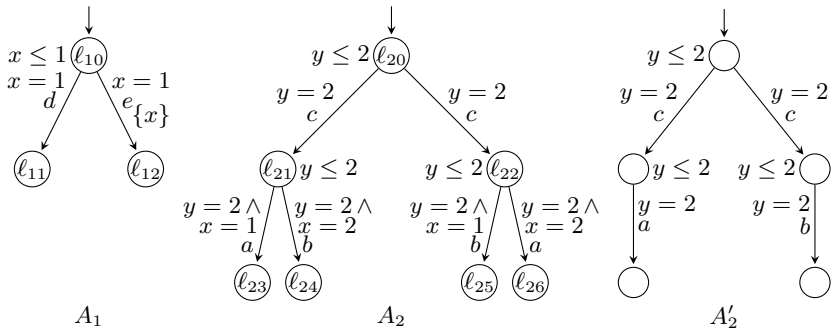


Fig. 3. A_2 needs to read the clocks of A_1 and $\mathrm{TTS}(A_1 \parallel A_2) \approx \mathrm{TTS}(A_1 \parallel A'_2)$

we have explained earlier why we allow A'_1 and A'_2 to use a larger synchronization alphabet than A_1 and A_2. The correspondence between the two alphabets will be done by a mapping ψ (this point will be refined later).

Now we have to impose that the behavior is preserved. The first idea that comes in mind is to impose bisimulation between $\psi(\mathrm{TTS}(A'_1 \parallel A'_2))$ (i.e. $\mathrm{TTS}(A'_1 \parallel A'_2)$ with synchronization actions relabeled by ψ) and $\mathrm{TTS}(A_1 \parallel A_2)$. But this is not sufficient, as illustrated by the example of Fig. 3 (where ψ is the identity). Intuitively A_2 needs to read x when in ℓ_{21} (and similarly in ℓ_{22}) at time 2, because this reading determines whether it will perform a or b, and the value of x cannot be inferred from its local state given by ℓ_{21} and the value of y. Anyway $\mathrm{TTS}(A_1 \parallel A'_2)$ is bisimilar to $\mathrm{TTS}(A_1 \parallel A_2)$, and A'_2 does not read x.

What we see here is that, if we focus on the point of view of A_2 and A'_2, these two automata do not behave the same. As a matter of fact, when A_2 fires one edge labeled by c, it has not read x yet, and there is still a possibility to fire a or b, whereas when A'_2 fires one edge labeled by c, there is no more choice afterwards. Therefore we need a relation between A'_2 and A_2, and in the general case, a relation between A'_1 and A_1 also.

4 Contextual Timed Transition Systems

As we are interested in representing a partial view of one of the components, we need to introduce another notion, that we call *contextual timed transition system*. This resembles the powerset construction used in game theory to capture the knowledge of an agent about another agent [24].

Notations. $\mathbb{S} = \Sigma_1^{\mathscr{I}} \cap \Sigma_2^{\mathscr{I}}$ denotes the set of common actions. Q_1 denotes the set of states of $\mathrm{TTS}(A_1)$. When $s = ((\ell_1, \ell_2), v)$ is a state of $\mathrm{TTS}(A_1 \parallel A_2)$, we also write $s = (s_1, s_2)$, where $s_1 = (\ell_1, v_{|X_1})$ is in Q_1, and $s_2 = (\ell_2, v_{|X_2 \setminus X_1})$, where $v_{|X}$ is v restricted to X.

Definition 3 (UR(s))**.** *Let* TTS(A_1) = $(Q_1, s_0, \Sigma_1, \to_1)$ *and* $s \in Q_1$*. The set of sates of* A_1 *reachable from* s *by local actions in 0 delay (and therefore not observable by* A_2*) is denoted by* $\mathrm{UR}(s) = \{s' \in Q_1 \mid \exists w \in \mathrm{TW}(\Sigma_1 \setminus \Sigma_2^{\not\epsilon}, 0) : s \xrightarrow{w}_1 s'\}$.

Contextual States. The states of this contextual TTS are called *contextual states*. They can be regarded as possibly infinite sets of states of TTS($A_1 \parallel A_2$) for which A_2 is in the same location and has the same valuation over $X_2 \setminus X_1$. A_2 may not be able to distinguish between some states (s_1, s_2) and (s'_1, s_2). In $\mathrm{TTS}_{A_1}(A_2)$, these states are grouped into the same contextual state. However, when $X_2 \cap X_1 \neq \emptyset$, it may happen that A_2 is able to perform a local action or delay from (s_1, s_2) and not from (s'_1, s_2), even if these states are grouped in a same contextual state.

Definition 4 (Contextual TTS). *Let* TTS($A_1 \parallel A_2$) = $(Q, q_0, \Sigma_1 \cup \Sigma_2, \Rightarrow)$*. Then, the* TTS of A_2 in the context of A_1*, denoted by* $\mathrm{TTS}_{A_1}(A_2)$*, is the TTS* $(S, s_0, (\Sigma_2 \setminus \mathbb{S}) \cup (\mathbb{S} \times Q_1), \to)$*, where*

- $S = \{(S_1, s_2) \mid \forall s_1 \in S_1, (s_1, s_2) \in Q\}$,
- $s_0 = (S_1^0, s_2^0)$*, s.t.* $(s_1^0, s_2^0) = q_0$ *and* $S_1^0 = \mathrm{UR}(s_1^0)$,
- $\to$ *is defined by*
  - *Local action: for any* $a \in \Sigma_2 \setminus \mathbb{S}$, $(S_1, s_2) \xrightarrow{a} (S'_1, s'_2)$ *iff* $\exists s_1 \in S_1 : (s_1, s_2) \overset{a}{\Rightarrow} (s_1, s'_2)$*, and* $S'_1 = \{s_1 \in S_1 \mid (s_1, s_2) \overset{a}{\Rightarrow} (s_1, s'_2)\}$
  - *Synchronization: for any* $(a, s'_1) \in \mathbb{S} \times Q_1$, $(S_1, s_2) \xrightarrow{a, s'_1} (\mathrm{UR}(s'_1), s'_2)$ *iff* $\exists s_1 \in S_1 : (s_1, s_2) \overset{a}{\Rightarrow} (s'_1, s'_2)$
  - *Local delay: for any* $d \in \mathbb{R}_{\geq 0}$, $(S_1, s_2) \xrightarrow{d} (S'_1, s'_2)$ *iff* $\exists s_1 \in S_1$, $w \in \mathrm{TW}(\Sigma_1 \setminus \Sigma_2^{\not\epsilon}, d) : (s_1, s_2) \overset{w}{\Rightarrow} (s'_1, s'_2)$*, and* $S'_1 = \{s'_1 \mid \exists s_1 \in S_1$, $w \in \mathrm{TW}(\Sigma_1 \setminus \Sigma_2^{\not\epsilon}, d) : (s_1, s_2) \overset{w}{\Rightarrow} (s'_1, s'_2)\}$

For example, consider A_1 and A_2 of Fig. 3. The initial state is $\big(\{(\ell_{10}, 0)\}, (\ell_{20}, 0)\big)$. From this contextual state, it is possible to delay 2 time units and reach the contextual state $\big(\{(\ell_{11}, 2), (\ell_{12}, 1)\}, (\ell_{20}, 2)\big)$. Indeed, during this delay, A_1 has to perform either e and reset x, or d. Now, from this contextual state, we can take an edge labeled by c, and reach $\big(\{(\ell_{11}, 2), (\ell_{12}, 1)\}, (\ell_{21}, 2)\big)$. Lastly, from this new state, a can be fired, because it is enabled by $((\ell_{12}, 1), (\ell_{21}, 2))$ in the TTS of the NTA, and the reached contextual state is $\big(\{(\ell_{12}, 1)\}, (\ell_{23}, 2)\big)$.

We say that there is no restriction in $\mathrm{TTS}_{A_1}(A_2)$ if whenever a local step is possible from a reachable contextual state, then it is possible from all the states (s_1, s_2) that are grouped into this contextual state. In the example above, there is a restriction in $\mathrm{TTS}_{A_1}(A_2)$ because we have seen that a is enabled only by $((\ell_{12}, 1), (\ell_{21}, 2))$, and not by all states merged in $\big(\{(\ell_{11}, 2), (\ell_{12}, 1)\}, (\ell_{21}, 2)\big)$. Formally, we use the predicate $noRestriction_{A_1}(A_2)$ defined as follows.

Definition 5 ($noRestriction_{A_1}(A_2)$)**.** *The predicate* $noRestriction_{A_1}(A_2)$ *holds iff for any reachable state* (S_1, s_2) *of* $\mathrm{TTS}_{A_1}(A_2)$*, both*

- $\forall a \in \Sigma_2 \setminus \mathbb{S}, (S_1, s_2) \xrightarrow{a} (S'_1, s'_2) \iff \forall s_1 \in S_1, (s_1, s_2) \overset{a}{\Rightarrow} (s_1, s'_2)$*, and*
- $\forall d \in \mathbb{R}_{\geq 0}, (S_1, s_2) \xrightarrow{d} \iff \forall s_1 \in S_1, \exists w \in \mathrm{TW}(\Sigma_1 \setminus \Sigma_2^{\not\epsilon}, d) : (s_1, s_2) \overset{w}{\Rightarrow}$

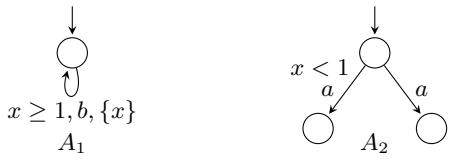


Fig. 4. $\mathrm{TTS}_{Q_1}(A_1) \otimes \mathrm{TTS}_{A_1}(A_2) \lessapprox \mathrm{TTS}_{Q_1}(A_1 \parallel A_2)$, although there is a restriction in $\mathrm{TTS}_{A_1}(A_2)$

Remark 2. If A_2 does not read X_1, then $noRestriction_{A_1}(A_2)$.

Sharing of Information on the Synchronizations. Later we assume that during a synchronization, A_1 is allowed to transmit all its state to A_2, that is why, in $\mathrm{TTS}_{A_1}(A_2)$, we distinguish the states reached after a synchronization according to the state reached in A_1. We also label the synchronization edges by a pair $(a, s_1) \in \mathbb{S} \times Q_1$ where a is the action and s_1 the state reached in A_1.

For the sequel, let $\mathrm{TTS}_{Q_1}(A_1)$ (resp. $\mathrm{TTS}_{Q_1}(A_1 \parallel A_2)$) denote $\mathrm{TTS}(A_1)$ (resp. $\mathrm{TTS}(A_1 \parallel A_2)$) where the synchronization edges are labeled by (a, s_1), where $a \in \mathbb{S}$ is the action, and s_1 is the state reached in A_1.

We can now state a nice property of unrestricted contextual TTS that is similar to the distributivity of TTS over the composition when considering TA with disjoint sets of clocks (see Remark 1). We say that a TA is *deterministic* if it has no ε-transition and for any location ℓ and action a, there is at most one edge labeled by a from ℓ.

Lemma 1. *If there is no restriction in* $\mathrm{TTS}_{A_1}(A_2)$*, then* $\mathrm{TTS}_{Q_1}(A_1) \otimes \mathrm{TTS}_{A_1}(A_2) \lessapprox \mathrm{TTS}_{Q_1}(A_1 \parallel A_2)$*. Moreover, when* A_2 *is deterministic, this condition becomes necessary.*

The example of Fig. 4 shows that the reciprocal does not hold when A_2 is not deterministic.

4.1 Need for Shared Clocks Revisited

We have argued in 3.3 that the existence of a NTA $A'_1 \parallel A'_2$ without shared clocks and such that $\psi(\mathrm{TTS}_{Q'_1}(A'_1 \parallel A'_2)) \approx \mathrm{TTS}_{Q_1}(A_1 \parallel A_2)$ is not sufficient to capture the idea that A_2 does not need to read the clocks of A_1. We are now equipped to define the relations we want to impose on the separate components, namely $\psi(\mathrm{TTS}_{Q'_1}(A'_1)) \approx \mathrm{TTS}_{Q_1}(A_1)$ and $\psi(\mathrm{TTS}_{A'_1}(A'_2)) \approx \mathrm{TTS}_{A_1}(A_2)$. And since we have seen the importance of using labeling the synchronization actions in contextual TTS by labels in $\mathbb{S} \times Q_1$ rather than in $\mathbb{S}$, the correspondence between the synchronization labels of $A'_1 \parallel A'_2$ with those of $A_1 \parallel A_2$ is now done by a mapping $\psi : \mathbb{S}' \times Q'_1 \to \mathbb{S} \times Q_1$.

This settles the problem of the example of Fig. 3 where $\mathrm{TTS}_{A_1}(A'_2) \not\approx \mathrm{TTS}_{A_1}(A_2)$ (here $A'_1 = A_1$), but as shown in Fig. 5, a problem remains. In

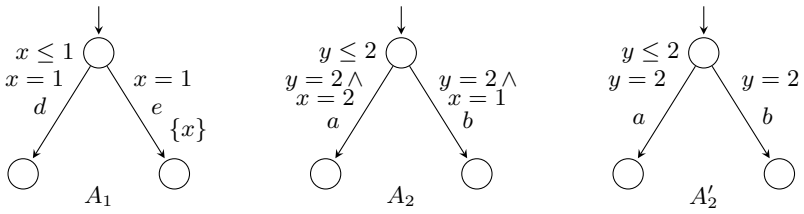


Fig. 5. A_2 needs to read the clocks of A_1 and $\mathrm{TTS}_{A_1}(A_2) \approx \mathrm{TTS}_{A_1}(A'_2)$.

this example, we can see that A_2 needs to read clock x of A_1 to know whether it has to perform a or b at time 2, and yet $\mathrm{TTS}_{A_1}(A_2) \approx \mathrm{TTS}_{A_1}(A'_2)$ (here also $A'_1 = A_1$). The intuition to understand this is that the contextual TTS merge too many states for the two systems to remain differentiable. However we remark that here, the first condition that we have required in Section 3, namely the global bisimulation between $\psi(\mathrm{TTS}(A'_1 \parallel A'_2))$ and $\mathrm{TTS}(A_1 \parallel A_2)$, does not hold.

Now we show that the conjunction of global and local bisimulations actually gives the good definition.

Definition 6 (Need for shared clocks). *Given $A_1 \parallel A_2$ such that A_1 does not read the clocks of A_2, A_2* does not need to read the clocks of A_1 *iff there exists a NTA $A'_1 \parallel A'_2$ without shared clocks (but with clock copies during synchronizations), using the same sets of local actions and a synchronization alphabet $\mathbb{S}'$ related to the original one by a mapping $\psi : \mathbb{S}' \times Q'_1 \to \mathbb{S} \times Q_1$, and such that*

1. $\psi(\mathrm{TTS}_{Q'_1}(A'_1 \parallel A'_2)) \approx \mathrm{TTS}_{Q_1}(A_1 \parallel A_2)$ *and*
2. $\psi(\mathrm{TTS}_{Q'_1}(A'_1)) \approx \mathrm{TTS}_{Q_1}(A_1)$ *and*
3. $\psi(\mathrm{TTS}_{A'_1}(A'_2)) \approx \mathrm{TTS}_{A_1}(A_2)$.

Notice that this does not mean that the clock constraints that read X_1 can simply be removed from A_2 (see Fig. 2).

Lemma 2. *When noRestriction$_{A_1}(A_2)$ holds, any NTA $A'_1 \parallel A'_2$ without shared clocks and that satisfies items 2 and 3 of Definition 6 also satisfies item 1.*

We are now ready to give a criterion to decide the need for shared clocks.

Theorem 1. *When noRestriction$_{A_1}(A_2)$ holds, A_2 does not need to read the clocks of A_1. When A_2 is deterministic, this condition becomes necessary.*

We remark from the proof that when there is a restriction in $\mathrm{TTS}_{A_1}(A_2)$, even infinite A'_1 and A'_2 would not help. Next section will be devoted to the constructive proof of the direct part of this theorem. The indirect part comes from Lemma 1.

The counterexample in Fig. 4 also works here to argue that the conditions of Lemma 2 and Theorem 1 are not necessary when A_2 is not deterministic. Indeed A'_2 with only one unguarded edge labeled by a and $A'_1 = A_1$ satisfy the three items of Definition 6 but there is a restriction in $\mathrm{TTS}_{A_1}(A_2)$.

5 Constructing a NTA Without Shared Clocks

This section is dedicated to proving Theorem 1 by constructing suitable A'_1 and A'_2. To simplify, we assume that in A_2, the guards on the synchronizations do not read X_1.

Construction. First, our A'_1 is obtained from A_1 by replacing all the labels $a \in \mathbb{S}$ on the synchronization edges of A_1 by $(a, \ell_1) \in \mathbb{S} \times L_1$, where ℓ_1 is the output location of the edge. Therefore the synchronization alphabet between A'_1 and A'_2 will be $\mathbb{S}' = \mathbb{S} \times L_1$, which allows A'_1 to transmit its location after each synchronization.

Then, the idea is to build A'_2 as a product $A_{1,2} \otimes A_{2,mod}$ ($\otimes$ denotes the product of TA as it is usually defined [3]), where $A_{2,mod}$ plays the role of A_2 and $A_{1,2}$ acts as a local copy of A'_1, from which $A_{2,mod}$ reads clocks instead of reading those of A'_1. For this, as long as the automata do not synchronize, $A_{1,2}$ will evolve, simulating a run of A'_1 that is compatible with what A'_2 knows about A'_1. And, as soon as A'_1 synchronizes with A'_2, A'_2 updates $A_{1,2}$ to the actual state of A'_1. If the clocks of $A_{1,2}$ always give the same truth value to the guards and invariants of $A_{2,mod}$ than the actual value of the clocks of A'_1, then our construction behaves like $A_1 \parallel A_2$. To check that this is the case, we equip A'_2 with an error location and edges that lead to it if there is a contradiction between the values of the clocks of A'_1 and the values of the clocks of $A_{1,2}$. The guards of these edges are the only cases where A'_2 reads clocks of A'_1. Therefore, if the error location is not reachable, they can be removed so that A'_2 does not read the clocks of A'_1. More precisely, a contradiction happens when $A_{2,mod}$ is in a given location and the guard of an outgoing edge is true according to $A_{1,2}$ and false according to A'_1, or vice versa, or when the invariant of the current location is false according to A'_1 (whereas it is true according to $A_{1,2}$, since $A_{2,mod}$ reads the clocks of $A_{1,2}$).

Namely, $\mathcal{S}_{mod} = A'_1 \parallel (A_{1,2} \otimes A_{2,mod})$ where $A_{1,2}$ and $A_{2,mod}$ are defined as follows. $A_{1,2} = (L_1, \ell_1^0, X'_1, \mathbb{S}' \cup \{\varepsilon\}, E'_1, Inv'_1)$, where

- each clock $x' \in X'_1$ is associated with a clock $c(x') = x \in X_1$ (c is a bijection from X'_1 to X_1). For any clock constraint γ, γ' denotes the clock constraint where any clock x of X_1 is substituted by x' of X'_1.
- $\forall \ell \in L_1, Inv'_1(\ell) = Inv_1(\ell)'$
- $E'_1 = \{\ell_1 \xrightarrow{g', \varepsilon_a, r'} \ell_2 \mid \exists a \in \Sigma_1 \setminus \Sigma_2^{\not g} : \ell_1 \xrightarrow{g, a, c(r')} \ell_2 \in E_1\}$
  $\cup\ \{\ell \xrightarrow{\top, (a,\ell_2), c} \ell_2 \mid \ell \in L_1 \wedge a \in \mathbb{S} \wedge \exists \ell_1 \xrightarrow{g,a,r} \ell_2 \in E_1\}$
  where c denotes the assignment of any clock $x' \in X'_1$ with the value of its associated clock $c(x') = x \in X_1$ (written $x' := x$ in Fig. 6).

$A_{2,mod} = (L_2 \cup \{\text{☹}\}, \ell_2^0, X_2 \cup X'_1, (\Sigma_2 \setminus \Sigma_1) \cup \mathbb{S}', E'_2, Inv'_2)$, where

- $\forall \ell \in L_2, Inv'_2(\ell) = Inv_2(\ell)'$ and $Inv'_2(\text{☹}) = \top$,

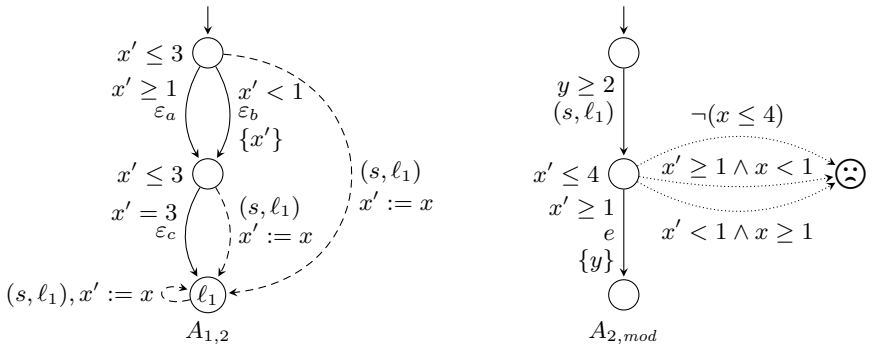


Fig. 6. $A_{1,2}$ and $A_{2,mod}$ for the example of Fig. 2

– $E'_2 = \{\ell_1 \xrightarrow{g',a,r} \ell_2 \mid \ell_1 \xrightarrow{g,a,r} \ell_2 \in E_2 \wedge a \notin \mathbb{S}\}$
$\cup \{\ell_1 \xrightarrow{g,(a,\ell),r} \ell_2 \mid \ell_1 \xrightarrow{g,a,r} \ell_2 \in E_2 \wedge a \in \mathbb{S} \wedge \ell \in L_1\}$
$\cup \{\ell \xrightarrow{\neg Inv_2(\ell),\varepsilon,\emptyset} ☹ \mid \ell \in L_2\}$
$\cup \{\ell \xrightarrow{g' \wedge \neg g,\varepsilon,\emptyset} ☹ \mid \ell \xrightarrow{g,a,r} \ell' \in E_2 \wedge a \notin \mathbb{S}\}$
$\cup \{\ell \xrightarrow{\neg g' \wedge g,\varepsilon,\emptyset} ☹ \mid \ell \xrightarrow{g,a,r} \ell' \in E_2 \wedge a \notin \mathbb{S}\}$.

For the example of Fig. 2, $A_{1,2}$ and $A_{2,mod}$ are pictured in Fig. 6.

Lemma 3. *☹ is reachable in $\mathcal{S}_{mod}$ iff there is a restriction in* $\mathrm{TTS}_{A_1}(A_2)$.

We now give a first simple case for which Theorem 1 can be proved easily. We say that A_1 has no urgent synchronization if for any location, when the invariant reaches its limit, a local action is enabled. Under this assumption, we can show that $A'_2 = A_{1,2} \otimes A'_{2,mod}$, where $A'_{2,mod}$ is $A_{2,mod}$ without location ☹ (that is never reached according to Lemma 3) and its ingoing edges, is suitable. Indeed, we can show that A'_2 does not read X_1 and is such that $\psi(\mathrm{TTS}_{A'_1}(A'_2)) \approx \mathrm{TTS}_{A_1}(A_2)$, where for any $((a, \ell_1), s_1) \in \mathbb{S}' \times Q'_1$, $\psi(((a, \ell_1), s_1)) = (a, s_1)$. Obviously, item 2 of Definition 6 holds, and Lemma 2 says that item 1 also holds.

When A_1 has urgent synchronizations, this construction allows one to check the absence of restriction in $\mathrm{TTS}_{A_1}(A_2)$, but it does not give directly a suitable A'_2. We will give the idea of the construction of A'_2 for the general case later.

In the example of Fig. 2, ☹ is not reachable in $\mathcal{S}_{mod}$ (see Fig. 6), therefore A_2 does not need to read X_1. For an example where ☹ is reachable, consider the same example with an additional edge $\xrightarrow{\top,f,\{x\}}$ from the end location of A_1 to a new location. Location ☹ can now be reached in $\mathcal{S}_{mod}$, for example consider a run where s is performed at time 2 leading to a state where $v(x) = 2$ and $v(x') = 2$, and then A_1 immediately performs f and resets x, leading to a state where the valuation v' is such that $v'(x) = 0$ and $v'(x') = 2$, and satisfies guard $x' \geq 1 \wedge x < 1$ in $\mathcal{S}_{mod}$. Therefore, with this additional edge in A_1, A_2 needs to

read X_1. Indeed, without this edge, A_2 knows that A_1 cannot modify x after the synchronization, but with this edge, A_2 does not know whether A_1 has performed f and reset x, while this may change the truth value of its guard $x \geq 1$.

Complexity. The reachability problem for timed automata is known to be PSPACE-complete [2]. We will reduce this problem to our problem of deciding whether A_2 needs to read the clocks of A_1. Consider a timed automaton A over alphabet Σ, with some location ℓ. Build the timed automaton A_2 as A augmented with two new locations ℓ' and ℓ'' and two edges, $\ell \xrightarrow{\top,\varepsilon,\emptyset} \ell'$ and $\ell' \xrightarrow{x=1,a,\emptyset} \ell''$, where x is a fresh clock, and a is some action in Σ. Let A_1 be the one of Fig. 4 with an action $b \notin \Sigma$. Then, ℓ is reachable in A iff A_2 needs to read x which belongs to A_1. Therefore the problem of deciding whether A_2 needs to read the clocks of A_1 is also PSPACE-hard.

Moreover, we can show that when A_2 is deterministic, our problem is in PSPACE. Indeed, by Theorem 1 and Lemma 3, ☹ is not reachable iff $noRestriction_{A_1}(A_2)$ iff A_2 does not need to read the clocks of A_1. Since the size of the modified system on which we check the reachability of ☹ is polynomial in the size of the original system, our problem is in PSPACE.

Dealing with Urgent Synchronizations. If we use exactly the same construction as before and allow urgent synchronizations, the following problem may occur. Remind that $A_{1,2}$ simulates a possible run of A'_1 while A'_1 plays its actual run. There is no reason why the two runs should coincide. Thus it may happen that the run simulated by $A_{1,2}$ reaches a state where the invariant expires and only a synchronization is possible. Then A'_2 is expecting a synchronization with A'_1, but it is possible that the actual A'_1 has not reached a state that enables this synchronization. Intuitively, A'_2 should then realize that the simulated run cannot be the actual one and try another run compatible with the absence of synchronization.

But it is simpler to avoid this situation, which we can do by forcing $A_{1,2}$ to simulate one of the runs of A'_1 (from the state reached after the last synchronization) that has maximal duration[1] before it synchronizes again with $A_{2,mod}$ (or never synchronizes again if possible). This choice of a run of A'_1 is as valid as the others, and it prevents the system from having to deal with the subtle situation that we described above.

For example, consider automaton A_1 in Fig. 2 without the edge labeled by c and with guard $x \leq 1$ instead of $x < 1$. We can see that $A_{1,2}$ has to fire b at time 1 and is able to wait 3 time units before synchronizing, although it is still able to synchronize at any time (we add the same dashed edges as in Fig. 6). This can be generalized for any A_1. The idea is essentially to force $A_{1,2}$ to follow the appropriate finite or ultimately periodic path in the region automaton [3] of A_1.

[1] There may not be any maximum if some time constraints are strict inequalities, but the idea can be adapted even to this case.

6 Conclusion

We have shown that in a distributed framework, when locality of actions and synchronizations matter, NTA with shared clocks cannot be easily transformed into NTA without shared clocks. The fact that the transformation is possible can be characterized using the notion of contextual TTS which represents the knowledge of one automaton about the other. Checking whether the transformation is possible is PSPACE-complete.

One conclusion is that, contrary to what happens when one considers the sequential semantics, NTA with shared clocks are strictly more expressive if we take distribution into account. This somehow justifies why shared clocks were introduced: they are actually more than syntactic sugar.

Another interesting point that we want to recall here, is the use of transmitting information during synchronizations. It is noticeable that infinitely precise information is required in general. This advocates the interest of updatable (N)TA used in an appropriate way, and more generally gives a flavor of a class of NTA closer to implementation.

Perspectives. Our first perspective is to generalize our result to the symmetrical case where A_1 also reads clocks from A_2. Then of course we can tackle general NTA with more than two automata.

Another line of research is to focus on transmission of information. The goal would be to minimize the information transmitted during synchronizations, and see for example where are the limits of finite information. Even when infinitely precise information is required to achieve the exact semantics of the NTA, it would be interesting to study how this semantics can be approximated using finitely precise information.

Finally, when shared clocks are necessary, one can discuss how to minimize them, or how to implement the model on a distributed architecture and how to handle shared clocks with as few communications as possible.

References

1. Akshay, S., Bollig, B., Gastin, P., Mukund, M., Narayan Kumar, K.: Distributed timed automata with independently evolving clocks. In: CONCUR. LNCS, vol. 5201, pp. 82–97. Springer (2008)
2. Alur, R., Dill, D.: Automata for modeling real-time systems. In: Automata, Languages and Programming, LNCS, vol. 443, pp. 322–335. Springer (1990)
3. Alur, R., Dill, D.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)
4. Balaguer, S., Chatain, Th., Haar, S.: A concurrency-preserving translation from time Petri nets to networks of timed automata. Formal Methods in System Design (2012)
5. Bengtsson, J., Jonsson, B., Lilius, J., Yi, W.: Partial order reductions for timed systems. In: CONCUR. LNCS, vol. 1466, pp. 485–500. Springer (1998)

6. Bérard, B., Cassez, F., Haddad, S., Roux, O., Lime, D.: Comparison of the expressiveness of timed automata and time Petri nets. In: FORMATS'05. LNCS, vol. 3829, pp. 211–225. Springer (2005)
7. Berwanger, D., Kaiser, Ł., Puchala, B.: Perfect-information construction for coordination in games. In: FSTTCS. Leibniz International Proceedings in Informatics, vol. 13, pp. 387–398. Leibniz-Zentrum für Informatik (2011)
8. Best, E., Devillers, R.R., Kiehn, A., Pomello, L.: Concurrent bisimulations in petri nets. Acta Inf. 28(3), 231–264 (1991)
9. Bouyer, P., Haddad, S., Reynier, P.A.: Timed unfoldings for networks of timed automata. In: ATVA. LNCS, vol. 4218, pp. 292–306. Springer (2006)
10. Bouyer, P., Dufourd, C., Fleury, E., Petit, A.: Updatable timed automata. Theoretical Computer Science 321(2-3), 291–345 (2004)
11. Boyer, M., Roux, O.H.: On the compared expressiveness of arc, place and transition time Petri nets. Fundamenta Informaticae 88(3), 225–249 (2008)
12. Bozga, M., Daws, C., Maler, O., Olivero, A., Tripakis, S., Yovine, S.: Kronos: a model-checking tool for real-time systems. In: CAV. LNCS, vol. 1427, pp. 546–550 (1998)
13. Cassez, F., Chatain, T., Jard, C.: Symbolic unfoldings for networks of timed automata. In: ATVA. LNCS, vol. 4218, pp. 307–321. Springer (2006)
14. Cassez, F., Roux, O.: Structural translation from time Petri nets to timed automata. Journal of Systems and Software (2006)
15. Cerans, K., Godskesen, J.C., Larsen, K.G.: Timed modal specification - theory and tools. In: CAV. LNCS, vol. 697, pp. 253–267. Springer (1993)
16. Dima, C.: Positive and negative results on the decidability of the model-checking problem for an epistemic extension of timed ctl. In: TIME. pp. 29–36. IEEE Computer Society (2009)
17. Dima, C., Lanotte, R.: Distributed time-asynchronous automata. In: ICTAC. pp. 185–200. Springer-Verlag (2007)
18. van Glabbeek, R.J., Goltz, U.: Refinement of actions and equivalence notions for concurrent systems. Acta Inf. 37(4/5), 229–327 (2001)
19. Halpern, J.Y., Fagin, R., Moses, Y., Vardi, M.Y.: Reasoning About Knowledge. MIT Press (1995)
20. Larsen, K.G., Pettersson, P., Yi, W.: Uppaal in a nutshell. International Journal on Software Tools for Technology Transfer (STTT) 1(1-2), 134–152 (1997)
21. Lugiez, D., Niebert, P., Zennou, S.: A partial order semantics approach to the clock explosion problem of timed automata. Theor. Comput. Sci. 345(1), 27–59 (2005)
22. Merlin, P.M., Farber, D.J.: Recoverability of communication protocols – implications of a theorical study. IEEE Transactions on Communications 24 (1976)
23. Minea, M.: Partial order reduction for model checking of timed automata. In: CONCUR. LNCS, vol. 1664, pp. 431–446. Springer (1999)
24. Reif, J.: The complexity of two-player games of incomplete information. Jour. Computer and Systems Sciences 29, 274–301 (1984)
25. Srba, J.: Comparing the expressiveness of timed automata and timed extensions of Petri nets. In: FORMATS. LNCS, vol. 5215, pp. 15–32. Springer (2008)
26. Wozna, B., Lomuscio, A.: A logic for knowledge, correctness, and real time. In: CLIMA. LNCS, vol. 3487, pp. 1–15. Springer (2004)

16

A Omitted Proofs

First we present two propositions that will help us prove Lemma 1.

Proposition 1.

1. *For any reachable state* (S_1, s_2) *of* $\mathrm{TTS}_{A_1}(A_2)$,
   $s_1 \in S_1 \implies (s_1, (S_1, s_2))$ *is a reachable state of* $\mathrm{TTS}_{Q_1}(A_1) \otimes \mathrm{TTS}_{A_1}(A_2)$
2. *noRestriction*$_{A_1}(A_2)$ *iff*
   for any reachable state (S_1, s_2) *of* $\mathrm{TTS}_{A_1}(A_2)$,
   $s_1 \in S_1 \iff (s_1, (S_1, s_2))$ *is a reachable state of* $\mathrm{TTS}_{Q_1}(A_1) \otimes \mathrm{TTS}_{A_1}(A_2)$

Proof. (1) For any reachable state (S_1, s_2), let us denote by $P(S_1, s_2)$ the fact that for any $s_1 \in S_1$, $(s_1, (S_1, s_2))$ is reachable in $\mathrm{TTS}_{Q_1}(A_1) \otimes \mathrm{TTS}_{A_1}(A_2)$. We give a recursive proof. First, the initial state (S_1^0, s_2^0) satisfies $P(S_1^0, s_2^0)$ because for any $s_1 \in S_1^0 = \mathrm{UR}(s_1^0)$, $\exists w \in \mathrm{TW}(\Sigma_1 \setminus \Sigma_2^{\mathscr{O}}, 0) : s_1^0 \xrightarrow{w}_1 s_1$ and hence $(s_1^0, (S_1^0, s_2^0)) \xrightarrow{w} (s_1, (S_1^0, s_2^0))$. Then, assume some reachable state (S_1, s_2) of $\mathrm{TTS}_{A_1}(A_2)$ satisfies $P(S_1, s_2)$ and show that any state (S_1', s_2') reachable in one step from (S_1, s_2) also satisfies $P(S_1', s_2')$. There can be three kinds of steps from (S_1, s_2) in $\mathrm{TTS}_{A_1}(A_2)$.

1. If for some $a \in \Sigma_2 \setminus \mathbb{S}$, $(S_1, s_2) \xrightarrow{a} (S_1', s_2')$, then for any $s_1' \in S_1' \subseteq S_1$, $(s_1', (S_1, s_2)) \xrightarrow{a} (s_1', (S_1', s_2'))$, i.e. $P(S_1', s_2')$ holds.
2. If for some $(a, s_1') \in \mathbb{S} \times Q_1$, $(S_1, s_2) \xrightarrow{a, s_1'} (S_1', s_2')$, then $S_1' = \mathrm{UR}(s_1')$, and for some $s_1 \in S_1$, $(s_1, (S_1, s_2)) \xrightarrow{a, s_1'} (s_1', (S_1', s_2'))$. By the same reasoning as for (S_1^0, s_2^0), for any $s_1'' \in S_1' = \mathrm{UR}(s_1')$, $\exists w \in \mathrm{TW}(\Sigma_1 \setminus \Sigma_2^{\mathscr{O}}, 0) : (s_1', (S_1', s_2')) \xrightarrow{w} (s_1'', (S_1', s_2'))$. Hence $P(S_1', s_2')$ holds.
3. If for some $d \in \mathbb{R}_{\geq 0}$, $(S_1, s_2) \xrightarrow{d} (S_1', s_2')$, then $\exists d_1 \leq d : (S_1, s_2) \xrightarrow{d_1} (S_1^1, s_2^1) \wedge$ $\exists s_1^1 \in S_1^1, s_1 \in S_1 : (s_1, s_2) \overset{d_1}{\Rightarrow} (s_1^1, s_2^1)$, that is $(s_1^1, (S_1^1, s_2^1))$ is reachable, and by time-determinism, $(S_1^1, s_2^1) \xrightarrow{d - d_1} (S_1', s_2')$.

For the third case, take d_1 small enough (but strictly positive) so that $S_1^1 = \{s_1' \mid \exists s_1 \in S_1 : (s_1, s_2) \overset{d_1}{\Rightarrow} (s_1^1, s_2^1) \wedge s_1' \in \mathrm{UR}(s_1^1)\}$. That is, after some local actions that take no time, A_1 is able to perform a delay d_1 during which no local action is enabled (such d_1 exists because of the non-zenoness assumption). With such d_1, any state $s_1' \in S_1^1$ is such that $s_1' \in \mathrm{UR}(s_1^1)$ for some s_1^1 so that $(s_1^1, (S_1^1, s_2^1))$ is reachable. Therefore, $\exists w \in \mathrm{TW}(\Sigma_1 \setminus \Sigma_2^{\mathscr{O}}, 0) : (s_1^1, (S_1^1, s_2^1)) \xrightarrow{w} (s_1', (S_1^1, s_2^1))$ and hence $P(S_1^1, s_2^1)$ holds.

Since A_1 is not Zeno, any delay in $\mathrm{TTS}_{A_1}(A_2)$ can be cut into a finite number of such smaller global delays. Hence, for any (S_1, s_2) that satisfies $P(S_1, s_2)$, for any $d \in \mathbb{R}_{\geq 0}$ such that $(S_1, s_2) \xrightarrow{d} (S_1', s_2')$, $P(S_1', s_2')$ holds.

(2, $\Rightarrow$) (1) already gives that $\forall s_1 \in S_1$, $(s_1, (S_1, s_2))$ is a reachable state. So it remains to prove that, when *noRestriction*$_{A_1}(A_2)$, if $(s_1, (S_1, s_2))$ is a reachable state, then $s_1 \in S_1$. We say that a reachable state $s = (s_1, (S_1, s_2))$ satisfies $P(s)$ iff $s_1 \in S_1$.

Assume $noRestriction_{A_1}(A_2)$ and $s = (s_1, (S_1, s_2)$ is a reachable state that satisfies $P(s)$. Then, any state s' reachable in one step from s by some local action or delay $a \in (\Sigma_1 \cup \Sigma_2) \setminus \mathbb{S} \cup \mathbb{R}_{\geq 0}$ or by some synchronization $(a, s'_1) \in \mathbb{S} \times Q_1$ matches one of the following cases.

- if $a \in \Sigma_1 \setminus \Sigma_2^{\not\sigma}$, then $s' = (s'_1, (S_1, s_2))$ such that $s'_1 \in \mathrm{UR}(s_1) \subseteq S_1$ (by construction, $s_1 \in S_1 \implies \mathrm{UR}(s_1) \subseteq S_1$),
- if $a \in \Sigma_2 \setminus \Sigma_1$, then $s' = (s_1, (S_1, s'_2))$,
- if $a \in \mathbb{R}_{\geq 0}$, then $s' = (s'_1, (S'_1, s'_2))$, where s'_1 such that $(s_1, s_2) \stackrel{a}{\Rightarrow} (s'_1, s'_2)$ is in $S'_1 = \{q'_1 \mid \exists q_1 \in S_1, w \in \mathrm{TW}(\Sigma_1 \setminus \Sigma_2^{\not\sigma}, a) : (q_1, s_2) \stackrel{w}{\Rightarrow} (q'_1, s'_2)\}$,
- if $(a, s'_1) \in (\mathbb{S} \times Q_1)$, then $s' = (s'_1, (\mathrm{UR}(s'_1), s'_2))$.

Therefore, any state s' reached in one step from s also satisfies $P(s')$, and recursively, since the initial state $s_0 = (s_1^0, (\mathrm{UR}(s_1^0), s_2^0))$ satisfies $P(s_0)$, any reachable state s of $\mathrm{TTS}_{Q_1}(A_1) \otimes \mathrm{TTS}_{A_1}(A_2)$ satisfies $P(s)$.

(2, $\Leftarrow$) By contradiction, assume there is a restriction in state (S_1, s_2) for local delay or action $a \in (\Sigma_2 \setminus \Sigma_1) \cup \mathbb{R}_{\geq 0}$ i.e. a is possible from some state (s'_1, s_2) but not from another state (s_1, s_2) such that $s'_1, s_1 \in S_1$. Then, after performing a from $(s_1, (S_1, s_2))$, that is reachable according to Proposition **??**, we reach state $(s_1, (S'_1, s'_2))$ such that $s_1 \notin S'_1$. □

Proposition 2. *If $noRestriction_{A_1}(A_2)$ then, for all timed word w over $(\Sigma_2 \setminus \mathbb{S}) \cup (\mathbb{S} \times Q_1)$, there exists at most one S_1 such that, for some s_2, $(S_1^0, s_2^0) \xrightarrow{w} (S_1, s_2)$ in $\mathrm{TTS}_{A_1}(A_2)$ (i.e. S_1 is uniquely determined by w, whatever the structure of A_2).*

Proof. Assume $noRestriction_{A_1}(A_2)$, we show that, for any (S_1^1, s_2^1) reachable in $\mathrm{TTS}_{A_1}(A_2)$, for any action or delay in $(\Sigma_2 \setminus \mathbb{S}) \cup (\mathbb{S} \times Q_1) \cup \mathbb{R}_{\geq 0}$, there is at most one S_1 such that, for some s_2, (S_1, s_2) is a successor of (S_1^1, s_2^1) by this action.

Indeed, by construction, and since there is no restriction,

- any successor of (S_1^1, s_2^1) by a local action is of the form (S_1^1, s'_2),
- any successor of (S_1^1, s_2^1) by a synchronization (a, s'_1) is of the form $(\mathrm{UR}(s'_1), s'_2)$,
- any successor of (S_1^1, s_2^1) by a delay d is of the form (S_1, s'_2) with $S_1 = \{s'_1 \mid \exists w \in \mathrm{TW}(\Sigma_1 \setminus \Sigma_2^{\not\sigma}, d), s_1 \in S_1^1 : s_1 \xrightarrow{w}_1 s'_1\}$.

Therefore, for any possible action or delay, S_1 does not depend on the state of A_2, and is uniquely determined by this action or delay.

Since (S_1^0, s_2^0) is unique, for any timed word w over $(\Sigma_2 \setminus \mathbb{S}) \cup (\mathbb{S} \times Q_1)$, either w does not describe a valid path in $\mathrm{TTS}_{A_1}(A_2)$, or there exists a unique S_1 such that for some s_2, $(S_1^0, s_2^0) \xrightarrow{w} (S_1, s_2)$ in $\mathrm{TTS}_{A_1}(A_2)$. □

Proof (Lemma 1). Assume $noRestriction_{A_1}(A_2)$, and define relation $\mathcal{R}$ as $(s_1, (S_1, s_2)) \ \mathcal{R} \ (s'_1, s'_2) \stackrel{def}{\iff} s_1 = s'_1 \wedge s_2 = s'_2$, for any reachable states $(s_1, (S_1, s_2))$ of $\mathrm{TTS}_{Q_1}(A_1) \otimes \mathrm{TTS}_{A_1}(A_2)$ and (s'_1, s'_2) of $\mathrm{TTS}_{Q_1}(A_1 \parallel A_2)$. By Proposition **??**, since $(s_1, (S_1, s_2))$ is reachable, $s_1 \in S_1$. We show that $\mathcal{R}$ is a strong timed bisimulation.

First, the initial states are $\mathcal{R}$-related: $(s_1^0,(S_1^0,s_2^0))\ \mathcal{R}\ (s_1^0,s_2^0)$. Then, if $(s_1,(S_1,s_2))\ \mathcal{R}\ (s_1',s_2')$, four kinds of steps are possible:

- if for some $a \in \Sigma_1 \setminus \Sigma_2^{\not\varepsilon}$, $(s_1,(S_1,s_2)) \xrightarrow{a} (s_1',(S_1,s_2))$, then $(s_1,s_2) \stackrel{a}{\Rightarrow} (s_1',s_2)$ and $(s_1',(S_1,s_2))\ \mathcal{R}\ (s_1',s_2)$, and conversely.
- if for some $a \in \Sigma_2 \setminus \Sigma_1$, $(s_1,(S_1,s_2)) \xrightarrow{a} (s_1,(S_1,s_2'))$, then, $\forall s_{11} \in S_1$, $(s_{11},s_2) \stackrel{a}{\Rightarrow} (s_{11},s_2')$ (because $noRestriction_{A_1}(A_2)$), and in particular, $(s_1,s_2) \stackrel{a}{\Rightarrow} (s_1,s_2')$ and $(s_1,(S_1,s_2'))\ \mathcal{R}\ (s_1,s_2')$, and conversely.
- if for some $(a,s_1') \in \mathbb{S}\times Q_1$, $(s_1,(S_1,s_2)) \xrightarrow{a,s_1'} (s_1',(S_1',s_2'))$, then $(s_1,s_2) \stackrel{a,s_1'}{\Longrightarrow} (s_1',s_2')$ and $(s_1',(S_1',s_2'))\ \mathcal{R}\ (s_1',s_2')$, and conversely.
- if for some $d \in \mathbb{R}_{\geq 0}$, $(s_1,(S_1,s_2)) \xrightarrow{d} (s_1',(S_1',s_2'))$, then $(s_1,s_2) \stackrel{d}{\Rightarrow} (s_1',s_2')$ (because $noRestriction_{A_1}(A_2)$), and $(s_1',(S_1',s_2'))\ \mathcal{R}\ (s_1',s_2')$, and conversely.

Now assume A_2 is deterministic. Let relation $\mathcal{R}$ be a strong timed bisimulation between $\mathrm{TTS}_{Q_1}(A_1) \otimes \mathrm{TTS}_{A_1}(A_2)$ and $\mathrm{TTS}_{Q_1}(A_1 \parallel A_2)$.

By contradiction, assume there is a restriction in $\mathrm{TTS}_{A_1}(A_2)$. Then there is a reachable state (S_1,s_2) of $\mathrm{TTS}_{A_1}(A_2)$, and a local delay or action $a \in (\Sigma_2 \setminus \Sigma_1) \cup \mathbb{R}_{\geq 0}$ such that, for some $s_1,s_1' \in S_1$, (s_1,s_2) enables a in $\mathrm{TTS}_{Q_1}(A_1 \parallel A_2)$, whereas (s_1',s_2) does not.

By definition of a bisimulation, there also exist two states $(p_1,(P_1,p_2))$ and $(p_1',(P_1',p_2')$ such that $(p_1,(P_1,p_2))\ \mathcal{R}\ (s_1,s_2)$ and $(p_1',(P_1',p_2'))\ \mathcal{R}\ (s_1',s_2)$. That is, in particular, $(p_1',(P_1',p_2'))$ does not enable a. Moreover, these states can be chosen so that they are reached by the same timed word over $(\Sigma_2 \setminus \mathbb{S}) \cup (\mathbb{S}\times Q_1)$, and since A_2 is deterministic, $p_2 = p_2' = s_2$.

Now, we can assume that (S_1,s_2) is chosen so that it is the first state with a restriction along an initial path. Then, the paths to (P_1,s_2) and (P_1',s_2) generate the same timed word over $(\Sigma_2 \setminus \mathbb{S}) \cup (\mathbb{S}\times Q_1)$, and by Proposition **??**, $P_1 = P_1' = S_1$.

Therefore, we have shown the existence of a state $(p_1',(S_1,s_2))$ in $\mathrm{TTS}_{Q_1}(A_1) \otimes \mathrm{TTS}_{A_1}(A_2)$ that does not enable a, which means that (S_1,s_2) does not enable a in $\mathrm{TTS}_{A_1}(A_2)$. This contradicts the fact that there exists $s_1 \in S_1$ such that (s_1,s_2) enables a. □

Proof (Lemma 2). When $noRestriction_{A_1}(A_2)$ holds, then by Lemma 1, $\mathrm{TTS}_{Q_1}(A_1) \otimes \mathrm{TTS}_{A_1}(A_2) \approxeq \mathrm{TTS}_{Q_1}(A_1 \parallel A_2)$. So for any NTA $A_1' \parallel A_2'$ satisfying items 2 and 3 of Definition 6, we have $\psi(\mathrm{TTS}_{Q_1'}(A_1')) \otimes \psi(\mathrm{TTS}_{A_1'}(A_2')) \approx \mathrm{TTS}_{Q_1}(A_1 \parallel A_2)$. It remains to show that $\psi(\mathrm{TTS}_{Q_1'}(A_1' \parallel A_2')) \approxeq \psi(\mathrm{TTS}_{Q_1'}(A_1')) \otimes \psi(\mathrm{TTS}_{A_1'}(A_2'))$. Remark that applying ψ to the labels before doing the product, allows more synchronizations than applying ψ on the TTS of the system since ψ may merge different labels. We show that, in our case, the two resulting TTS are bisimilar anyway.

For this, let $\mathcal{R}_1$ be a bisimulation relation between $\psi(\mathrm{TTS}_{Q_1'}(A_1'))$ and $\mathrm{TTS}_{Q_1}(A_1)$, and $\mathcal{R}_2$ be a bisimulation relation between $\psi(\mathrm{TTS}_{A_1'}(A_2'))$ and $\mathrm{TTS}_{A_1}(A_2)$. We will build inductively a bisimulation $\mathcal{R}$ between $\psi(\mathrm{TTS}_{Q_1'}(A_1' \parallel A_2'))$ and $\psi(\mathrm{TTS}_{Q_1'}(A_1')) \otimes \psi(\mathrm{TTS}_{A_1'}(A_2'))$ such that for any (q_1,q_2) and (r_1,r_2) such that $(q_1,q_2)\ \mathcal{R}\ (r_1,r_2)$, there exists a state s_1 of $\mathrm{TTS}_{Q_1}(A_1)$ and a state

s_2 of $\mathrm{TTS}_{A_1}(A_2)$ such that $q_1 \mathcal{R}_1 s_1$ and $r_1 \mathcal{R}_1 s_1$ and $q_2 \mathcal{R}_2 s_2$ and $r_2 \mathcal{R}_2 s_2$. The inductive definition of $\mathcal{R}$ is as follows. The initial states (which are the same in both sides) are in relation; $\mathcal{R}$ is preserved by delays; $\mathcal{R}$ is preserved by playing local actions. The key is the treatment of synchronizations: when $(q_1, q_2)\ \mathcal{R}\ (r_1, r_2)$ and $q_1 \xrightarrow{a_1} q_1'$ in $\mathrm{TTS}_{Q_1}(A_1)$ and $q_2 \xrightarrow{a_2} q_2'$ in $\mathrm{TTS}_{A_1}(A_2)$ with $\psi(a_1) = \psi(a_2) = a$, then the existence of the s_1 and s_2 mentioned earlier ensures that there exists a state (r_1', r_2') in $\psi(\mathrm{TTS}_{Q_1'}(A_1' \parallel A_2'))$ such that $(r_1, r_2) \xrightarrow{a} (r_1', r_2')$, and we set $(q_1', q_2')\ \mathcal{R}\ (r_1', r_2')$ for any such (r_1', r_2'). □

Proof (Theorem 1, necessary condition when A_2 is deterministic). Like in the proof of Lemma 2, we show that for any NTA $A_1' \parallel A_2'$ satisfying items 2 and 3 of Definition 6, $\psi(\mathrm{TTS}_{Q_1'}(A_1' \parallel A_2')) \approx \mathrm{TTS}_{Q_1}(A_1) \otimes \mathrm{TTS}_{A_1}(A_2)$. But, by Lemma 1, when A_2 is deterministic and $\mathrm{TTS}_{A_1}(A_2)$ has restrictions, $\mathrm{TTS}_{Q_1}(A_1) \otimes \mathrm{TTS}_{A_1}(A_2)$ is not timed bisimilar to $\mathrm{TTS}_{Q_1}(A_1 \parallel A_2)$ (not even weakly timed bisimilar since there are no ε-transitions). Hence any NTA $A_1' \parallel A_2'$ satisfying items 2 and 3 of Definition 6, does not satisfy item 1. □

To prove the other part of Theorem 1, we will use Lemma 3. We first prove the correspondence between a state of $\mathcal{S}_{mod}$ and two states of $\mathrm{TTS}(A_1 \parallel A_2)$ that are merged into the same state of $\mathrm{TTS}_{A_1}(A_2)$. This is stated in the following proposition, that will be used to prove Lemma 3. A state of $\mathcal{S}_{mod}$ is denoted as $(s_1, s_{1,2}, s_2) = \big((\ell_1, v_{|X_1}), (\ell_{1,2}, v_{|X_1'}), (\ell_2, v_{|X_2 \setminus X_1})\big)$. For a given state of $A_{1,2}$, $s_{1,2} = (\ell_{1,2}, v_{|X_1'})$, we denote by $s_{1,2}'$ the state $(\ell_{1,2}, v')$, where $v' : X_1 \to \mathbb{R}_{\geq 0}$ is defined as: for any $x \in X_1$, $v'(x) = v(x')$ (i.e. $s_{1,2}'$ is a state of A_1). Reciprocally, for a given state of A_1, $s_{1,2}' = (\ell_{1,2}, v')$, $s_{1,2}$ denotes the state $(\ell_{1,2}, v)$, where $v : X_1' \to \mathbb{R}_{\geq 0}$ is defined as: for any $x' \in X_1'$, $v(x') = v'(x)$.

Proposition 3. *Let $(s_1, s_{1,2}, s_2)$ be a state of $\mathcal{S}_{mod}$. If along one path that leads to $(s_1, s_{1,2}, s_2)$ no edge leading to ☹ is enabled, then there exists S_1 such that (S_1, s_2) is a reachable state of $\mathrm{TTS}_{A_1}(A_2)$ and s_1 and $s_{1,2}'$ are both in S_1.*

Conversely, let (S_1, s_2) be a reachable state of $\mathrm{TTS}_{A_1}(A_2)$, and s_1 and $s_{1,2}'$ be some states in S_1. Then $(s_1, s_{1,2}, s_2)$ is a state of $\mathcal{S}_{mod}$.

*Proof (Proposition **??**).* Let $(s_1, s_{1,2}, s_2)$ be a reachable state of $\mathcal{S}_{mod}$, such that there is a path ρ from the initial state $(s_1^0, s_{1,2}^0, s_2^0)$ to $(s_1, s_{1,2}, s_2)$ that does not enable any edges leading to ☹ (except maybe from $(s_1, s_{1,2}, s_2)$). We give a recursive proof. First, for the initial state $(s_1^0, s_{1,2}^0, s_2^0)$ of $\mathcal{S}_{mod}$, s_1^0 and $s_{1,2}^{0\prime}$ are bot in S_1^0 such that (S_1^0, s_2^0) is the initial state of $\mathrm{TTS}_{A_1}(A_2)$. Now, assume this is true for some $(p_1, p_{1,2}, p_2)$ visited along ρ. That is, there exists P_1 such that (P_1, p_2) is reachable and $p_1, p_{1,2}' \in P_1$. Then, the next state s' visited along ρ is reached after one of the following steps:

- local action in A_1': $s' = (q_1, p_{1,2}, p_2)$ such that $q_1 \in \mathrm{UR}(p_1) \subseteq P_1$,
- local action in $A_{1,2}$: $s' = (p_1, q_{1,2}, p_2)$ such that $q_{1,2}' \in \mathrm{UR}(p_{1,2}') \subseteq P_1$,
- local action in A_2: $s' = (p_1, p_{1,2}, q_2)$ such that there exists S_1' such that (S_1', q_2) is reachable from (P_1, q_2) by the same action, and, since no edge leading to ☹ is enabled, both (p_1, p_2) and $(p_{1,2}', p_2)$ enable this step in $\mathrm{TTS}(A_1 \parallel A_2)$. Therefore, $p_1, p_{1,2}' \in S_1'$.

– synchronization: $s' = (q_1, q_{1,2}, q_2)$ such that there exists $S'_1 = \mathrm{UR}(q_1)$ such that (S'_1, q_2) is reachable from (P_1, q_2) by the same action, and $q_1 = q'_{1,2} \in S'_1$.

By recursion, $(s_1, s_{1,2}, s_2)$ also satisfies the property, that is, there exists S_1 such that (S_1, s_2) is reachable and $s_1, s'_{1,2} \in S_1$.

Conversely, let denote by $P(S_1, s_2)$ the fact that for any reachable state (S_1, s_2) of $\mathrm{TTS}_{A_1}(A_2)$, for any states $s_1, s'_{1,2} \in S_1$, $(s_1, s_{1,2}, s_2)$ is a reachable state of $\mathcal{S}_{mod}$. First, for any $s_1, s'_{1,2} \in S^0_1 = \mathrm{UR}(s^0_1)$, $(s_1, s_{1,2}, s^0_2)$ is a reachable state, because by construction, $A_{1,2}$ can only mimic (as long as there is no synchronization) one possible behavior of A_1 to reach $s_{1,2}$ from s^0_1, therefore $P(S^0_1, s^0_2)$ holds. Assume that for some reachable state (S_1, s_2) $P(S_1, s_2)$ holds. Then any state reachable in one step from (S_1, s_2) is reached by one of the following steps.

– If for some $a \in \Sigma_2 \setminus \mathbb{S}$, $(S_1, s_2) \xrightarrow{a} (S'_1, s'_2)$, then for any $s_1, s'_{1,2} \in S'_1 \subseteq S_1$, $(s_1, s'_{1,2}, s_2) \xrightarrow{a} (s_1, s'_{1,2}, s'_2)$, i.e. $P(S'_1, s'_2)$ holds.
– If for some $(a, s'_1) \in \mathbb{S} \times Q_1$, $(S_1, s_2) \xrightarrow{a, s'_1} (S'_1, s'_2)$, then $S'_1 = \mathrm{UR}(s'_1)$, and for any $s_1, s'_{1,2} \in S'_1$, $(s_1, s_{1,2}, s'_2)$ can be reached from some $(p_1, p_{1,2}, s_2)$ such that $p_1, p'_{1,2} \in S_1$. Indeed, in $\mathcal{S}_{mod}$, synchronization $((a, \ell'_1), s'_1)$ resets $A_{1,2}$ in the same state as A_1 and then A_1 performs some local actions while $A_{1,2}$ also performs some local actions mimicking one possible behavior of A_1 (that is why $s'_{1,2} \in S'_1$). Hence $P(S'_1, s'_2)$ holds.
– If for some $d \in \mathbb{R}_{\geq 0}$, $(S_1, s_2) \xrightarrow{d} (S'_1, s'_2)$, then we use the same reasoning as for a synchronization. Since $A_{1,2}$ is built so that it mimics any possible behavior of A_1 between synchronizations, any state $s'_{1,2} \in S'_1$ reachable by A_1 during this delay corresponds to a state $s_{1,2}$ reachable by $A_{1,2}$. Hence $P(S'_1, s'_2)$ also holds.

By recursion, $P(S_1, s_2)$ holds for any reachable state (S_1, s_2). □

Proof (Lemma 3). Assume ☹ is not reachable in $\mathcal{S}_{mod}$. From Proposition **??**, we know that for any state (S_1, s_2) of $\mathrm{TTS}_{A_1}(A_2)$, for any s_1, $s'_{1,2}$ in S_1, there is a corresponding state $s = \big((\ell_1, v_{|X_1}), (\ell_{1,2}, v_{|X'_1}), (\ell_2, v_{|X_2 \setminus X_1})\big) = (s_1, s_{1,2}, s_2)$ of $\mathcal{S}_{mod}$. Moreover, for any such s, if there is an outgoing edge towards ☹ from ℓ_2, then this edge is never enabled. That is, for any time constraint γ read in ℓ_2 in the original system $\mathcal{S}$ (invariant of ℓ_2 or guard of an outgoing edge with a local action), $v_{|X_2 \cup X_1} \models \gamma \iff v_{|(X_2 \setminus X_1) \cup X'_1} \models \gamma'$. Hence for any enabled step from (S_1, s_2), s_1 and $s'_{1,2}$ are in the same restriction. Therefore, $\mathit{noRestriction}_{A_1}(A_2)$.

Assume ☹ is reachable in $\mathcal{S}_{mod}$. From Proposition **??**, we know that for any state $s = \big((\ell_1, v_{|X_1}), (\ell_{1,2}, v_{|X'_1}), (\ell_2, v_{|X_2 \setminus X_1})\big) = (s_1, s_{1,2}, s_2)$ of $\mathcal{S}_{mod}$, reached after a path that does not enable edges leading to ☹ (except maybe from this last state), there is a corresponding state (S_1, s_2) of $\mathrm{TTS}_{A_1}(A_2)$ such that s_1 and $s'_{1,2}$ are both in S_1. If ☹ can be reached, then consider a path that reach ☹ and such no edge leading to ☹ was enabled before along the path. The last state s of $\mathcal{S}_{mod}$ visited before ☹ is such that for some time constraint γ evaluated at s from ℓ_2, $v_{|X_2 \cup X_1} \models \gamma$ and $v_{|(X_2 \setminus X_1) \cup X'_1} \not\models \gamma'$ (or conversely). Therefore, a

local action or local delay is possible from (s_1, s_2) and not from $(s'_{1,2}, s_2)$. Hence (S_1, s_2) is a state with a restriction. □

Proof (Theorem 1, direct part, when no urgent synchronization in A_1). Assume $noRestriction_{A_1}(A_2)$. We consider $A'_2 = A_{1,2} \otimes A'_{2,mod}$ where $A'_{2,mod}$ is $A_{2,mod}$ without ☹ (that is never reached according to Lemma 3) and its ingoing edges. Therefore, $A'_{2,mod}$ does not read X_1 and neither does $A'_2 = A_{1,2} \otimes A'_{2,mod}$. Below we show that A'_2 is a suitable candidate because $\psi(\mathrm{TTS}_{A'_1}(A'_2)) \approx \mathrm{TTS}_{A_1}(A_2)$ ($\psi(\mathrm{TTS}_{Q'_1}(A'_2)) \approx \mathrm{TTS}_{Q_1}(A_1)$ obviously holds).

Let $\mathcal{R}$ be the relation such that for any reachable state (S_1, s_2) of $\mathrm{TTS}_{A_1}(A_2)$, and any reachable state (S'_1, s'_2) of $\psi(\mathrm{TTS}_{A'_1}(A'_2))$,

$$(S_1, s_2)\ \mathcal{R}\ (S'_1, s'_2) \overset{def}{\iff} \begin{cases} s_2 = (\ell_2, v_2) \text{ and } s'_2 = ((\ell_{1,2}, \ell_2), v'_2) \text{ s.t.} \\ \forall x \in X_2 \setminus X_1, v_2(x) = v'_2(x) \\ S_1 = S'_1 \end{cases}$$

i.e. A_2 and $A'_{2,mod}$ are both in ℓ_2 and their local clocks have the same value, and A_1 and A'_1 are in indistinguishable states (states merged in a same contextual state S_1). Obviously, the initial states, (S_1^0, s_2^0) and $(S_1^0, s_2^{0\prime})$, are $\mathcal{R}$-related. Since there is no marked state in $\mathrm{TTS}_{A_1}(A_2)$ (resp. in $\mathrm{TTS}_{A'_1}(A'_2)$), for any state $s = (S_1, s_2)$ (resp. $s' = (S'_1, s'_2)$) of this TTS, all time constraints read by automaton 2 in ℓ_2 (invariant of ℓ_2 and guards of the outgoing edges) have the same truth value for all the states (s_1, s_2) such that $s_1 \in S_1$ (resp. $s_1 \in S'_1$). In the sequel, we say that valuation V of s (resp. V' of s') satisfies constraint g, when the valuations of all states (s_1, s_2) in s (resp. in s') satisfy g. Assume now that for some reachable states (S_1, s_2) and (S'_1, s'_2), $(S_1, s_2)\ \mathcal{R}\ (S'_1, s'_2)$.

Local Action. If $a \in \Sigma_2 \setminus \Sigma_1$ is enabled from (S_1, s_2), then, there is an associated edge in A_2, $\ell_2 \xrightarrow{g,a,r} p_2$ such that guard g is satisfied by V. Let g' be the guard on the corresponding outgoing edge $(\ell_{1,2}, \ell_2) \xrightarrow{g',a,r} (\ell_{1,2}, p_2)$ in A'_2. g uses clocks in X_2, and by construction, g' has the same form but with clocks in $(X_2 \setminus X_1) \uplus X'_1$. $(S_1, s_2)\ \mathcal{R}\ (S'_1, s'_2)$ says that v_2 and v'_2 coincide on $X_2 \setminus X_1$, and since ☹ is never reached in $\mathcal{S}_{mod}$, V satisfies the constraints of g on X_1 iff V' satisfies the constraints of g' on X'_1. That is, $V \models g \iff V' \models g'$. Therefore A'_2 can also perform a from (S_1, s'_2) and the states reached in both systems are $\mathcal{R}$-related: $(S_1, q_2)\ \mathcal{R}\ (S_1, q'_2)$, because $q_2 = (p_2, v_2[r])$ and $q'_2 = ((\ell_{1,2}, p_2), v'_2[r])$. This also holds reciprocally.

Synchronization. Assume for some $(a, s'_1) \in \mathbb{S} \times Q_1$, $(S_1, s_2) \xrightarrow{a,s'_1} (S'_1, q_2)$. That is, there is an edge $\ell_2 \xrightarrow{g_2,a,r_2} p_2$ in A_2 such that $v_2 \models g_2$ and $q_2 = (p_2, v_2[r_2])$ and, for some $(\ell_1, v_1) \in S_1$, an edge $\ell_1 \xrightarrow{g_1,a,r_1} p_1$ in A_1 such that $v_1 \models g_1$ and $s'_1 = (p_1, v_1[r_1]) \in S'_1$. Hence, synchronization $((a, p_1), s'_1)$ is also enabled from state (S_1, s'_2) because $A_{2,mod}$ is in the same location as A_2, and has the same clock values over $X_2 \setminus X_1$, and A'_1 is also in some state of S_1, therefore, there is also the same state $(\ell_1, v_1) \in S_1$ which enables (a, p_1). We do not consider

$A_{1,2}$ because it is always ready to synchronize. Moreover, the state reached in $\psi(\mathrm{TTS}_{A'_1}(A'_2))$ after this synchronization is (S'_1, q'_2) such that $(S'_1, q_2)\ \mathcal{R}\ (S'_1, q'_2)$, because $q_2 = (p_2, v_2[r_2])$ and $q'_2 = \big((p_{1,2}, p_2), (v'_2[r_2])[c]\big)$ where c denotes the copy of the clocks of X_1 into their associated clocks of X'_1 and therefore c modifies only clocks that we do not consider in relation $\mathcal{R}$, and $r_2 \subseteq C_2 \subseteq (X_2 \setminus X_1)$ resets the same clocks in both systems. And reciprocally.

Local Delay. Assume for some $d \in \mathbb{R}_{\geq 0}$, $(S_1, s_2) \xrightarrow{d} (S'_1, q_2)$. Then, $V + d \models \mathit{Inv}_2(\ell_2)$, and since ☹ is never reached in $\mathcal{S}_{mod}$, $V + d \models \mathit{Inv}_2(\ell_2) \iff V' + d \models \mathit{Inv}'_2(\ell_2)$. That is, the same delay is enabled from (S_1, s'_2) while $A_{1,2}$ may perform some local steps: $(S_1, s'_2)(\xrightarrow{g_0,\varepsilon,r_0})^* \xrightarrow{d_0} (\xrightarrow{g_n,\varepsilon,r_n})^* \dots \xrightarrow{d_n} (S''_1, q'_2)$, where $\sum_{i=0}^{n} d_i = d$, g_i is a guards over X'_1 and r_i is a reset included in X'_1. This works because we assumed that A_1 has no urgent synchronization (and so does A'_1). Therefore, $A_{1,2}$ cannot force a synchronization.

Reciprocally, if we can perform a delay d from (S_1, s'_2), then $V' + d \models \mathit{Inv}'_2(\ell_2) \wedge \mathit{Inv}'_1(\ell_{1,2})$. And since $V + d \models \mathit{Inv}_2(\ell_2) \iff V' + d \models \mathit{Inv}'_2(\ell_2)$, we can perform the same delay from (S_1, s_2).

Moreover, we reach equivalent states in both systems. Indeed, A_2 and $A'_{2,mod}$ stay in the same location, the clocks in $X_2 \setminus X_1$ increase their value by d, and the set of states of A_1 and A'_1 becomes $S'_1 = S''_1 = \{s'_1 \mid \exists s_1 \in S_1, w \in \mathrm{TW}(\Sigma_1 \setminus \Sigma_2^{\not\sigma}, d) : (s_1, s_2) \overset{w}{\Rightarrow} (s'_1, q_2)\}$.

Therefore, $\mathcal{R}$ is a weak timed bisimulation and $\psi(\mathrm{TTS}_{A'_1}(A'_2)) \approx \mathrm{TTS}_{A_1}(A_2)$. Lastly, by Lemma 2, $\psi(\mathrm{TTS}_{Q'_1}(A'_1 \parallel A'_2)) \approx \mathrm{TTS}_{Q_1}(A_1 \parallel A_2)$ also, and A_2 does not need to read X_1. □

Exploitation de la Hiérarchies et des Symétries dans les systèmes répartis

F. Kordon[1]

LIP6, CNRS UMR 7606, Université P. & M. Curie – Paris 6
4, place Jussieu, F-75252 Paris Cedex 05, France
Fabrice.Kordon@lip6.fr

1 Résumé

Le *Model Cheking* est une technique de vérification formelle maintenant couramment acceptée pour l'analyse de systèmes complexes. Cependant, elle est difficile à mettre en œuvre dans les systèmes émergents actuels qui sot à la fois répartis sur un ensemble de machines (avec de la communication asynchrone) et composés hiérarchiquement (o pale désormais de "systèmes de systèmes").

Cet exposé présentera comment la structure régulière et hiérarchique de ces systèmes répartis modernes peut-être utilisé dans leur modélisation et exploitée pour combattre l'explosion combinatoire lors de leur vérification par *Model Checking*.

Plusieurs techniques seront présentées:

- la modélisation avec desmulti-ensembles (*bags*) dans les réseaux de Petri colorés [4],
- l'usage de représentations basées sur les symétries [1],
- l'usage de diagrammes de décision hiérarchiques [6].

L'exposé montrera également comment ces techniques peuvent être combinées et se "superposer", aboutissant ainsi à une meilleur performance des outils ainsi implémentés. Des résultats préliminaires seront présentés sur le prototype expérimental que nous avons développé: Crocodile [2].

Une autre expérimentation basée sur les réseaux de Petri temporels de Romeo [3] sera également abordée [5].

References

1. G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. Stochastic well-formed coloured nets for symmetric modelling applications. *IEEE Transactions on Computers*, 42(11):1343–1360, Nov. 1993.
2. M. Colange, S. Baarir, F. Kordon, and Y. Thierry-Mieg. Crocodile: a Symbolic/Symbolic tool for the analysis of Symmetric Nets with Bag. In *32nd International Conference on Petri Nets and Other Models of Concurrency (ICATPN 2011)*, volume 6709 of *Lecture Notes in Computer Science*, pages 338–347, Newcastle, UK, June 2011. Springer.
3. G. Gardey, D. Lime, M. Magnin, and O. H. Roux. Roméo: A tool for analyzing time Petri nets. In *17th International Conference on Computer Aided Verification (CAV'05)*, volume 3576 of *LNCS*. Springer, July 2005. http://romeo.rts-software.org/.

4. S. Haddad, F. Kordon, L. Petrucci, J. Pradat-Peyre, and L. Treves. Efficient state-based analysis by introducing bags in petri nets color domains. In *American Control Conference, 2009. ACC'09.*, pages 5018–5025. IEEE, 2009.
5. Y. Thierry-Mieg, B. Bérard, F. Kordon, D. Lime, and O. H. Roux. Compositional Analysis of Discrete Time Petri nets. In *1st workshop on Petri Nets Compositions (CompoNet 2011)*, volume 726, pages 17–31, Newcastle, UK, June 2011. CEUR.
6. Y. Thierry-Mieg, D. Poitrenaud, A. Hamez, and F. Kordon. Hierarchical Set Decision Diagrams and Regular Models. In *15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*, volume 5505 of *Lecture Notes in Computer Science*, pages 1–15, York, UK, March 2009. Springer.

Session du groupe de travail COSMAL

Composants Objets Services : Modèles, Architectures et Langages

Architectural Constraint Components

Chouki Tibermacine[1], Salah Sadou[2], Christophe Dony[1], and Luc Fabresse[3]

[1] LIRMM, CNRS and Montpellier-II University, France
[2] IRISA, Université de Bretagne Sud, Vannes, France
[3] École des Mines de Douai, France
{tibermacin,beniere,dony}@lirmm.fr,fabresse@ensm-douai.fr,sadou@univ-ubs.fr

1 Introduction and General Approach

When defining component-based software architecture descriptions, architecture constraints are generally intended for the validation of some specific architectural elements (components, in most cases). This limits their potential reuse with architectural elements of other architecture descriptions. In addition, this kind of documentation often includes some parts which can be used individually for documenting parts of design decisions [2–5, 7]. Unfortunately, there is no means to extract these parts, to make them parametrized entities that can be factorized and used in different reuse contexts. We defend thus in this paper the idea of defining blocks of constraints as customizable and reusable entities. We observed that in the literature, these issues have not been addressed deeply. To doing so, we turned towards component-based software development.

It is well known that some of the main "ilities" of component-based software engineering are reusability, composability and customizability. Reusability represents the ability for a given piece of software to be reused by developers. While shifting from design to implementation, developers are thus able to concretize a given design element by using a pre-developed software entity (development by reuse). Within the development process, developers are responsible for putting on shelves the produced software artifacts (components) during implementation for future system development (development for reuse). The second non-functional characteristic is inherent to component-based software development. Indeed, in this development paradigm, software building blocks that explicit their dependencies with their environment offer a connection capability of these different pieces of software to build a complex system. Customizability is the ability for a software to be changed by developers in order to adapt it to a given context. There are different methods to reach customizability. One of the most known techniques used for this purpose is parametrization. Indeed defining parameters in the signature of a given software entity allows the developer to customize the software entity behavior according to the passed arguments.

The goal of the work presented in this paper is to propose a way to build basic constraints as checkable entities embedded in a special kind of software components, that can be reused, assembled, composed into higher-level ones and customized using standard component-based techniques. The purpose is as well to put reusable constraint-component on shelves (design for reuse) and to produce new constraints by composition of existing ones (design by reuse) and then to simplify the expression and definition of constraints (ascending design). An additional fundamental goal is to define a uniform paradigm to develop business and non-functional (constraint-) components. In synthesis, we aim at proposing an operational component-based design environment providing new capabilities to express architecture constraints that can be executed at design-time to check the conformity of architecture designs and in which business components can be compiled into instructions of a component-based programming language.

In order to answer these challenges, we propose an approach where constraints are embedded in a special kind of software components. These components provide services via ports for checking these constraints at design time. These components do not exist at implementation or execution time. They are assembled with business (functional[4]) components which require the checking of constraints on their internal architecture description. In the proposed approach, each time the

[4] Constraint components can be considered as non-functional components here.

architect wants to check a given constraint on a business component, she/he should specify a new kind of required (non-functional or constraint) port that is removed when the business component is implemented. This required port is then connected to a provided port of a constraint component. The architect can add multiple required constraint ports if she/he needs to check several constraints on the internal architecture of her/his business components.

A provided administration port is integrated automatically to business components which have required constraint ports. This port allows to check all the constraints that are connected to these components. Each business component that has an administration port is instantiated at design time for using its administration port and thus for checking the constraints that are associated to the component. We chose to use a port for the checking of non-functional properties (structural constraints) of a component in order to ensure consistency of using components in the design stage.

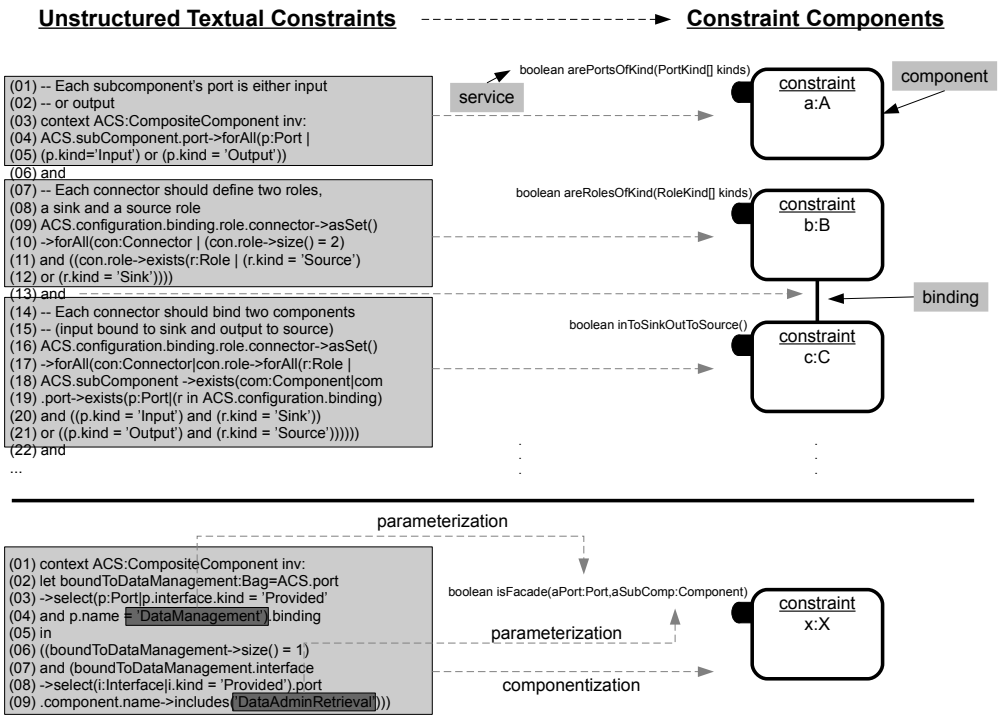


Fig. 1. Component-based Specification of Constraints

In the left part of figure 1) we give an example of an architectural constraint for pipeline property and in the right part corresponds to a set of constraint-components representing the same constraint. The architectural constraint is described using the ACL language [6]. Each component provides a single port for checking a part of the constraint through a service. On the other hand, the business components contain one required port by a non-functional property needed. To meet the non-functional property (pipeline) of the business component from our example, all obtained constraint components should be organized in a composite constraint component, which will provide an interface to check that the component conforms to the pipeline architecture style. Thus, binding a required port of a business component to a provided port of constraint component means that the latter is responsible for checking the validity of the concerned non-functional property. As for the provided administration port, in business components, the ports requiring constraint components exist only at design-time.

The services provided by constraint components can be parametrized by some architectural elements used in the constraint. This is illustrated in the bottom of Figure 1. In this way constraints become more generic and can be reused in different contexts (with different business components). Besides this, constraints become a modeling element that can be connected together to build more complex constraint components (as illustrated through the binding at the right of Figure 1).

2 Architecture Constraints as Components

Our solution is embedded into an operational software suite (CLACS-SCL) made of an Architecture Description Language (ADL) called CLACS, and of a component-oriented programming language named SCL [1]. CLACS is as a modeling alternative for SCL. Using that suite, component-based application architectures can be graphically composed in CLACS and deployed in SCL for execution; architecture constraints can be defined, composed and executed in CLACS.

In order to not add (yet-)other constructs for constraint-component modeling, we chose to use the same constructs as for business component modeling. SCL Business components and CLACS constraint components share most of their characteristics.

2.1 Specifying Constraint-Components

When designing a software architecture, the developer can connect constraint-components to business ones. The binding used to connect these two model elements makes it possible to validate the architecture design according to the constraints embedded in the constraint-component. This subsection proposes an example of a constraint-component definition and the following sub-section an example of such a connection.

Figure 2 depicts the definition of a simple constraint-component descriptor. This component allows to check the *Facade* pattern presented in the previous section. This descriptor can be instantiated in a given architecture description. Each *extitFacade checker*, instance of this descriptor, owns one provided port named `Checking` that exports a constraint checking service having the following signature : `boolean isFacade(aPort:Port,aSubComp:Component)`. Each *extitFacade checker* can then be connected, through that checking port, to any business component requiring this constraint service.

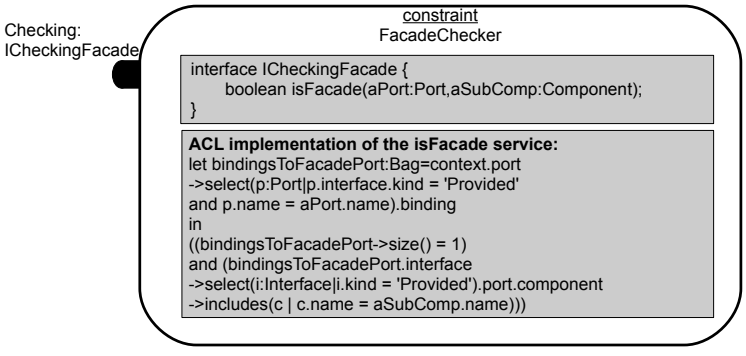


Fig. 2. Example of a primitive Constraint-Component

When invoked within our modeling environment, a constraint-component provided service returns true if the architecture of the business component to which it is connected fulfills the constraint. When such a connection is established and a constraint evaluated, the constraint expressions interpretor automatically binds the `context` identifier, used in constraints expressions

(see again Figure 2), to the business-component to which the constraint will be applied. When composite constraint-components are built in which a constraint-component is connected to another one, a transitive closure is computed on that link until a business-component is found.

2.2 Connecting Constraints to Business Components

Figure 3 presents two connected sub-components: a `Pass-wordGenerator` named `pg` and a `FacadeChecker` named `fc`. A `PasswordGenerator` uses random numbers to automatically generate passwords. It has a (business) provided port `Generation` and a (business) required port (`Randomizing`). At design stage, it also holds another provided port (`ConstraintChecking`) and another required (constraint) port (`Checking`). Through the latter it is connected to the `Checking` provided port of the `FacadeChecker` constraint component. Both ports are drawn in gray in the figure, to indicate their temporary nature (exist only at design-time) comparatively to the other business ports.

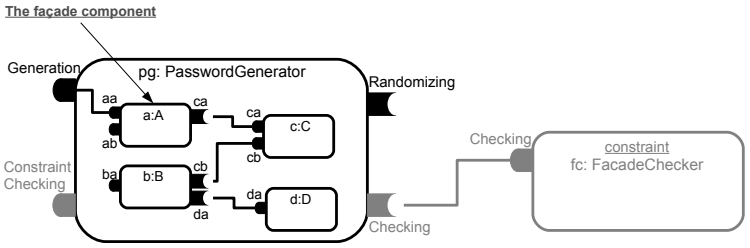


Fig. 3. Example of a Constraint-Component Assembled with a Business One

The semantics of the constraint checking binding is that the `PasswordGenerator` designer wants to check that the internal organization of this component conforms to the facade architecture pattern.

We can observe in Figure 3 that component `a:A` in the modeled architecture plays the role of a extitFacade (the only sub-component of `PasswordGenerator` that is connected by its provided port to the external ports of its encompassing component).

2.3 Composing Constraint-Components

Before detailing how constraint-components can be assembled, we expose in Figure 4 a constraint-component (`InputSinkOutputSourceRestrictor`) with a required port. This component represents the checking of a part of the constraint that formalizes the Pipleine architecture style. In this component descriptor there are two interface specifications. The first is named `IInSinkOutSrc-Restriction` and represents the type of the provided port, and the second (put in the dashed box in the figure) is named `IPortsRestriction` specifies a type for the required port `IO_PortRestriction`.

The required port is used in the body of the constraint (see again Figure 4) to invoke the service `arePortsOfKind`. We can observe in Figure 4 (underlined expression) how the service invocation is associated with the remaining of the constraint provided by the constraint-component `InputSinkOutputSourceRestrictor` using an `and` operator.

A constraint-component can be assembled with (or bound to) other constraint-components to build more complex ones. As in UML and many other component models, bindings can be either of type `Delegation` or `Assembly`. Delegation bindings of kind "`functional`" are used exclusively between business components (this is not discussed in this section). A delegation binding of kind "`constraintChecking`" is used for building a composite constraint-component starting from other constraint-components. This binding allows the composite to delegate the checking of part of

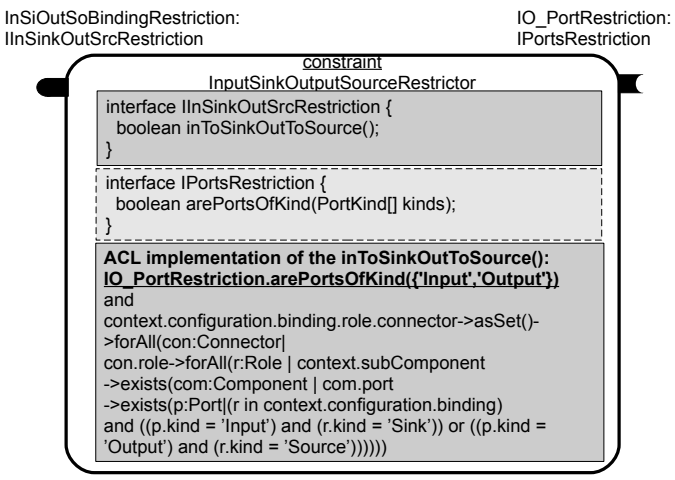


Fig. 4. Example of a Constraint-Component with a Required Port

the architecture constraint to its subcomponents. Figure 5 illustrates such a composition. The composite component (x:X in the Figure) asks its subcomponents a:A and c:C, which are connected to it by using internal required ports (see the left part of Figure 5), to make a constraint checking. The checking results are used in the service provided by the port `xx` of the composite component. These can be combined by an `and`, an `or` or any other logical operator in the provided service specification. In this service specification we can invoke the service (`serviceA()`) provided by the port `aa` of component a:A and the service (`serviceC()`) provided by the port `cc` of component c:C in the same way as stated in the previous subsection: `aa.serviceA() and cc.serviceC()`. Arguments are passed in the binding specification as indicated previously. A delegation binding can link a required external port of a subcomponent to a required external port of its composite component (see the right side of Figure 5). A constraint checking in this case is expected from another constraint-component connected with the composite component.

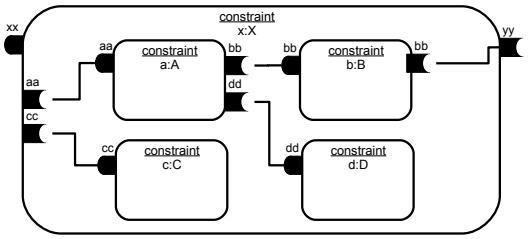


Fig. 5. Example of a Constraint-Component Composition

Assembly bindings link a required port of a given component to the provided port of another component of the same hierarchical level (for example, the components a:A and b:B). In Figure 5, we defined two assembly bindings, one between a:A and b:B, and the other between a:A and d:D. In the same way as for delegation bindings, it is in the specification of the service provided by the component a:A that we define how the results returned by the other components (b:B and d:D) are combined. In this example, we consider that a:A specifies its own constraint (represented by A) which is composed with the constraints provided by B and D. Here, x:X specifies only a composition of other constraint-components. It states that the constraint provided by a:A must be respected and the constraint implemented by c:C must not. This is equivalent to: `A and (not C)`.

3 Conclusion and Future Work

In this paper, we presented architecture constraints as recurrent non-functional solutions to recurrent documentation problems, to address the customizability and reusability challenges presented in the introduction. Architecture constraints are "white-box" assets that can be customized for a given application context. "Variability points" represent the architectural elements to be constrained. "Rules for usage" represent component assembly principles, which are based in this work on binding construction and thus on traditional interface matching.

Sometimes, defined manually (from scratch) this kind of architectural decisions' documentation is complex, error-prone and time-consuming. Having a means to define such documentations by hierarchical composition of constraints is beneficial for two accounts: First, by decomposing the models of architecture constraints in several small interfaced documentation parts, a common repository of reusable (parametrized) assets is provided for software architects; and second, this is a logical way of doing in the continuum of artifact development in component-based software engineering[5]. The development process obtained in this work starts with component architecture design and documentation with CLACS, and ends with component implementation and execution with SCL and its runtime environment.

References

1. L. Fabresse, C. Dony, and M. Huchard. Foundations of a Simple and Unified Component-Oriented Language. *Journal of Computer Languages, Systems & Structures*, 34/2-3:130–149, 2008.
2. A. Jansen and J. Bosch. Software architecture as a set of architectural design decisions. In *Proc. of WICSA'05*, 2005.
3. P. Kruchten. An ontology of architectural design decisions in software intensive systems. In *Proceedings of the 2nd Groningen Workshop Software Variability*, pages 54–61, 2004.
4. A. Tang, M. A. Babar, I. Gorton, and J. Han. A survey of the use and documentation of architecture design rationale. In *Proc. of WICSA'05*, Pittsburgh, Pennsylvania, USA, November 2005. IEEE CS.
5. C. Tibermacine, R. Fleurquin, and S. Sadou. On-demand quality-oriented assistance in component-based software evolution. In *Proc. of CBSE'06*, pages 294–309, Vasteras, Sweden, June 2006. Springer LNCS.
6. C. Tibermacine, R. Fleurquin, and S. Sadou. A family of languages for architecture constraint specification. *In Journal of Systems and Software (JSS), Elsevier*, 2010.
7. J. Tyree and A. Akerman. Architecture decisions: Demystifying architecture. *IEEE Software*, 22(2):19–27, March/April 2005.

[5] In the same spirit, the Eiffel language has been proposed for, at the same time, programming applications' business-logic and formalizing functional constraints (contract programming with assertions).

E{Java, CaesarJ, Scala} : un exercice d'intégration de la programmation par objets, par aspects et par évènements ⋆

Jacques Noyé

ASCOLA - École des Mines de Nantes/INRIA, LINA
Jacques.Noye@mines-nantes.fr

Résumé

La programmation par évènements et la programmation par aspects sont des paradigmes de programmation qui s'avèrent compléter utilement la programmation par objets dans une très large gamme d'applications. Leur utilisation concomitante, bien que possible dans un langage comme Java, est toutefois malaisée. Malgré leur très grande proximité, les solutions proposées présentent de nombreuses faiblesses et des irrégularités qui sont des sources notables de perplexité et de complexité. Considérons ces paradigmes deux par deux.

Programmation par objets et programmation par évènements En Java, la programmation par événements fait essentiellement appel au schéma de conception « observateur » ou à ses variantes. Cette solution pose divers problèmes, en particulier l'ajout de code d'infrastructure qui vient cacher la logique de base du programme. Pour pallier à ces inconvénients, C# permet de définir un évènement sous la forme d'un champ d'un objet et d'y associer dynamiquement des gestionnaires. Ces évènements se comportent toutefois étrangement vis-à-vis de l'héritage.

Programmation par évènements et programmation par aspects Il est tout à fait naturel de considérer, les *points de jonction* de la programmation par aspects comme des évènements générés par l'exécution du *programme de base* [1], évènements que l'on dira *implicites* pour les distinguer des évènements *explicites*, considérés ci-dessus, dont le déclenchement est explicitement programmé. On peut alors voir l'*action* exécutée par un aspect comme un gestionnaire d'évènement. Mais quelle place doit accorder cette vision évènementielle aux *points de coupe*, ces prédicats chargés de la sélection des points de jonction ?

Programmation par objets et programmation par aspects Pour AspectJ, le standard de programmation par aspects en Java, les aspects sont des classes sans en être. Ils sont instanciés mais implicitement, possèdent des membres, points de coupe inclus, mais ces derniers sont statiques, et les possibilités d'héritage sont très limitées. Finalement, la possibilité offerte aux aspects d'intervenir sans restriction sur l'ensemble du programme de base casse le raisonnement modulaire propre à la programmation par objets.

Notre proposition pour résoudre ces problèmes est de centrer l'intégration des trois paradigmes autour des principes de la programmation par objets et de réaligner autour de ces principes l'ensemble des concepts rencontrés :

- il n'y a plus d'aspects, seulement des classes dont certaines, que l'on pourrait qualifier d'aspectuelles, jouent un rôle habituellement dévolu aux aspects ;
- les notions de point de jonction, de point de coupe et d'action disparaissent au profit des notions d'occurrence d'évènement, d'évènement et de gestionnaire d'évènement ;
- de la même manière qu'un point de coupe en programmation par aspects sélectionne des points de jonctions et peut, en tant que prédicat, être défini comme une composition de prédicats, un évènement sélectionne des occurrences d'évènements et peut être défini *déclarativement* par composition d'autres évènements, ultimement des évènements implicites ou explicites ;

⋆. Ce travail a été effectué en collaboration avec Angel Núñez, Jurgen Van Ham, Vaidas Gasiūnas et Mira Mezini. Il a été partiellement financé par le projet AMPLE : Aspect-Oriented, Model-Driven, Product Line Engineering (STREP IST-033710).

– évènements et gestionnaires sont des membres d'instance. Par exemple, les règles habituelles liées à l'héritage (redéfinition et utilisation de `super`) s'appliquent.

Cette combinaison de caractéristiques fournit un modèle régulier et très flexible de programmation qui permet de mélanger les paradigmes existants, tout en en corrigeant certaines faiblesses, mais donne aussi accès à des hybridations intéressantes comme la possibilité de définir des évènements déclaratifs portant sur des évènements explicites. Ce modèle a été implémenté avec quelques variations dans EJava [2], ECaesarJ [3,4,2] et EScala [5], qui étendent respectivement Java, CaesarJ et Scala.

Suivant les possibilités de quantification fournies pour définir les évènements implicites, le modèle, dans sa généralité, ne garantit pas la préservation d'un raisonnement modulaire en présence de classes aspectuelles [1]. C'est toutefois le cas dès que les évènements observés par les classes aspectuelles sont définis au niveau des classes observées. Dans tous les cas, la définition des évènements en tant que membres d'instance permet de contrôler la portée des évènements qui ne peuvent être observés que par l'intermédiaire des relations de leurs instances englobantes. Au contraire, les propositions connexes, citons [6] et [7], mettent en avant le *type* des évènements (ou des points de jonction suivant la terminologie employée). Ceux-ci ont alors une portée globale et réduire leur observation à un ensemble restreint de sources demande de faire appel à des filtres, potentiellement couteux, ou de revenir à la programmation fine de sujets et d'observateurs. La possibilité d'utiliser du sous-typage est toutefois attractive. Une analyse fine de ces différentes approches et de leur adaptation à différents types d'applications reste à faire. Il est très probable que cette analyse montre qu'aucune approche n'est véritablement meilleure que les autres sur l'ensemble des applications et on peut dès à présent se poser la question de les combiner. Pour finir, signalons que ces travaux ont démarré avec l'objectif de prendre en compte la concurrence suite à des réflexions à un niveau conceptuel [8,9]. Cette dimension a été laissée de côté dans un premier temps mais est de nouveau d'actualité et bénéficie du riche environnement fourni par Scala.

Références

1. Douence, R., Motelet, O., Südholt, M. : A formal definition of crosscuts. In : Meta-Level Architectures and Separation of Crosscutting Concerns, Third International Conference (Reflection 2001), Springer (2001)
2. Núñez, A. : A Programming Model Integrating Classes, Events and Aspects. PhD thesis, École des Mines de Nantes and Université de Nantes (2011)
3. Núñez, A., Noyé, J., Gasiūnas, V. : Declarative definition of contexts with polymorphic events. In : Proc. of the International Workshop on Context-Oriented Programming at ECOOP'09 (COP '09), ACM (2009)
4. Núñez, A., Noyé, J., Gasiūnas, V., Mezini, M. : Product Line Implementation with ECaesarJ. In : Building Software Product Lines : The AMPLE Approach. Cambridge University Press (2011)
5. Gasiūnas, V., Satabin, L., Mezini, M., Núñez, A., Noyé, J. : EScala : Modular event-driven object interactions in Scala. In : Proc. of the 10th International Conference on Aspect-Oriented Software Development (AOSD 2011), ACM (2011)
6. Rajan, H., Leavens, G.T. : Ptolemy : A language with quantified, typed events. In : Proc. of the 22nd European Conference on Object-Oriented Programming (ECOOP 2008), Springer (2008)
7. Inostroza, M., Tanter, E., Bodden, E. : Join point interfaces for modular reasoning in aspect-oriented programs. In : Proc. of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE '11), ACM (2011)
8. Douence, R., Le Botlan, D., Noyé, J., Südholt, M. : Concurrent aspects. In : Proc. of the 5th International Conference on Generative Programming and Component Engineering (GPCE '06), ACM (2006)
9. Núñez, A., Noyé, J. : An event-based coordination model for context-aware applications. In : 10th International Conf. on Coordination Models and Languages (COORDINATION 2008), Springer (2008)

1. EScala est plus strict et préserve un raisonnement modulaire. EJava et ECaesarJ laissent le choix au programmeur.

Towards Flexible Evolution of Dynamically Adaptive Systems*

Gilles Perrouin[1], Brice Morin[2], Franck Chauvel[2], Franck Fleurey[2], Jacques Klein[3], Yves Le Traon[3], Olivier Barais[4], and Jean-Marc Jézéquel[4]

[1] PRECISE, University of Namur, Belgium, gilles.perrouin@fundp.ac.be
[2] SINTEF IKT, OSLO, Norway, {Brice.Morin,Franck.Chauvel,Franck.Fleurey}@sintef.no
[3] SnT - University of Luxembourg, Luxembourg, {jacques.klein,yves.letraon}@uni.lu
[4] IRISA, University of Rennes 1, France, {jezequel,barais}@irisa.fr

1 Introduction

We are now living in *societies of digital systems* [1] where various devices interact in an unpredictably changing environment offering services to humans ranging from mobile music experience to assistance in crises management. To support such digital societies, we can recourse to *Dynamically Adaptive Systems* (DAS). DAS can be seen as open distributed systems that have the faculty to adapt themselves to the ongoing circumstances while accomplishing their functionalities. Engineering such systems is complex since they combine several adaptation mechanisms ensuring that functional and non functional-properties, such as security and performance, are always satisfied.

Two main reasoning paradigms have been explored for DAS: *Event-Condition-Action* (ECA) rules and goal optimization. ECA-based techniques [2] directly link the features of the system to specific context fragments through a set of rules while goal-based ones [4] propose to describe which (QoS) properties should be optimized in which contexts. While DAS offer reconfiguration possibilities, combining distinct reasoning approaches is hardly supported. This lack of flexibility prevents proper support for unexpected events and hinders evolution: current approaches for developing and executing DAS rely on hand-written *ad hoc* control loops, which cannot change at runtime.

2 Dynamic Adaptation of the Control Loop

Our claim is that we can flexibly evolve a DAS by dynamically adapting the components of its control loop. Such a *meta-adaptation* mechanism is able to extend the reasoning abilities of a DAS and thus make it resilient to a larger number of unexpected situations. In our work, we focus on the the well-known *Monitoring-Analysis-Planning-Execution* loop (MAPE) [5].

The MAPE loop can be implemented as a component-based system, so that a MAPE loop can potentially be reconfigured at runtime. Thus, we use a MAPE loop to manage another MAPE loop, as illustrated by Figure 1.

* Summary of paper with same title published in ICSE 2012

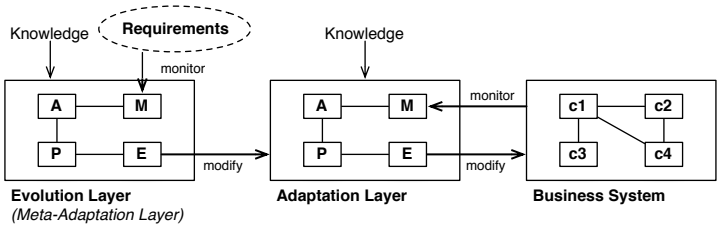


Fig. 1. An Overview of the Proposed Approach: Dynamic Adaptation of the Adaptation Loop to Support Evolution

1) Business Layer (right): This layer describes the architecture of the business application. This layer is managed by the adaptation layer, which can dynamically reconfigure the business architecture (addition/removal of components/bindings) depending on the monitored context.

2) Adaptation Layer (middle): This layer is responsible for managing the business layer. It basically consists of a MAPE loop usually found in most adaptive systems [5]. This MAPE loop is fed with knowledge (mainly variability and reasoning models) to determine when and how to adapt the business architecture. The novelty is that this loop is dynamically adaptive and managed by the evolution layer.

3) Evolution (Meta-Adaptation) Layer (left): This layer is responsible for managing the adaptation layer *i.e.*, to reconfigure the MAPE loop that manages the business system. This layer is also a MAPE loop, but it does not evolve at runtime. Yet, additional meta-adaptation layers can be defined [3]. It monitors the requirements of the business architecture. More precisely, it observes the knowledge of the adaptation layer and can decide to dynamically reconfigure the adaptation layer to make it to take full advantage of this new knowledge.

The dynamic reconfiguration of the adaptation loop is mainly driven by the evolution of the models (knowledge) describing the different facets of the business system. Model evolution can be monitored similarly to the evolution of the execution context [6].

References

1. Carzaniga, A., Denaro, G., Pezze, M., Estublier, J., Wolf, A.L.: Toward deeply adaptive societies of digital systems. In: ICSE COMPANION. pp. 331–334. IEEE (2009)
2. David, P., Ledoux, T.: Safe dynamic reconfigurations of fractal architectures with fscript. In: 5th Fractal Workshop at ECOOP. vol. 4067 (2006)
3. Edwards, G., Garcia, J., Tajalli, H., Popescu, D., Medvidovic, N., Sukhatme, G., Petrus, B.: Architecture-driven self-adaptation and self-management in robotics systems. In: SEAMS workshop@ICSE. pp. 142–151. IEEE (2009)
4. Elkhodary, A., Esfahani, N., Malek, S.: Fusion: A framework for engineering self-tuning self-adaptive software systems. In: Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering. pp. 7–16. ACM (2010)
5. Kephart, J., Chess, D.: The vision of autonomic computing. Computer 36(1), 41–50 (Jan 2003)
6. Morin, B., Ledoux, T., Ben Hassine, M., Chauvel, F., Barais, O., Jézéquel, J.M.: Unifying Runtime Adaptation and Design Evolution. In: CIT. Xiamen, China (Oct 2009)

Session du groupe de travail Compilation

Défis des architectures à venir, quelle compilation pour demain

Auteur : Erven Rohou, Inria/Irisa

Résumé :
Depuis une dizaine d'années, l'amélioration des performances des processeurs ne se fait plus sous forme d'augmentation de la fréquence d'horloge, mais de parallélisme additionnel. Les conséquences pour l'industrie du logiciel sont fantastiques : la plupart des applications existantes ont été développées avec un modèle séquentiel, et même les applications parallèles contiennent des sections séquentielles. Nous présentons certains défis posés par l'émergence de ces architectures parallèles et souvent hétérogènes, ainsi que des pistes pour la compilation de demain.

Quels langages pour l'ère post-Moore ?

Auteur : Pierre Boulet, LIFL, Université Lille 1

Résumé :
La fin prochaine de la loi de Moore provoquera très probablement une révolution architecturale. Les architectures multi-cœurs n'en sont que les prémisses. Nous allons inexorablement vers des architectures massivement parallèles hétérogènes et reconfigurables. Au delà de ces architectures basées sur le CMOS, des accélérateurs construits avec d'autres nanotechnologies sont envisagées pour continuer à améliorer les performances des dispositifs de traitement de l'information. Ces architectures seront vraisemblablement variables. Du côté du logiciel, l'abstraction simplificatrice de l'architecture de Von Neumann vole en éclat. Il faut donc repenser notre façon de concevoir les logiciels si on veut pouvoir exploiter ces futures architectures. Dans cet exposé, j'examinerai les défis que posent ces architectures aux langages de programmation et proposerai quelques pistes de recherche dans ce domaine.

Analyses statiques de programmes multitâches

Auteur : Antoine Miné, CNRS/ENS

Résumé :
Il est important de s'assurer de la correction des logiciels avant leur déploiement, en particulier dans les systèmes embarqués critiques. Dans cet exposé, nous présenterons l'analyseur Astrée pour les logiciels embarqués C synchrones et son extension AstréeA aux logiciels embarqués multi-tâches. Ces analyseurs détectent toutes les erreurs arithmétiques et de mémoire. Ils sont basés sur la théorie de l'interprétation abstraite, qui permet la construction d'analyseurs efficaces et sûrs (n'oubliant aucune erreur). En contrepartie, ils peuvent produire des fausses alarmes. Nous montrerons les abstractions qui nous ont permis de gérer l'explosion combinatoire des entrelacement des threads, les politiques d'ordonnancement des systèmes temps-réels et la consistance des mémoires partagées, afin de construire un analyseur sûr, raisonnablement précis et efficace sur des applications industrielles.

Session du groupe de travail FORWAL

Formalismes et Outils pour la Vérification et la Validation

Dépliage de réseaux de Petri généraux

Jean-Michel Couvreur
LIFO, Université d'Orléans, Orléans, France,
Jean-Michel.Couvreur@univ-orleans.fr

Denis Poitrenaud
LIP6, Université Pierre et Marie Curie, Paris, France,
Denis.Poitrenaud@lip6.fr

Pascal Weil
LaBRI, Université de Bordeaux I, Talence, France,
Pascal.Weil@labri.fr

L'étude du comportement des modèles de la concurrence exige habituellement la définition de modèles plus abstraits. Dans le cadre de réseaux de Petri, principalement deux modèles ont été retenus : les réseaux d'occurrences et les structures d'événements. Les deux modèles ont été proposés par Nielsen, Plotkin et Winskel dans leur article fondateur [6], afin de donner une sémantique de la concurrence pour les réseaux de Petri sains. La description d'une exécution d'un réseau de Petri sain est présentée par un réseaux causal étiqueté, appelé processus. Grosso modo, les réseaux causaux sont des réseaux de Petri acycliques dont les places sont sans branchement. Notamment, leurs places et leurs transitions sont partiellement ordonnées, et cet ordre, limité à des transitions, induit un ordre partiel sur les occurrences des transitions dans le réseau de Petri d'origine. Pour la représentation des conflits entre comportements, les branchements sur les places sont autorisés. Cela conduit à la définition des réseaux d'occurrences étiquetés, appelé processus arborescents. L'ensemble de tous les comportements du système peut être capturé par un seul et même processus arborescent, appelé dépliage du système, dont les transitions sont appelées événements. En restreignant la relation de causalité et de conflit aux événements, on obtient une structure d'événement appelé structure d'événements première. Depuis la publication de [6], il y a eu de nombreuses tentatives pour étendre ces résultats à des réseaux de Petri généraux.

Dans ces tentatives, l'accent a souvent été mis en utilisant les mêmes classes que dans le cas des réseaux de Petri sains, à savoir des réseaux causaux et des réseaux d'occurrences. Engelfriet [2] a limité son travail aux réseaux 1-valués et initialement sains, et il a obtenu de bonnes propriétés algébriques sur les processus arborescent (une structure de treillis) qui ont conduit à la notion de dépliage.

Pour d'autres (par exemple Best et Devillers [1], Meseguer, Montanari et Sassone [5]), les jetons sont individualisés. Comme l'a soutenu dans [4], ceci est contraire à une vue purement multi-ensemble des réseaux de Petri généraux, et les systèmes modélisés par des réseaux de Petri justifient rarement l'individualisation de jetons. De même, Haar [3], étendu par Vogler dans [7], propose une approche qui vise à traduire un réseau de Petri général en un réseau sain, en introduisant une place pour chaque marquage possible d'une place du réseau d'origine. Non seulement, il augmente considérablement la taille de la structure, mais il introduit artificiellement des conflits entre les transitions ayant en commun une place voisine. Ainsi, il s'écarte fortement de la sémantique attendue des réseaux de Petri.

A l'opposé de ces approches, Hoogers, Kleijn et Thiagarajan [4] proposent une nouvelle structure. Dans cette structure d'événements, les jetons ne sont pas colorées. Leur théorie est complète pour les réseaux de Petri co-sain (voir [4]), et elle peut être étendu aux réseaux généraux. Cependant, elle ne présente pas les propriétés attendues dans le cas général.

Dans notre étude, nous proposons une approche orientée réseaux de Petri, qui n'impose pas d'individualiser les jetons, et qui intègre les solutions de [1, 2, 5]. Notre cadre formel nous permet d'identifier une nouvelle structure, appelée vrai dépliage, qui possède les bonnes propriétés algébriques identifiées par Engelfriet [2], et qui est applicable aux réseaux généraux.

Le point de départ de notre approche est une extension de la définition des réseaux d'occurrences, qui peut être arbitrairement valués et non sains, et dont les configurations sont des multi-ensembles de transitions. Les processus arborescents sont définies comme des réseaux occurrences étiquetés par les éléments du réseau d'origine. L'ensemble des processus arborescents d'un réseau est muni d'une relation d'ordre conduisant à la définition du dépliage vue comme le plus grand processus arborescent.

Nous identifions deux classes de processus arborescents et deux types de dépliages : sain et vrai. Le cas sain coïncide avec ceux de [1, 2, 5]. Le cas vrai est plus intéressant par ses bonnes propriétés de treillis. Pour les réseaux sains, et plus généralement la classe des réseaux de Petri identifiée par Engelfriet dans [2], il existe un unique dépliage, et les concepts de dépliage sain et vrai coïncident. En général, toutefois, les deux notions de dépliage diffèrent.

Nous avons également formalisé la notion de processus dans notre contexte multi-ensemble. Contrairement à d'autres recherches, notre définition ne repose pas sur des réseaux causaux, mais sur le concept de configuration d'un processus arborescent. Encore une fois, il s'avère que le cas sain n'offre pas suffisamment de bonnes propriétés (*e.g.*, nous ne pouvons pas définir le plus grand minorant de deux processus). En revanche, les propriétés attendues sont vérifiées pour les processus vrais. Cela provient du fait qu'un processus vrai est représenté de manière unique dans le vrai dépliage.

L'inconvénient de notre approche algébrique est que les concepts de conflit et de causalité ne sont plus explicitement donnés par la structure du modèle. Nous perdons un lien direct avec les structures d'événements premières, comme dans [4].

Références

[1] E. Best and R. Devillers. Sequential and concurrent behaviour in Petri net theory. *Theoretical Computer Science*, 55 :87–136, 1987.

[2] J. Engelfriet. Branching processes of Petri nets. *Acta Informatica*, 28 :575–591, 1991.

[3] S. Haar. Branching processes of general S/T-systems and their properties. *Electronic Notes in Theoretical Computer Science*, 18, 1998.

[4] P.W. Hoogers, H.C.M. Kleijn, and P.S. Thiagarajan. An event structure semantics for general Petri nets. *Theoretical Computer Science*, 153 :129–170, 1996.

[5] J. Meseguer, U. Montanari, and V. Sassone. On the model of computation of Place/Transition Petri nets. In *Proc. of the 15th Int. Conference on Application and Theory of Petri Nets*, volume 815 of *Lecture Notes in Computer Science*, pages 16–38. Springer Verlag, 1994.

[6] M. Nielsen, G. Plotkin, and G. Winskel. Petri nets, events structures and domains, Part I. *Theoretical Computer Science*, 13(1) :85–108, 1981.

[7] W. Vogler. Executions : A new partial-order semantics of Petri nets. *Theoretical Computer Science*, 91 :205–238, 1991.

Boucles et sur-boucles pour les automates cheminant

P.-C. Héam et V. Hugot et O. Kouchnarenko

FEMTO-ST - CNRS UMR 6174 - INIRA CASSIS

1 Introduction

Les résultats énoncés ici ont été publiés dans [4]. Les automates d'arbre cheminant ont été introduits dans les années 60. Ce sont des automates finis dont la tête de lecture navigue dans une structure d'arbre, à la manière des *two-way automata* sur les mots finis. Il s'agit d'une approche qui contraste avec les automates d'arbre classiques qui ont un mode de reconnaissance en une seule passe en parallèle sur l'arbre.

Les automates d'arbre cheminant ont connu un regain d'intérêt récent pour l'étude théorique des documents structurés type XML [6, Chapter 12] : ils sont proches du fonctionnement d'une requête qui parcourt un document. Les principaux résultats fondamentaux récents sont, d'une part que les automates d'arbres cheminant ne reconnaissent qu'un sous ensemble strict des langages réguliers d'arbres [3] et, d'autre part, qu'ils ne peuvent pas être déterminisés [2]. Le lecteur intéressé pourra lire [6] ou [1] pour plus de renseignements et de pointeurs sur le sujet.

2 Automates d'arbre cheminant

Pour simplifier on ne considère que des arbres binaires complets dont les noeuds sont étiquetés par des éléments d'un alphabet fini $\Sigma = \Sigma_0 \cup \Sigma_2$, où les éléments de Σ_0 étiquettent les feuilles et ceux de Σ_2 les autres noeuds. On note $\mathbb{T} = \{r, f_1, f_2\}$ les types de noeuds d'un arbre, respectivement : racine, fils gauche, fils droit. On note par $\mathbb{M} = \{\circlearrowleft, \swarrow, \searrow, \uparrow\}$ l'ensemble des mouvements possibles : rester sur place, descendre à gauche, descendre à droite, remonter. Un automate d'arbre cheminant est un tuple $(Q, \Sigma, \Delta, I, F)$ où Q est un ensemble fini d'état, $I \subseteq Q$ est l'ensemble des états initiaux, $F \subseteq Q$ est l'ensemble des états finaux et $\Delta \subseteq Q \times \mathbb{T} \times \mathbb{M} \times \Sigma \times Q$ est l'ensemble des transitions. L'élément de $\mathbb{T}$ est une garde sur la transition : elle ne peut être tirée que si le noeud sur lequel on pointe dans le calcul est du type désigné. L'élément de $\mathbb{M}$ désigne comment on bouge le pointeur dans l'arbre. Un calcul dans un automate d'arbre cheminant sur un arbre donné pointe initialement sur la racine en étant dans un état de I. Il s'exécute ensuite selon les transitions. Il est réussi s'il pointe à la fin sur la racine en étant dans un état de F. Considérons par exemple l'automate de la figure 1, où $\Sigma_2 = \{f\}$ et $\Sigma_0 = \{a, b\}$.

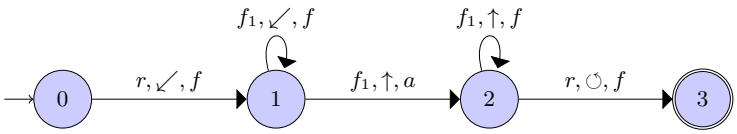


Figure 1. Exemple d'un automate cheminant.

Dans cet exemple, l'automate reconnaît tous les arbres dont la feuille en bas à gauche est un a et qui n'est pas réduit à sa racine. En effet, on commence par utiliser l'une des transitions entre 0 et 1 pour descendre vers le fils gauche de la racine. Puis, tant qu'on pointe

sur un noeud interne on va vers son fils gauche en bouclant sur l'état 1. Une fois dans la feuille la plus en bas à gauche, si elle est étiquetée par un a on passe dans l'état 2, puis on remonte jusqu'à la racine en restant dans l'état 2. Une fois dans la racine, on fait du sur-place en pointant encore sur la racine mais en passant dans l'état 3.

3 Contributions

Rappelons que les automates cheminant reconnaissent des langages réguliers d'arbre. Les algorithmes proposés dans la littérature transforment un automate d'arbre cheminant en un automate d'arbre *bottom-up* reconnaissant le même langage avec $|\Sigma|.|\mathbb{T}|2^{|Q|^2}$ états. En introduisant le nouveau concept de *sur-boucle*, nous avons montrer le résultat suivant.

Théorème *Soit $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$ un automate d'arbre cheminant. On peut calculer un automate d'arbre* bottom-up *reconnaissant le même langage ayant $|\mathbb{T}|2^{|Q|^2}$ états. Si $\mathcal{A}$ est déterministe, on peut en trouver un avec $|\mathbb{T}|2^{|Q|\log(|Q|+1)}$.* états.

En pratique les algorithmes font un calcul à la volée et les gains en temps sont similaires aux gains en taille. Il faut noter que gagner un facteur $|\Sigma|$ n'est pas négligeable dans un contexte de documents structurés où l'alphabet code les différentes balises possibles et donc avoir une taille conséquente. Nous avons testé l'approche, dans le cas déterministe, en utilisant un générateur uniforme d'automates cheminant déterministes [5]. En pratique les gains obtenus sont significatifs, avec des ratios de taille entre 2 et 18 en moyenne.

Par ailleurs, le problème du vide pour les automates cheminant utilise la transformation en automates d'arbre équivalents. Afin d'éviter en pratique les constructions lourdes ci-dessus, nous avons mis au point des heuristiques s'appuyant sur des approximations, dont les performances expérimentales sont excellentes.

Références

1. M. Bojanczyk. Tree-walking automata. In C. Martín-Vide, F. Otto, and H. Fernau, editors, *LATA*, volume 5196 of *Lecture Notes in Computer Science*, pages 1–2. Springer, 2008.
2. M. Bojanczyk and T. Colcombet. Tree-walking automata cannot be determinized. *Theor. Comput. Sci.*, 350(2-3) :164–173, 2006.
3. M. Bojanczyk and T. Colcombet. Tree-walking automata do not recognize all regular languages. *SIAM J. Comput.*, 38(2) :658–701, 2008.
4. P.-C. Héam, V. Hugot, and O. Kouchnarenko. Loops and overloops for tree walking automata. In B. Bouchou-Markhoff, P. Caron, J.-M. Champarnaud, and D. Maurel, editors, *CIAA*, volume 6807 of *Lecture Notes in Computer Science*, pages 166–177. Springer, 2011.
5. P.-C. Héam, C. Nicaud, and S. Schmitz. Parametric random generation of deterministic tree automata. *Theor. Comput. Sci.*, 411(38-39) :3469–3480, 2010.
6. H. Hosoya. *Foundations of XML Processing, The Tree Automata Approach.* Cambridge University Press, 2010.

Abstract Tree Regular Model Checking for Lattice Based Automata

Auteur : Thomas Genet (INRIA/IRISA), Tristan Le Gall (INRIA/IRISA), Axel Legay (INRIA/IRISA)

Résumé :
L'ingénierie des lignes de produits logiciels est un paradigme d'ingénierie du logiciel dont le but est de permettre le développement efficace de grandes familles de logiciels. Cependant, dans le cas des systèmes critiques, peu de membres d'une famille de produits sont généralement vendus aux clients. En effet, les techniques d'assurance qualité utilisées dans le développement de ce type de systèmes, telles que le model checking, ne peuvent traiter que des systèmes sans variabilité. En conséquence, il est très coûteux d'appliquer ces techniques à tous les systèmes d'une même famille.

Si les différences entres les systèmes sont exprimées en terme de features, le nombre de systèmes possibles est exponentiel en fonction du nombre de features. Deux défis importants pour les techniques d'assurance qualité sont donc le développement de modèles et d'algorithmes qui passent à l'échelle. Nous traitons ces défis pour une technique spécifique : le model checking.

La technique de model checking proposée est basée sur les Featured Transition Systems (FTS), un nouveau formalisme introduit dans cette thèse. Les FTS sont un modèle mathématique pour représenter le comportement d'une famille de systèmes de fa con compacte. Un FTS est essentiellement un système de transitions dans lequel la présence d'une transition dépend des features. Nous proposons et étudions des algorithmes permettant de vérifier des propriétés temporelles sur des FTS. Ces algorithmes peuvent prouver que tous les systèmes d'une famille satisfont une propriété donnée, ou bien ils identifient ceux qui ne le font pas. Pour spécifier les propriétés, nous proposons feature LTL and feature CTL, des extensions des deux logiques temporelles classiques.

En plus des fondements mathématiques, nous décrivons deux implémentations des FTS utilisables par des non-experts. La première utilise une représentation symbolique de l'espace d'états et a été implémentée au sein du model checker NuSMV. La deuxième, appelée SNIP, utilise un algorithme semi-symbolique à la volée. Le langage de modélisation de SNIP est basé sur le langage Promela. Enfin, nous discutons une évaluation théorique et empirique de nos résultats. Le cas de base utilisé pour l'évaluation empirique est l'application d'un algorithme de model checking classique à chaque produit. Les expériences con- duites avec les deux outils montrent que nos algorithmes peuvent atteindre des gains en vitesse de plusieurs ordres de grandeur par rapport au cas de base.

Session de l'action IDM

Ingénierie dirigée par les modèles

Introduction à l'approche ADM

Modernisation du patrimoine logiciel par les modèles

Olivier Le Goaer

Université de Pau et des Pays de l'Adour
Avenue de l'Université, 64000 PAU
olivier.legoaer@univ-pau.fr

Abstract. Le patrimoine logiciel existant (*legacy*) a été l'un des obstacles majeur à la percée de l'approche MDA de l'OMG. En effet, la place des systèmes logiciels est désormais prépondérante, voire critique, dans le fonctionnement des entreprises et des organisations. Ces structures ne demandent pourtant qu'à pouvoir transformer et moderniser leurs systèmes d'informations vieillissants, mais pas à n'importe quel prix ! En effet, il n'est pas concevable de faire table rase des systèmes en place – et de toute la connaissance qui y est enfouie – pour en redévelopper de nouveaux à partir de zéro, même à l'aide de l'outillage MDA.

Face à ce constat empirique, un groupe de travail s'est vu assigné l'objectif de construire et de promouvoir les standards qui seront utilisés pour la modernisation de l'existant. Cette initiative de l'OMG, baptisée ADM (*Architecture-driven modernization*), se veut le miroir du MDA, en partant du code pour remonter vers les modèles. En d'autres termes, l'ADM cible la rétro-ingénierie tandis que le MDA cible l'« ingénierie vers l'avant ».

L'ADM fournit des spécifications pour traiter des aspects variés des systèmes logiciels patrimoniaux. Parmi celles-ci, KDM (*Knowledge Discovery Metamodel*) constitue l'épine dorsale de l'approche puisqu'elle offre une représentation pivot d'un système, destinée à favoriser l'interopérabilité avec le reste des standards de l'OMG et avec les outils du marché qui s'y rapportent.

L'approche ADM est incomplète; le processus de standardisation du groupe de travail est toujours en cours. Néanmoins, sa version finalisée jouera probablement pleinement son rôle de catalyseur du MDA tant les besoins de modernisation sont immenses.

Keywords: Ingénierie des modèles, OMG, modernisation, patrimoine logiciel

adfa, p. 1, 2011.

Dealing with Multiple Abstractions in Software Process Modeling

Fahad R. Golra and Fabien Dagnat

IRISA / Université Européenne de Bretagne
Institut Mines-Télécom / Télécom Bretagne
Brest, France
{fahad.golra,fabien.dagnat}@telecom-bretagne.eu

Abstract. Software development processes are of varying nature based on their dynamics, abstraction and automation. The corresponding process modeling approach should be able to deal with these variations. CAMA Process Modeling Framework is presented to deal with these processes in a way that all these variations can be modeled effectively. CPMF deals with multiple abstraction levels in terms of development phases and in terms of logical abstractions within an activity.

1 Introduction

A process modeling approach that has the capability of assisting in modern development paradigms like MDE, should be able to deal with models in an effective manner. This support for MDE needs to be the focal point of the current process modeling endeavors. A process modeling approach should allow the flexibility for process improvement, not only at organizational level but also within a running project, on the fly. This calls for reactive and dynamic process modeling approaches that can deal with abstractions like meta-levels (modeling hierarchies).

Process modeling domain has mostly been dominated by business process models that focus towards the workflow and sequence of the processes. This results in the lack of appropriate mechanisms to deal with dynamics, abstraction and automation. We also believe that software process modeling being targeted for the software industry should match the perspectives of the domain experts. The software development domain has its roots in concepts like execution, initiation, implementation and typing. We propose a software process modeling language that takes benefit of these key concepts by the inspirations from component based paradigm.

2 Approach

We argue that a unique process model cannot capture all the semantics of the processes at different development phases. For this reason, our approach presents three metamodels: the *Process Specification Metamodel(PspM)* [2], the

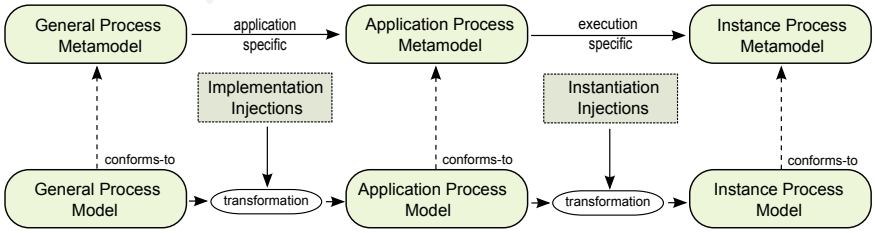


Fig. 1. Multi-phase development in CPMF

Process Implementation Metamodel(PimM) [3] and the *Process Instantiation Metamodel(PinM)*, as illustrated in Figure 1. A *PspM* is used to document the process best practices. It is not specific to certain organization or project. When it is applied to a specific project by some organization, it is refined to guide the development process, thus transformed into a *PimM*. A *PimM* is responsible for the execution of the processes for a project, thus it takes into account the time plan and status of the project. For example, if we take the ECSS software development standard, we can model it as a *PspM*. A specific implementation model for an application conforming to this standard, can be modeled as *PimM*. And finally, for the execution of this application model, we have a *PinM*, that serves as a support for project management. Model transformations are used between the models conforming to these metamodels, which inject implementation and instantiation details to the process model as it advances. We are looking forward to provide the tool support for carrying out these transformations.

A process in our approach is a collection of activities that can be arranged in a sequence through dependencies based on their contracts. An activity is decomposable, overall presenting the process model as a hierarchical structure. Activities can be shared amongst different processes. Activity definitions are kept separate from their implementations where the later provides the implementation of the former. An activity can have various implementations but all of these implementations have to follow the contract of the activity. This gives us the flexibility of replacing one activity implementation with another without affecting the context of the activity. This black box characteristic of an activity promotes process improvement. Furthermore, the reuse of activity and its implementations is ensured by the use of contract as the interface of an activity.

An activity behaves as a black box component, where its interface to its context and content (in case of composite activity) uses its contract. The contracts of the activity types, at the abstract level are abstract contracts, where as for the activity implementations, they are concrete contracts. Abstract contracts contain artifacts and concrete contracts contain events. Events are responsible for triggering activity implementations. Each of these events of concrete contract map to an artifact of the abstract contract. Inspired from the component based paradigm, the only source of transition between two activities is through this contract. Contracts of an activity may be required or provided. The required contract provides the requirements to the activity from its context so that it can

be executed. The provided contract provides the producible from the activity to its context. In case of composite activities, for each external required contract, there is a corresponding implicit internal provided contract and for each external provide contract, there is a corresponding implicit external required contract.

A collection of activities connected together through dependencies is a process. Dependencies allow the sequencing of the activities based on their contracts (required or provided). No explicit control flow is defined for the activities. The properties of the activity and the dependencies of the contract together allow the dynamic sequencing of flow for the processes. This dynamic sequence of processes gives a reactive control for process models that have the capability of restructuring the control flow at runtime. Artifacts, either produced or required by the activities are dealt as models, which conform to their metamodels. Thus each activity converts an input artifact model to an output artifact model, where each of them conforms to their respective metamodels. This gives us the flexibility to support model transformations in our approach.

3 Conclusion

This approach presents activities as components which have their defined contracts. The activities support abstractions in terms of multiple activity implementations for a single activity type. Use of structured artifacts allows the processes to support model driven engineering. The controlflow at the concrete level through event management system, guides the data flow at the abstract level. The instantiation semantics for these processes is current under development, which would allow us to have an executable software development process modeling approach that can support model driven engineering. Furthermore we are looking forward for defining a formal semantics of these processes. Such formal semantics would allow us to verify the properties of the processes all way starting from the requirements phase till the deployment. The tooling support for our framework is under development, for which we are extending an existing open source tool for process modeling, Openflexo [1].

References

1. Agile Birds. Openflexo. `http://openflexo.com`, 2011.
2. Fahad R. Golra and Fabien Dagnat. Using component-oriented process models for multi-metamodel applications. In *Proc. of the 9th International Conference on Frontiers of Information Technology*. IEEE, Dec 2011.
3. Fahad R. Golra and Fabien Dagnat. Using component-oriented process models for multi-metamodel applications. In *Proc. of th International Conference on Software and System Process, ICSSP*. IEEE, June 2012.

Teaching MDE through the Formal Verification of Process Models

Benoit Combemale[2], Xavier Crégut[1], Arnaud Dieumegard[1], Marc Pantel[1], and Faiez Zalila[1]

[1] Firstname.Lastname@enseeiht.fr
Université de Toulouse, IRIT – France
[2] Firstname.Lastname@irisa.fr
Université de Rennes 1, IRISA – France

Abstract. Model Driven Engineering (MDE) and formal methods (FM) play a key role in the development of Safety Critical Systems (SCS). They promote user oriented abstraction and formal specification using Domain Specific Modeling Languages (DSML), early Validation and formal Verification (V&V) using efficient dedicated technologies and Automatic Code and Documentation Generation. Their combined use allow to improve system qualities and reduce development costs. However, in most computer science curriculae, both domains are usually taught independently. MDE is associated to practical software engineering and FM to theoretical computer science. This contribution relates a course about MDE for SCS development that bridges the gap between these domains. It describes the content of the course and provides the lessons learned from its teaching. It focuses on early formal verification using model checking of a DSML for development process modeling. MDE technologies are illustrated both on language engineering for CASE tool development and on development process modeling. The case study also highlights the unification power of MDE as it does not target traditional executable software.

1 Introduction

The course presented in this contribution was designed in 2007 for M2 students in System and Software Engineering in Toulouse where Aeronautics, Automotive and Space transportations, and especially the development of Safety Critical Systems, are the key industries. These industries have been using model based design for the last 20 years and have experimented with the use of formal specification and verification for 10 years. However, in the academic curriculae, these technologies are taught independently in very different manners as Model Driven Engineering (MDE) is associated to practical software engineering and Formal Methods (FM) to theoretical computer science. The growing complexity of critical systems can only be mastered by using both MDE techniques with user level languages and formal methods through tool level languages. This course gathers these elements to bridge the gap between domain-specific languages promoted by MDE and verification-specific languages promoted by formal methods. The usual verification done in MDE is made of structural OCL constraints (as in [1],[2]). This course relies on model checking[3] based on Temporal Petri nets to verify behavioral correctness of development process models. Furthermore, the unification granted by the MDE is emphasized in this case study through the translation from end-user domain-specific languages to formal verification-specific languages. This course also explains how software modeling and formal verification can coexist during a system development and gives a practical overview of technologies easing the introduction of formal verification technologies such as model-checking. The key principle was to apply the various MDE technologies that allows the implementation of a Domain Specific CASE tool, such as TOPCASED[4], to a verification driven case study.

This contribution is structured as follows. Section 2 presents the simple process modeling language use case that structures the whole course. Section 3 lists the main topics addressed

[3] The authors wish to thank F. Vernadat that gives that part of the course and initiated the whole activity.
[4] http://www.topcased.org

by the course and further described in the next sections: metamodeling and static semantics constraints (section 4), concrete syntaxes (section 5) and transformations (section 6). Section 7 presents some extensions to the initial use case that allows student to develop and validate the acquired knowledge. Finally, section 8 explains how the course is conducted in the different contexts and gives some insights on future evolutions.

2 Formal Verification of Processes: *SimplePDL to Tina* Case Study

The main target of this course is the development of safety critical systems using Domain Specific Modeling Languages (DSML) and formal verification technologies. A simple yet realistic concrete case study is used all over the course in order to illustrate the different concepts and associated tools of MDE that are used to create DSMLs and to connect them to existing tools. The starting point is a very simple SPEM-based process modeling language called SimplePDL and the verification that all activities terminates.

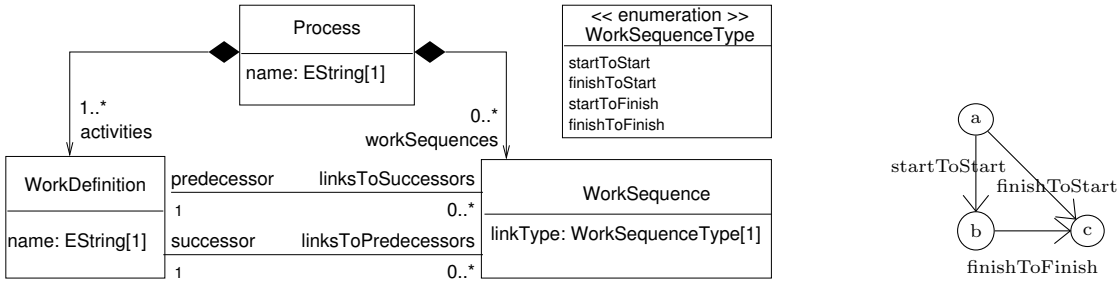


Fig. 1. First metamodel of SimplePDL (left) and one conforming model (right)

The SimplePDL metamodel (Figure 1) defines the process concept (*Process*) composed of a set of activities (*WorkDefinition*) representing the activities to be performed during the development. The concept of *WorkSequence* illustrates temporal dependencies between work definitions: the target activity can only be started or finished if the source activity is already started or finished. The kind of constraint (*linkType*) is expressed using the enumeration *WorkSequenceType*. For example, the *c* activity (right of figure 1) can only be started when *a* is finished and finished when *b* is finished. This first metamodel is obviously oversimplified but some extensions are proposed in section 7 to gain in expressiveness and validate the acquired knowledge.

The end user purpose is to check properties on a SimplePDL model. For example, he may asks whether the modeled process can finish (all activities from the process have been finished while respecting the sequencing constraints expressed by the work sequences).

To answer those questions, the students must reuse model-checking tools that they have studied in previous courses. We rely currently on the Tina toolkit[5] for the verification activities using Temporal Petri nets as modeling language and SE-LTL (*State Event Linear Temporal Logic*) as property language. The principle of the verification is thus to translate a process model into a behaviourally equivalent Petri net and then to express the common end users questions as LTL formulae that will be checked by the Tina model-checker. A PetriNet metamodel is used to avoid being directly wired to the Tina input syntax. Thus, as depicted in Figure 2, the overall approach is composed of a model to model (M2M) transformation (SimplePDL to PetriNet) and two model to text (M2T) transformations (PetriNet model to Tina concrete syntax for the first one and SimplePDL model to LTL concrete syntax for the second).

[5] http://www.laas.fr/tina/

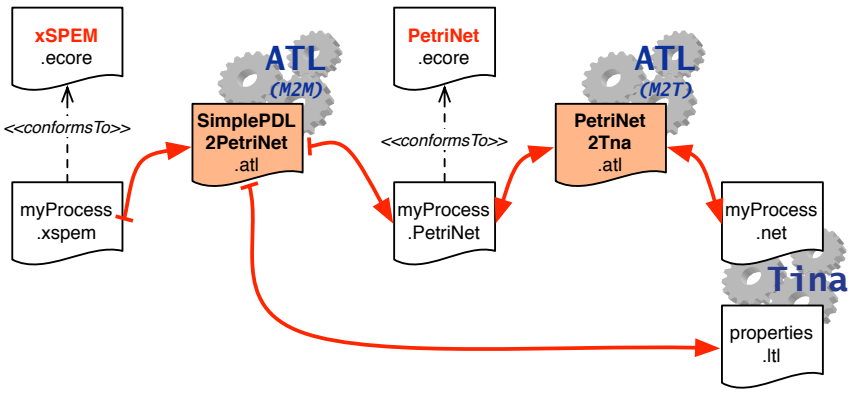


Fig. 2. Approach to evaluate behavioral properties on a process model

3 Content and Schedule of the Course

The course presents the main concepts and tools of MDE as a set of 7 topics[6] grouped into 3 themes (metamodeling, concrete syntaxes and model transformations) and ends with a project as listed in the following table and developed in the next sections.

Metamodeling (section 4)	1. Metamodeling using Eclipse/EMF
	2. Static semantics using OCL
Concrete Syntaxes (section 5)	3. Textual concrete syntaxes using Xtext
	4. Graphical concrete syntaxes using GMF generators
Model transformations (section 6)	5. Model to Model transformation using ATL
	6. Model to Text transformation using xPand
	7. Model to Text transformation using ATL
Project (section 7)	Implementing extensions of the case study

One lecture (≈2h) and a practical work session (≈2h) is allocated to each topic. Lectures present the main concepts and associated standards. For each topic[7], except the two last one, the practical session has the same three-part structure. The first part is a tutorial that presents the concepts and tools illustrated on the SimplePDL language. The second part provides exercises to ensure that the students got a good understanding. Exercises consists in completing the work done in the first part. The third part is an open exercise that asks the student to do a similar work on the Petri net language, without any guidance. The first two parts are usually finished in about two hours of supervised work but the last one is often only started in the supervised time slot and students have to finish it on their own.

Finally, a project is done by students: the implementation of several extensions to the main case study. Section 7 describes the extensions. First, a 6 hours assignment is used to assess that students master the MDE concepts and tools. It is based on the E_1 and E_2 extensions of section 7. An oral presentation and demonstration is organized. Then, the other extensions are implemented by the students (18 hours of personal work). These extensions can be done independently in order to avoid unnecessary difficulties in regard to the evaluation of their understanding. Students pass an oral examination where they show the result of their work and explain how they conducted this work, what were their main choices and why they made them.

All tools used in practical work are based on the open-source Eclipse platform and are part of (or can be added in) the Eclipse Modeling Tools[8]. This choice allows students to go on with their work using any computer, and have a unified environment for all the tasks they have to do.

[6] An optional topic called "The Tina toolkit" is a 20 minutes tutorial that explains to students who have never used the Tina toolkit how to perform model-checking. It is not described here as it is not strictly related to MDE technologies.

[7] Teaching materials are available at http://cregut.perso.enseeiht.fr/2010/ but, unfortunately, currently only in French.

[8] cf. http://www.eclipse.org/downloads/packages/eclipse-modeling-tools/indigor

4 Metamodeling

The first topic focuses on metamodeling using the Ecore language from the Eclipse Modeling Framework[9]. Students use the Ecore graphical editor to load the initial metamodel of SimplePDL (Figure 1). They use EMF to generate Java classes to load and store models, and a structured editor that is then used to edit small process models. The structured editor provides a good understanding of the *containment* attribute of an *EReference* (and a concrete example of the composition relationship already seen in UML classes that took place in the previous years). Furthermore, the *eOpposite* attribute also helps in the understanding of the UML association (when one reference is updated, the opposite reference is also updated).

The second part of this topic consists in modifying the metamodel by adding a *ProcessElement* metaclass to generalize WorkDefinition and WorkSequence elements and a Guidance element that allows to associate a natural language description to any process element. Aside manipulating the Ecore editor, students discusses advantages and drawbacks of their metamodels.

Finally, students have to define a metamodel for Petri nets from scratch. The starting point is a textual description of Petri nets (taken from wikipedia) with some model examples. Examples are easier to understand by the students but they lead them to define a partial metamodel that has to be completed using the information provided in the textual description. It is a very interesting exercise because several metamodels are usually provided by the various students and many exchanges take place in order to define the best one according to various criteria.

Furthermore, the Petri net metamodel does not capture all the constraints of Petri net models. It thus demonstrates the need to define the static semantics. We use OCL for that purpose.

OCL and static semantics is presented in the next topic. The TOPCASED OCL checker is used for practical activities. When an invariant does not hold, the context element is red marked. First, students must write some OCL constraints in order to understand the basics of this language. The use of the tools allows to stress that it is important to define an invariant at the right place. For example, expressing that activity names are unique may be expressed as an invariant on *Process* but marking a *Process* element as wrong is of little help to find the bad activities. Thus, this invariant is better expressed on *WorkDefinition* elements so that bad activities are highlighted.

Thanks to the Petri net metamodel, students can notice that there is a balance to be found between having a simpler metamodel with less concepts and relationships but more OCL constraints, or a more complex metamodel with less OCL constraints.

5 Concrete syntax

A metamodel only describes an abstract syntax. Models are stored as XML or XMI files that could be considered as concrete syntaxes but these are not designed as editing tools for the end user. Thus, it is mandatory to provide concrete syntaxes, like the Ecore diagram defines a graphical concrete syntax for Ecore models. In this course, textual concrete syntaxes are presented using Xtext and graphical ones using GMF Tooling.

5.1 Textual Concrete Syntaxes

Xtext[10] is used as a tool to define concrete syntaxes. Figure 3 proposes two examples of concrete syntaxes for the SimplePDL model presented on the right of figure 1. The first one is used to explain the concepts of Xtext: both the concrete syntax and the Xtext files are provided. Then, the Xtext file corresponding to the second syntax is given and students have to find the concrete syntax it corresponds to. They first have to write it on a paper to ensure they understood the grammar as defined in Xtext, then they can test it using the generated Eclipse editor. Finally, they have to define their own textual concrete syntax for Petri nets.

[9] See the Eclipse Modeling Project: http://www.eclipse.org/modeling
[10] http://www.eclipse.org/Xtext

```
process dvp {
    wd a
    wd b
    wd c
    ws s2s from a to b
    ws f2f from b to c
    ws f2s from a to c
}
```

```
process dvp {
    wd a
    wd b starts if a started
    wd c finishes if b finished
      starts if a started
}
```

Fig. 3. Two possible textual concrete syntaxes for SimplePDL

Xtext can generate the metamodel that is closely related to the structure of the grammar provided in Xtext for the concrete syntax. The metamodel corresponding to the first syntax is close to the one of Figure 1 whereas the second one is very different. Students are thus shown that a compromise must be found between the structure of the grammar and the structure of the generated metamodel. Using an existing metamodel is also possible but is a bit tricky.

The main lesson is that the "natural" metamodel defined for a domain is generally not well-suited for defining the concrete syntax. It is thus better to let Xtext generate its own metamodel and then use a M2M transformation to translate this one into the other. Xtext allows to automatically use an Xtend transformation in that purpose but we do not teach that feature to students.

5.2 Graphical Concrete Syntaxes

Graphical concrete syntaxes allow graphical editors to be built for the respective metamodels. It is very attractive for the students to be able to do so in a very short time.

From a pedagogical point of view, graphical editors are a good illustration of MDE principles, and generative approaches. The user only has to describe in a declarative manner the features of the expected editors, and the different generators get the things done. Moreover, while most of the generative approaches are defined for a given DSML to automate tasks from the conforming models, the use of such generative approaches can highlight the usefulness of a metametamodel.

We initially used the simple graphical editor generator provided by TOPCASED. All aspects required to describe the editor were blurred in the same configuration model (Views, Controler and mappings to the Model in the MVC pattern) with very few possible customizations.

For three years now, we have switched to GMF Tooling[11] that is part of the Eclipse Modeling bundle. GMF Tooling puts a better stress on separation of concerns as it is based on different models that describe each part of a graphical editor: tools, graphical representation of elements, palette and mappings among these models and the Ecore metamodel. It also uses OCL to define some behavior, for example to prevent a reflexive transition (illustrated on the *WorkSequence* element). Unfortunately GMF tooling has suffered from nasty bugs for years that generate files with obvious mistakes that have to be hand-corrected by students, which is very confusing.

The main drawback of these tools is that they are quite complex and, often students only perform the required actions without reading the explanations and understanding the underlying motivations. Thus, at first sight, students consider those tools as tedious. Furthermore, they lack many features and does not help when they want to build their own editor for PetriNet models without writing Java code. It is why we plan to switch to OBEO Designer[12] to have a more robust and sophisticated tool.

6 Model Transformation

The last topics concern model transformation. Model to model transformations (M2M) are presented using ATL. ATL and xPand are used for model to text transformations (M2T).

[11] http://wiki.eclipse.org/GMF
[12] http://www.obeodesigner.com

6.1 Model to Model Transformations

M2M transformations are a key concept of the MDE course. A presentation of QVT [3] specification puts the focus on the operational and declarative parts. Many languages are available. We could have used operational languages such as Kermeta, xTend, SmartQVT, EMP Operational QVT, Epsilon Transformation Language, etc. But we feared that students would only consider them as classical programming languages. Indeed, they have to manage explicitly the control flow, track links to already created target elements, etc. Their major benefit over a language like Java is their powerful query language and collections operators derived from OCL.

We have chosen ATL and we mainly use its declarative part so that students can concentrate on how to map source elements to target elements. In practice, they write a rule for each source element and describe the elements that have to be created in the target model.

Before handling technical details, we first ask students to find how to translate a SimplePDL model into a Petri net in order to verify that the modeled process terminates. We take a very simple process like the one on the left of figure 4. The corresponding Petri net is on the right of this figure. The purpose is that they gain a good understanding of the mapping between process models and Petri nets. Each *WorkDefinition* is translated into three places characterizing its state (*notStarted*, *running* and *finished*) linked by two transitions. These transitions model the actions that we want to observe on an activity: one can *start* an activity and then *finish* it. An activity is considered as *started* if it is running or finished. This is recorded by the place called *started*. A *WorkSequence* becomes a *read-arc*[13] from one place of the source activity (either *started* or *finished*) to a transition of the target activity (either *start* or *finished*) according to the kind of *WorkSequence*. This allows to illustrate a property driven approach to formal model design.

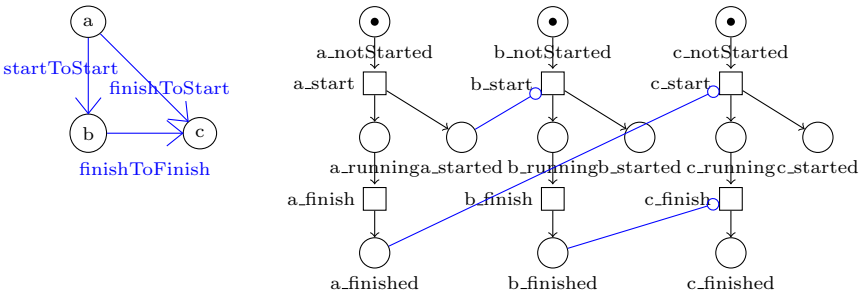


Fig. 4. Translation of a process composed of work definitions a, b and c into a Petri net

To show the principles of ATL, we give the rule that translates a Process into a PetriNet and the beginning of the rule that translates a WorkDefinition into nodes, places and arcs. Process to PetriNet is a 1 to 1 rule. Students have to complete the WorkDefintion2PetriNet rule. This rule creates 4 places, 2 transitions and 5 arcs for each work definition (1 to n rule). The only difficulty is that a process element does not have direct access to its process container. An ATL helper is provided to access the missing reference.

Finally, they add a new rule to translate one work sequence into a PetriNet read arc (1 to 1 rule). This last rule illustrates the use of `resolveTemp` ATL operator (`resolveIn` in QVT) which allows to select one element of the target model among those built from a given source model element (in a 1 to n rule).

Operational constructs or more advanced concepts like rule inheritance, called rules and so on are mentioned but not used. One important aspect of the use case is that it allows this kind of simple declarative transformation. It would be useful to illustrate also an example that is really ill fitted and requires a more imperative transformation, however we lack time to have the students do the experiment and can only give insights on this problem.

[13] A read-arc checks that enough tokens are in the input place. Tokens are not withdrawn when the transition fires.

6.2 Model to Text Transformations

A first example shows how xPand may be used to generate the graphical concrete syntax of SimplePDL proposed in Fig. 4. Then, students have to generate a DOT file for textual graph representation. The final exercise consists in generating the Tina text from a Petri net model.

Students are quite familiar with template languages due to the use of JSP in previous courses and have no problem to use xPand despite its verbose syntax and the use of non common characters (even for french students).

ATL also provides M2T transformations. We provide the students with the ATL query that translates a Petri net model into the Tina syntax. All aspects of Petri net are handled except for time constraints. So, once students have noticed that ATL allows to define helpers on metamodel elements and uses a query language very close to OCL, they complete the query. They are then able to write from scratch the second M2T transformation that generates the LTL formulae corresponding to the questions asked on the process (they obviously depends on the process model). The ATL main drawback is that mistakes are generally only detected at runtime.

7 Case Study Extensions

The initial case study allows to introduce MDE concepts and tools and to check that students got a basic understanding of the technologies. Several extensions are proposed to students that they have to develop on their own in order to improve their practice. These extensions are listed hereafter with a short description of their strong points. For each extension, students have to extend the core course realizations: extend the SimplePDL metamodel, add new OCL constraints, update concrete syntaxes, M2M transformation to handle extensions and, eventually the generation of LTL formulae. This work is done in autonomy and is eventually assessed. The stress is also put on how they can validate their work.

E_1: Resources. Every work definition requires a set of resources to be executed.

E_1 is quite easy to implement. SimplePDL metamodel is completed with two new concepts *Resource* and *Allocation* to describe how many occurrences of a resource are needed by an activity. OCL constraints specifies restrictions on the amount of allocated resources with respect to the total amount of resources. Extending concrete syntaxes is easy. The M2M transformation is extended with two rules. The first creates one place for each *Resource*, the number of token is the amount of available resources. The second adds two arcs so that an activity takes the resources when it starts and releases them when it finishes.

E_2: Time constrained. Are activities able to finish in a defined time interval?

The idea is to add an observer on the activities to record if they finish too early, in time or too late. The observer allows to answer both the initial question and the new one: may the process finish? May it finish while respecting time constraints?

E_3: Hierarchical work definitions. A work definition can be split into sub work definitions.

It is mainly a modeling problem. Students have to find how to model hierarchical activities. It also allows to illustrate the composite design pattern. On the ATL side, rule inheritance is useful to avoid code redundancy.

E_4: Suspending a work definition. A work definition may be suspended and its resources released. It can then be resumed if the required resources are available at that time.

This extension is not difficult if we consider that time is not stopped when an activity is suspended. The main interest is that the SimplePDL metamodel is unchanged. Only its semantics (encoded by the translation from process models to Petri nets) changes as an activity may be suspended that leads to new possible scenarios to ensure the process finishes. Only the M2M transformation is concerned by this extension.

E_5: Different possible sets of resources. A work definition can work with different sets of resources (alternative resources).

The difficulty is to find how to represent the "or" between resource sets in Petri nets.

8 Discussions

Modularity of the course Even if the purpose of the course was to present most of the MDE concepts and tools, it is very modular and parts of it can be dropped of. In an alternate version, only one lecture presents the main MDE concepts, then each of the 7 topic (section 3) is done using 2 hours sessions. The first part of each topic is used to explain the concepts and see how tools work. Students are asked to finish topics on their own. Finally, students have a 6 hours session to implement the two first extensions and have to write a short report. Another one is taught in only 8 hours. It consists in an overview of MDE at the end of the M2 level. There is only a 2 hours lecture to present the main concepts of MDE and 6 hours of practical sessions, 2 hours for metamodeling with SimplePDL and PetriNet (OCL is left as homework) and 4 hours for model transformation (focusing on M2M transformation). Concrete syntaxes have already been studied during M1.

Formal verification use case is not a difficulty Even if the course is based on a verification case study. We have experienced that the lack of knowledge in formal verification and model-checking is not an issue. Petri net are indeed an easy to understand formalism and students are able to define its metamodel and propose a translation of SimplePDL model into Petri net. Furthermore, it allows to present model-checking to students and put the focus on the importance of early verification in the development process.

Technical overload The practical work causes some troubles to students. In particular, when making the metamodel evolve, students often do not entail the consequences of their choices and need some help to validate their proposals. Furthermore, all the steps required to verify a model are run manually: running the M2M transformation, running the two M2T transformations to generate the Petri net file and the LTL formulae, running the Tina model-checker. Presenting tools to automate these steps (like Ant or a workflow engine) could favor the adoption. We are also surprised that students are not asking for such tools.

Studying verification does not imply the verification of the study We observed that students are not really concerned about the validity of their development, despite the fact they had to develop verification tools and thus are aware of that. A major problem is that students do not rigorously verify their work (OCL constraints, transformation...). Students only use the example provided in the subject to validate their work whereas they should define many wrong models to ensure that all inconsistencies are caught by the OCL invariants, define several models to test the different aspects of their transformations, etc. In fact students are not really experienced at defining good test cases that provide a significant coverage.

Students' difficulties The students' difficulties we have seen in this course are twofold: one is the necessary ability to find the right level of abstraction, especially making the right choices in the design of language is essential to reduce the accidental complexity generated each time the metamodel changes. The other difficulty comes from the technical environment that may appear complicated to learn, especially the lack of maturity and performance of certain tools, and the proliferation of tools which continues to evolve. We noticed that students encounter in particular one of these difficulties according to their curriculums. While some students are comfortable with a complex technical environment but have difficulty with abstraction, other students easily manipulate abstractions but are struggling to implement them using the MDE tools.

9 Conclusion and Perspectives

In this paper, we have presented how a case study may help in presenting MDE concepts and tools in an attractive way that furthermore make students aware of formal verification technologies that are getting more and more interest in the safety critical transportation systems industry. Despite its verification focus that can be considered as difficult for most students, we have noticed that

it can also been taught to student with no formal background because Petri net are an easy to understand language in its concepts and semantics.

For the future, we strongly believe that an MDE course should be taught at M1 level because it is becoming a very important domain, not only in academy but also in industry. Our main fear about this is to know whether students will have enough background and experience on modeling to be able to handle metamodeling and generative tools.

We also think that MDE is quite close to Software Language Engineering (abstract syntax, concrete syntax, transformation of abstract syntax, etc) and thus it could be interesting to merge these modules to gain on efficiency and stress the important points and favor separation of concerns: abstract syntax, concrete syntax, transformation. We now plan to rely on MDE technologies in our compiler course instead of classical attribute grammar technologies.

References

1. Brosch, P., Kappel, G., Seidl, M., Wimmer, M.: Teaching model engineering in the large. (2009) In: Educators' Symposium @ Models 2009.
2. Gjosaeter, T., Prinz, A.: Teaching model driven language handling. ECEASST (2010) In: Educators' Symposium @ Models 2010.
3. Object Management Group, Inc.: Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT) Specification, version 1.0. (April 2008)

Session du groupe de travail LaMHA

Langages et Modèles de Haut-niveau pour la programmation parallèle, distribuée, de grilles de calcul et Applications

Une bibliothèques vérifiée de squelettes algorithmiques sur tableaux équitablement répartis

Wadoud Bousdira[1] Frédéric Loulergue[1] Julien Tesson[2]

[1] LIFO, University of Orléans, France, Firstname.Lastname@univ-orleans.fr
[2] Kochi University of Technology, Japan, tesson.julien@kochi-tech.ac.jp

Jusqu'à récemment, la rapidité des unités de calcul a été en constante augmentation, et les programmes séquentiels ont profité directement de cette accélération du matériel. Cependant, depuis quelques années maintenant, les constructeurs sont confrontés à des problèmes de dissipation de la chaleur et surtout de consommation d'énergie qui les ont poussés à ne plus augmenter la rapidité des unités de calcul mais à multiplier leur nombre sur une même puce. Les architectures parallèles, jusqu'à récemment confinées aux centres de calcul scientifique, se retrouvent maintenant dans tous les ordinateurs grand public, et jusque dans les téléphones portables.

Si les architectures parallèles sont maintenant partout, il n'en est pas de même pour la programmation parallèle. Le plus souvent axée sur des modèles de programmation non structurés et de relativement bas niveau, la programmation parallèle nécessite des formes plus structurées pour être plus simple d'accès et de ce fait plus répandue. Plusieurs formes de parallélisme structuré ont été proposées dans les années 90 pour pallier les problèmes soulevés par la programmation des architectures parallèles : le parallélisme de données [1], le modèle de programmation parallèle quasi-synchrone [7] et les squelettes algorithmiques [2] par exemple.

Issus des approches de programmation fonctionnelle, les squelettes algorithmiques sont des fonctions d'ordre supérieur implantées en parallèle et fournies au programmeur. Ces fonctions permettent d'encapsuler les détails du parallélisme en proposant au programmeur des primitives correspondant à des motifs de calcul couramment utilisés, que ce soit pour le calcul sur des structures de données ou pour l'agencement de tâches de calcul. Différentes implantations d'un même squelette peuvent exister, permettant d'obtenir des performances optimales pour telle ou telle architecture. De nombreux bibliothèques et langages reposant sur le modèle de programmation par squelettes algorithmiques existent, un panorama récent de ceux-ci est dressé dans [3], auquel on peut ajouter l'*Orléans Skeleton Library* (OSL).

OSL est une bibliothèque de programmation parallèle en C++, méta-programmée pour plus d'efficacité. Nous nous sommes récemment intéressés au prototypage d'OSL à l'aide du langage parallèle fonctionnel *Bulk Synchronous Parallel ML* (BSML). L'objectif de ce prototypage consiste à vérifier l'implantation de ces squelettes en BSML à l'aide de l'assistant de preuve Coq [6]. OSL traite des tableaux répartis de manière quelconque. Dans une première étape, nous considérons des squelettes algorithmiques sur des tableaux équitablement répartis. Ce sous-ensemble correspond aux squelettes de la bibliothèque SkeTo [4] sur les tableaux. Le principe de la vérification est le suivant :

- les squelettes algorithmiques sont implantés directement dans le langage fonctionnel de Coq, en utilisant une axiomatisation de BSML [5] dans Coq,
- une spécification fonctionnelle de ces squelettes est également donnée en Coq,
- nous avons prouvé que ces implantations sont correctes par rapport à cette spécification.

Les implantations compilables et exécutables sur des architectures parallèles sont *extraites* de ce développement Coq. Nous obtenons ainsi des implantations sûres de la bibliothèque de squelettes algorithmiques.

Spécification. Les tableaux équitablement répartis sont spécifiés, pour l'utilisateur de la bibliothèque, sans exhiber de parallélisme. Un tableau est une structure indexée sur laquelle des fonctions d'ordre supérieur sont définies, ici `map` et `zip` (la spécification complète indique également que la taille des tableaux est préservée par ces fonctions) :

Module Type TYPE.
  Parameter distributedArray: **Type** →**Type**.

```
Parameter get: ∀(A:Type)(da: distributedArray A)(position: N)(default:A), A.
Parameter map: ∀(A B: Type)(f:A→B)(da: distributedArray A),
    { da' : distributedArray B | ∀d position, get da' position (f d) = f (get da position d) }.
Parameter zip: ∀(A B C: Type)(f:A→B→C)
 (da1: distributedArray A)(da2: distributedArray B){H: compatible da1 da2},
  { da' : distributedArray C | ∀d1 d2 position,
            get da' position (f d1 d2) = f (get da1 position d1)(get da2 position d2) }.
```

...

La fonction `zip` a un argument supplémentaire par rapport à des versions plus classiques : une preuve que les deux tableaux doivent être "compatibles". Deux tableaux répartis sont compatibles s'ils ont la même répartition. Dans le cas d'une répartition équitable, il suffit de montrer que les deux tableaux sont de même taille (la propriété `compatible` est définie ainsi). L'utilisation des classes de types Coq, permet dans la plupart des cas, de construire la preuve de compatibilité automatiquement, évitant ainsi à l'utilisateur de la donner explicitement.

Implantation. Les tableaux répartis sont implantés comme des vecteurs parallèles (BSML) de listes. Des informations utiles à l'implantation des squelettes peuvent être calculées sur cette représentation, mais elles nécessitent cependant un coût de calcul non négligeable car incluant des communications. Il s'agit de la taille totale du tableau, de la répartition du tableau sur les processeurs et enfin pour chaque processeur, de l'indice global dans le tableau du premier élément détenu par ce processeur. Par conséquent, plutôt que d'être calculées, elles sont mémorisées dans la structure de données des tableaux répartis. De plus, un champ supplémentaire indique que ces informations sont cohérentes entre elles :

```
Record da (A:Type) := make_da {
    da_content: par (list A);
    da_length: N;
    da_distribution: list N;
    da_firstLocalIndex: par N;
    da_coherence: ... }.
```

Le type de tableau équitablement réparti est ensuite défini ainsi :

```
Definition distributedArray (A:Type) := { d : da A | evenlyDistributed d }.
```

Les implantations des squelettes sont partiellement définies par des programmes BSML (en Coq) calculant les quatre premiers champs d'une valeur de type da. Le champ da_coherence est obtenu par preuve, ainsi que le fait que le résultat est équitablement réparti. Pour établir la correction de ces implantations, nous prouvons de plus qu'elles respectent la spécification.

Références

1. Bougé, L. : The Data-Parallel Programming Model : A Semantic Perspective. In : Perrin, G.R., Darte, A. (eds.) The Data Parallel Programming Model. pp. 4–26. LNCS 1132, Springer (1996)
2. Cole, M. : Algorithmic Skeletons : Structured Management of Parallel Computation. MIT Press (1989), available at http ://homepages.inf.ed.ac.uk/mic/Pubs
3. González-Vélez, H., Leyton, M. : A survey of algorithmic skeleton frameworks : high-level structured parallel programming enablers. Software, Practrice & Experience 40(12), 1135–1160 (2010)
4. Matsuzaki, K., Iwasaki, H., Emoto, K., Hu, Z. : A Library of Constructive Skeletons for Sequential Style of Parallel Programming. In : InfoScale'06 : Proceedings of the 1st international conference on Scalable information systems. ACM Press (2006)
5. Tesson, J., Loulergue, F. : A Verified Bulk Synchronous Parallel ML Heat Diffusion Simulation. In : International Conference on Computational Science (ICCS). pp. 36–45. Procedia Computer Science, Elsevier (2011)
6. The Coq Development Team : The Coq Proof Assistant. http ://coq.inria.fr
7. Valiant, L.G. : A bridging model for parallel computation. Comm. of the ACM 33(8), 103 (1990)

SGL: vers la programmation parallèle hétérogène et hiérarchique

Chong Li[1,2] et Gaétan Hains[1,2]

[1] LACL, Université Paris-Est, 94000 Créteil, France
[2] EXQIM SAS, 24 rue de Caumartin, 75009 Paris, France
chong.li@exqim.com gaetan.hains@univ-paris-est.fr

Résumé. Cet article présente un modèle pour la programmation parallèle base de diffusion-constraction et réalisé sous la forme d'un langage impératif simple appelé *scatter-gather language* (SGL). Le concept de SGL est basé sur l'expérience avec la programmation BSP. Les nouvelles caractéristiques de SGL sont motivées par les évolutions de la dernière décennie vers les architectures multi-niveaux et hétérogènes, comme des processeurs multi-coeurs, des accélérateurs graphiques ou des réseaux de routage hiérarchiques. SGL est conforme au modèle Multi-BSP de Valiant, tout en offrant une interface de programmation encore plus simple que les primitives de BSML (bulk-synchronous parallel ML). SGL semble couvrir un grand sous-ensemble des algorithmes BSP tout en centralisant la sémantique des communications. Comme tous les systèmes inspirés de BSP, il permet l'équilibrage de charge systèmatique et des performances prévisibles, portables et évolutives.

1 Introduction

Parallel programming and data-parallel algorithms have been the main techniques supporting high-performance computing for many decades. Like all non-functional properties of software, the conversion of computing resources into scalable and predictable performance involves a delicate balance of abstraction and automation with semantic precision. With these goals in mind we apply the notion of *explicit processes* because parallel speed-up cannot be part of programming semantics without an explicit notion of the number of physical parallel processors. A language for parallel algorithms must not express a function from computation events to physical units (which may not be injective) but just the inverse.

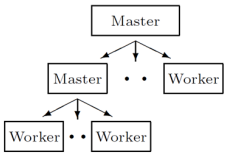
For bridging the gap between parallel systems and parallel algorithms, a major conceptual step was taken by L. Valiant [Val90] who introduced his Bulk-Synchronous Parallel (BSP) model. Inspired by complexity theory's PRAM model of parallel computers, Valiant proposed that parallel algorithms be designed and measured by accounting not only for the classical balance between time and parallel space (the number of processors) but also for communication and synchronization. BSP is thus a *bridging model* relating parallel algorithms to hardware architectures.

During the 2000's decade it became obvious that models like BSP should adapt or generalize to the new variety of heterogeneous and hierarchical architectures. Yet, programming simplicity and performance portability should be retained as much as possible. With these goals in mind, Valiant introduced Multi-BSP [Val11] a multi-level variant of the BSP model and showed how to design scalable and predictable algorithms for it. The main new feature of Multi-BSP is its hierarchical nature with nested levels that correspond to physical architectures' natural layers. In that sense it preserves the notion of explicit processes and, as we will show in this paper with our SGL design, allows to solve three pending problems with BSP programming: 1) The flat nature of BSP is not easily reconciled with divide-and-conquer parallelism [Hai98]. 2) BSP was designed natively to scale out with the number of processors, but parallelism is also used to scale up the computing power nowadays. 3) We do not know how to design algorithms for nested-level systems with the original flat BSP model.

2 SGL model

The SGL model defines a set of sequential processors composed of a computation element ("core") and memory unit. The processors are arranged in a tree structure with the root being called a "master" and its children that are either masters themselves or "leaf-workers". The number of children is not limited so that the BSP/PRAM concept of a flat p-vector of processors is easily simulated in SGL. Different forms are possible:

1. Only 1 worker (without master) → a sequential machine
2. Master + workers → a parallel machine e.g. BSP
3. A hierarchical machine (Figure) of any shape



An SGL *program execution* is a sequence of *super-steps*. The initial computing data and final result can be either distributed in workers or centralized in the *root-master*. Each *super-step* is composed of 4 phases: 1) a scatter communication phase initiated by the master; 2) an asynchronous computation phase performed by the children; 3) a gather communication phase centered on the master; 4) a local computation phase on the master.

SGL's main goal of expressing parallel algorithms requires a precise notion of the execution time for a program. We give here the mathematical form of this *cost model*:

$$Cost_{Master} = \max_{i=1..p}(Cost_{child_i}) + w_0 \times c_0 + k_\downarrow \times g_\downarrow + k_\uparrow \times g_\uparrow + 2l$$
$$Cost_{Worker} = w_i \times c_i$$

which clearly cover the possibility of a heterogeneous architecture. The definition of parameters can be found in [LH11].

SGL is an imperative language for which we have defined a *big-step* [Win93] operational semantics. The complex definition can be found in [LH11]. Here we show only the semantics of vector expressions:

$$\frac{\langle w,\ \sigma\rangle\rightarrow<v_1,v_2,\ldots v_\ell> \quad \forall_{i=1..numChd}\langle \overrightarrow{V}_i{:}\sim v_i,\ \sigma\rangle\rightarrow\sigma'_i}{\langle \textbf{scatter}\ w\ \textbf{to}\ \overrightarrow{V},\ \sigma\rangle\rightarrow\sigma'} \qquad \frac{\langle \widetilde{W}{:}\sim<\overrightarrow{V}_1,\overrightarrow{V}_2,\ldots\overrightarrow{V}_{numChd}>,\ \sigma\rangle\rightarrow\sigma'}{\langle \textbf{gather}\ \overrightarrow{V}\ \textbf{to}\ \widetilde{W},\ \sigma\rangle\rightarrow\sigma'}$$

$$\frac{\langle v,\ \sigma\rangle\rightarrow<n_1,n_2,\ldots n_\ell> \quad \forall_{i=1..numChd}\langle X_i{:}\sim n_i,\ \sigma\rangle\rightarrow\sigma'_i}{\langle \textbf{scatter}\ v\ \textbf{to}\ X,\ \sigma\rangle\rightarrow\sigma'} \qquad \frac{\langle \overrightarrow{V}{:}\sim<X_1,X_2,\ldots X_{numChd}>,\ \sigma\rangle\rightarrow\sigma'}{\langle \textbf{gather}\ X\ \textbf{to}\ \overrightarrow{V},\ \sigma\rangle\rightarrow\sigma'}$$

$$\frac{\forall_{i=1..numChd}\langle c,\ \sigma_i\rangle\rightarrow\sigma'_i}{\langle \textbf{pardo}\ c,\ \sigma\rangle\rightarrow\sigma'} \qquad \frac{\langle numChd{=}0,\ \sigma\rangle\rightarrow\textbf{true} \quad \langle c_2,\ \sigma\rangle\rightarrow\sigma'}{\langle \textbf{if master}\ c_1\ \textbf{else}\ c_2,\ \sigma\rangle\rightarrow\sigma'} \qquad \frac{\langle numChd{=}0,\ \sigma\rangle\rightarrow\textbf{false} \quad \langle c_1,\ \sigma\rangle\rightarrow\sigma'}{\langle \textbf{if master}\ c_1\ \textbf{else}\ c_2,\ \sigma\rangle\rightarrow\sigma'}$$

3 Experimentation

We have performed experiments to observe how the measured computation time of an SGL algorithm compares with the cost model's prediction. The overall system is a SGI Altix ICE 8200 with 2 IRUs. We build a 2-level SGL abstract machine to represent the physical machine:

Unit	Children	Communication
Root-master	Nodes	InfiniBand
Node-master	Cores	Front-Side Bus
Worker	N/A	N/A

We have tested SGL variants of some of the most important basic parallel algorithms: parallel reduction, parallel prefix reductions also called scan and a sorting algorithm. For each one we wrote a SGL algorithm, implemented it by hand using the MPI/OpenMP operations, and then compared the model's predicted vs observed run time for increasing data sizes. The three tests illustrate SGL's relative programming simplicity on a relevant set of algorithms, convincing proof of the cost model's quality and initial evidence that SGL programming has no intrinsic cost overhead. Figure 3 shows the predictions and the measurements for SGL parallel scan. Others can be found in [LH11].

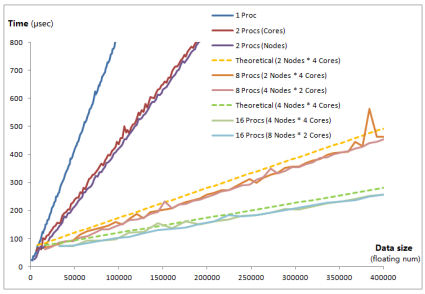


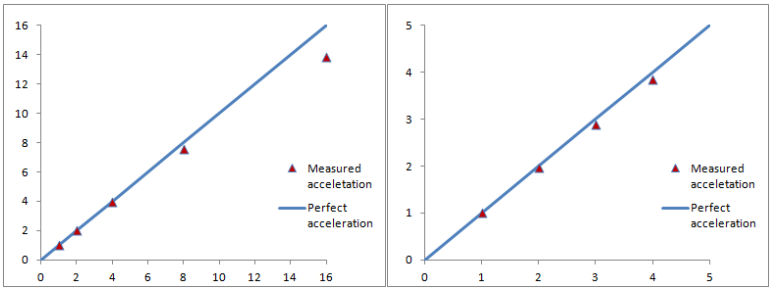
Fig. 1: Predicted and actual execution times for SGL Scan

We tested speed-up and efficiency using the reduction algorithm. The formulas we used are:

$$Speedup = \frac{ExecutionTime_1}{ExecutionTime_p} \quad \& \quad Efficiency = \frac{Speedup_p}{p}$$

where p is the number of used nodes (resp. cores).

Our measurements show that core-level scale-out is more efficient than node-level scale-out. The theoretical explanation is that the cost of node-level communication is more expensive than core-level communication. Figures 2a and 2b illustrate the two tables from the point of view of acceleration.



(a) Node-level acceleration (b) Core-level acceleration

Fig. 2: Accelerations of SGL parallel scan

4 Conclusion

BSP's advantages for parallel software can be enhanced by the recursive hierarchical and heterogeneous machine structure of SGL, while simplifying the programming interface even further by replacing point-to-point messages with logically centralized communications. Our initial experiments with language definition, programming and performance measurement show that SGL combines a clean semantics, simplified programming of BSP-like algorithms and dependable performance measurement. One remaining open problem is the implicit treatment of horizontal communication needed for operations like sample-sort or bucket-sort [SCO+92]. We are confident that the large body of knowledge on algorithms for cost-explicit models will yield many solutions to this problem. It is our opinion that SGL's lesser expressive power with respect to message-passing structures is not only necessary for software reliability but also coherent with one of the most basic concept in computing, namely recursive structures.

References

[Hai98] Gaétan Hains. Subset synchronization in BSP computing. In H. R. Arabnia, editor, *PDPTA'98 International Conference on Parallel and Distributed Processing Techniques and Applications*, volume I, pages 242–246, Las Vegas, July 1998.

[LH11] C. Li and G. Hains. A simple bridging model for high-performance computing. In *(HPCS2011) International Conference on High Performance Computing and Simulation*, Istanbul, Turkey, July 2011. IEEE Computer Society.

[SCO+92] Hanmao Shi, Resewrch Council, Grant Number Ogp, Jonathan Schaeffer, and Jonathan Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14:361–372, 1992.

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33:103–111, August 1990.

[Val11] Leslie G. Valiant. A bridging model for multi-core computing. *J. Comput. Syst. Sci.*, 77(1):154–166, 2011.

[Win93] Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT Press, Cambridge, MA, USA, 1993.

CAPH : An actor-based language for implementing stream-processing applications on reconfigurable hardware

Jocelyn Sérot, François Berry, Sameer Ahmed
Institut Pascal, UMR 6602 CNRS / Université Blaise Pascal, Clermont-Ferrand
{*jocelyn.serot,francois.berry,sameer.ahmed*}*@lasmea.univ-bpclermont.fr*

I. INTRODUCTION

Digital circuits based upon reconfigurable logic (FPGAs) offer large opportunities for exploiting fine grain parallelism. But, in the current state-of-the art, programming FPGAs essentially remains an hardware-oriented activity, relying on dedicated Hardware Description Languages (such as VHDL or Verilog). Using these languages requires expertise in digital design and this practically limits the applicability of FPGA-based solutions.

We introduce CAPH, a high-level, domain-specific language (DSL) dedicated to the implementation of stream-processing applications on reconfigurable harware. By *stream-processing* application, we mean applications operating on the fly on continuous streams of data, such as real-time image processing for example. These applications often require a computing power which still beyond the capacity of general purpose processors (GPPs) but generally exhibit a large amount of fine-grained parallelism, making them good candidates for implementation on FPGAs.

II. THE CAPH LANGUAGE

CAPH is based upon the *actor* / *dataflow* model of computation : an application is described as a network of autonomous processing elements (actors) exchanging tokens through unidirectional channels (FIFOs).

The behavior of actors is specified using a set of *generalized transition rules*, where a each rule consists of a set of *patterns*, involving inputs and possibly local variables, and a set of *expressions*, describing modifications of outputs and local variables. Tokens circulating on channels and manipulated by actors are divided into *data tokens* (carrying actual values, such as pixels for example) and control tokens (acting as structuring delimiters). This approach allows fine grain processing (down to the pixel level) to be expressed without the need of global control and/or synchronization.

The actor described in Fig. 1, for example, computes the sum of a list of values. Given the input stream `<1 2 3> <4 5 6>`, for example, it will produce the values `6, 15`. For this it uses two local variables : An accumulator `s` and a state variable `st`. The latter indicates whether the actor is actually processing a list or waiting for a new one to start. In the first state, the accumulator keeps track of the running sum. The first rule can be read as : when waiting for a list (`st=S0`) and reading the start of a new one (`a='<`), then reset accumulator (`s:=0`) and start processing (`st=S1`). The second rule says : When processing (`st=S1`) and reading a data value (`a='v`), then update accumulator (`s:=s+v`). The last rule is fired at the end of the list (`a='>`); the final value of the accumulator is written on output `c`.

```
actor suml ()
  in (a: signed<8> dc)
 out (c: signed<16>)
 var st: {S0,S1}=S0
 var s : signed<16>
 rules (st,a,s)-> (st,c,s)
   (S0, '<, _) -> (S1, _, 0)
 | (S1, 'v, s) -> (S1, _, s+v)
 | (S1, '>, s) -> (S0, s, _)
```

Figure 1. An actor computing the sum of values along lists

The structure of the actor network is described using a strongly-typed, polymorphic, higher-order, purely functional language called FGN (Functional Graph Notation) [1]. By encoding dataflow dependencies using functional equations, this language allows complex networks to be described in a very abstract concise manner. For example, the network depicted in Fig. 2, which computes $f(x) = (x+1) \times (x-1)$ for each element x of its input stream `i`, can be described with the following equations :

```
net (x,y) = dup i
net o = mul (inc x, dec y)
```

where `f x` denotes application of function `f` to argument `x`, `(x,y)` denotes a pair of values and the `net` keyword serves to introduce bindings.

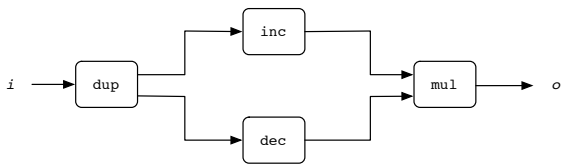


Figure 2. A dataflow process network

III. The CAPH toolset

The current tool chain supporting the CAPH language is sketched on Fig. 3. It comprises a graph visualizer, a reference interpreter and compiler producing both SystemC and synthetizable VHDL code.

The **graph visualizer** produces representations of the actor network in the `.dot` format for visualisation with the GRAPHVIZ suite of tools.

The **reference interpreter** is based on the fully formalized semantics of the language [3], written in axiomatic style. Its role is to provide reference results to check the correctness of the generated SystemC and VHDL code. It can also be used to test and debug programs, during the first steps of application development (in this case, input/output streams are read from/written to files). Several tracing and monitoring facilities are provided. For example, it is possible to compute statistics on channel occupation or on rule activation.

The **compiler** is the core of the system. It relies on an *elaboration phase*, turning the AST into a target-independent intermediate representation, and a set of dedicated back-ends. The intermediate representation (IR) is basically a process network in which each process is represented as a generalized finite-state machine (GFSM) and channels as unbounded FIFOs. Two back-ends are provided : the first produces cycle-accurate SystemC code for simulation and profiling, the second VHDL code for hardware synthesis. Execution of the SystemC code provides informations which are used to refine the VHDL implementation (for example : the actual size of the FIFOs used to implement channels).

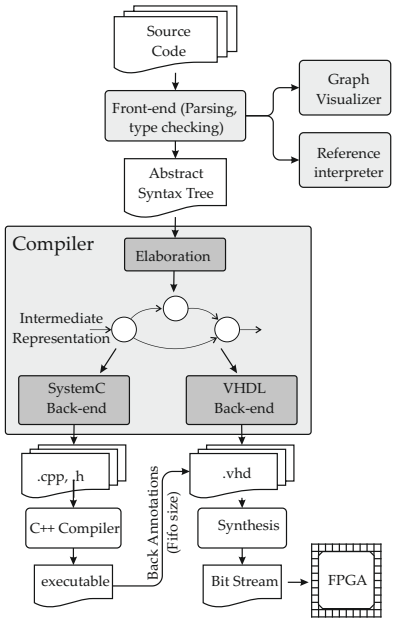


Figure 3. The CAPH toolset

IV. Experimental results

We have implemented several real-time image processing applications using a smart-camera integrating an FPGA board and an image sensing device as the target platform [2], [4]. In this configuration, two dedicated VHDL processes provide interfacing of the generated actor network to the physical I/O devices (camera and display). The VHDL code produced by the CAPH compiler is compiled and downloaded to the FPGA using the Altera Quartus toolset.

Table I reports the number of lines of code (LOC) both for the CAPH source code and the generated VHDL for a motion-detection application involving eight actors and operating on the fly on video streams of $512 \times 512 \times 8$ bit images at 15 FPS, with a clock frequency of 150 MHz. The tenfold factor observed between the volume of the CAPH source code and the VHDL code (automatically) generated by the compiler gives an idea of the gain in productivity offered by our approach. Moreover, this gain is not obtained at the price of performance since a hand-crafted VHDL version of this application does not offer significant gain both in ressource usage nor in performance.

Actor	CAPH	Generated VHDL
asub	7	78
d1f	15	124
thr	7	68
hproj	11	91
vproj	18	136
peaks	17	134
win	17	134
vwin	20	190
network, decls	13	285
Total	125	1240

Table I
Line of Code (LOC) for a motion detection application

The CAPH language reference manual and toolset are available from http://wwwlasmea.univ-bpclermont.fr/Personnel/Jocelyn.Serot/caph.html.

References

[1] J. Sérot. The semantics of a purely functional graph notation system. 9th Symposium on Trends in Functional Programming, 2008.

[2] J. Sérot, F. Berry, S. Ahmed. Implementing stream-processing applications on FPGAs : a DSL-based approach. 21st International Conference on Field Programmable Logic and Applications, Chania, Crete, 2-5 sep 2011

[3] J. Sérot The CAPH Reference Manual. http://wwwlasmea.univ-bpclermont.fr/Personnel/Jocelyn.Serot/caph/caph-lrm.pdf

[4] J. Sérot, F. Berry, S. Ahmed. Caph : A language for implementing stream-processing applications on FPGAs. In Embedded Systems Design with FPGAs, P. Athanas, D. Pnevmatikatos, N. Sklavos eds., Springer, 2012 (to appear: ISBN 978-1-4614-1361-5).

SPOC :Programmation GPGPU avec OCaml

Mathias Bourgoin
Emmanuel Chailloux et Jean-Luc Lamotte

Laboratoire d'Informatique de Paris 6 (LIP6 - UMR 7606)
Université Pierre et Marie Curie (UPMC - Paris 6)
Sorbonne Universités
4 place Jussieu, 75005 Paris, France
{Mathias.Bourgoin,Emmanuel.Chailloux,Jean-Luc.Lamotte}@lip6.fr

Nous avons développé une bibliothèque OCaml, appelée SPOC [1] (Stream Processing with OCaml), qui permet la programmation GPGPU (General Purpose on Graphics Processing Units). Elle consiste en une extension de syntaxe pour OCaml associée à une bibliothèque d'exécution. L'extension permet la déclaration de noyaux GPGPU externes utilisables depuis un programme OCaml, tandis que la bibliothèque offre les fonctions pour exploiter ces noyaux ainsi que les données nécessaires pour leur exécution. SPOC offre une abstraction sur les deux environnements de développement, OpenCL et Cuda, en les unifiant en une seule bibliothèque. Par ailleurs, SPOC abstrait les données avec un type spécifique vecteur, et gère automatiquement leurs transferts entre dispositifs.

Abstraire les environnements de programmation GPGPU Comme il existe différents environnements de programmation GPGPU, il peut être difficile de garantir la portabilité ainsi que l'efficacité. Cuda et OpenCL proposent différentes façons d'exprimer les noyaux de calcul, de les exécuter ou de manipuler la mémoire sur les GPGPU. SPOC unifie ces deux environnements de développement, permettant d'exécuter un programme avec tout type d'architecture GPGPU, séparément ou conjointement. Cela implique qu'elle n'offre que les fonctions partagées par les deux environnements de développement, tandis qu'elle utilise leurs spécificités dans son cœur pour offrir les meilleures performances. Cela permet aux programmes SPOC de fonctionner sur les architectures avec de multiples dispositifs GPGPU et d'utiliser conjointement des dispositifs constitués de plusieurs appareils GPGPU gérés par des environnements de développement incompatibles.

Abstraire les transferts Le *Stream Processing* n'offre de hautes performances que pour un grand rapport de quantité de calcul sur la taille des données, d'autre part, pour exploiter les nombreuses unités de calcul des appareils GPGPU actuels, il est (souvent) nécessaire d'utiliser d'importants jeux de données. SPOC introduit le type *vector* pour représenter ces données. Basés sur les *bigarrays* d'OCaml, ils sont facilement interopérables avec C et donc avec Cuda et OpenCL. Pour obtenir de bonnes performances avec des programmes complexes, il est important d'optimiser l'ordonnancement des transferts. L'optimisation la plus simple consiste à réduire le nombre de transferts. Les noyaux de calcul sont des procédures qui modifient par effets de bord la mémoire globale du GPGPU. Il est, ainsi, difficile de savoir quelle valeur contient le résultat des

1. http ://www.algo-prog.info/spoc/

calculs. Une solution pour proposer des transferts automatiques est de transférer tous les vecteurs nécessaires à l'exécution d'un noyau sur l'appareil GPGPU puis, de tous les ramener en mémoire CPU une fois le calcul terminé. Ceci assure que chaque vecteur modifié par le calcul est accessible depuis le CPU mais peut impliquer des transferts inutiles. Notre alternative est de déplacer les données à la demande. Les vecteurs de SPOC enregistrent leur position courante (CPU/ GPGPU). Lorsqu'on lance un noyau, nous vérifions si les données sont déjà sur l'appareil GPGPU. Si besoin, elles sont transférées. Après le calcul, nous ne déplaçons pas les données. Lorsqu'un vecteur est lu ou écrit par le CPU, SPOC vérifie à nouveau sa position, le transférant à nouveau si nécessaire. Si elle ne rapatrie pas les données dans la mémoire CPU et les maintiens en mémoire GPGPU, la bibliothèque SPOC pourrait rapidement remplir la mémoire du dispositif GPGPU. Ceci est résolu en liant les vecteurs au gestionnaire de mémoire d'OCaml et à son ramasse-miette (GC). Ainsi, quand les vecteurs deviennent inutiles au programme, leur espace mémoire est libéré, que ce soit dans la mémoire CPU ou dans la mémoire globale des GPGPU. Le GC peut être déclenché par le programme OCaml mais aussi, si lors d'un transfert, l'espace mémoire est insuffisant.

Utilisation de noyaux externes SPOC permet l'utilisation de noyaux de calcul externes, écrits en assembleur Cuda (ptx) ou C99 pour OpenCL. Elle offre une extension de syntaxe à OCaml pour déclarer ces noyaux externes d'une manière similaire à la déclaration de fonctions externes C en OCaml. Cette extension permet la vérification du type des arguments du noyau lors de la compilation et ainsi de réduire les risques d'erreurs difficiles à déboguer durant l'exécution. Lors du lancement d'un noyau, SPOC va automatiquement rechercher le fichier et le noyau correspondant, le compiler et l'exécuter avant de reprendre le cours de l'exécution du programme OCaml. La compilation est effectuée dynamiquement, en s'appuyant sur l'environnement de programmation Cuda ou OpenCL lié au dispositif GPGPU chargé d'exécuter le noyau.
L'exploitation de noyaux externes permet en particulier d'utiliser les bibliothèques GPGPU optimisées existantes. Nous avons, dans ce cadre effectué un binding avec la bibliothèque de calcul numérique Cublas pour l'environnement de programmation Cuda.

SPOC permet d'exploiter les dispositifs GPGPU avec le langage OCaml. Cette bibliothèque unifie les deux environnements de développement, Cuda et OpenCL. Elle permet ainsi d'utiliser indifféremment et conjointement des dispositifs compatibles avec l'un ou l'autre de ces environnements de développement. Elle offre, de plus, une gestion des transferts entre dispositifs automatique. En s'appuyant sur le gestionnaire de mémoire automatique d'OCaml et sur son ramasse-miette elle limite les transferts tout en limitant les risques de remplir la mémoire des dispositifs GPGPU. Ainsi, elle simplifie la programmation GPGPU, tout en apportant davantage de sûreté et de portabilité. Elle [2] offre, enfin, une bonne base pour le développement futur d'abstractions supplémentaires comme des squelettes de parallélisme ou des *DSL*.

2. M. Bourgoin, E. Chailloux, and J. L. Lamotte. Spoc : GPGPU Programming Through Stream Processing with OCaml. In *Parallel Processing Letters*,2012.

Session du groupe de travail LTP

Langages, Types et Preuves

Construction d'espaces topologiques pour la représentation d'objets musicaux

Antoine Spicher

LACL – Université Paris-Est Créteil
antoine.spicher@u-pec.fr

Cet article présente les premiers résultats issus du rapprochement récent de deux domaines de recherche : le *calcul spatial* et l'*informatique musicale*. Il s'agit d'un résumé de l'article [1].

1 Contexte

La nature algébrique des objets musicaux a été très tôt reconnue comme une caractéristique importante des représentations musicales : définition des tempéraments, analyse des structures canoniques, etc. Ces notions algébriques ont permis l'étude des propriétés combinatoires et la classification de nombreuses structures musicales. Plus récemment, un point de vue nouveau sur ces structures a émergé, considérant des représentations de nature topologique [2] ou géométrique [3].

Nous poursuivons dans cette voie en proposant l'application des principes du *calcul spatial* aux représentations musicales. Le *calcul spatial* est un domaine émergent de l'informatique [4,5] soulignant l'importance de l'espace dans les calculs : les calculs sont spécifiés comme un ensemble d'interactions entre des données « voisines » dans un certain espace. Cet espace abstrait peut représenter des contraintes *physiques* (une distribution spatiale des données, une localisation des ressources, etc.) ou *logiques* (inhérentes au problème à résoudre). Le calcul spatial est notamment à l'origine d'une forme de programmation originale, la *programmation spatiale*, reposant de façon centrale sur l'espace comme moyen, ressource, entrée et sortie d'un calcul.

Les représentations classiques en informatique musicale étant souvent de nature spatiale, nous avons appliqué dans [1] le paradigme du calcul spatial pour exprimer de manière élégante différentes problématiques musicales dans un langage de programmation spatiale, MGS [6].

2 Quelques résultats en informatique musicale

Dans [1], nous nous sommes intéressés à deux sujets en particulier : la représentation spatiale d'un ensemble d'accords et l'étude des *séries tous-intervalles*.

Auto-assemblage d'accords. Le concept musical de *tonalité* peut être interprété de manière topologique en considérant une structure combinatoire fondée sur une définition simpliciale des accords triadiques. L'espace ainsi engendré est un ruban de Möbius aux propriétés topologiques intéressantes, mises en avant dans un premier temps par A. Schoenberg puis reprises et développées par G. Mazzola dans une formalisation catégorique de la musique [2].

Nous montrons que ce type de construction, faite jusqu'ici à la main, peut facilement être automatisée et s'étendre à tout ensemble arbitraire d'accords à travers un processus d'auto-assemblage, l'idée générale étant de laisser *réagir* les accords entre eux à la façon de molécules chimiques dans une solution. L'expression extrêmement concise de cet algorithme en MGS est générique et indépendante de la dimension des accords, permettant la génération systématique d'espaces. Nous avons utilisé cet algorithme pour deux problématiques musicales. La première a consisté à étudier la topologie de la représentation de la tonalité à partir des accords à 4 sons. Le volume généré s'avère être toroïdal avec pour bord un réseau hexagonal. La seconde a consisté à associer un espace à la progression harmonique du prélude Op.28 No.4 de F. Chopin. L'étude de cet espace permet une analyse des choix de composition de F. Chopin liés à une propriété musicale importante, appelée *minimum voice-leading*.

Énumération et classification des séries tous-intervalles (STI). Il s'agit d'un problème connu depuis longtemps à la fois en musique (dès 1928 dans la *Suite Lyrique* d'A. Berg) mais également en programmation par contrainte où il fait partie des 50 problèmes de la CSPLib.

Nous utilisons ce problème pour illustrer une méthodologie de l'expression « spatiale » d'algorithmes de recherche. Il s'agit de construire un espace topologique structuré où l'objet recherché peut être exprimé en termes spatiaux uniquement. L'espace utilisé pour cette recherche permet également de classer de façon élégante les STI suivant leurs caractéristiques topologiques. L'intérêt musicologique d'une telle classification des STI est d'autant plus important qu'elle permet de retrouver les STI ayant un contenu musical particulier [7]. Nous montrons que notre classification permet d'exhiber les STI contenant consécutivement les notes d'une gamme particulière. L'analyse de la combinatoire de cet espace nous a également permis d'étudier la géométrie des STI par rapport à différentes opérations algébriques très largement utilisées en informatique musical.

3 Conclusion et perspectives

Nous avons présenté dans [1] la manipulation d'objets musicaux à la lumière du calcul spatial avec d'une part la construction d'un espace représentant la structure de la gamme diatonique, et d'autre part l'énumération et la classification des STI. Sans révolutionner l'informatique musicale, ces premiers résultats sont prometteurs par la généricité de l'approche spatiale. Ainsi, la construction d'un espace associé à un ensemble d'accords a permis de développer une nouvelle approche des *Tonnetz* (*i.e.*, réseaux de notes) dont nous poursuivons l'étude pour l'aide à l'analyse musicale. La formalisation topologique des STI repose sur une recherche de chemins hamiltoniens ; l'hamiltonicité est une notion très présente chez certains compositeurs contemporains (*e.g.*, *Hamiltonian Cycle : Saxophone* de R. Morris, *Maximum Changes* de M. Winter, *Corale #4, preludio for cello and string orchestra* de G. Albini) ou pour exprimer des contraintes musicales.

Nous pensons que ces travaux préliminaires montrent l'intérêt d'un outil permettant la conception et la construction systématiques des espaces abstraits utilisés dans les analyses musicales. Ces espaces abstraits sont définis de manière algébrique, ce qui les rend bien plus adaptés à la programmation que les approches fondées sur des ensembles de points (*point set topology*). Enfin, l'un des atouts majeurs de cette approche est l'expressivité et la concision de la formulation des algorithmes. Ainsi, une seule règle de réécriture MGS a été suffisante pour construire l'espace des accords et un motif MGS a suffi pour spécifier une propriété musicale complexe comme les STI.

Remerciements

L'auteur tient à remercier L. Bigo et O. Michel du LACL qui ont participé activement à travaux, ainsi que l'équipe *Représentations Musicales* de l'IRCAM, en particulier M. Andreatta et J.-L. Giavitto pour leur clairvoyance et les nombreuses discussions sur l'utilisation du calcul spatial en informatique musicale.

Références

1. Bigo, L., Giavitto, J.L., Spicher, A.: Building spaces for musical objects. In: Mathematics and Computation in Music. Communications in Computer and Information Science. Springer (2011)
2. Mazzola, G., et al.: The topos of music: geometric logic of concepts, theory, and performance. Birkhäuser (2002)
3. Tymoczko, D.: The geometry of musical chords. Science **313**(5783) (2006) 72
4. De Hon, A., Giavitto, J.L., Gruau, F., eds.: Computing Media and Languages for Space-Oriented Computation. Number 06361 in Dagsthul Seminar Proceedings (3-8 september 2006)
5. SCW: the "Spatial Computing Workshops" series and related events. List on the "Spatial Computing Home Page" `http://www.spatial-computing.org/doku.php?id=events:start` (accessed in january 2011)
6. Giavitto, J.L., Spicher, A.: Topological rewriting and the geometrization of programming. Physica D **237**(9) (jully 2008) 1302–1314
7. Riotte, A., Mesnage, M. : Formalismes et modèles musicaux. Delatour (2006)

Une intégration à base de traces des prouveurs SAT et SMT dans l'assistant de preuve Coq ⋆

M. Armand[1], G. Faure[2], B. Grégoire[1], C. Keller[2], L. Théry[1], and B. Werner[2]

[1] INRIA Sophia-Antipolis {Michael.Armand,Benjamin.Gregoire,Laurent.Thery}@inria.fr
[2] INRIA - Saclay–Île-de-France at LIX, École Polytechnique
{Germain.Faure,Chantal.Keller,Benjamin.Werner}@inria.fr

1 Introduction

Un utilisateur d'un assistant de preuve comme Coq [4] part d'un ensemble de définitions afin d'en dériver automatiquement ou interactivement tout un ensemble de conséquences. Il est donc tout à fait crucial pour l'utilisateur d'avoir une confiance presque absolue dans la correction de ces dérivations. C'est pourquoi les développeurs d'assistants de preuve attachent un soin tout particulier à l'architecture de leur système afin d'en favoriser la correction. Par exemple, dans Coq, le code critique est relativement restreint et est presque intégralement concentré dans l'implémentation de l'algorithme de typage.

Dans ce contexte, pour profiter, sans perte de confiance, de la puissance d'automatisation proposée par les prouveurs SAT et SMT dans un assistant de preuve comme Coq, il y a deux possibilités : soit on construit de toutes pièces un prouveur certifié ; soit on utilise de façon externe un prouveur existant qui est capable de générer une trace d'exécution, et on construit seulement à l'intérieur de l'assistant de preuve un vérificateur certifié de ces traces. Les deux approches ont leurs avantages et inconvénients. L'approche prouveur certifié, suivie par exemple par [7], est de loin la plus séduisante mais si on vise un prouveur relativement efficace elle nécessite de prouver un code bien plus élaboré que celui d'un simple vérificateur de traces. En contre-partie, les traces sont de gros objets et il est loin d'être évident que l'assistant de preuve puisse dans un temps et un espace mémoire raisonnables vérifier de tels objets. Dans cette présentation, nous suivons l'approche à base de traces et nous allons évoquer les ingrédients qui ont rendu possible l'intégration efficace de prouveurs SAT et SMT à l'intérieur du système Coq. Pour une présentation plus détaillée, le lecteur peut consulter [2].

2 Intégration des prouveurs SAT

Du point de vue de l'intégration à base de traces des prouveurs SAT, la situation est quasi idéale. Il existe en effet un consensus sur le format de trace qu'implémente à quelques variantes près la quasi totalité des prouveurs existants fournissant des traces. La brique de base d'un tel format est la règle de résolution entre deux clauses

$$\frac{v \vee C \qquad \bar{v} \vee D}{C \vee D}$$

Une trace sera ainsi composée de listes de séquences de résolution. Chaque listejustifie l'addition d'une nouvelle clause : la liste des résolutions explique pourquoi la nouvelle clause est une conséquence logique des clauses déjà présentes.

Vérifier efficacement de telles traces dans un cadre purement fonctionnel s'est révélé difficile. Pour contourner ces difficultés, on a introduit dans le système Coq deux traits impératifs : les entiers modulaires natifs et les tableaux persistants [3]. Ces deux additions permettent une représentation concise en mémoire des clauses et leur manipulation efficace avec ce nouveau cadre quasi-impératif.

⋆. Ce travail a été en partie supporté par l'ANR DECERT.

3 Intégration des prouveurs SMT

Grossièrement l'exécution d'un prouveur SMT peut-être vue comme un dialogue entre un prouveur SAT qui constitue le cœur du SMT et un ensemble de composants dédiés à chaque théorie spécifique du problème initial. Le prouveur SMT essaie de construire un modèle pour la partie propositionnelle du problème. Ce modèle est ensuite envoyé à chaque composant qui essaie de le réfuter en trouvant un lemme de leur théorie qui contredit le modèle proposé. Si aucun composant n'est capable de générer un tel lemme, la formule initiale est déclarée satisfiable. Si un tel lemme est produit, il est passé au prouveur SAT qui est appelé de nouveau pour générer un modèle alternatif qui satisfasse aussi le nouveau lemme. S'il y réussit, on recommence le processus, autrement la formule est déclarée insatisfiable.

Une trace d'un prouveur SMT sera donc a priori une extension de celle d'un prouveur SAT où en plus apparaîtront les lemmes de théorie générés ainsi que leur justification. Il n'y a malheureusement actuellement pas de consensus sur un format standard de trace SMT. Notre architecture propose un cadre général et modulaire pour la construction d'un vérificateur de telles traces. Il n'en reste pas moins vrai que tout un travail spécifique est nécessaire à l'intégration d'un SMT particulier. Dans notre cas, on s'est intéressé au prouveur veriT [6] qui a le grand avantage de fournir une trace détaillée. Dans ce travail d'intégration, une attention toute particulière a été apportée à transformer la trace générée par le prouveur SMT avant de l'envoyer à Coq. De nombreux pré-calculs sont mis en œuvre pour simplifier la tâche du vérificateur. De plus, à l'intérieur de Coq, le vérificateur de traces travaille sur une structure de données stratifiée et avec partage maximum. Ceci permet d'accélérer la vérification (on peut par exemple utiliser l'égalité de pointeur) mais aussi l'espace mémoire nécessaire pour vérifier une trace.

4 Conclusion

Les premiers résultats de cette intégration sont encourageants. Ils laissent espérer que très prochainement on sera capable de profiter pleinement de toute l'automatisation proposée par les prouveurs SMT à l'intérieur de Coq. Beaucoup de problèmes restent à résoudre comme l'intégration des quantificateurs, la traduction d'une logique riche comme celle de Coq vers une logique plus faible comme celle des SMT, mais ils ne semblent pas insurmontables.

Références

1. SMT-LIB, `http://www.smtlib.org`
2. Armand, M., Faure, G., Grégoire, B., Keller, C., Théry, L., Werner, B. : A modular integration of sat/smt solvers to coq through proof witnesses. In : CPP. LNCS, vol. 7086, pp. 135–150. Springer (2011)
3. Baker, H.G. : Shallow Binding Makes Functional Arrays Fast. ACM SIGPLAN notices 26, 1991–145 (1991)
4. Bertot, Y., Castéran, P. : Interactive Theorem Proving and Program Development, Coq'Art : The Calculus of Inductive Constructions. Springer (2004)
5. Böhme, S., Weber, T. : Fast LCF-Style Proof Reconstruction for Z3. In : Kaufmann, M., Paulson, L.C. (eds.) ITP. Lecture Notes in Computer Science, vol. 6172, pp. 179–194. Springer (2010)
6. Bouton, T., de Oliveira, D., Déharbe, D., Fontaine, P. : veriT : An Open, Trustable and Efficient SMT-Solver. In : CADE. LNCS, vol. 5663, pp. 151–156. Springer (2009)
7. Lescuyer, S., Conchon, S. : Improving Coq Propositional Reasoning Using a Lazy CNF Conversion Scheme. In : FroCos. LNCS, vol. 5749, pp. 287–303. Springer (2009)

Formalisation de la méthode de Wu simple en Coq

Jean-David Génevaux, Julien Narboux, and Pascal Schreck

LSIIT UMR 7005 CNRS - Université de Strasbourg ⋆ ⋆⋆

Dans l'optique de faciliter l'usage des assistants de preuves, notamment Coq, les chercheurs ont mis au points des techniques visant à automatiser certaines parties de la preuve (parce qu'elles paraissent triviale ou inintéressantes au niveau où se situe la preuve). Par exemple, un preuve sur des algorithmes géométriques pourra s'appuyer sur des théorèmes de géométrie bien connus de l'utilisateur qu'il n'a pas envie de les re-démontrer. Nous nous inscrivons dans cette démarche, et dans cet article, nous nous concentrons sur une des méthodes de déduction automatique en géométrie les plus efficaces - elle permet de prouver de nombreux théorèmes qui sont difficiles à prouver à la main - à savoir la méthode de Wu [Wu78]. Nous intégrons cette méthode dans l'assistant de preuve Coq.

L'implantation directe en Coq de la méthode de Wu complète qui inclue l'implantation de méthodes de factorisation de polynômes à plusieurs variables sur des extensions algébriques ou transcendantes de Q, est coûteuse en temps chercheur. C'est pourquoi, dans un premier temps, nous avons opté pour une approche à base de certificats déjà employée par Grégoire, Pottier et Théry. Nous modifions la procédure de décision afin qu'elle produise, en plus du résultat, un certificat qui peut être vérifié par un outil externe appelé *validateur*. C'est ce validateur, qui est plus générique et plus simple à prouver que la procédure originale, que l'on prouve formellement. Avec une telle approche, la complétude de la méthode n'est pas assurée car le générateur de certificats peut générer des certificats non valides ou même échouer. Mais si on prouve formellement le validateur on obtient une garantie de correction des résultats aussi forte qu'en prouvant la procédure de décision directement. Cette approche possède de nombreux avantages : on peut générer le certificat de manière indépendante du validateur, ainsi le générateur peut être écrit par personnes différentes et dans un langage différent. De plus comme nous ne prouvons pas la procédure, on peut utiliser diverses optimisations sans avoir à les prouver. La difficulté consiste à générer un certificat et à ce que le certificat puisse être vérifié en un temps raisonnable.

Les bases de Gröbner sont à l'origine d'une autre méthode bien connue et déjà testée pour la déduction des théorèmes en géométrie [Kap86] qui a été intégrée à Coq par B. Grégoire, L. Pottier et L. Théry [GPT11]. Nous avons étendu leurs travaux à la méthode de Wu. L'existence d'un certificat est justifiée par le théorème de la *Nullstellensatz* d'Hilbert.

Formaliser la méthode de Wu nous permet aussi de prouver des théorèmes en géométrie en se basant sur un test d'appartenance à un idéal comme les bases de Gröbner, mais la méthode de Wu est souvent très efficace. De plus, en géométrie, les énoncés fournis par l'utilisateur sont souvent faux, il faut les compléter par ce qu'on appelle des conditions de non dégénérescence *i.e.* des hypothèses sur la non colinéarité de trois points ou sur le non parallélisme de deux droites nécessaires à la preuve du théorème. Un avantage de la méthode de Wu réside dans le fait qu'elle permet de générer ces conditions *automatiquement*.

D'une manière schématique, voici les principales étapes qui composent la méthode de Wu :

1. Mettre l'énoncé sous forme algébrique.
2. Choisir un repère et montrer que prouver l'énoncé dans ce repère est suffisant pour prouver l'énoncé dans le cas général.
3. Simplifier les équations suite au choix du repère.
4. Trianguler la liste d'hypothèses et générer un certificat pour cette triangulation.

⋆. Ce travail a été partiellement financé par le projet ANR Galapagos.

⋆⋆. Ceci est un résumé de l'article *Formalization of Wu's simple method in Coq*, CPP 2011 First International Conference on Certified Programs and Proofs, Taiwan.

5. Pseudo-diviser le but successivement par les différentes hypothèses triangulées. Si le reste final est nul, l'énoncé est prouvé sous les hypothèses des conditions de non-dégénérescence. Générer un certificat pour cette étape.
6. Vérifier le certificat en utilisant un algorithme prouvé formellement en Coq.

Les résultats que nous obtenons sont résumés dans le tableau 1. Les résultats montrent que le temps de vérification du certificat est conséquent par rapport au temps de génération du certificat et que le ratio varie grandement suivant les exemples. Nous prévoyons d'étendre notre implantation OCAML en implantant la méthode de Wu complète ou une de ses variantes développées par Dongming Wang [Wan01].

Table 1. Résultats obtenus avec un Intel(R) Core(TM) i5 CPU 750 @ 2.67GHz avec 4Gb RAM.

Théorème	Génération de certificat en utilisant **Wu** et vérification en utilisant **Ocaml**	Génération de certificat en utilisant **Wu** et vérification en utilisant **Coq**	Génération de certificat en utilisant **GB** et vérification en utilisant **Coq**	Vitesse relative de la méthode de Wu par rapport à GB
Pascal_2	0.013	21	-	-
Pascal_1	0.024	22	1652	×75
Ptolemy95	0.010	10	30	×3
Pappus	0.043	3	8	×2.6
Altitudes	0.002	3	7	×2.3
Simson	0.002	5	8	×1.6
Perp-bisect	0.001	2	3	×1.5
Pythagore	0.001	1	1	×1
Feuerbach	0.038	15	15	×1
Isoceles	0.001	1	1	×1
Euler Line	0.063	9	6	×0.6
Medians	0.001	3	2	×0.6
Chords	0.015	4	2	×0.5
Thales	0.003	6	3	×0.5
Bissectors	0.001	6	3	×0.5
Desargues	0.027	99	10	×0.1
Ceva	0.025	98	6	×0.06

Références

[GPT11] Benjamin Grégoire, Loïc Pottier, and Laurent Théry. Proof Certificates for Algebra and their Application to Automatic Geometry Theorem Proving. In *Post-proceedings of Automated Deduction in Geometry (ADG 2008)*, number 6701 in Lecture Notes in Artificial Intelligence, 2011.

[Kap86] Deepak Kapur. Geometry Theorem Proving using Hilbert's Nullstellensatz. In *SYMSAC '86 : Proceedings of the fifth ACM symposium on Symbolic and algebraic computation*, pages 202–208, New York, NY, USA, 1986. ACM Press.

[Wan01] Dongming Wang. *Elimination Method*. Springer-Verlag, 2001.

[Wu78] Wen-Tsün Wu. On the Decision Problem and the Mechanization of Theorem Proving in Elementary Geometry. In *Scientia Sinica*, volume 21, pages 157–179. 1978.

Session du groupe de travail MFDL

Méthodes Formelles dans le Développement Logiciel

Apport des méthodes formelles pour l'exploitation de logs informatiques dans un contexte contractuel

Auteur : Lacramioara Astefanoaei, Gregor Gossler, Daniel Le Métayer, Eduardo Mazza, Marie Laure Potet

Résumé :
Dans cet article, nous présentons la démarche adoptée dans le projet LISE pour spécifier de manière formelle les responsabilités des parties dans un contrat portant sur des logiciels. Nous décrivons deux options, l'une reposant sur une attribution a priori des responsabilités en fonction des dysfonctionnements constatés dans les logs, l'autre sur une analyse de causalité, et nous les illustrons sur un exemple de système de réservation d'hôtels.

Le contrôle d'accès et d'usage a posteriori.

Auteur : Nora Cuppens-Boulahia, Frédéric Cuppens (Télécom Bretagne)

Résumé :
La "privacy" et la confidentialité sont deux préoccupations majeures dans certains environnements tels que les établissements de santé. Elles sont généralement satisfaites au moyen du contrôle d'accès et d'usage. Les mécanismes traditionnels de contrôle d'accès et d'usage empêchent l'accès illégal en vérifiant l'autorisation d'accès avant l'exécution des mesures de prévention. Ils appartiennent à une classe de solutions de sécurité qualifiées d'a priori. De ce point de vue, ils présentent certaines limitations, comme l'inflexibilité dans des circonstances imprévues (situations d'urgence, par exemple). Récemment, des contrôles d'accès et d'usages de type a posteriori ont été proposés qui viennent en support aux contrôles d'accès classiques et apportent la flexibilité nécessaire dans certains contextes. Ce sont des contrôles dissuasifs qui n'empêchent pas les accès, mais se fondent sur l'imputabilité et les sanctions en cas de violation de la politique de sécurité. Pour être déployée, une solution de contrôle d'accès a posteriori doit recueillir des preuves d'actions effectuées, par qui, quand, comment et dans quelles circonstances. Comme ils sont extraits des dossiers de journalisation et de logs, l'efficacité de la détection des violations dépend du processus qui permet de vérifier la conformité de ces enregistrements avec la politique de sécurité. Cette présentation traitera de ce nouveau type de sécurité.

Calcul de points d'observation pour l'analyse dynamique de programmes

Auteur : Antoine Ferlin, Virginie Wiels (ONERA)

Résumé :
Plusieurs techniques existent pour la vérification du logiciel : les tests qui sont exécutés sur le matériel réel, l'analyse statique qui ne nécessite pas l'exécution du programme et l'analyse dynamique qui est parfois présentée comme une utilisation plus légère de techniques de vérification formelle et qui cherche à analyser des propriétés sur certaines exécutions du programme. Cet article présente des travaux en cours pour mettre en oeuvre des techniques d'analyse dynamique de propriétés temporelles sur des programmes C. L'approche proposée utilise l'analyse statique pour définir les points d'observation nécessaires à la vérification d'une propriété donnée

Session du groupe de travail MTV^2

Méthodes de test pour la validation et la vérification

Le slicing améliore une méthode de vérification combinant l'analyse statique et l'analyse dynamique
(Extended Abstract)

Omar Chebaro[1], Nikolai Kosmatov[2], Alain Giorgetti[3,4], and Jacques Julliand[3]

[1] ASCOLA (EMN-INRIA, LINA), École des Mines de Nantes, 44307 Nantes, France
[2] CEA, LIST, Laboratoire Sûreté des Logiciels, PC 174, 91191 Gif-sur-Yvette, France
[3] University of Franche-Comté, FEMTO-ST, UMR 6174, Besançon, F-25030, France
[4] Inria, Villers-lès-Nancy, F-54600, France, CASSIS project
firstname.lastname@{mines-nantes[1], cea[2], femto-st[3]}.fr

La validation des logiciels est une partie cruciale de leur cycle de développement. Deux techniques de vérification et de validation se sont démarquées au cours de ces dernières années : l'analyse statique et l'analyse dynamique. Les points forts et faibles des deux techniques sont complémentaires.

Dans [1], nous avons présenté une combinaison originale de ces deux techniques. Dans cette combinaison, l'analyse statique signale les instructions risquant de provoquer des erreurs à l'exécution, par des alarmes dont certaines peuvent être de fausses alarmes, puis l'analyse dynamique (génération de tests) est utilisée pour confirmer ou rejeter ces alarmes. Appliquée à des programmes de grande taille, la génération de tests peut manquer de temps ou d'espace mémoire avant de confirmer certaines alarmes comme de vraies erreurs ou conclure qu'aucun chemin d'exécution ne peut atteindre l'état d'erreur de certaines alarmes et donc rejeter ces alarmes. Les expérimentations ont montré que la génération de tests sur le programme complet peut perdre beaucoup de temps en essayant de couvrir des chemins d'exécution ou des parties de code qui ne sont pas pertinentes pour les alarmes. Nous proposons de réduire la taille du code source par le *slicing* avant de lancer la génération de tests (cf Fig. 1). Le *slicing* transforme un programme en un autre programme plus simple, appelé *slice*, qui est équivalent au programme initial selon certains critères.

Une autre motivation importante de ce travail est de fournir automatiquement à l'ingénieur de validation autant d'informations que possible sur chaque erreur détectée.

Quatre utilisations du *slicing* sont étudiées. La première utilisation est nommée *all*. Elle consiste à appliquer le *slicing* une seule fois, le critère de simplification étant l'ensemble de toutes les alarmes du programme qui ont été détectées par l'analyse statique. L'inconvénient de cette utilisation est que la génération de tests peut manquer de temps ou d'espace et les alarmes les plus faciles à classer sont pénalisées par l'analyse d'autres alarmes plus complexes. Dans la deuxième utilisation, nommée *each*, le *slicing* est effectué séparément par rapport à chaque alarme. Cependant, la génération de tests est exécutée pour chaque programme et il y a un risque de redondance d'analyse si des alarmes sont incluses dans d'autres slices.

Pour pallier ces inconvénients, nous avons étudié les dépendances entre les alarmes et nous avons introduit deux utilisations avancées du *slicing*, nommées *min* et *smart*, qui exploitent ces dépendances. Dans l'utilisation *min*, le *slicing* est effectué par rapport à un ensemble minimal de sous-ensembles d'alarmes. Ces sous-ensembles sont

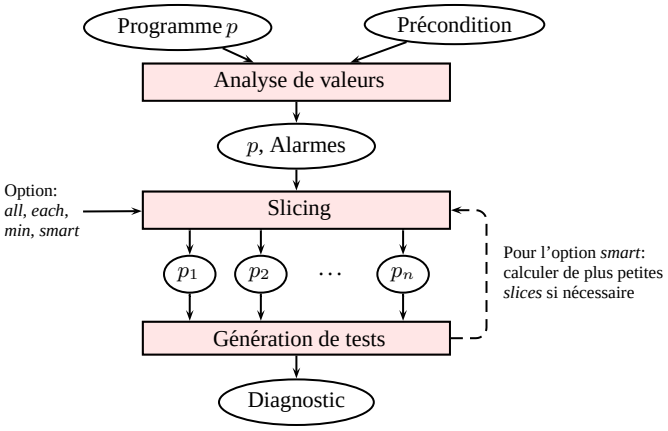


Fig. 1. Principe de la méthode de vérification

choisis en fonction de dépendances entre les alarmes et l'union de ces sous-ensembles couvre l'ensemble de toutes les alarmes. Avec cette utilisation, on a moins de *slices* qu'avec *each*, et des *slices* plus simples qu'avec *all*. Cependant, l'analyse dynamique de certaines *slices* peut manquer de temps ou d'espace mémoire pour classer certaines alarmes, tandis que l'analyse dynamique d'une *slice* éventuellement plus simple permettrait de les classer. L'utilisation *smart* consiste à appliquer l'utilisation précédente itérativement en réduisant la taille des sous-ensembles quand c'est nécessaire. Lorsqu'une alarme ne peut pas être classée par l'analyse dynamique d'une slice, des slices plus simples sont calculées.

Ces travaux sont implantés dans SANTE, notre outil qui relie l'outil de génération de tests PATHCRAWLER [2] et la plate-forme d'analyse statique FRAMA-C [3]. Des expérimentations ont montré que la génération de tests sur les programmes simplifiés est plus efficace (accélération moyenne autour de 43%, allant jusqu'à 99% pour certains exemples), et laisse moins d'alarmes non classées dans un temps donné. En outre, une erreur est signalée avec des informations plus précises et illustrée sur un programme plus simple. Ceci facilite l'analyse des erreurs détectées et des alarmes restantes. Les utilisations du slicing et les expérimentations sont présentées en détail dans [4].

References

1. Chebaro, O., Kosmatov, N., Giorgetti, A., Julliand, J.: Combining static analysis and test generation for C program debugging. In: TAP. Volume 6143 of LNCS. (2010) 652–666
2. Botella, B., Delahaye, M., Hong-Tuan-Ha, S., Kosmatov, N., Mouy, P., Roger, M., Williams, N.: Automating structural testing of C programs: Experience with PathCrawler. In: AST, Vancouver, Canada, IEEE Computer Society (May 2009) 70–78
3. Cuoq, P., Signoles, J.: Experience report: Ocaml for an industrial-strength static analysis framework. In: ICFP. (2009) 281–286
4. Chebaro, O., Kosmatov, N., Giorgetti, A., Julliand, J.: Program slicing enhances a verification technique combining static and dynamic analysis. In: SAC, ACM (2012) 1284–1291

Isabelle/*Circus* : a Process Specification and Verification Environment

Abderrahmane Feliachi, Marie-Claude Gaudel and Burkhart Wolff

[1] Univ. Paris-Sud, Laboratoire LRI, UMR8623, Orsay, F-91405, France
[2] CNRS, Orsay, F-91405, France
{Abderrahmane.Feliachi, Marie-Claude.Gaudel, Burkhart.Wolff}@lri.fr

Résumé *Circus* est un langage de spécification qui permet de spécifier des structures de données et des comportements complexes. Sa sémantique est basée sur le modèle UTP (unifying theories of programming) proposé par Hoare et He.
Nous proposons, sur la base de Isabelle/UTP, notre théorie de la sémantique de UTP en Isabelle/HOL, une sémantique formelle mécanisée, basée sur une intégration superficielle (shallow-embedding) de *Circus* en Isabelle/UTP. Nous dérivons des règles de preuve à partir de cette sémantique et mettons en oeuvre des tactiques qui permettent d'écrire des preuves de raffinement sur des processus *Circus* (impliquant à la fois des données et des comportements complexes).
Afin de faciliter son utilisation, l'environnement de preuve développé supporte une syntaxe très proche de la représentation textuelle de *Circus*.

Abstract The *Circus* specification language combines elements for complex data and behavior specifications, using an integration of Z and CSP with a refinement calculus. Its semantics is based on Hoare and He's unifying theories of programming (UTP).
Based on Isabelle/UTP, our semantic theory of UTP based on Isabelle/HOL, we develop a machine-checked, formal semantics based on a "shallow embedding" of *Circus* in Isabelle/UTP. We derive proof rules from this semantics and implement tactic support that finally allows for proofs of refinement for *Circus* processes (involving both data and behavioral aspects).
This proof environment supports a syntax for the semantic definitions which is close to textbook presentations of *Circus*.

Keywords: *Circus*, denotational semantics, Isabelle/HOL, Process Algebras, Refinement

1 Introduction

Many systems involve both complex (sometimes infinite) data structures and interactions between concurrent processes. Refinement of abstract specifications of such systems into more concrete ones, requires an appropriate formalisation of refinement and appropriate proof support.

There are several combinations of process-oriented modeling languages with data-oriented specification formalisms such as Z or B or CASL; examples are discussed in [3, 9, 16, 13]. In this report, we consider *Circus* [17], a language for refinement, that supports modeling of high-level specifications, designs, and concrete programs. It is representative of a class of languages that provide facilities to model data types, using a predicate-based notation, and patterns of interactions, without imposing architectural restrictions. It is this feature that makes it suitable for reasoning about both abstract and low-level designs.

We present a "shallow embedding" of the *Circus* semantics enabling state variables and channels in *Circus* to have arbitrary HOL types. Therefore, the entire handling of typing can be completely shifted to the (efficiently implemented) Isabelle type-checker and is therefore implicit in proofs. This drastically simplifies the definitions, proofs, and makes the reuse of standardized proof procedures possible. Compared to implementations based on a "deep embedding" such as [18] this drastically improves the usability of the resulting proof environment.

Our representation brings particular technical challenges and contributions concerning some important notions about variables. The main challenge was to represent alphabets and bindings in a typed way that preserves the semantics and improves deduction. We provide a representation of bindings without an explicit management of alphabets. Hpwever, the representation of some core concepts in the unifying theories of programming (UTP) and *Circus* constructs (variable scopes and renaming) became challenging. Thus, we propose a (stack-based) solution that allows the coding of state variables scoping with no need for renaming. This solution is even a contribution to the UTP theory that does not allow nested variable scoping. Some challenging and tricky definitions (e.g. channels and name sets) are explained in this report.

This report is organized as follows. The next section gives an introduction to the basics of our work: Isabelle/HOL, UTP and *Circus* with a short example of a *Circus* process. In section 3, we present our embedding of the basic concepts of the *Circus* language (alphabet, variables ...). We introduce also the representation of the *Circus* actions and process, with an overview of the Isabelle/*Circus* syntax. In section 4, we explain by an example, how Isabelle/*Circus* can be used to write specifications. We give some details on what is happening "behind the scenes" when the system parses each part of the specification. In the last part of this section, we show how to write proofs based on specifications, and give a refinement proof example.

2 Background

2.1 Isabelle, HOL and Isabelle/HOL

Isabelle [11] is a generic theorem prover implemented in SML. It is based on the so-called "LCF-style architecture", which makes it possible to extend a small trusted logical kernel by user-programmed procedures in a logically safe way. New object logics can be introduced to Isabelle by specifying their syntax and semantics, by deriving its inference rules from there and program specific tactic support for the object logic. Isabelle is based on a typed λ-calculus including a Haskell-style type-system including type-classes (e.g. in $\alpha :: \text{order}$, the type-variable ranges over all types that posses a partial ordering.)

Higher-order logic (HOL) [7, 1] is a classical logic based on a simple type system. It provides the usual logical connectives like $\_ \wedge \_$, $\_ \Rightarrow \_$, $\neg \_$ as well as the object-logical quantifiers $\forall x \bullet P\, x$ and $\exists x \bullet P\, x$; in contrast to first-order logic, quantifiers may range over arbitrary types, including total functions $f :: \alpha \Rightarrow \beta$. HOL is centered around extensional equality $\_ = \_ :: \alpha \Rightarrow \alpha \Rightarrow \text{bool}$. HOL is more expressive than first-order logic, since, e. g., induction schemes can be expressed inside the logic. Being based on some polymorphically typed λ-calculus, HOL can be viewed as a combination of a programming language like SML or Haskell and a specification language providing powerful logical quantifiers ranging over elementary and function types.

Isabelle/HOL is an instance of Isabelle with higher-order logic. It provides a rich collection of library theories like sets, pairs, relations, partial functions lists, multi-sets, orderings, and various arithmetic theories which only contain rules derived from conservative, i. e. logically safe definitions. Setups for the automated proof procedures like `simp`, `auto`, and arithmetic types such as `int` are provided.

2.2 Advanced Specification Constructs in Isabelle/HOL

Constant definitions. In its easiest form, constant definitions are definitional logical axioms of the form $c \equiv E$ where c is a fresh constant symbol not occurring in E which is closed (both wrt. variables and type variables). For example:

```
definition upd::(α⇒β)⇒α⇒β⇒(α⇒β)    ("_(⦇_ := _⦈)")
where      upd f x v ≡ λ z. if x=z then v else f z
```

The pragma `("_(⦇ _ := _⦈)")` for the Isabelle syntax engine introduces the notation `f(⦇x:=y⦈)` for `upd f x y`. Moreover, some elaborate preprocessing allows for recursive definitions, provided that a termination ordering can be established; such recursive definitions are thus internally reduced to definitional axioms.

Type definitions. Types can be introduced in Isabelle/HOL by different ways. The most general way to safely introduce new types is the type definition using `typedef` construct. This allows one to introduce a type as a non-empty subset of an existing type. More precisely, the new type is specified to be isomorphic to this non-empty subset. For instance:

```
typedef mytype = "{x::nat. x < 10}"
```

This definition requires that the set is non-empty: ∃x. x∈{x::nat. x<10}, which is easy to prove in this case:

```
by (rule_tac x = 1 in exI, simp)
```

where *rule_tac* is a tactic that applies an introduction rule and `exI` corresponds to the introduction of the existential quantification.

In the same way, the `datatype` command allows one to define inductive datatypes. This command introduces a datatype using a list of *constructors*. For instance, a logical compiler is invoked for the following introduction of the type `option`:

```
datatype α option  = None | Some α
```

which generates the underlying type definition and derives distinctness rules and induction principles. Besides the *constructors* `None` and `Some`, the following match-operator and his rules are also generated:

case x of None ⇒ ... | Some a ⇒ ...

Extensible records. Isabelle/HOL's support for *extensible records* is of particular importance for our work. Record types are denoted, for example, by:

```
record T = a::T1
           b::T2
```

which implicitly introduces the record constructor (|a:=e1,b:=e2|) and the update of record r in field a, written as r(|a:= x|). Extensible records are represented internally by cartesian products with an implicit free component δ, i.e. in this case by a triple of the type $T_1 \times T_2 \times \delta$. The third component can be referenced by a special selector `more` available on extensible records. Thus, the record `T` can be extended later on using the syntax:

```
record ET = T + c::T3
```

The key point is that theorems can be established, once and for all, on `T` types, even if future parts of the record are not yet known, and reused in the later definition and proofs over `ET`-values. Using this feature, we can model the effect of defining the alphabet of UTP processes incrementally while maintaining the full expressivity of HOL wrt. the types of T_1, T_2 and T_3.

2.3 *Circus* and its UTP Foundation

Circus is a formal specification language [17] which integrates the notions of states and complex data types (in a Z-like style) and communicating parallel processes inspired from CSP. From Z, the language inherits the notion of a schema used to model sets of (ground) states as well as syntactic machinery to describe pre-states and post-states; from CSP, the language inherits the concept of *communication events* and typed communication channels, the concepts of deterministic and non-deterministic choice (reflected by the process combinators $P \,\Box\, P'$ and $P \sqcap P'$), the concept of concealment (hiding) $P \backslash A$ of events in A occurring in in the evolution of process P. Due to the presence of state variables, the *Circus* synchronous communication operator syntax is slightly different frome CSP: $P \;[\![\; n \mid c \mid n' \;]\!]\; P'$ means that P and P' communicate via the channels mentioned in c; moreover, P may modify the variables mentioned in n only, and P' in n' only, n and n' are disjoint name sets.

Moreover, the language comes with a formal notion of refinement based on a denotational semantics. It follows the failure/divergence semantics [14], (but coined in terms of the UTP [12]) providing a notion of execution trace `tr`, refusals `ref`, and divergences. It is expressed in terms of the UTP [10] which makes it amenable to other refinement-notions in UTP. The semantics allows for a rich set of algebraic rules for specifications and their transitions to program models.

A simple *Circus* specification is `FIG`, the fresh identifiers generator given in fig. 1:

$$[ID]$$

channel req
channel $ret, out : ID$

process $FIG \mathrel{\widehat=}$ **begin**
state $S == [\, idS : \mathbb{P}\, ID \,]$
$Init \mathrel{\widehat=} idS := \emptyset$

Out

ΔS
$v! : ID$

$v! \notin idS$
$idS' = idS \cup \{v!\}$

Remove

ΔS
$x? : ID$

$idS' = idS \setminus \{x?\}$

$\bullet\ Init \,;\ \mathbf{var}\ v : ID \bullet$
$(\mu\ X \bullet (req \rightarrow Out \,;\ out!v \rightarrow Skip \ \Box\ ret?x \rightarrow Remove) \,;\ X)$

end

Fig. 1. The Fresh Identifiers Generator in (Textbook) *Circus*

Predicates and Relations. The UTP is a semantic framework based on an alphabetized relational calculus. An *alphabetized predicate* is a pair (*alphabet*, *predicate*) where the free variables appearing in the predicate are all in the alphabet, e.g. $(\{x, y\}, x > y)$. As such, it is very similar to the concept of a *schema*

in Z. In the base theory Isabelle/UTP of this work, we represent alphabetized predicates by sets of (extensible) records, e.g. `{A. x A > y A}`.

An *alphabetized relation* is an alphabetized predicate where the alphabet is composed of input (undecorated) and output (dashed) variables. In this case the predicate describes a relation between input and output variables, for example $(\{x, x', y, y'\}, x' = x + y)$ which is a notation for: `{(A,A').x A' = x A + y A}`, which is a set of pairs, thus a relation.

Standard predicate calculus operators are used to combine alphabetized predicates. The definition of these operators is very similar to the standard one, with some additional constraints on the alphabets.

Designs and processes. In UTP, in order to explicitly record the termination of a program, a subset of alphabetized relations is introduced. These relations are called *designs* and their alphabet should contain the special boolean observational variable `ok`. It is used to record the start and termination of a program. A UTP design is defined as follows in Isabelle:

$$\texttt{(P} \vdash \texttt{Q)} \equiv \lambda\ \texttt{(A,A').\ (ok A} \wedge \texttt{P (A,A'))} \longrightarrow \texttt{(ok A'} \wedge \texttt{Q (A,A'))}$$

Following the way of UTP to describe reactive processes, more observational variables are needed to record the interaction with the environment. Three observational variables are defined for this subset of relations: `wait`, `tr` and `ref`. The boolean variable `wait` records if the process is waiting for an interaction or has terminated. `tr` records the list (trace) of interactions the process has performed so far. The variable `ref` contains the set of interactions (events) the process may refuse to perform. These observational variables defines the basic alphabet of all reactive processes called "`alpha_rp`".

Some healthiness conditions are defined over `wait`, `tr` and `ref` to ensure that a process satisfies some properties [5] (see table 2). Four healthiness conditions, $H1$ to $H4$, are defined to characterize designs and three other ones, $R1$, $R2$ and $R3$, for reactive processes.

Finally, a CSP process is a UTP reactive process that satisfies two additional healthiness conditions called $CSP1$ and $CSP2$ (all well-formedness conditions are summarized in table 2). A process that satisfies $CSP1$ and $CSP2$ is said to be CSP healthy.

3 Isabelle/*Circus*

The Isabelle/*Circus* environment allows the representation of processes in a syntax which is close to the textbook presentations of *Circus* (see Fig. 2). Similar to other specification constructs in Isabelle/HOL, this syntax is "parsed away", i. e. compiled into an internal representation of the denotational semantics of *Circus*, which is a formalization in form of a shallow embedding of the (essentially untyped) paper-and-pencil definitions by Oliveira et al. [12], based on UTP. *Circus* actions are defined as CSP healthy reactive processes.

$H1$: A design may not make any prediction on variable values until the program has started.
`P = λ(A,A'). ok A →P (A,A')`
$H2$: A design may not require non-termination.
`P(A,A'(|ok:=False|))→P(A,A'(|ok:=True|))`
$H3$: If the precondition of a design is satisfiable, its postcondition must be satisfiable too.
`P = P ;; Π`
$H4$: Exclude miracle design.
`P ;; true = true`
$R1$: The execution of a reactive process never undoes any event that has already been performed.
`P = P ∧ λ (A,A'). tr A ≤ tr A'`
$R2$: The behaviour of a reactive process is oblivious to what has gone before.
`P = λ(A,A'). P(A(|tr:=[]|),A'(|tr:=(tr A' - tr A)|))`
$R3$: Intermediate stable states do not progress.
`P = Πrea ◁ wait o fst ▷ P`
$CSP1$: Extension of the trace is the only guarantee on divergence.
`P = P ∨ (λ (A,A'). ¬ ok A ∧ tr A ≤ tr A')`
$CSP2$: A process may not require non-termination.
`P = P ;; J`
`J = λ(A,A'). (ok A →ok A') ∧ tr A' = tr A ∧ wait A' = wait A`
`∧ ref A' = ref A ∧ more A' = more A`

Where:

- `;;` is the sequential composition operator over relations,
- Π is the relational Skip,
- Π`rea` is the Skip reactive process,
- `◁ ▷` is the conditional operator over relations,
- `o` is the HOL functional composition operator,
- `fst` returns the first element of a pair.

Table 2. UTP Healthiness conditions

In the UTP representation of reactive processes we have given in a previous paper [8], the process type is generic. It contains two type parameters that represent the channel type and the alphabet of the process. These parameters are very general, and they are instantiated for each specific process. This could be problematic when representing the *Circus* semantics, since some definitions rely directly on variables and channels (e.g assignment and communication). In this section we present our solution to deal with this kind of problems, and our representation of the *Circus* actions and processes.

In the following, we describe the foundation as well as the semantic definition of the process operators of *Circus*. A distinguishing feature of *Circus* processes are explicit state variables which do not exist in other process algebras like, e.g., CSP. These can be:

- *global* state variables, i. e. they are declared via alphabetized predicates in the `state` section, or Z-like Δ operations on global states that generate alphabetized relations, or

Process ::= **circusprocess** Tpar* name = PParagraph* **where** Action
PParagraph ::= AlphabetP | StateP | ChannelP | NamesetP | ChansetP | SchemaP
| ActionP
AlphabetP ::= **alphabet** [vardecl+]
vardecl ::= name :: type
StateP ::= **state** [vardecl+]
ChannelP ::= **channel** [chandecl+]
chandecl ::= name | name type
NamesetP ::= **nameset** name = [name+]
ChansetP ::= **chanset** name = [name+]
SchemaP ::= **schema** name = SchemaExpression
ActionP ::= **action** name = Action
Action ::= **Skip** | **Stop** | Action ; Action | Action □ Action | Action ⊓ Action
| Action \ chansetN | var := expr | guard & Action | comm → Action
| **Schema** name | ActionName | μ var • Action | **var** var • Action
| Action ⟦ namesetN | chansetN | namesetN ⟧ Action

Fig. 2. Isabelle/*Circus* syntax

– *local* state variables, i. e. they are result of the variable declaration statement **var** var • Action. The scope of local variables is restricted to Action.

On both kind of state variables, logical constraints may be expressed.

3.1 Alphabets and Variables

In order to define the set of variables, the *Circus* semantics describes the alphabet of its components, be it on the level of alphabetized predicates, alphabetized relations or actions. We recall that these items are represented by sets of records or sets of pairs of records, following the idea that an alphabet is used to establish a "binding" of variables to values. The *alphabet of a process* is defined by extending the reactive process alphabet (cf. Section 2.3) with the corresponding variable names and types. Considering the example *FIG*, where the global state variable *idS* is defined, this is reflected in Isabelle/Circus by the extension of the process alphabet by this variable, i.e. by the extension of the Isabelle/HOL record:

```
record α alpha = α alpha_rp +  idS :: ID set
```

This introduces the record type `alpha` that contains the observational variables of a reactive process, plus the variable `idS`. Note that our *Circus* semantic representation allows "built-in" bindings of alphabets in a typed way. Moreover, there is no restriction on the associated HOL type. However, the inconvenience of this representation is that variables cannot be introduced "on the fly"; they must be known statically i.e. at type inference time. Another consequence is that a "syntactic" operation such as variable renaming has to be expressed as a "semantic" operation that maps one record type into another.

Updating and accessing global variables. Since the alphabets are represented by HOL records, i.e. a kind binding "$name \mapsto value$", we need a certain

infrastructure to access data in them and to update them. The Isabelle representation as records gives us already two functions (for each record) "select" and "update". The "select" function returns the value of a given variable name, and the "update" functions updates the value of this variable. Since we may have different HOL types for different variables, a unique definition for select and update cannot be provided. There is an instance of these functions for each variable in the record. The name of the variable is used to distinguish the different instances: for the select function the name is used directly and for the update function the name is used as a prefix e.g. for a variable named "x" the names of the *select* and *update* functions are respectively `x` of type α and `x_update`.

Since a variable is characterized essentially by these functions, we define a general type (synonym) called `var` which represents a variable as a pair of its select and update function (in the underlying state σ).

```
types (β, σ) var = "(σ ⇒ β) * ((β ⇒ β) ⇒ σ ⇒ σ)"
```

For a given alphabet (record) of type σ, `(`β`, the type `σ`)var` represents the type of the variables whose value type is β in this alphabet. One can then extract the select and update functions from a given variable with the following functions:

```
definition select :: "(β, σ) var ⇒σ ⇒ β"
  where select f ≡ (fst f)

definition update :: "(β, σ) var ⇒ β ⇒ σ ⇒ σ"
  where update f v ≡ (snd f) (λ _ . v)
```

Finally, we introduce a function called `VAR` to implement a syntactic translation of a variable name to an entity of type `var`.

```
syntax "_VAR" :: "id ⇒(β, σ) var" ("VAR _")
translations VAR x => (x, _update_ name x)
```

Note that in this syntactic translation rule, `_update_ name x` stands for the concatenation of the string `_update_` with the content of the variable `x`; the resulting `_update_x` in this example is mapped to the field-update function of the extensible record `x_update` by a default mechanism. On this basis, the assignment notation can be written as usual:

```
syntax
  "_assign" :: "id ⇒(σ ⇒β) ⇒(α, σ) action" ("_ ‘:=‘ _")
translations
  "x ‘:=‘ E"   => "CONST ASSIGN (VAR x) E"
```

and mapped to the *semantics* of the program variable `(x,x_update)` together with the universal `ASSIGN` operator defined in Section 3.3.

Updating and accessing local variables. In *Circus*, local program variables can be introduced on the fly, and their scopes are explicitly defined, as can

be seen in the *FIG* example. In textbook *Circus*, nested scopes are handled by variable renaming which is not possible in our representation due to the implicit representation of variable names. Instead, we represent local program variables by global variables, using the `var` type defined above, where selection and update involve an explicit stack discipline. Each variable is mapped to a list of values, and not to one value only (as for state variables). Entering the scope of a variable corresponds to adding a new value as the head of the corresponding values list. Leaving a variable scope corresponds to removing the head of the values list. The select and update functions correspond to selecting and updating the head of the list.

Note that this encoding scheme requires to make local variables lexically distinct from global variables; local variable instances are just distinguished from the global ones by the stack discipline. The results in dynamic scoping which is required by the operational semantics.

3.2 Synchronization infrastructure: Name sets and channels.

Name sets. An important notion, used in the definition of parallel *Circus* actions, is name sets as seen in Section 2. A name set is a set of variable names, which is a subset of the alphabet. This notion cannot be directly expressed in our representation since variable names are not explicitly represented. Its definition is a bit tricky and relies on the characterization of the variables in our representation. As for variables, name sets are defined by their functional characterization. Name sets are only used in the definition of the binding merge function *MSt*: $\forall\, v \bullet (v \in ns1 \Rightarrow v' = (1.v)) \land (v \in ns2 \Rightarrow v' = (2.v)) \land (v \notin ns1 \cup ns2 \Rightarrow v' = v)$.

The disjoint name sets $ns1$ and $ns2$ are used to determine which variable values (extracted from local bindings of the parallel components) are used to update the global binding of the process. A name set can be functionally defined as a binding update function, that copies values from a local binding to the global one. For example, a name set *NS* that only contains the variable x can be defined as follows in Isabelle/Circus:

```
definition NS lb gb ≡ x_update (x lb) gb
```

where `lb` and `gb` stands for local and global bindings, `x` and `x_update` are the select and update functions of variable `x`. Then the merge function can be defined by composing the application of the name sets to the global binding.

Channels. Reactive processes interact with the environment via synchronizations and communications. A synchronization is an interaction via a channel without any exchange of data. A communication is a synchronization with data exchange. In order to reason about communications in the same way, a datatype *channels* is defined using the channels names as constructors. For instance, in:

```
datatype channels = chan1 | chan2 nat | chan3 bool
```

we declare three channels: `chan1` that synchronizes without data , `chan2` that communicates natural values and `chan3` that exchanges boolean values.

This definition makes it possible to reason globally about communications since they have the same type. A drawback is that the channels may not have the same type: in the above example the types of `chan1`, `chan2` and `chan3` are respectively of types `channels`, `nat` ⇒ `channels` and `bool` ⇒ `channels`. However, in the definition of some *Circus* operators, we need to compare two channels, and one can't compare for example `chan1` with `chan2` since they don't have the same type. A solution would be to compare `chan1` with (`chan2 v`). The types are equivalent in this case, but the problem remains because comparing (`chan2 0`) to (`chan2 1`) will state inequality just because the communicated values are not equal. We can of course define an inductive function over the datatype `channels` to compare channels, but this is only possible when all the channels are known *a priori*.

Thus, we add some constraint to the generic channels type: we require the `channels` type to implement a function `chan_eq` that tests the equality of two channels. Fortunately, Isabelle/HOL provides a construct for this kind of restriction: the type classes (sorts) seen in the first section. We define a type class (interface) `chan_eq` that contains a signature of the `chan_eq` function.

```
class chan_eq =
  fixes chan_eq :: "α ⇒α ⇒ bool"
begin end
```

Concrete channels type must implement the interface (class) " `chan_eq`" that can be easily defined for this concrete type. Moreover, one can use this class to add some definition that depends on the channel equivalence function. For example, a trace equivalence function can be defined as follows:

```
fun tr_eq where
  tr_eq [] [] = True | tr_eq xs [] = False | tr_eq [] ys = False
| tr_eq (x#xs) (y#ys) = if chan_eq x y then tr_eq xs ys else False
```

This function will be applicable for traces of elements whose type belongs to the sort `chan_eq`.

3.3 Actions and Processes

The *Circus* actions type is defined as the set of all the CSP healthy reactive processes. The type `(α,σ)relation_rp` is the reactive process type where α is of `channels` type and σ is a record extensions of `action_rp`, i. e. the global state variables. On this basis, we can encode the concept of a process for a family of possible state instances. We introduce the vital type `action` via the type-definition:

```
typedef(Action)
 (α::chan_eq,σ) action = {p::(α,σ)relation_rp. is_CSP_process p}
proof - {...} qed
```

As mentioned before, a type-definition introduces a new type by stating a set. In our case it is the set of reactive processes that satisfy the healthiness-conditions for CSP-processes, isomorphic to the new type.

Technically, this construct introduces two constants definitions `Abs_Action` and `Rep_Action` respectively of type `(α,σ) relation_rp ⇒(α,σ) action` and `(α,σ)action ⇒(α,σ)relation_rp` as well as the usual two axioms expressing the bijection `Abs_Action(Rep_Action(X))=X` and `is_CSP_process p ⟹ Rep_Action(Abs_Action(p))=p` where `is_CSP_process` captures the healthiness conditions.

Every *Circus* action is an abstraction of an alphabetized predicate. Below, we introduce the definitions of all the actions and operators using their denotational semantics. We must provide for each action, the proof that this predicate is CSP healthy. In this section we show all *Circus* basic actions and operators definitions. We also show how a whole *Circus* process is represented in the UTP framework. The environment contains the definitions of all the *Circus* operators shown in the next section.

Moreover, the environment contains a proof for a theorem stating that every reactive design — based on the above and the subsequent definitions — is CSP healthy.

Basic actions. `Stop` is defined as a reactive design, with a precondition `true` and a postcondition stating that the system deadlocks and the traces are not evolving.

```
definition
Stop ≡  Abs_Action (R (true ⊢λ(A, A'). tr A' = tr A ∧ wait A'))
```

`Skip` is defined as a reactive design, with a precondition *true* and a postcondition stating that the system terminates and all the state variables are not changed. We represent this fact by stating that the `more` field is not changed, since this field is mapped to all the state variables. Recall that the `more`-field is a tribute to our encoding of alphabets by extensible records and stands for all future extensions of the alphabet (e.g. state variables).

```
definition Skip ≡  Abs_Action (R (true ⊢ λ (A, A'). tr A' = tr A
                                  ∧ ¬ wait A' ∧ more A = more A'))
```

The universal assignment action. In the previous section 3.1, we described already how global and local variables were represented by access- and updates functions introduced by fields in extensible records. In these terms, the "lifting" to the assignment action in *Circus* processes is straightforward:

```
definition
  ASSIGN::"(β, σ) var ⇒(σ ⇒ β) ⇒(α::ev_eq, σ) action"
where
  ASSIGN x e ≡ Abs_Action (R (true ⊢ Y))
```

```
where
 Y = λ(A, A'). tr A' = tr A ∧ ¬ wait A' ∧
                more A' = (assign x (e (more A))) (more A)
```

where `assign` is the projection into the update operation of a semantic variable described in section 3.1.

Internal and External Choice. For the internal choice operator the semantics is quite simple. It is defined as the relational disjunction of the two actions.

```
definition ndet (infixl "⊓") where
P ⊓Q ≡  Abs_Action ((Rep_Action P ) ∨(Rep_Action Q))
```

The external choice semantics is more complicated, it is defined in our environment as follows:

```
definition det (infixl "□") where
P □Q ≡  Abs_Action(R (X ⊢ Y))
where
X =  ¬(Spec F F (Rep_Action P)) ∧¬ (Spec F F (Rep_Action Q))
and
Y = (Spec T F (Rep_Action P)) ∧ (Spec T F (Rep_Action Q))
           ◁ λ (A, A'). tr A = tr A' ∧ wait A' ▷
     (Spec T F (Rep_Action P)) ∨ (Spec T F (Rep_Action Q))
```

where the operation *Spec* is defined as follows:

```
definition Spec x y P = λ(A, A'). P (A(|wait := y|), A'(|ok := x|))
```

Guarded Actions. A guarded action is an action that can be executed if the guard value is *true* only, otherwise it stops. A guard is defined as a predicate over the action variables and the semantics of the guarded action is defined as follows:

```
definition guard ("_ & _") where
g & P ≡  Abs_Action(R (X ⊢ Y))
where
X = (g o more o fst) ⟶¬ (Spec F F (Rep_Action P))
and
Y = ((g o more o fst) ∧ (Spec T F (Rep_Action P))) ∨
    (¬ (g o more o fst) ∧ (λ (A, A'). tr A' = tr A ∧ wait A'))
```

Given this definition, it is easy to prove that if the value of the guard is *false* the action will stop. The proof of this property is given by the following:

```
lemma "false & P = Stop"
by (simp add: Guard_def Stop_def csp_defs design_defs utp_defs
                                                        rp_defs)
```

Sequencing. The sequence operator is defined using the UTP sequential composition operator: the semantics of composing two actions is given by the relational composition of their corresponding relations.

```
definition seq (infixl ";") where
P ; Q ≡ Abs_Action (Rep_Action P ;; Rep_Action Q)
```

Schema Expressions. In order to define the semantics of schema expressions, the function `Pre` is introduced. This function verifies that the precondition of a schema expression is *true* before applying the schema operation. This function is defined by ignoring the output alphabet of the schema, since the precondition depends only on the input variables.

```
definition Pre ::"'α relation ⇒'α predicate" where
Pre sc ≡ λ A. ∃ A'. sc (A, A')
```

The semantics of schema expressions is then:

```
definition
Schema sc ≡ Abs_Action(R (X ⊢ Y))
where
X = λ(A, A'). (Pre sc) (more A)
and
Y = λ(A, A'). sc (more A, more A') ∧ ¬ wait A' ∧ tr A = tr A'
```

Communications. The definition of prefixed actions is based on the definition of a special relation `do_I`. In the *Circus* denotational semantics, various forms of prefixing were defined. In our theory, we define one general form, and the other forms are defined as special cases.

```
definition do_I c x P ≡  X  ◁ wait o fst ▷  Y
where
X = (λ (A, A'). tr A = tr A' ∧ ((c ‘ P) ∩ ref A') = {})
and
Y = (λ (A, A'). hd ((tr A') - (tr A)) ∈ (c ‘ P) ∧
     (c (select x (more A))) = (last (tr A')))
```

where `c` is a channel constructor, `x` is a variable (of `var` type) and `P` is a predicate. The `do_I` relation gives the semantics of an interaction: if the system is ready to interact, the trace is unchanged and the waiting channel is not refused. After performing the interaction, the new event in the trace corresponds to this interaction.

The semantics of the whole action is given by the following definition:

```
definition Prefix c x P S ≡Abs_Action(R (X ⊢ Y)) ; S
where
X = true
and
Y =  do_I c x P ∧ (λ (A, A'). more A' = more A)
```

where c is a channel constructor, x is a variable (of type var), P is a predicate and S is an action. This definition states that the prefixed action semantics is given by the interaction semantics (do_I) composed with the semantics of the continuation (action S).

Different types of communication are considered:

- Inputs: the communication is done over a variable.
- Constrained Inputs: the input variable value is constrained with a predicate.
- Outputs: the communications exchanges only one value.
- Synchronizations: only the channel name is considered (no data).

The semantics of these different forms of communications is based on the general definition above.

```
definition read c x P ≡ Prefix c x true P
definition write1 c a P ≡ Prefix c (λs. a s, (λ x. λy. y)) true P
definition write0 c P ≡ Prefix (λ_.c) (λ_._, (λ x. λy. y)) true P
```

where read, write1 and write0 corresponds respectively to *input*, *output* and *synchronization*, *constrained input* corresponds to the general definition.

We configure the Isabelle syntax-engine such that it parses the usual communication primitives and gives the corresponding semantics:

```
translations
  c ? p →P      == CONST read c (VAR p) P
  c ? p : b →P  == CONST Prefix c (VAR p) b P
  c ! p →P      == CONST write1 c p P
  a → P         == CONST write0 (TYPE(_)) a P
```

Hiding. The hiding operator is interesting because it depends on a channel set. This operator P \ cs is used to encapsulate the events that are in the channel set cs. These events become no longer visible from the environment. The semantics of the hiding operator is given by the following reactive process:

```
definition
Hide ::"[(α, σ) action , α set] ⇒(α, σ) action" (infixl "\")
where
P \ cs ≡Abs_Action( R(λ (A, A').
          ∃ s. (Rep_Action P)(A, A'(|tr :=s, ref := (ref A') ∪ cs|))
              ∧ (tr A' - tr A) = (tr_filter (s - tr A) cs))); Skip
```

The definition uses a filtering function tr_filter that removes from a trace the events whose channels belong to a given set. The definition of this function is based on the function chan_eq we defined in the class chan_eq. This explains the presence of the constraint on the type of the action channels in the hiding definition, and in the definition of the filtering function below:

```
fun tr_filter::"a::chan_eq list ⇒a set ⇒a list" where
  tr_filter [] cs = []
```

```
| tr_filter (x#xs) cs = (if (¬ chan-in_set x cs)
                           then (x#(tr_filter xs cs))
                            else (tr_filter xs cs))
```

where the `chan-in_set` function checks if a given channel belongs to a channel set using `chan_eq` as equality function.

Parallel Composition. The parallel composition of actions is one of the most important definitions in our environment. It involves two important notions presented in the last section which are name sets and channel sets. As explained in Sections 2 and 3.2, name sets are used in the state merge function *MSt* that merges the final values of the local states into the global one. The name sets are defined as state update functions that can be composed to build the global state by the *MSt* function. On this basis, we define the *MSt* function in Isabelle/Circus as follows:

```
definition
MSt s1 s2 = (λ (S, S'). (S' = s1 S)) ;; (λ (S, S'). (S' = s2 S))
```

where `s1` and `s2` are disjoint name sets.

The second important notion in this definition is channel set. As explained in section 3.2, channels are defined as datatype constructors. As channels are usually defined over different types, channel sets cannot be directly defined since the types of elements may be not the same (as explained in section 3.2). To avoid this problem, we use the communications type `channels` as the type of elements in the channel sets. Thus, in the case of constructors communicating values, we apply them to some dummy values to obtain a value of type `channels`. One can define, for instance, the channel set `cs = {chan1, chan2(Some x)}`, and define a new membership function over this channel set using the function `chan_eq` to check if a channel belongs to some given channel set.

Given these definitions, the parallel composition operator is stated as follows:

```
definition Par ("_ ⟦_ | _ | _ ⟧_") where
A1 ⟦ ns1 | cs | ns2 ⟧ A2 ≡  Abs_Action(R (X ⊢ Y))
where
X = (λ (S, S'). ¬∃ tr1 tr2. (Spec F F (Rep_Action A1) ;;
(λ (S, S'). tr1 = tr S)) (S, S') ∧(Spec F (wait S) (Rep_Action A2) ;;
(λ (S, S'). tr2 = tr S)) (S, S') ∧(tr_filter tr1 cs) = (tr_filter tr2 cs)
 ∧ ¬ ∃ tr1 tr2. (Spec F (wait S) (Rep_Action A1) ;;
(λ (S, S'). tr1 = tr S)) (S, S') ∧(Spec F F (Rep_Action A2) ;;
(λ (S, S'). tr2 = tr S)) (S, S') ∧(tr_filter tr1 cs) = (tr_filter tr2 cs))
and
Y = (λ (S, S'). (∃ s1 s2. (λ (A, A'). (Spec T F (Rep_Action A1) (A, s1))
   ∧ Spec T F (Rep_Action A2) (A, s2)) ;; M_par s1 ns1 s2 ns2 cs) (S, S'))
```

where `A1` and `A1` are *Circus* actions, `ns1` and `ns2` are name sets defined as update functions over the state of the actions `A1` and `A2`. Finally, `cs` is a channel set defined over the communications type of the actions `A1` and `A2`.

The environment contains also the definitions of `tr_filter`, `M_par` and some other functions used in these definitions.

Recursion. To represent the recursion operator "μ" over actions, we use the universal least fix-point operator "*lfp*" defined in the HOL library for lattices and we follow again [12]. The use of least fix-points in [12] is the most substantial deviation from the standard CSP denotational semantics, which requires Scott-domains and complete partial orderings. The operator *lfp* is inherited from the "*Complete Lattice class*" under some conditions, and all theorems defined over this operator can be reused. In order to reuse this operator, we have to show that the least-fixpoint over functionals that enrich pairs of failure - and divergence trace sets monotonely, produces an `action` that satisfies the CSP healthiness conditions. This consistency proof for the recursion operator is the largest contained in the Isabelle/*Circus* library.

In order to reuse the *lfp* operator and its inherited proofs, we must prove that the *Circus* actions type defines a complete lattice. This leads to prove that the actions type belongs to the "*Complete Lattice class*" of HOL. Since type classes in HOL are hierarchic, we provide a proof in three steps. First, we prove that the *Circus* actions type forms a lattice by instantiating the HOL "*Lattice class*". In the second step, we prove that actions type instantiates a subclass of lattices called "*Bounded Lattice class*". The last step is to prove the instantiation from the "*Complete Lattice class*". The details of these proofs are not given here.

```
instantiation action :: (ev_eq, type) lattice
begin
definition inf_action:
    inf P Q ≡ Abs_Action ((Rep_Action P) ⊓ (Rep_Action Q))
definition sup_action:
    sup P Q ≡ Abs_Action ((Rep_Action P) ⊔ (Rep_Action Q))
definition leq_action:
    P ≤ Q ≡ P ⊑ Q
definition less_action:
    P < Q ≡ P ⊑ Q ∧ ¬ Q ⊑ P
instance proof
  {...}
end
```

A lattice is a partial order with infimum and supremum of any two actions, the $\sqcap$ (meet) and $\sqcup$ (join) operations select such infimum and supremum actions. The instantiation proof of the lattice class requires the introduction of the definitions of the *meet*, the *join* and the ordering operators $\leq$ and $<$. In addition to the definition, the instantiation provides some proof obligations to ensure that all the notions are well defined (e.g. the ordering relation is reflexive and transitive). After proving these properties, the action type is considered as a lattice.

```
instantiation action :: (ev_eq, type) bounded_lattice
begin
definition bot_action:
    bot ≡ Abs_Action true
definition top_action:
```

```
    top ≡ Abs_Action (R(true ⊢false))
instance proof
  {...}
end
```

For the instantiation of the bounded lattice class, we add definitions of bounds (top and bottom of the lattice) and prove that these bounds are well defined w.r.t the ordering relation.

```
instantiation action :: (ev_eq, type) complete_lattice
begin
definition Sup_action:
    Sup S ≡ if S={} then bot else Abs_Action ⨆(Rep_Action ‘S)
definition Inf_action:
    Inf S ≡ if S={} then top else Abs_Action ⨅(Rep_Action ‘S)
instance proof
  {...}
end
```

Finally, a complete lattice is a partial order with general (infinitary) infimum of any set of actions, a general supremum exists as well. The general $\bigsqcap$ (meet) and $\bigsqcup$ (join) operations select such infimum and supremum actions. These operations indeed determine bounds on this complete lattice structure. The Knaster-Tarski Theorem (in its simplest formulation) states that any monotone function on a complete lattice has a least fixed-point. This is a consequence of the basic boundary properties of the complete lattice operations. Instantiating the complete lattice class allows one to inherit these properties with the definition of the least fixed-point for monotonic functions over *Circus* actions.

Circus **Processes.** A *Circus* process is defined in our environment as a local theory by introducing qualified names for all its components. This is very similar to the notion of *namespaces* popular in programming languages. Defining a *Circus* process locally allows us to encapsulate definitions of alphabet, channels, schema expressions and actions in the same namespace. It is important for the foundation of Isabelle/*Circus* to avoid the ambiguity between local process entities definitions (e.g. `FIG.Out` and `DFIG.Out` in the example of section 4).

4 Using Isabelle/*Circus*

We describe the front-end interface of Isabelle/*Circus*. In order to support a maximum of common *Circus* syntactic look-and-feel, we have programmed at the SML level of Isabelle a compiler that parses and (partially) pretty prints *Circus* process given in the syntax presented in Fig.2.

4.1 Writing specifications

A specification is a sequence of paragraphs. Each paragraph may be a declaration of alphabet, state, channels, name sets, channel sets, schema expressions or

actions. The main action is introduced by the keyword `where`. Below, we illustrate how to use the environment to write a *Circus* specification using the `FIG` process example presented in Figure 1.

```
circusprocess FIG =
  alphabet = [v::nat, x::nat]
  state = [idS::nat set]
  channel = [req, ret nat, out nat]
  schema Init = idS := {}
  schema Out = ∃a. v' = a ∧ v' ∉ idS ∧ idS' = idS ∪{v'}
  schema Remove = x ∉ idS ∧ idS' = idS - {x}
  where var v · Schema Init; (μ X ·(req →Schema Out; out!v →Skip)
                                □ (ret?x →Schema Remove); X)
```

Each line of the specification is translated into the corresponding semantic operators discussed in the previous chapter. In the following, we describe the result of executing each command:

- the compiler introduces a scope of local components whose names are qualified by the process name (`FIG` in the example).
- `alphabet` generates a list of record fields to represent the binding. These fields map names to value lists.
- `state` generates a list of record fields that corresponds to the state variables. The names are mapped to single values. This command, together with `alphabet` command, generates a record that represents all the variables (for the `FIG` example the command generates the record `FIG_alphabet`, that contains the fields `v` and `x` of type `nat list` and the field `idS` of type `nat set`).
- `channel` introduces a datatype of typed communication channels (for the `FIG` example the command generates the datatype `FIG_channels` that contains the constructors `req` without communicated value and `ret` and `out` that communicate natural values).
- `schema` allows the definition of schema expressions represented as an alphabetized relation over the process variables (in the example the schema expressions `FIG.Init`, `FIG.Out` and `FIG.Remove` are generated).
- `action` introduces definitions for *Circus* actions in the process. These definitions are based on the denotational semantics of *Circus* actions. The type parameters of the action type are instantiated with the locally defined channels and alphabet types.
- `where` introduces the main action as in `action` command (in the example the main action is `FIG.FIG` of type `(FIG_channels, FIG_alphabet)action`).

4.2 Relational and Functional Refinement in Circus

The main goal of Isabelle/*Circus* is to provide a proof environment for *Circus* processes. The "shallow-embedding" of *Circus* and UTP in Isabelle/HOL offers the possibility to reuse proof procedures, infrastructure and theorem libraries

already existing in Isabelle/HOL. Moreover, once a process specification is encoded and parsed in Isabelle/*Circus*, proofs of, eg, refinement properties can be developped using the ISAR language for structured proofs.

To show in more details how to use Isabelle/*Circus*, we provide a small example of action refinement proof. The refinement relation is defined as the universal reverse implication in the UTP. In *Circus*, it is defined as follows:

```
definition A1 ⊑c A2 ≡(Rep_Action A1) ⊑utp (Rep_Action A2)
```

where A1 and A2 are *Circus* actions, ⊑c and ⊑utp stands respectively for refinement relation on *Circus* actions and on UTP predicate.

This definition assumes that the actions A1 and A2 share the same alphabet (binding) and the same channels. In general, refinement involves an important data evolution and growth. The data refinement is defined in [15, 4] by backwards and forwards simulations. In this report, we restrict ourselves to a special case, the so-called *functional* backwards simulation. This refers to the fact that the abstraction relation R that relates concrete and abstract actions is just a function:

```
definition Simulation ("_ ≼_ _") where
 A1 ≼R A2 = ∀a b.(Rep_Action A2)(a,b) ⟶(Rep_Action A1)(R a,R b)
```

where A1 and A2 are *Circus* actions and R is a function mapping the corresponding A1 alphabet to the A2 alphabet.

4.3 Refinement Proofs

We can use the definition of simulation to transform the proof of refinement to a simple proof of implication by unfolding the operators in terms of their underlying relational semantics. The problem with this approach is that the size of proofs will grow exponentially with respect to the size of the processes. To avoid this problem, some general refinement laws were defined in [4] to deal with the refinement of *Circus* actions at operators level and not at UTP level. We introduced and proved a subset of theses laws in our environment (see table 3).

In table 3, the relations "$x \sim_S y$" and "$g_1 \simeq_S g_2$" record the fact that the variable x (repectively the guard g_1) is refined by the variable y (repectively by the guard g_2) w.r.t the simulation function S.

These laws can be used in complex refinement proofs to simplify them at the *Circus* level. More rules can be defined and proved to deal with more complicated statements like combination of operators for example. Using these laws, and exploiting the advantages of a shallow embedding, the automated proof of refinement becomes surprisingly simple.

Coming back to our example, let us consider the DFIG specification below, where the management of the identifiers via the set idS is refined into a set of removed identifiers retidS and a number max, which is the rank of the last issued identifier.

$$\frac{P \preceq_S Q \qquad P' \preceq_S Q'}{P;\ P' \preceq_S Q;\ Q'}\ \text{SeqI} \qquad \frac{P \preceq_S Q \qquad g_1 \simeq_S g_2}{g_1 \& P \preceq_S g_2 \& Q}\ \text{GrdI}$$

$$\frac{P \preceq_S Q \qquad x \sim_S y}{var\ x \bullet P \preceq_S var\ y \bullet Q}\ \text{VarI} \qquad \frac{P \preceq_S Q \qquad x \sim_S y}{c?x \to P \preceq_S c?y \to Q}\ \text{InpI}$$

$$\frac{P \preceq_S Q \qquad P' \preceq_S Q'}{P \sqcap P' \preceq_S Q \sqcap Q'}\ \text{NdetI} \qquad \frac{P \preceq_S Q \qquad x \sim_S y}{c!x \to P \preceq_S c!y \to Q}\ \text{OutI}$$

$$\frac{\begin{array}{c}[X \preceq_S Y] \\ \vdots \\ P\ X \preceq_S Q\ Y \qquad mono\ P \qquad mono\ Q\end{array}}{\mu\ X \bullet P\ X \preceq_S \mu\ Y \bullet Q\ Y}\ \text{MuI} \qquad \frac{P \preceq_S Q \qquad P' \preceq_S Q'}{P \Box P' \preceq_S Q \Box Q'}\ \text{DetI}$$

$$\frac{\begin{array}{cc}[Pre\ sc_1\ (S\ A)] & [Pre\ sc_1\ (S\ A) \qquad sc_2\ (A, A')] \\ \vdots & \vdots \\ Pre\ sc_2\ A & sc_1\ (S\ A, S\ A')\end{array}}{schema\ sc_1 \preceq_S schema\ sc_2}\ \text{SchI} \qquad \frac{P \preceq_S Q}{a \to P \preceq_S a \to Q}\ \text{SyncI}$$

$$\frac{P \preceq_S Q \quad P' \preceq_S Q' \quad ns_1 \sim_S ns_1' \quad ns_2 \sim_S ns_2'}{P [\![ns_1 \mid cs \mid ns_2]\!] P' \preceq_S Q [\![ns_1' \mid cs \mid ns_2']\!] Q'}\ \text{ParI} \qquad \frac{}{Skip \preceq_S Skip}\ \text{SkipI}$$

Table 3. Proved refinement laws

```
circusprocess DFIG =
  alphabet = [w::nat, y::nat]
  state = [retidS::nat set, max::nat]
  schema Init = retidS' = {} ∧max' = 0
  schema Out = w' = max ∧ max' = max+1 ∧ retidS' = retidS - {max}
  schema Remove = y < max ∧ y ∉ retidS ∧  retidS' = retidS ∪ {y}
                    ∧ max' = max
  where var w · Schema Init; (μ X ·(req →Schema Out; out!w →Skip)
                                □ (ret?y →Schema Remove); X)
```

We provide the proof of refinement of `FIG` by `DFIG` just instantiating the simulation function `R` by the following abstraction function, that maps the underlying concrete states to abstract states:

```
definition Sim A = FIG_alphabet.make (w A) (y A)
                                ({a. a < (max A) ∧ a ∉ (retidS A)})
```

where A is the alphabet of `DFIG`, and `FIG_alphabet.make` yields an alphabet of type `FIG_Alphabet` initializing the values of `v`, `x` and `idS` by their corresponding values from `DFIG_alphabet`: `w`, `y` and `{a. a < max ∧ a ∉ retidS}`).

To prove that `DFIG` is a refinement of `FIG` one must prove that the main action `DFIG.DFIG` refines the main action `FIG.FIG`. The definition is then simplified, and the refinement laws are applied to simplify the proof goal. Thus, the full proof consists of a few lines in ISAR:

```
theorem "FIG.FIG ≼Sim DFIG.DFIG"
  apply (auto simp: DFIG.DFIG_def FIG.FIG_def mono_Seq
             intro!: VarI SeqI MuI DetI SyncI InpI OutI SkipI)
```

```
  apply (simp_all add: SimRemove SimOut SimInit Sim_def)
done
```

First, the definitions of `FIG.FIG` and `DFIG.DFIG` are simplified and the defined refinement laws are used by the auto tactic as introduction rules. The second step replaces the definition of the simulation function and uses some proved lemmas to finish the proof. The three lemmas used in this proof: `SimInit`, `SimOut` and `SimRemove` give proofs of simulation for the schema `Init`, `Out` and `Remove`.

5 Conclusions

We have shown for the language *Circus*, which combines data-oriented modeling in the style of Z and behavioral modeling in the style of CSP, a semantics in form of a shallow embedding in Isabelle/HOL. In particular, by representing the somewhat non-standard concept of the *alphabet* in UTP in form of extensible records in HOL, we achieved a fairly compact, typed presentation of the language. In contrast to previous work based on some deep embedding [18], this shallow embedding allows arbitrary (higher-order) HOL-types for channels, events, and state-variables, such as, e.g., sets of relations etc. Besides, systematic renaming of local variables is avoided by compiling them essentially to global variables using a stack of variable instances. The necessary proofs for showing that the definitions are consistent — i. e. satisfy altogether `is_CSP_healthy` — have been done, together with a number of algebraic simplification laws on *Circus* processes.

Since the encoding effort can be hidden behind the scene by flexible extension mechanisms of the Isabelle, it is possible to have a compact notation for both specifications and proofs. Moreover, existing standard tactics of Isabelle such as `auto`, `simp` and `metis` can be reused since our *Circus* semantics is representationally close to HOL. Thus, we provide an environment that can cope with combined refinements concerning data and behavior. Finally, we demonstrate its power — w.r.t. both expressivity and proof automation — with a small, but prototypic example of a process-refinement.

In the future, we intend to use Isabelle/*Circus* for the generation of test-cases, on the basis of [6], using the HOL-TestGen-environment [2].

6 Acknowledgement

We would like to thank Markarius Wenzel for his valuable help with the Isabelle framework. Furthermore, we are greatly indebted to Ana Cavalcanti for her comments on the semantic foundation of this work.

References

1. Peter B. Andrews. *Introduction to Mathematical Logic and Type Theory: To Truth through Proof*. Kluwer Academic, 2nd edition, 2002. now published by Springer.

2. Achim D. Brucker and Burkhart Wolff. On theorem prover-based testing. *Formal Aspects of Computing*, 2012. To appear.
3. Michael Butler. CSP2B: A practical approach to combining CSP and B. *Formal Aspects of Computing*, 12:182–196, 2000.
4. A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. A Refinement Strategy for *Circus*. *Formal Aspects of Computing*, 15(2 - 3):146 — 181, 2003.
5. A. L. C. Cavalcanti and J. C. P. Woodcock. A Tutorial Introduction to CSP in Unifying Theories of Programming. In *Refinement Techniques in Software Engineering*, volume 3167 of *LNCS*, pages 220 – 268. Springer-Verlag, 2006.
6. Ana Cavalcanti and Marie-Claude Gaudel. Testing for refinement in Circus. *Acta Informatica*, 48(2):97–147, 2011.
7. Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, June 1940.
8. Abderrahmane Feliachi, Marie-Claude Gaudel, and Burkhart Wolff. Unifying theories in Isabelle/HOL. In *UTP 2010, 3rd Int. Symp. on Unifying Theories of Programming*, volume 6445 of *LNCS*, pages 188–206. Springer Verlag, 2010.
9. Clemens Fischer. How to combine Z with process algebra. In *11th Int. Conf. of Z Users on The Z Formal Specification Notation*, pages 5–23. Springer-Verlag, 1998.
10. C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice Hall International Series in Computer Science, 1998.
11. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.
12. Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock. A denotational semantics for Circus. *Electron. Notes Theor. Comput. Sci.*, 187:107–123, 2007.
13. Markus Roggenbach. CSP-CASL: a new integration of process algebra and algebraic specification. *Theor. Comput. Sci.*, 354:42–71, 2006.
14. A. W. Roscoe, C. A. R. Hoare, and Richard Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.
15. A. C. A. Sampaio, J. C. P. Woodcock, and A. L. C. Cavalcanti. Refinement in *circus*. In L. Eriksson and P. A. Lindsay, editors, *FME 2002: Formal Methods – Getting IT Right*, volume 2391 of *Lecture Notes in Computer Science*, pages 451—470. Springer-Verlag, 2002.
16. K. Taguchi and K. Araki. The state-based CCS semantics for concurrent Z specification. In *1st Int. Conf. on Formal Engineering Methods*, ICFEM '97, pages 283–292. IEEE, 1997.
17. J. C. P. Woodcock and A. L. C. Cavalcanti. The semantics of Circus. In *ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of *LNCS*, pages 184—203. Springer-Verlag, 2002.
18. Frank Zeyda and Ana Cavalcanti. Encoding Circus programs in ProofPowerZ. In *Unifying Theories of Programming, 2nd Int. Symp., UTP 2008, Revised Selected Papers*, volume 5713 of *LNCS*. Springer-Verlag, 2009.

Nouveaux mécanismes de filtrage de tests basés sur le modèle

T. Triki[1], Y. Ledru[1], L. du Bousquet[1], F. Dadeau[2], and J. Botella[3]

[1] UJF-Grenoble 1/Grenoble-INP/UPMF-Grenoble 2/CNRS,
LIG UMR 5217, F-38041, Grenoble, France
{Taha.Triki, Yves.Ledru, Lydie.du-Bousquet}@imag.fr
[2] FEMTO-ST/INRIA CASSIS Project, 16 route de Gray, 25030 Besançon, FRANCE
frederic.dadeau@femto-st.fr
[3] Smartesting, Besançon, France
julien.botella@smartesting.com

Résumé L'animation de tests sur une spécification permet d'écarter les tests invalides, typiquement ceux qui ne vérifient pas la précondition d'opérations. Dans le projet ANR TASCCC, les tests sont dépliés à partir d'un patron de test en utilisant l'outil de test combinatoire Tobias. Un patron de test représentant un sénario complexe du système et/ou utilisant plusieurs valeurs d'entrée pourra déclencher une explosion combinatoire. Dans ce papier, nous présentons de nouveaux mécanismes de filtrage de tests en se basant sur le modèle, qui permettent de maîtriser l'explosion combinatoire.

1 Génération de tests

Pour générer des tests, nous utilisons l'outil de test combinatoire Tobias [1]. Cet outil déplie un patron de test représentant un ensemble de scénarios en une suite de tests. Dans ce patron, nous pouvons définir un groupe de valeurs possibles pour un paramètre d'opération ou un groupe d'appels d'opérations. A partir de ce patron, Tobias calcule et génère toutes les combinaisons possibles d'opérations et de valeurs représentant la suite de tests. Plusieurs tests générés sont des tests invalides par rapport à la spécification. Par exemple, supposons qu'on veuille créditer un système de porte-monnaie éléctronique avec un montant négatif. Cet appel d'opération viole la précondition d'opération et tout test contenant cet appel doit être écarté de la suite. Pour effectuer ce type de filtrage, nous devons animer chaque test sur la spécification du système.

2 Animation de tests

Dans le projet ANR TASCCC, un test généré est animé sur une spécification UML/OCL du système sous test en utilisant l'animateur CertifyIt de l'entreprise Smartesting [4]. L'animateur évalue ainsi pour chaque appel d'opération du test s'il vérifie la pré et la post condition OCL de l'opération. Si une condition n'est pas vérifiée, le test est considéré comme non conforme à la spécification et est écarté de la suite de tests.

4. http ://www.smartesting.com

3 Problématique

La génération de tests à partir d'un patron complexe peut résulter en une explosion combinatoire du nombre de tests produits. L'outil Tobias a été conçu pour générer des millions de tests mais ceci peut se révéler insuffisant. Ainsi dans [2] nous discutons d'un patron de test correspondant à 19^{18} cas de test. L'élimination des cas de test invalides ne donne qu'une réponse partielle à ce problème. Quand un patron de test correspond à une suite de très grande taille, l'explosion combinatoire peut empêcher Tobias de la générer, mais peut aussi entraîner l'échec de la compilation de cette suite de test ou de son animation. Dans la section qui suit, nous allons voir les solutions proposées pour résoudre ces problèmes.

4 Nouveaux mécanismes de filtrage

Outre le filtrage par précondition qui est insuffisant pour résoudre les problèmes cités avant, nous proposons trois nouveaux mécanismes de filtrage :

- Filtrage par prédicat : Un prédicat OCL est inséré à un point donné de la séquence d'appels d'opérations. Ce prédicat exprime qu'une propriété doit être vérifiée à ce stade. Les tests dont l'animation ne satisfont pas ce prédicat sont écartés de la suite de tests.
- Filtrage par comportement : Une opération correspond généralement à plusieurs comportements correspondant à des chemins possibles dans son flux de contrôle. Le filtrage par comportement permet de ne conserver que les tests correspondant à des comportements donnés.
- Filtrage par clé : Il consiste à définir des points de filtrage incrémental de la suite de tests. A chacun de ces points on rejette l'ensemble des tests invalides et on ne conserve qu'une proportion donnée de tests valides.

5 Conclusion

Nous avons défini une approche proposant de nouveaux mécanismes pour filtrer une suite de tests combinatoire. Dans [2] nous avons montré par une expérimentation comment réaliser le dépliage et l'animation de tests d'un patron de test correspondant à 19^{18} cas de test. Le résultat de l'expérimentation démontre l'efficacité de la technique incrémentale pour trouver des cas de test complexes dans un espace de recherche assez grand.

Remerciements Ce travail est financé par le projet ANR TASCCC (ANR-09-SEGI-014).

Références

1. Ledru, Y., du Bousquet, L., Maury, O., Bontron, P. : Filtering TOBIAS combinatorial test suites. In : FASE. LNCS, vol. 2984, pp. 281–294. Springer (2004)
2. Triki, T., Ledru, Y., du Bousquet, L., Dadeau, F., Botella, J. : Model-based filtering of combinatorial test suites. In : FASE. LNCS, vol. 7212, pp. 439–454 (2012)

Session du groupe de travail RIMEL

Rétro-Ingénierie, Maintenance et Evolution des Logiciels

Faciliter l'apprentissage de transformation de modèles par appariement automatique d'exemples ⋆

X. Dolques[1], A. Dogui[2], J.R. Falleri[3], M. Huchard[4], C. Nebut[4], and F. Pfister[5]

[1] INRIA, Centre Inria Rennes - Bretagne Atlantique, Campus universitaire de Beaulieu, 35042 Rennes, xavier.dolques@inria.fr
[2] Supélec Paris, aymen.dogui@supelec.fr
[3] Université de Bordeaux, falleri@labri.fr
[4] LIRMM, Université de Montpellier 2 et CNRS, Montpellier, first.last@lirmm.fr
[5] LGI2P, Ecole des Mines d'Alès, Nîmes, francois.pfister@mines-ales.fr

Présentation du contexte et motivations

Les transformations de modèles sont la partie opérationnelle de l'ingénierie dirigée par les modèles (IDM) et plusieurs langages de transformation ont été proposés pour les développer. Écrire une transformation requiert deux compétences différentes : une forte expérience en métamodélisation et avec des langages de transformation et une bonne compréhension de la sémantique des domaines sources et cibles. Tandis qu'un développeur de transformation possèdera la première compétence, la seconde compétence est généralement l'apanage des experts du domaine. Cela rend le développement lent et difficile, car le développeur doit interagir avec les experts des domaines dans le but de récupérer suffisament de connaissances pour être en mesure d'écrire une transformation correcte.

Parmi les différentes approches proposées pour assister le développement de transformations de modèles, nous nous sommes intéressés aux approches de transformation de modèles par l'exemple (MTBE). Ces approches ont pour but d'inférer la transformation [2,3,4] ou le résultat de la transformation à partir d'un ensemble d'exemples de transformation.

Appliquer la MTBE nécessite des exemples de transformation qui se composent d'un modèle source, un modèle transformé et les liens entre les éléments sources et transformés. Alors qu'il est facile d'obtenir des modèles sources et cibles grâce aux experts du domaine, récupérer les liens entre les éléments de ces modèles est long et fastidieux, et aucun des environnements de métamodélisation actuels n'est capable de les créer lors de l'édition manuelle des modèles. Ainsi ces liens sont généralement cherchés et rajoutés manuellement. Nous pensons que la majorité de ces liens peuvent être trouvés automatiquement. En effet, quand le modèle transformé est créé, les noms des éléments transformés sont généralement identiques, à une convention de nommage près, aux noms des éléments sources. Par ailleurs, les métamodèles utilisés peuvent être différents, mais bien souvent le voisinage d'un élément dans le modèle source est transformé en un élément que l'on retrouve dans le voisinage de l'élément transformé.

⋆. Cet article est un résumé de l'article [1].

Notre Approche
L'approche proposée par cet article étend l'approche d'alignement de schémas Anchor-Prompt [5]. Elle s'appuie sur la découverte automatique de paires d'ancres (paires d'éléments pour lesquels la probabilité d'appariement est forte) pour la découverte d'un appariement complet. L'algorithme de découverte de paires d'ancres parcourt les modèles sources et cibles et va sélectionner les paires d'éléments répondant à une condition d'appariement suffisamment contraignante pour obtenir une bonne précision.

À partir de ce premier appariement considéré comme fiable, l'approche Anchor-Prompt recherche des appariements dans les voisinages des paires d'ancres. En effet, si une paire d'éléments est située dans le voisinage d'une paire d'ancres, alors la probabilité que cette paire appartienne à l'appariement augmente. Ainsi on incrémente un coefficient d'appariement sur chaque paire d'éléments lorsqu'elles apparaissent sur des chemins correspondants (deux chemins connectant deux paires d'ancres). En filtrant les coefficients on obtient un second appariement couvrant plus largement les deux modèles.

Nous avons réalisé une expérimentation montrant que sur 22 transformations simples, la précision obtenue est très satisfaisante, égale à 1 dans la majorité des cas, mais le rappel est souvent faible nécessitant une intervention de l'utilisateur pour vérifier et compléter le résultat. Les résultats montrent malgré tout que l'approche apporte une assistance à la création d'appariements qui permettent dans la majorité des cas de réduire la tâche du développeur.

Cet article propose une première approche pour assister l'appariement de modèles lors du développement de transformations à partir d'exemples. Les résultats obtenus montrent que le processus d'extension de l'appariement a diminué légèrement la précision de l'appariement dans un nombre significatif de transformations. Nous espérons pouvoir réduire cet effet et améliorer le rappel dans nos travaux à venir notamment en rajoutant de nouvelles phases de découverte d'appariements avec différentes contraintes.

Références

1. Dolques, X., Dogui, A., Falleri, J.R., Huchard, M., Nebut, C., Pfister, F. : Easing model transformation learning with automatically aligned examples. In : 7th European Conference, ECMFA 2011. (2011) 189–204
2. Wimmer, M., Strommer, M., Kargl, H., Kramler, G. : Towards model transformation generation by-example. In : HICSS '07 :, IEEE Computer Society (2007) 285b
3. Balogh, Z., Varró, D. : Model transformation by example using inductive logic programming. Software and Systems Modeling (2008) Appeared online.
4. Dolques, X., Huchard, M., Nebut, C. : From transformation traces to transformation rules : Assisting model driven engineering approach with formal concept analysis. In : Supplementary Proceedings of ICCS'09. (2009) 15–29
5. Noy, N.F., Musen, M.A. : Anchor-prompt : Using non-local context for semantic matching. In : Proc. of the Workshop on Ontologies and Information Sharing at IJCAI-2001, Seattle (USA) (2001) 63–70

Supporting Simultaneous Versions for Software Evolution Assessment

Jannik Laval[1], Stéphane Ducasse[2], Jean-Rémy Falleri[1]

[1] Univ. Bordeaux, LaBRI, UMR 5800
F-33400 Talence, France
Email: {jlaval,falleri}@labri.fr
[2] RMoD Team - INRIA - Lille Nord Europe
USTL - CNRS UMR 8022
Lille, France
Email: stephane.ducasse@inria.fr

1 Introduction

Software architecture evolution, change impact analysis, software quality prediction, remodularization are important tasks in a reengineering process [BA96]. They often require developers to make choices about future system structure such as changing the dependencies between packages. While software maintainers would greatly benefit from the possibility to *assess different choices*, in practice they mostly rely on experience or intuition because of the lack of approaches providing comparison between possible variations of a change. Software reengineers do not have the possibility to easily *apply analyses on different version branches of a system and compare them* to pick up the most adequate changes.

There is a need to (1) *navigate* into *multiple, alternative* futures of the same system, (2) *apply various analyses on such futures*, and (3) *compare results* of such analyses. Since we want to be able to manipulate different futures of the same system *simultaneously*, directly changing source code to build each alternative does not scale up. There is a need for a software model of the source code that allows a reengineer to perform the operations mentioned above. In the paper [LDDF11] we raise the problem of the scalability of such multiple futures and branching versions. We propose the approach named *Orion*.

2 Orion design and dynamics

2.1 The need for sharing entities between versions

Models for reengineering are typically large because they reify lots of information to perform meaningful analyses. For example, one system under study with Orion is Pharo[3], an open-source Smalltalk platform comprising 1800 classes in 150 packages. Its FAMIX representation counts more than 800,000 entities, because it includes entities for variables, accesses, invocations... It becomes a major concern because we need several such models in memory to enable an interactive experience for the reengineer.

The most straightforward strategy is the full copy, where a version is created by copying all entities from its parent version. Then two versions are two independent models in memory and tools run as is on each model. However, this approach has a prohibitive cost both in term of memory space and creation time. In early experiments analysing Pharo, each model took 350Mo in memory and, more annoyingly, copy took more than one hour to allocate and create the 800,000 instances for each version. This was useless in the context of our approach.

Our solution is a variation of the partial copy approach, but requires an adaptation of the access of entities through links. The trade-off is between the memory cost of large models and the time cost of running queries on such models. Only entities which are directly changed are

[3] `http://www.pharo-project.org/home`

copied (then modified) in our approach. Other entities are left unchanged in their version, making the copy "sparse" and efficient. Changed and unchanged entities are reachable from the current version through a reference table, which is copied from the parent when creating the new version and modified by actions. Thus entities are effectively shared across different versions. However, dynamics are more complex than a simple Copy-on-Write standard approach as explained below.

2.2 Running queries in the presence of shared entities

Queries are the foundations for tools as they enable navigation between entities of the model. Basic queries represent direct relationships between entities: a class can be queried for its methods, a method can be queried for its outgoing invocations (i.e., method calls within the method), a package for its classes, ... More complex queries made by tools are composed from such queries.

Sharing entities across different models have an important impact on the way queries are run in a version. In particular, starting from a given version, a query may run on shared entities from older versions: results returned by such shared entities must always be interpreted in the context of the starting version, as older entities may link to entities which have changed since. This specific aspect makes our solution more subtle to implement than a simple Copy-on-Write. The challenge of running queries over shared entities is summarized as follows:

1. basic queries retrieve entities which may or may not reside in a parent version;
2. then Orion should resolve each retrieved entity to its most recent entity (sharing the same *orionId*) reachable from the current version.

The challenge is akin to late binding in object-oriented languages. An entity residing in a parent version is always interpreted in the context of the current version where the query is run, same as a method invocation is always resolved against the dynamic class of *this*, even when the call comes from a method in a superclass. In our solution, there is no look-up through parent versions to resolve the most recent entity, but a direct access through the reference table of the current version.

3 Conclusion

We claim that software maintainers should be able to assess and compare multiple change scenarios. The main concern detailed in this paper is the efficient manipulation of simultaneous versions in memory of large source code models. Copy approach does not scale up in memory for such models. In Orion, only changed entities are copied between versions, unchanged entities being effectively shared. Then, basic queries take care of retrieving a consistent view of entities in the analysed version. The benchmarks provided in the original paper [LDDF11] report the large gain in memory for our approach with an acceptable overhead in query running time. Overall, it allows Orion to scale up and be usable. We have started to use Orion on two large case studies.

References

[BA96] S. A Bohner and R.S. Arnold. *Software Change Impact Analysis.* IEEE Computer Society Press, 1996.

[LDDF11] Jannik Laval, Simon Denier, Stéphane Ducasse, and Jean-Rémy Falleri. Supporting simultaneous versions for software evolution assessment. *Sci. Comput. Program.*, 76(12):1177–1193, May 2011.

Reverse Engineering Architectural Feature Models

Mathieu Acher[1], Anthony Cleve[1], Philippe Collet[3],
Philippe Merle[2], Laurence Duchien[2], and Philippe Lahire[3]

[1] PReCISE Research Centre, University of Namur, Belgium
{mac, acl}@info.fundp.ac.be
[2] INRIA Lille-Nord Europe, Univ. Lille 1 - CNRS UMR 8022, France
{philippe.merle,laurence.duchien}@inria.fr
[3] Université de Nice Sophia Antipolis - I3S (CNRS UMR 6070), France
{collet,lahire}@i3s.unice.fr

Software product line engineering (SPLE) aims at generating tailor-made software variants for the needs of particular customers or environments. SPLE is gaining more and more attention in different application domains as a means of efficiently producing and maintaining multiple similar software variants. The overall principle is to exploit what software variants – forming a software product line – have in common while managing their differences. Yet the design and implementation of a complete mass customization production line is not always feasible. In many cases, SPLE practitioners rather have to deal with legacy software systems, that were not initially designed as product lines. It is the case of FraSCAti [7], a large and highly configurable component and plugin based system, that have constantly evolved over time and now offers a large number of variants, with many configuration options and extension points.

The variability of such existing and feature rich systems should be properly modeled and managed. A first and essential step is to explicitly identify and represent the variability of a system, including complex constraints between architectural elements. We rely on *feature models* (FMs) that are widely used to model the variability of a system in terms of mandatory, optional and exclusive features as well as Boolean constraints over the features [5,6,9]. FMs specify the set of valid combination of features, called *configurations*, supported by a system – in our case, an architecture. Reverse engineering the FM of an existing system is a challenging activity [8]. The architect knowledge is essential to identify features and to explicit interactions or constraints between them. But the manual creation of FMs is both time-consuming and error-prone. On a large scale, it is very difficult for an architect to guarantee that the resulting FM correctly represents the configurations supported by the architecture. The scope defined by the FM should not be too large (otherwise some unsafe composition of the architectural elements are authorized) or too narrow (otherwise it is a symptom of unused flexibility of the architecture). Both automatic extraction from existing parts and the architect knowledge should be ideally combined to achieve this goal.

We present a comprehensive, tool supported process for reverse engineering architectural FMs [1]. At the starting point of the process, an intentional model of the variability (i.e., an FM) is elaborated by the software architect. As the software architect FM may contain errors, we develop automated techniques to extract and combine different variability descriptions of an architecture, namely

a hierarchical software architecture model and a plugin dependencies model. Then, the extracted FM and the software architect feature model are reconciled in order to reason about their differences. Advanced editing techniques are incrementally applied to integrate the software architect knowledge. The reverse engineering process is made possible by the combined use of FM management operators (aggregate, merge, slice, compare, diff) [3,4], integrated in the language FAMILIAR [2].

We illustrate the process when applied to a representative software system, FraSCAti. Our experience in the context of FraSCAti shows that the automated procedures produce both correct and useful results, thereby significantly reducing manual effort. Firstly, the gap between the FM extracted by the procedure and the FM elaborated by the software architect appears to be manageable, due to an important similarity between the two FMs. The most time-consuming task was to reconcile the granularity levels of both FMs. For this specific activity, tool supported, advanced techniques, such as the safe removal of a feature, are not desirable but mandatory [3,4], since basic manual edits of FMs are not sufficient. Secondly, the extraction procedure recovers most of the variability expressed by the software architect and encourages the software architect to correct his initial model. Thirdly, we learn that the software architect knowledge is required *i)* to scope the architecture (e.g., by restricting the set of configurations of the extracted FM), especially when software artefacts do not correctly document the variability of the system and *ii)* to control the accuracy of the automated procedure. As ongoing work, we are investigating a systematic, tool-supported process for extracting FMs of other FraSCAti versions.

References

1. Mathieu Acher, Anthony Cleve, Philippe Collet, Philippe Merle, Laurence Duchien, and Philippe Lahire. Reverse engineering architectural feature models. In *ECSA'11*, LNCS, pages 220–235. Springer, 2011.
2. Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert France. In *SAC'11*, pages 1333–1340. ACM.
3. Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert France. Separation of Concerns in Feature Modeling: Support and Applications. In *AOSD'12*, pages 1–12. ACM, 2012.
4. Mathieu Acher, Patrick Heymans, Philippe Collet, Clément Quinton, Philippe Lahire, and Philippe Merle. Feature Model Differences. In *CAiSE'12*, LNCS, 2012.
5. K. Czarnecki and A. Wasowski. Feature diagrams and logics: There and back again. In *SPLC'07*, pages 23–34, 2007.
6. P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Generic semantics of feature diagrams. *Comput. Netw.*, 51(2):456–479, 2007.
7. Lionel Seinturier, Philippe Merle, Romain Rouvoy, Daniel Romero, Valerio Schiavoni, and Jean-Bernard Stefani. A component-based middleware platform for reconfigurable service-oriented architectures. *Software: Practice and Experience*, 42(5):559–583, 2012.
8. S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. Reverse engineering feature models. In *ICSE'11*, pages 461–470. ACM, 2011.
9. T. Thüm, D. Batory, and C. Kästner. Reasoning about edits to feature models. In *ICSE'09*, pages 254–264. IEEE, 2009.

Table ronde Logiciels et Industriels

Création d'un Club des Partenaires Industriels du GDR-GPL : Table ronde sur les attentes de membres potentiels de ce club

Animée par Christel Seguin (Onera)

Le Groupement De Recherche du CNRS sur le Génie de la Programmation et du Logiciel (GDR GPL http://gdr-gpl.cnrs.fr/) souhaite se doter d'un "Club des Partenaires Industriels". Ce club aurait pour vocation de permettre une meilleure connaissance mutuelle et faciliter le partage des informations entre recherche et industrie.
La table ronde rassemble des industriels de différents secteurs de l'industrie du logiciel (télécommunication, multi-média, systèmes embarqués, ...) dans le but de préciser les attentes des membres potentiels de ce club, tant sur les finalités du club que sur ses modalités de fonctionnement.

Posters et démonstrations

A Domain Specific Language (DSL) for temporal knowledge

POSTERS ET DÉMONSTRATIONS

Cyril Faucher, Jean-Yves Lafaye, Frédéric Bertrand

L3i (Informatique, Image et Interaction) - University of La Rochelle – France
cyril.faucher@univ-lr.fr - http://l3i.univ-larochelle.fr/Faucher-Cyril.html

SUPPORTED BY ANR

This work is granted by the French National Research Agency (ANR-Contint, RelaxMultiMedias 2 project)
http://relaxmultimedia2.univ-lr.fr

Objectives

- Modeling intensional temporal expressions (*vs* extensional) for representing periodical occurrences e.g., "on the 1st Tuesday of every Month".
- Providing a textual concrete syntax for this DSL.
- Validating candidate expressions first with constraint programming thanks to constraint checking (OCL class invariants) and second with Prolog predicates.
- Querying temporal knowledge with intensional temporal expressions.

Context

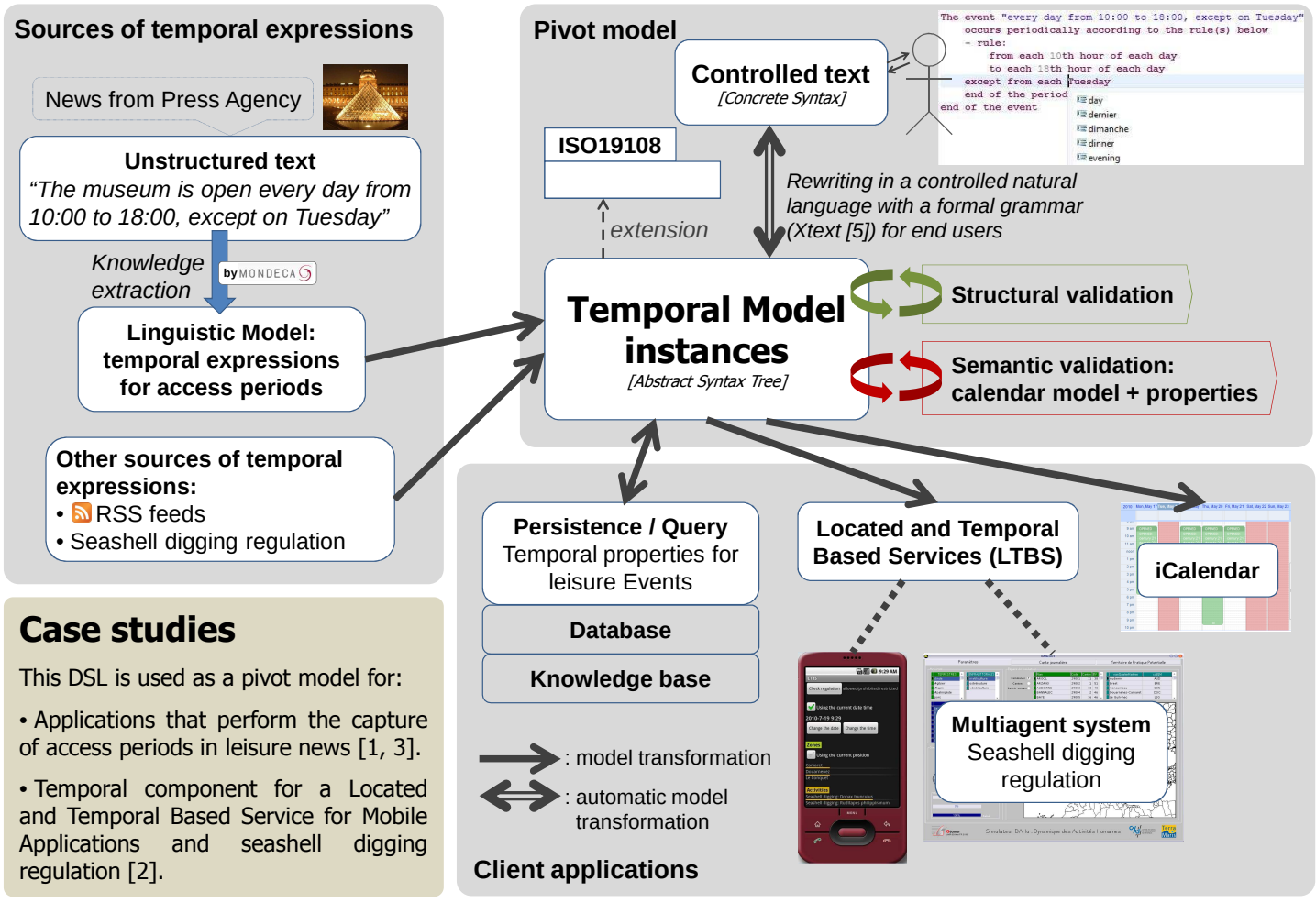
- Temporal data modeling.
- Model Driven Engineering: model transformation, grammar and pivot model.
- Knowledge engineering: acquisition, modeling and querying.

Structural and Semantic validation

Temporal expression: "6th week of each year" is syntactically and semantically correct, but "6th week of each month" is semantically meaningless although syntactically correct. Therefore, we must specify both a structural and semantic validation process ➡ "calendar model".

Temporal Model

We extended the ISO 19108 [4] standard and proposed an object model (Temporal Model) for intensional temporal expressions. This model includes the main concepts for modeling: Frequency, Periodic Interval, etc.



Case studies

This DSL is used as a pivot model for:

- Applications that perform the capture of access periods in leisure news [1, 3].
- Temporal component for a Located and Temporal Based Service for Mobile Applications and seashell digging regulation [2].

References

1. Faucher, C., Teissèdre, C., Lafaye, J.Y., Bertrand, F., Temporal Knowledge Acquisition and Modeling. In: EKAW 2010, October 2010, Lisbon (Portugal), volume 6317 of LNCS
2. Faucher, C., Tissot, C., Lafaye, J.Y., Bertrand, F., Benefits of a periodic temporal model for the simulation of human activities. In: GeoVA(t) Workshop at AGILE 2010, May 2010, Guimaraes (Portugal)
3. TKA (Temporal Knowledge Acquisition): http://client2.mondeca.com/AccessPeriodEditor/
4. ISO 19108: http://www.iso.org/iso/catalogue_detail.htm?csnumber=26013
5. XText: http://www.eclipse.org/Xtext/

Composition de lignes de produits logiciels dirigée par les modèles

Simon Urli (urli@i3s.unice.fr)

Université Nice-Sophia Antipolis, I3S / CNRS UMR 7271

1 Problématique

Les Lignes de Produits Logiciels (LPL) permettent la construction de logiciels à moindre coût par réutilisation d'assets logiciels spécifiques à un domaine d'application [1]. Ces artefacts représentent à la fois les éléments variables et communs d'une famille de logiciels. La variabilité est habituellement gérée en considérant que toutes les variantes d'un logiciel dérivent de la même racine. Cependant, la construction de logiciel par assemblage de composants externes provenant de différents domaines est maintenant une pratique répandue et encouragée. Il devient donc nécessaire d'être en mesure de gérer la composition de produits à partir de multiples LPL représentant des domaines différents [2].

2 YourCast : vers une composition de ligne de produits

Cette problématique générale découle d'un cas concret d'utilisation, le projet ANR Emergence YourCast[1], dont le but est de permettre la réalisation d'un outil de création de système de diffusion d'informations utilisable par tout un chacun. Dans ce contexte, l'utilisation d'une LPL semble particulièrement pertinente de par l'importance de la variabilité latente aux systèmes de diffusion. Fort d'une expérience de plusieurs années sur un système pré-existant (JSeduite), nous avons pu constater que les LPL et le formalisme des modèles de variabilité souffrent de certaines limitations auxquelles nous avons été confrontés.

JSeduite[2] a été écrit à l'origine afin de répondre aux besoins des institutions académiques [3]. Il supporte la diffusion d'informations provenant de différentes instances (par exemple, des réseaux de transports ou du restaurant universitaire), vers de multiples dispositifs (smartphone, PDA, écrans d'ordinateurs, écrans publiques).

JSeduite est actuellement déployé dans trois institutions : l'école d'ingénieur Polytech'Sophia et deux instituts spécialisés pour les personnes présentant un handicap visuel.

Cependant, si ce système fonctionne à l'heure actuelle, il n'offre que peu de possibilités d'évolution, et n'est pas destiné à être adapté par l'utilisateur. Par ailleurs, le système n'a été déployé que pour une utilisation à petite échelle de l'ordre d'une dizaine d'écrans. YourCast a pour objectif de permettre (i) une utilisation de l'ordre de la centaines d'écrans, et (ii) une construction du système entièrement dirigée par un utilisateur final sélectionnant les modules qu'il souhaite utiliser.

Les choix des différentes parties du logiciel devront porter sur des concepts très différents :
- *Les sources d'informations* : il peut s'agir de services tels que Twitter, FlickR, des Flux RSS, des calendriers, ou encore des services internes à des entreprises.
- *Les rendus d'affichage* : il s'agit d'un processus traitant les informations afin de les afficher de la façon souhaitée par l'utilisateur ; par exemple, un album photo affiché sous forme de mosaïque ou en diapositives.
- *Le design général* : il s'agit d'un squelette de l'écran d'affichage, contenant les différentes zones d'affichage et le style général de la page ; le design est adapté à un support.
- *Les politiques* : il s'agit d'un processus effectuant des opérations sur les sources d'informations ; par exemple, une politique de filtrage d'informations, ou de tri sur des photographies.

De plus, les choix qui peuvent être faits ici sont extrêmement liés aux contextes d'utilisation : un diffuseur d'information dans un institut spécialisés pour les personnes ayant un handicap

[1] http://yourcast.unice.fr
[2] http://www.jseduite.org

visuel sera extrêmement différent d'un diffuseur réalisé pour un grand rassemblement associatif. La manière même dont ils sont construits sera très différente, privilégiant un aspect ou un autre.

3 Travaux en cours

La figure 1 présente le processus global de construction d'un système de diffusion d'informations par un utilisateur.

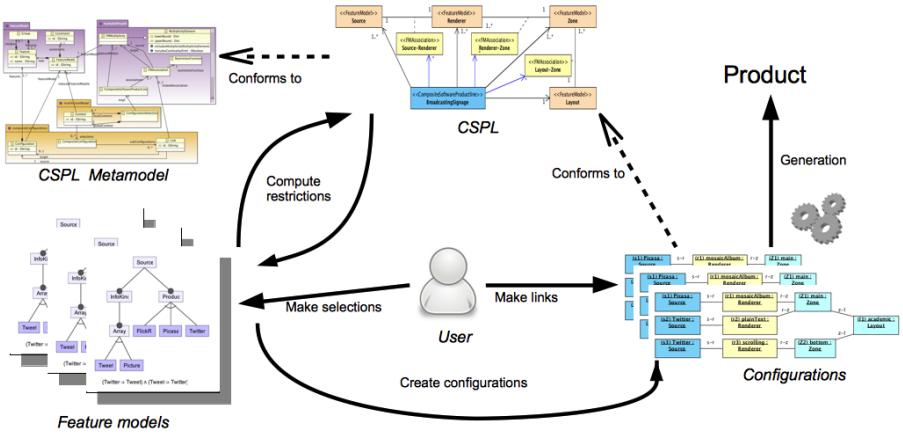


Fig. 1. Global process

Nous considérons que chacun des concepts des systèmes de diffusions peuvent être représentés par un modèle de fonctionnalités (*Feature Model* ou *FM*) indépendant dans lequel l'utilisateur peut choisir un ou plusieurs produits [4].

Le choix d'un produit peut cependant avoir un impact sur les futurs choix de l'utilisateur. Par exemple une source affichant des photos ne pourra pas être diffusée en utilisant une méthode de rendu ne pouvant traiter que du texte. De ce fait, chacun des choix de l'utilisateur doit pouvoir impacter les autres FMs : ces restrictions vont être guidées par un modèle du système représentant les différents FM, les liens entre ceux-ci et les opérations de restrictions à effectuer. Nous utiliserons entre autres pour ces restrictions les opérateurs définis dans [5]. Ce modèle devra être conforme à un méta-modèle que nous définissons.

Enfin, les différents produits sélectionnés par l'utilisateur doivent être liés entre eux pour qu'une configuration du système soit correctement instancié. Par exemple, une source sélectionnée doit obligatoirement être liée à une méthode de rendu afin de pouvoir être affichée. Ces configurations doivent être conformes au modèle afin que le système de diffusion d'information puisse être généré.

References

[1] Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag (2005)

[2] Bosch, J.: Toward compositional software product lines. IEEE Software **27** (2010) 29–34

[3] Blay-Fornarino, M., Hourdin, V., Joffroy, C., Lavirotte, S., Mosser, S., Pinna-Déry, A.M., Renevier, P., Riveill, M., Tigli, J.Y.: Architecture pour l'adaptation de Systèmes d'Information Interactifs Orientés Services. Revues des Sciences et Technologies de l'Information (RSTI) (December 2007) 93–118

[4] Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.A., Spencer Peterson, A.: Feature-oriented domain analysis (FODA) feasibility study. Technical Report November, The Software Engineering Institute (1990)

[5] Acher, M., Collet, P., Lahire, P., France, R.: Separation of Concerns in Feature Modeling: Support and Applications. In: Aspect-Oriented Software Development(AOSD'12) AR=25%. , ACM (March 2012)

VPraxis: a Uniform Approach to Support Mining Software Repositories

Cédric Teyton, Jannik Laval, Jean-Rémy Falleri, and Xavier Blanc
name.surname@labri.fr

Univ. Bordeaux, LaBRI, UMR 5800,F-33400 Talence, France

1 VPraxis at a glance

Mining software repositories is a challenging task in the context of software evolution. Along a software project, both developers and managers frequently ask questions related to the software evolution [1, 2, 4, 5]. Both categories target distinct elements. Managers want to thoroughly understand the evolution of their software development project. For example, they want to get answers to questions such as : "*Which operations change together ?*". On the other hand, developers look for convenience in their collaborative tasks and one of their questions is typically "*Who has made changes to my modules?*". Thus, there is a need to consider the source code elements or software objects (e.g. a class, method or function) and the metadata provided by the software repository. In that case, it becomes possible to associate operations that affect the evolution of the code with time and author information.

VPraxis is an approach to uniformly support mining software repositories. The key idea is to represent a software evolution as a **history** of editing actions. The contributions are listed as follows :

- VPraxis uses an **operation-based** model that unifies both source code information and versioning data.
- The VPraxis model is **independent** of the language and the versioning system being used.
- It includes an extensible **framework** that implements the model and the algorithm for the history construction.
- The framework currently supports two main programming **languages** (Java and C) and two well-known **versioning** systems (SVN and Git).
- **Queries** can be run against the histories. Queries are supported in 3 distinct formats languages, **SQL**, **Prolog** and **XQuery**.

2 Model and History Construction

A VPraxis *state* represents the sequences of actions for a particular version. Each action holds two versioning information, one temporal indicator of when the action happened, that is the revision number, and the author of the action (the person who performed the commit). VPraxis features 6 editing actions:

- `create(e,t,r,a)` (resp. `delete(e,t,r,a)`) creates (resp. deletes) an element e with the tag t. A created (resp. deleted) element has neither properties nor references. e also represents the unique identifier of the element in the digraph.
- `addProperty(e,t,v,r,a)` (resp. `remProperty(e,t,v,r,a)`) adds (resp. removes) the value v to the property tagged t of the element e.
- `addReference(e,t,l,r,a)` (resp. `remReference(e,t,l,r,a)`) links (resp. unlinks) the target element l to the reference tagged t of the element e.

A VPraxis history is built incrementally by appending a *diff* between two sequences of actions corresponding to the *states* of the software at time r and $r+1$ respectively. The history thus

contains the sequence of actions necessary to lead the initial state of the software to its final state, regarding only its source code elements evolution.

Figure 1 represents a sample software composed by 2 revisions. On the left, there is the source code view, where on the right there is the VPraxis representation. The 2nd revision features the removal of the property *abstract* of the class *Vehicle* and a replacement of visibility for the attribute *Driver*, and one method has been added. Those two *diff* are part of the software *history*.

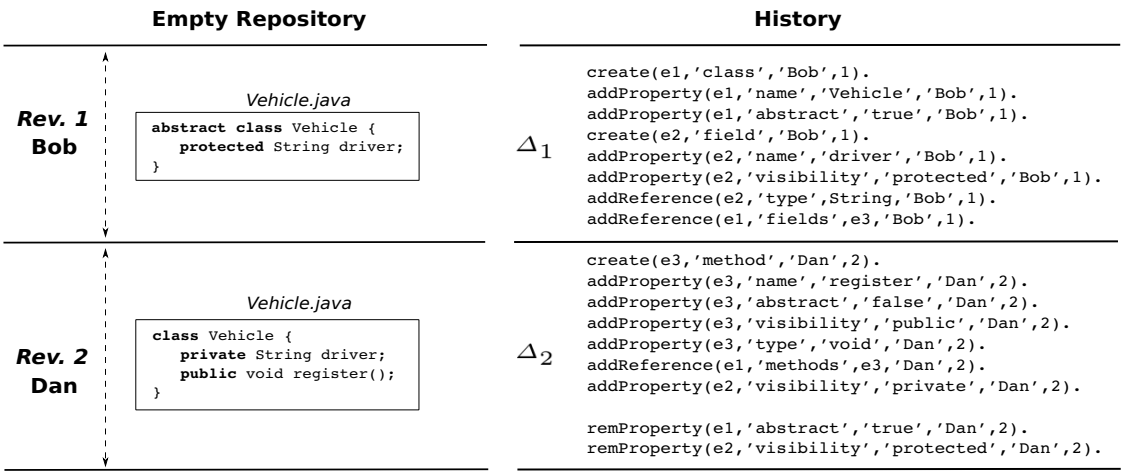


Fig. 1. Evolution of a sample software and its corresponding VPraxis history

3 Implemented Architecture

VPraxis architecture features separated modules for the three keys steps of the history construction: (i) a **software repository extractor** is in charge of repository connection, revisions browsing and versioned files retrieval, (ii) an **actions generator** parses and analyses the source code files, generates lists of corresponding actions, and (iii) a **history management** computes *diff* between two list of actions, maintains the history and stores it in a persistent manner in a given format. Our tool is available online (http://code.google.com/p/harmony/).

4 History Querying

Queries can be run against a VPraxis history to answer questions targeted by developers and managers. The format exposed in Figure 1 is suitable for **Prolog** as it stands as a database facts. Queries can be formulated to focus on one of any combination of the following factors : source code **properties** and **references**, **temporal** window, and the **social** aspect.

References

1. D'Ambros, M., Gall, H., Lanza, M., Pinzger, M.: Analysing software repositories to understand software evolution. In: Mens, T., Demeyer, S. (eds.) Software Evolution, pp. 37–67. Springer (2008)
2. Fritz, T., Murphy, G.C.: Using information fragments to answer the questions developers ask. In: Kramer et al. [3], pp. 175–184
3. Kramer, J., Bishop, J., Devanbu, P.T., Uchitel, S. (eds.): Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010. ACM (2010)
4. LaToza, T.D., Myers, B.A.: Developers ask reachability questions. In: Kramer et al. [3], pp. 185–194
5. LaToza, T.D., Venolia, G., DeLine, R.: Maintaining mental models: a study of developer work habits. In: Osterweil, L.J., Rombach, H.D., Soffa, M.L. (eds.) ICSE. pp. 492–501. ACM (2006)

Découpez vos Modèles avec *Kompren* : une Démonstration

Arnaud Blouin, Benoit Combemale, and Benoit Baudry

Triskell, INRIA/IRISA, Rennes
{ablouin,bcombemale}@irisa.fr bbaudry@inria.fr

1 Introduction

Le découpage de modèles (*model slicing*) est une opération qui extrait un sous-ensemble d'un modèle dans un but précis, tel que la compréhension de modèles et l'amélioration de performances. Cependant, les approches actuelles de découpage de modèles sont dédiées à un domaine de modélisation (métamodèle) particulier. L'apparition régulière de nouveaux domaines nécessite alors la conception et l'implantation de nouvelles fonctionnalités de découpage de modèles. Dans nos récents travaux, nous avons proposé Kompren, un environnement pour définir et générer des découpeurs de modèles (*model slicers*) pour tout type de domaine de modélisation [1]. Cette démonstration présente les différents outils de Kompren [1] et trois cas d'utilisation illustrant l'expressivité du langage.

2 Présentation de *Kompren*

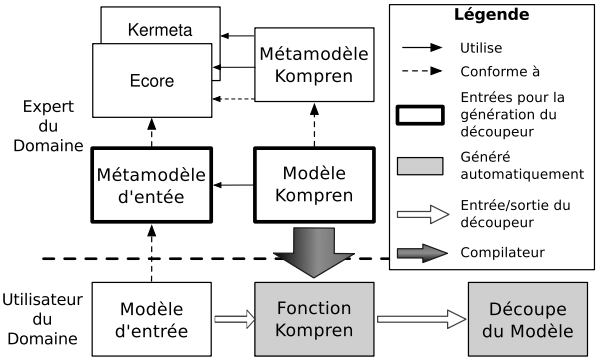


Figure 1. Vue d'ensemble du processus d'utilisation de *Kompren*

La figure 1 présente une vue d'ensemble de l'utilisation de *Kompren*. Tous les concepts et relations de *Kompren* sont définis dans le métamodèle de *Kompren*.

1. http ://people.irisa.fr/Arnaud.Blouin/software_kompren.html

Un modèle *Kompren* décrit un découpeur de modèles en sélectionnant dans le métamodèle d'entrée les classes et les propriétés d'intérêts qui seront extraites lors de l'exécution, le métamodèle d'entrée étant un modèle Ecore. Une fois un modèle *Kompren* défini (une syntaxe textuelle et son éditeur sont fournis), il peut être compilé en une fonction *Kompren* consistant en du byte-code Scala/Java exécutable. Cette fonction prend en entrée un modèle à découper, conforme au métamodèle d'entrée, et produit en sortie une découpe de ce modèle.

Par défaut, l'exécution d'un découpeur (fonction *Kompren*) sur un modèle d'entrée assure la conformité du modèle de sortie (*slice*) en extrayant les éléments nécessaires à la préservation de cette conformité même s'ils ne sont pas explicitement sélectionnés dans le modèle *Kompren*. Cependant, la conformité n'est pas requise dans certains cas ; c'est pourquoi *Kompren* offre des fonctionnalités relâchant cette conformité afin de permettre aux développeurs de paramétrer les découpeurs de modèles, notamment à l'aide du langage d'action Kermeta [2].

3 Description de la Démonstration

La démonstration vise à expliquer comment *Kompren* fonctionne et les différentes utilisations qu'il est possible d'en faire. Pour cela, cette démonstration se compose des scénarios suivants :

- **Compréhension de modèles**. Le zoom sémantique et le filtrage dynamique sont des techniques de visualisation dont le but est d'afficher uniquement les informations relatives à une préoccupation donnée, ce qui consiste donc en une forme de découpage de modèles. Nous montrons comment *Kompren* peut être utilisé pour définir de telles techniques de visualisation puis intégré dans un visualiseur de modèles.
- **Optimisation**. Étant donné un point d'entrée dans un modèle, le découpage de modèles peut être utilisé pour supprimer les éléments non pertinents relativement à ce point d'entrée. Nous illustrons ce principe en expliquant comment *Kompren* est utilisé dans l'environnement Kermeta pour réduire le temps de compilation des programmes Kermeta 2.
- **Analyse statique**. Étant donnée une opération *op* dans un modèle exécutable, le découpage de modèles peut être utilisé pour identifier les variables non utilisées déclarées dans *op*. Nous présentons comment *Kompren* permet de définir une telle analyse pour des opérations Kermeta.

Cette démonstration s'effectue à l'aide des outils de développement de *Kompren*, notamment son éditeur textuel, son compilateur et Kermeta 2.

Références

1. Arnaud Blouin, Benoît Combemale, Benoit Baudry, and Olivier Beaudoux. Modeling model slicers. In *Proceedings of MODELS'11*, pages 62–76, 2011.
2. Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving executability into object-oriented meta-languages. In *Proc. of MODELS'05*, pages 264–278, 2005.

Ingénierie dirigée par les modèles pour structurer et partager un référentiel d'exigences de sûreté dans la durée

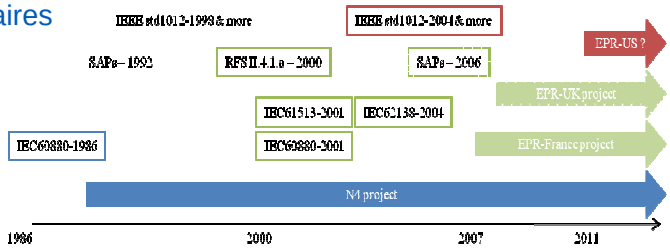
Nicolas Sannier* **

** EDF R&D STEP P1A – 6 quai Watier 78401 Chatou cedex*

** Inria Rennes EPI Triskell – Campus de Beaulieu 35042 Rennes cedex*

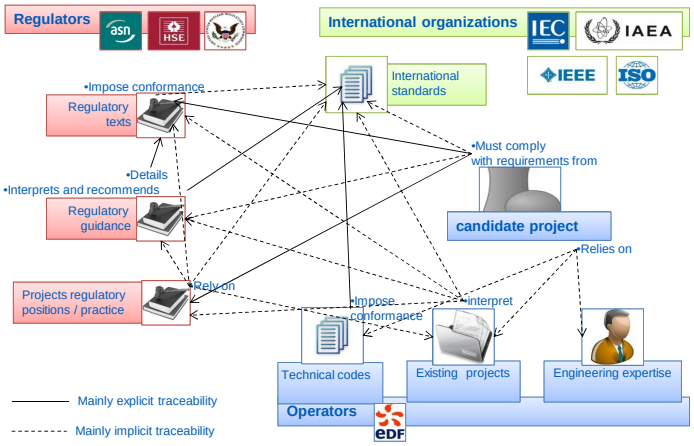
Variabilité et traçabilité des exigences réglementaires

- Une pratique non formalisée sur des projets à longue durée de vie
- Des exigences volontairement ambigües pour certaines et potentiellement interprétées différemment d'un pays à un autre
- Des corpus réglementaires différents et réévalués
- Des attentes et des approches différentes des autorités de sûreté
- Comprendre et rapprocher ces différences



Variabilité des exigences dans le temps et dans l'espace

Publié à Model-driven Requirements Engineering (MoDRE) 2011



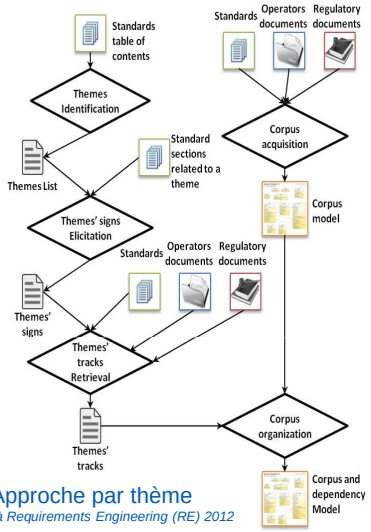
Sources d'exigences dans les projets et traçabilité

soumis à Requirements Engineering (RE) 2012

Ingénierie des exigences dirigée par les modèles

Un métamodèle pour :

- exprimer les concepts clés du monde réglementaire
  - Différents types de textes, différents types d'exigences, contextes …
- organiser la connaissance en un ensemble de relation
  - Références, pratiques acceptables …
  - Représenter différemment les connaissances (support à une interface et à analyse)



Approche par thème

soumis à Requirements Engineering (RE) 2012

Des actions centrées sur les documents et sur les expertises métier

- Acquérir un corpus d'exigences
- Capturer et Représenter ce corpus dans un modèle d'exigences
- Déterminer et représenter les premiers éléments de traçabilité dans le modèle
- Déterminer les grandes préoccupations (les thèmes) dans ce corpus
- Déterminer les régions du corpus qui sont relatives à ces thèmes
- Rendre la main aux experts pour continuer à enrichir le modèle
- Aider à retrouver certaines pratiques non formalisées et connaissances tacites

6.2 Self-supervision

6.2.A The software of the computer-based system **shall** supervise the hardware during operation within specified time intervals and the software behaviour **(A.2.2)**.
This is considered to be a primary factor in achieving high overall system reliability.

6.2.B Those parts of the memory that contain code or invariable data **shall** be monitored to detect unintended changes.

6.2.C The self-supervision **should** be able to detect to the extent practicable:
- *Random failure of hardware components;*
- *Erroneous behavior of software (e.g. deviations from specified software processing and operating conditions or data corruption);*
- *Erroneous data transmission between different processing units.*

6.2.D **If** a failure is detected by the software during plant operation, the software **shall** take appropriate and timely response. Those **shall** be implemented according to the system reactions required by the specification and to **IEC 61513** system design rules.
This **may** require giving due consideration to avoiding spurious actuation.

6.2.E Self-supervision **shall not** adversely affect the intended system functions.

6.2.F It **should** be possible to automatically collect all useful diagnostic information arising from software self-supervision.

Partitioning, **definition**, **reference**, information, scope, *composition*, characterization

Possibles acquisitions d'information sur les textes

présenté à Model-Driven Requirements Engineering (MoDRE) 2011

Contributions

- **Défis pour la variabilité et la traçabilité des exigences en ingénierie système**, Nicolas Sannier & Benoît Baudry, *in INFORSID 2011*, Lille, 24-26 may 2011
- **Formalizing standards and regulations variability in longlife projects. A challenge for model-driven engineering**, Nicolas Sannier, Benoît Baudry & Thuy Nguyen, *in 1st workshop MODRE Model-Driven Requirements Engineering*, Trento, Italy, 29th august 2011.
- **Retrieving Themes' Tracks in Nuclear International Standards**, Nicolas Sannier, Benoît Baudry & Thuy Nguyen, *soumis à RE2012*

Contact : *Nicolas Sannier – nicolas.sannier@inria.fr @edf.fr*

Prototyping Static Analysis Certification using Why3

Frederic Besson, Pierre-Emmanuel Cornilleau, and Thomas Jensen

INRIA Rennes - Bretagne Atlantique
prenom.nom@inria.fr,
WWW home page: http://www.irisa.fr/celtique/ext/chk-sa-boogie/

Keywords: static analysis, result certification, intermediate verification language

1 Introduction.

Software reliability can be established through many means. One is to use static program analysers to detect errors at compile-time, or even prove the absence of run-time errors. However, such *proofs* rely on the *soundness* of the analyser: there is no verifiable proof object, only the fact that the analyser did not report any error. Hence the analyser is included in the Trusted Computing Base(TCB): the set of all component critical to the security of the application. But a precise and efficient static analyser is a complicated piece of software and may not be acceptable in the TCB, depending on the application.

One solution to reduce the TCB is to obtain a *foundational* proof that the program is free of run-time errors: a proof relying only on a proof-checker and a formal semantic definition of run-time safety. It amounts to counting the proof-checker in the TCB in place of the analyser. A foundational proof of safety can be obtained by certifying the analyser inside a proof-checker. This approach requires to develop and prove in Coq the whole analyser which is a formidable effort of certification and raises efficiency concerns, Coq being a pure lambda-calculus language. Another way to obtain a foundational proof of safety is to certify, inside the proof-checker, a dedicated verifier of analysis result rather than the analyser.

We revisit the result certification approach and try to alleviate the need for dedicated checkers. We concentrate on analysers formalised in the abstract interpretation framework [2], and use Intermediate Verification Languages to establish trust in the results.

2 Abstract Interpretation result certification.

In the Abstract Interpretation framework, an analyser calculates abstractions of programs' behaviours by a post-fixpoint iterative process. A concretisation function associates an abstraction to a set of states of execution in the semantics of the analysed program. If this set contains no error state, the analyser concludes that the program is runtime-error free. Therefore, checking the result of an analyser amounts to i) making sure the abstraction is a fixpoint, and ii) checking that the concretisation contains no error state.

An Intermediate Verification Languages (IVL) is a simple language with a clear axiomatic semantic used in Hoare style program verification as a middleground between source languages (*e.g.*, C, Java) and the logic of a reasoning engine (*e.g.*, automated theorem provers, proof assitants). It is associated with a Weakest Precondtion calculus (WP) which gives Verification Condition (VC) sufficient to ensure that i) a propriety is an program invariant, and ii) a precondition is sufficient to prove that a program respects semantic conditions, such as array bound check for array access.

We propose to translate the abstraction of a program into program invariants using the concretisation function, reducing the proof that the abstraction was a fixpoint to a proof of program invariants. To describe the semantics of the analysed language in the IVL, we propose to implement in the IVL an interpreter for the analysed language. The semantic condition for error free execution are described in the logic of the IVL by precondition on the interpreter, and semantic states are described by data-structures of the IVL. A program and its abstraction are represented

by a fixed set of functions and predicate implemented in the IVL, describing the control flow and instruction set of the program, and the invariants deduced from the abstraction. These functions and predicates are used to implement the interpreter and to specify its pre and post-conditions.

To prove a particular program free of runtime errors, we need to prove that the implementation of the functions and predicate representing the program and the invariants validates the interpreter's VC. It can be done using usual verification techniques available for the IVL, such as translation of VCs in formulae fed to Automated Theorem Provers.

3 Prototyping using Why3.

To prototype our approach, we instantiate in the Why3 [1] framework the scheme on two analyses performed on different programs: an interval analyser and a relational polyhedric analyser. The concretisation of program behaviour abstractions to program invariants is straightforward, as both numeric analyses calculate sets of linear constraints on the content of program variables at each program point.

The proof of the interpreter's VC is done in two part. To implement the interpreter, we need only to declare the functions and predicates representing programs, and the relations between them are defined by axioms in the IVL. In a first time, these axioms are used to prove the interpreter VC in the general case. This proof is done once, and can be done manually. Then, to establish this result for a particular program, the program specific functions and predicates are implemented and are proved to validate the axioms. The certification scheme requires automation of this proof.

For all the tested programs and analysers results, ATPs (Z3 and Alt-Ergo in our experiments) were able to prove that the Why3 functions and predicates representing programs and abstractions validate the axioms of the interpreter. The results illustrate the need for cooperation between ATPs to discharge all proofs, and the relevance of transformation of goals at the level of the IVL.

Apart from splitting the proof of interpreter's VCs in two using axiom, another way to help the ATPs is to instantiate the axioms on a particular program point. During our experiments, some of the full theorems could not be proved automatically, whereas instantiation of those theorems on relevant program points could be. And with these instantiated lemmas in their context, the ATPs were finally able to discharge the full theorem. These instantiations can be generated automatically, and no manual proof or prior knowledge is needed to obtain a full proof of the program. Some of the predicates used in the axiomatisation of analysis results could be expressed using combinations of pre, post and the functions describing the program, reducing the number of axioms needed to prove the interpreter. Nevertheless, introducing redundancy and intermediate predicate makes it easier for automatic provers to discharge the verification conditions, and the axioms introduced by these redundant predicates are discharged easily by ATPs and do not introduce any noticeable overhead.

Completeness. If the soundness of the approach is formally proved, its completeness is not clearly established, as the axiomatisation theorems may not be appropriate for all analysers. They are sufficient, as the proof of the interpreter established, but they may not be necessary. The use of weaker theorems may complicate the proof of the interpreter, which is not a problem as it is done only once and may even simplify the automation of the proof of analysers results. However, it is not necessary in our case studies, as every verification condition is discharged using ATPs. Nevertheless the amount of automation is a concern. The statement of the semantics of the language and the axiomatisation theorems were carefully crafted, as minor syntactic differences can make a huge difference when discharging the verification conditions.

References

1. François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011*, pages 53–64, August 2011.
2. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL 1977*, pages 238–252. ACM, 1977.

Comparaison de modèles filtrée pour le test de transformations de modèles

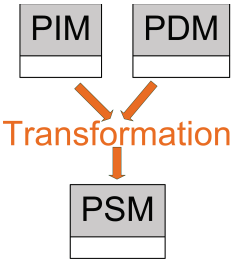
Olivier FINOT Ecole doctorale STIM LINA UMR CNRS 6241 Equipe : AeLoS
olivier.finot@univ-nantes.fr

Transformation de modèles

Les modèles sont à la base de l'IDM. Ils évoluent sous l'effet de transformations, qui sont automatisées.

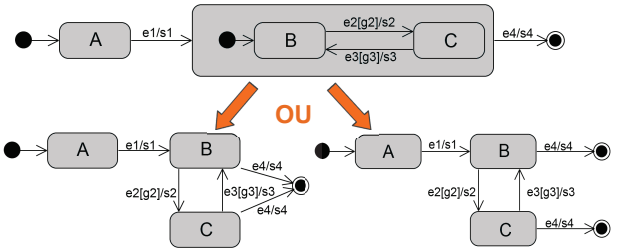
Exemple :
Model Driven Architecture

- Modélisation d'un système indépendant de la plateforme (PIM)
- Modélisation de la plateforme (PDM)
- Transformation en un modèle spécifique à la plateforme (PSM)



Sorties polymorphes

Certaines transformations appliquées à un modèle d'entrée donné peuvent produire plusieurs variantes d'un même modèle de sortie.

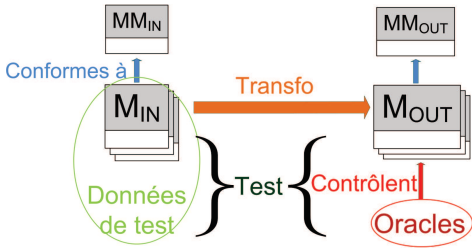


Exemple : mise à plat d'une machine à état

- Élimination des états composites
- Équivalence sémantique entre les variantes d'un même modèle de sortie

Test de transformations

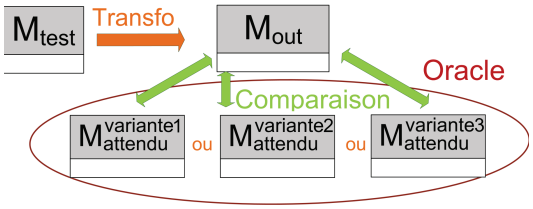
Les transformations de modèles doivent être testées pour éviter propagation et dissémination d'erreurs.



- Des données de test sont sélectionnées en entrée
- Les oracles contrôlent les modèles obtenus
  - Typiquement : comparaison d'un modèle de sortie obtenu avec un modèle de référence

Oracle du test

L'oracle du test d'une transformation aux sorties polymorphes doit contrôler le modèle de sortie obtenu par rapport à toutes les variantes possibles.

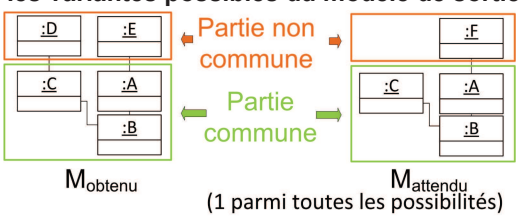


PROBLEMATIQUE:

- Coût de la définition et du contrôle des multiples versions
- Difficulté de définir un oracle unique pour comparer le résultat à toutes les variantes

Comparaison partielle

PROPOSITION :
Contrôler la partie du modèle obtenu commune à toutes les variantes possibles du modèle de sortie.



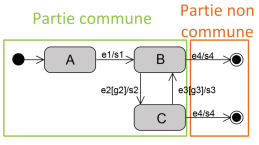
(1 parmi toutes les possibilités)

- Utilisation d'un unique modèle de référence complet
- Comparaison de la partie commune des modèles
- Obtention d'un verdict partiel utile pour
  - Augmenter la confiance dans la transformation
  - Localiser des erreurs de la partie commune

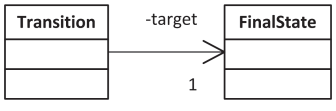
Mise en œuvre

Mise en œuvre de notre proposition dans le framework EMF d'Eclipse avec EMFCompare pour comparer les modèles.

- Filtrage du résultat de la comparaison pour rejeter les résultats concernant la partie non commune :



- Éléments de la partie non commune à filtrer définis dans un pattern composé de métaclasses :



- Pour la **mise à plat d'une machine à état** : comparaison des modèles hors les états finaux et leurs transitions

Directeur de thèse :
Christian ATTIOGBE (AeLoS)

Co-encadrants :
Jean-Marie MOTTU (AeLoS), Gerson SUNYE (ATLAS GDD)

Sélection automatique de configurations de pour le test d'applications

Aymeric HERVIEU
KEREVAL – INRIA aymeric.hervieu@Kereval.com

CONTEXTE ET PROBLÉMATIQUE

La variabilité dans le logiciel existe sous de nombreuses formes :

- **dans les applications à base de composants** comme Eclipse **(fig. 1.)**.
- **Dans les environnements d'exécution** : le logiciel Business Everywhere (BIEW) de Orange a été spécifié pour être validé sur plus de 600000 environnements différents
- **Les applications mobiles** devant fonctionner sur des milliers de téléphones différents

Cette variabilité mène généralement à des milliers ou des millions de cas possibles. Comment peut on capturer cette variabilité et comment peut on l'échantillonner pour réduire ces combinaisons ?

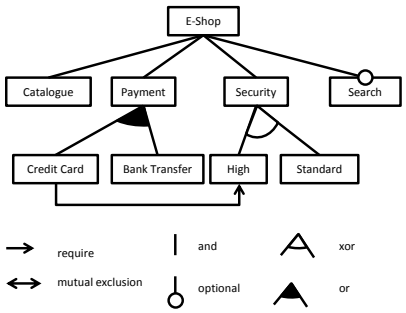


Figure 2. Feature model représentant une boutique en ligne configurable.

SÉLECTION DE CONFIGURATIONS

Les features models représentent un très grand nombre de configurations possibles (plusieurs millions).
Lors des activités de test il est impossible de tester toutes ces configurations, il est nécessaire d'en sélectionner.
Le critère PAIRWISE est un critère de sélection de configurations, respectant le modèle où toutes les interactions possibles 2 à 2 des features sont testées.
L'application de ce critère a abouti à la création d'un outil : PACOGEN, utilisant des techniques de programmation par contraintes.

Modèles	Nombre de features	Nombre total de configurations	Nombre de configurations pairwise
STACK FM	17	432	10
CRISIS MANAGEMENT	17	20	5
FAME DBMS	21	320	8
SMART HOME	35	1 048 576	8
INVENTORY	37	2 028 096	15
SIENNA	38	2 520	20
WEB PORTAL	43	2 120 800	16
DOC GENERATION	44	55 700 000	17
ARCADE GAME	61	3 300 000 000	13
MODEL TRANSFORMATION	88	$1{,}65.10^{13}$	25
COCHE ECOLOGICO	94	23 200 000	92
ELECTRONIC SHOPPING	287	$2{,}26.10^{49}$	37

Figure 4. Résultat de l'échantillonnage des configurations en utilisant PACOGEN sur différents modèles issus du site splot-research.com

	Java	Java EE	C/C++	C/C++ Linux	RCP / RAP	Modeling
RCP/Platform	✓	✓	✓	✓	✓	✓
CVS	✓	✓	✓	✓	✓	✓
EGIT			✓	✓	✓	✓
EMF	✓	✓				✓
GET	✓	✓				✓
JDT	✓	✓			✓	✓
Mylyn	✓	✓	✓	✓	✓	✓
Web Tools		✓				
Linux Tools			✓	✓		
Java EE Tools		✓			✓	
XML Tools	✓	✓				
RSE		✓	✓	✓		

Figure 1. Variabilité au sein de Eclipse. Les colonnes représentent les différentes versions téléchargeables, les lignes les composants inclus dans chaque version

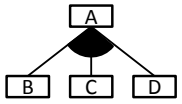
UN MODÈLE DE VARIABILITÉ : LE FEATURE MODEL

Un feature model représente un ensemble de **configurations**. Ces configurations sont une sélection de features respectant les règles du modèle. Une feature est **sélectionnée** si sa valeur est égale à 1, sinon elle est égale à 0 **(fig 3.).**

Les règles du modèles sont représentées par des opérateurs reliant les features entre elles. Chaque opérateur a une sémantique particulière : par exemple l'opérateur xor n'autorise la sélection que d'une seule feature à la fois (ci contre une boutique électronique a une politique de sécurité standard ou élevée).

A	B	C	D
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	1
1	1	1	0

Ensemble de configurations pairwise invalide ([2] ne respecte pas le modèle) : si A n'est pas sélectionné, B,C et D ne peuvent l'être.



Modèle liant 4 features par l'opérateur or

A	B	C	D
1	0	1	1
1	1	1	0
1	1	0	1
0	0	0	0

Ensemble de configurations pairwise valide

Figure 3. Configuration Pairwise sur un modèle simple

PUBLICATIONS

PACOGEN : Automatic Generation of Pairwise Test Configurations from Feature Models (Aymeric Hervieu, Benoit Baudry, Arnaud Gotlieb), In Proceedings of the International Symposium on Software Reliability Engineering (ISSRE'11), 2011.

Managing environment variablity during software Testing : an industrial experience (Aymeric Hervieu, Benoit Baudry, Arnaud Gotlieb), soumis à SPLC 2012

EXTENSION DU MODEL-BASED TESTING POUR LA PRISE EN COMPTE DE LA VARIABILITE DANS LES SYSTEMES COMPLEXES

Hamza Samih, Benoit Baudry, Hélène Le Guen

{hamza.samih,benoit.baudry}@inria.fr, helene.leguen@all4tec.net
ALL4tec, INRIA

1 Introduction

Le test est la phase la plus couteuse dans un cycle de développement. Plusieurs méthodes de test sont utilisées pour optimiser cette phase, parmi ces solutions le Model-based Testing (MBT) est de plus en plus utilisé pour le test de systèmes complexes dans l'industrie.

Le MBT est un processus en évolution qui génère automatiquement des tests de validation à partir d'un modèle de test, décrivant certains aspects du système sous test (SUT) (Mark Utting, 2007).

Le modèle de test représente le champ des utilisations possibles du système. Le modèle peut contenir aussi des scénarios non-fonctionnels tels que la portabilité, la compatibilité, la monté en charge, etc. Il est possible de définir ensuite plusieurs stratégies de test, pour dériver de ce modèle un ensemble de cas de tests pertinents dans le but de réaliser une campagne de test. Nous nous intéressons en particulier aux modèles de test pouvant être assimilés à des chaînes de Markov pour qualifier l'usage du système comme sous MaTeLo[1], qui possède un diagramme d'Etat-Transition probabilisé gérant un système déterministe (Le Guen, 2003).

Les systèmes testés sont soumis aux contraintes de développements industriels, ils évoluent dans le temps et peuvent être déclinés selon de nombreuses variantes. Ces systèmes se regroupent dans ce que l'on appelle une ligne de produits.

Dans le domaine automobile par exemple, les systèmes testés sont souvent utilisés sur plusieurs gammes de véhicules et peuvent être configurés avec différents jeux d'options. Dans le ferroviaire, des systèmes génériques sont modifiés et adaptés à chaque projet qui est contraint par les normes locales et la configuration de la ligne de train. D'autre part, certains de ces systèmes sont critiques et sont soumis à des contraintes de développement spécifiques. A l'heure actuelle, les méthodes employées pour gérer la variabilité sont disjointes du processus de validation du système. L'objectif de ce travail est d'intégrer ces problématiques dans une solution complète de MBT. Cette activité de recherche se fait au sein de la société ALL4TEC et les résultats seront expérimentés dans l'atelier MaTeLo.

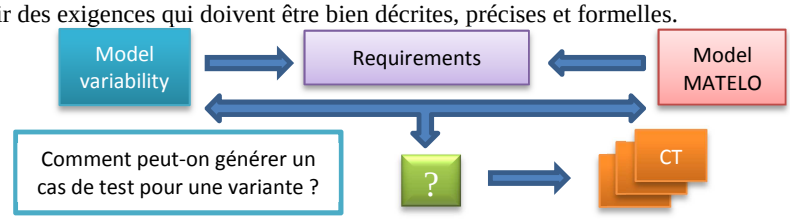
---

[1] L'acronyme de Markov Test Logic. Un outil automatique de génération de tests fonctionnels et de validation. Outil développé par ALL4TEC

2 Objectif

Notre objectif est de formaliser la notion de variantes dans une solution MBT. Cela en proposant une plateforme d'information commune intégrant les différentes caractéristiques fonctionnelles, ou non-fonctionnelles.

Pour enrichir le modèle de test avec la variabilité, il faut extraire ces propriétés à partir des exigences qui doivent être bien décrites, précises et formelles.



Il s'agit également de bâtir des stratégies de validation pertinentes au regard des différentes problématiques, ceci à la fois pour la génération de séquences de test mais également au niveau du suivi d'indicateurs qualitatifs.

Ce travail se concentre sur les verrous scientifiques suivants :

- **Test des dérivées d'une ligne de produit** : lors de la création d'une nouvelle version d'un ancien modèle de voiture, certaines fonctionnalités ont déjà été validées dans l'ancien contexte, reste à savoir : comment sélectionner un sous ensemble de tests à exécuter pour valider la nouvelle version ? Quels cas de test doivent être réalisés sur tous les produits ? Comment tester plus particulièrement une fonctionnalité critique ?
- **Gestion de la variabilité** : pour gérer les changements entre le nouveau et l'ancien modèle, il faut identifier les fonctionnalités qui sont différentes entre chaque produit, les contraintes par rapport aux fonctionnalités existantes, ou récemment ajoutées. En effet, modéliser la variabilité est possible et plusieurs travaux ont été effectués dans ce sens comme les travaux de (Klaus pohl, 2005). Mais avant, il faut déterminer les formes et les dimensions de variabilité en ingénierie système, pour choisir la modélisation adéquate. Ainsi il faut décider comment positionner la variabilité par rapport au modèle de test.
- **Cohérence des exigences** : les exigences sont une description des fonctionnalités, si elles sont incohérentes les suites de cas de test générées à partir du modèle de test seront incohérentes avec le contexte du SUT. Il est donc nécessaire de proposer des méthodes d'analyse de la cohérence exigences-fonctionnalités pour établir des mécanismes robustes de sélection de test pour une ligne de produits.

Référence :

Le Guen, H. L. (2003, septembre). *Thelin. Practical experiences about statistical usage testing. In STEP*. Amsterdam, The Netherlands.

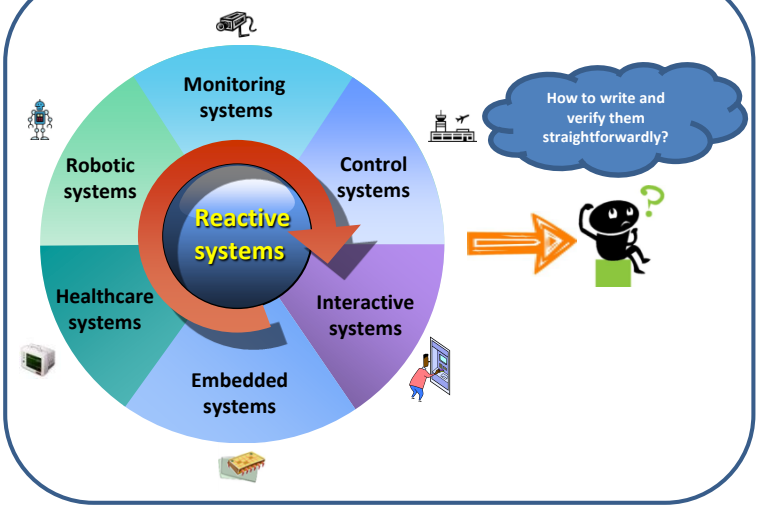
Klaus pohl, G. B. (2005). Software product line engineering, and techniques.

Mark Utting, B. L. (2007). *Pratical Model-Based testing A tools Approach.*

Combining Rule-based and Event-based Programming Paradigms for the Development of Reactive Systems

Truong-Giang Le
LISITE-ISEP, 28 rue Notre-Dame des Champs, 75006 Paris, France
Email: le-truong.giang@isep.fr

1. Motivation



2. Our Approach

Rule-based style
Event-based style
→ A new programming language called INI

Major features

- Rules and events can be defined independently or in combination.
- Programmers may write user-defined events in Java or in C/C++, then integrate them into INI programs.
- Events may run in parallel either asynchronously or synchronously.
- Events can be reconfigured at runtime to change their behaviors.

3. Examples

A simple HTTP server

```
function main() {
  @init() {
    start_http_server(8080)
  }
}
function start_http_server(port) {
  @init() {
    s = socket_server(port)
    clear(c)
    println("Server started on " +
     port)
  }
  s {
    c = socket_accept(s)
  }
  @update[variable = c](oldc, newc) {
    // Handle HTTP requests here
    client = socket_address(c)
    ...
  }
}
```

A prototype for an M2M gateway

```
function main() {
  @init() {
    dataFile = file("faceData.csv")
  }
  $(e) f:@faceDetect[period = 100](point1,
   point2) {
    case {
      !file_exists(dataFile) {
        create_file(dataFile)
      }
    }
    fwriteln(dataFile, to_string(time()) + ","
     + point1 + "," + point2)
  }
  $(f) e:@every[time = 5000]() {
    upload_ftp("host", "username", "password",
     dataFile, to_string(time()) +
     "dataUpload.csv")
    delete_file(dataFile)
  }
}
```

Syntax:
Rule: <Condition> {<Action>}
Event: $(<List of synchronized events>) id:@event[<Input parameters>](<Output parameters>) {<Action>}

4. Type System in INI

Common types

- Number types (Double, Float, Long, Int, Byte), Char type, String type, and Map types (List, Set).

User-defined types

```
type Company = [name:String, numOfEmployee:Int]
function main() {
  @init() {
    c = Company[name = "XYZABC",numOfEmployee = 100]
    println("The " + c.name + " company has " +
     c.numOfEmployee + " employees.")
  }
}
```

Type inference and checking engine

- No need to declare any type explicitly.
- Type conflicts are prohibited.

5. Model Checking INI Programs

INI can be converted to Promela for model checking with SPIN. Possible applications are:
- Detecting infinite loops and unreachable code.
- Verifying properties expressed in Linear Temporal Logic formulas.

Example: Detecting unreachable code

```
function main() {
  @init() {
    v=1
  }
  v < 5 {
    v++
  }
  v == 6 {
    v = v+2
  }
}
```

→

```
int v = 1
proctype main() {
  do
    ::v < 5 -> v++;
    ::v == 6 -> v = v+2;
    ::else -> break;
  od;
}
init {
  run main();
}
```

Verification result

```
...
unreached in proctype main
  UnreachableCode.pml:5, state 4,
   "v = (v+2)"
  (1 of 10 states)
unreached in init
  (0 of 2 states)
...
```

Project: MCUBE **Sponsors:** SYSTEMATIC PARIS REGION SYSTEMS & ICT CLUSTER, l'Europe s'engage en Ile-de-France

Reference:
https://sites.google.com/site/inilanguage

On Model Subtyping

Clément Guy, Triskell and Cairn teams, IRISA-Inria
clement.guy@irisa.fr

CONTEXT

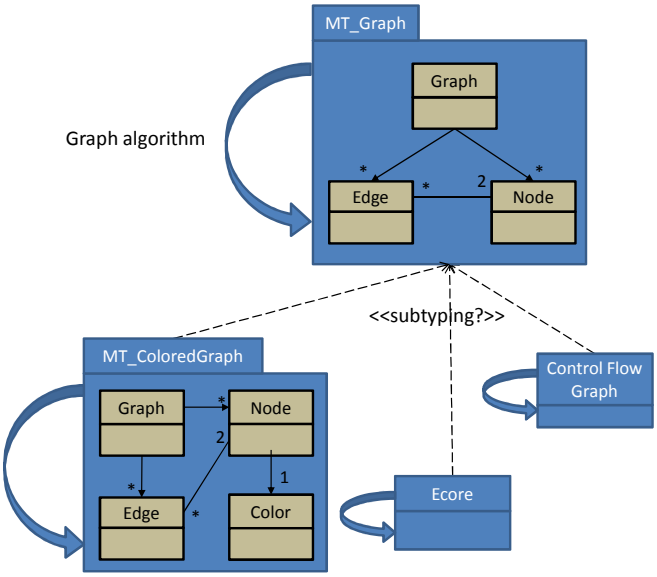
- Rapid increase of the number of modeling languages
  - More and more model manipulation operators
- Need for a systematic engineering
  - Providing design methods and **facilities** (reuse of operators and structures, advanced tools…)
- Existing approaches remain disconnected from each other

➢ Need for a unified theory

MODEL-ORIENTED TYPE SYSTEMS

- Model-oriented type systems should provide facilities such as abstraction, reuse, safety, auto-completion…

A type of a model is a set of types (MOF classes) of objects which may belong to the model, and their relations.



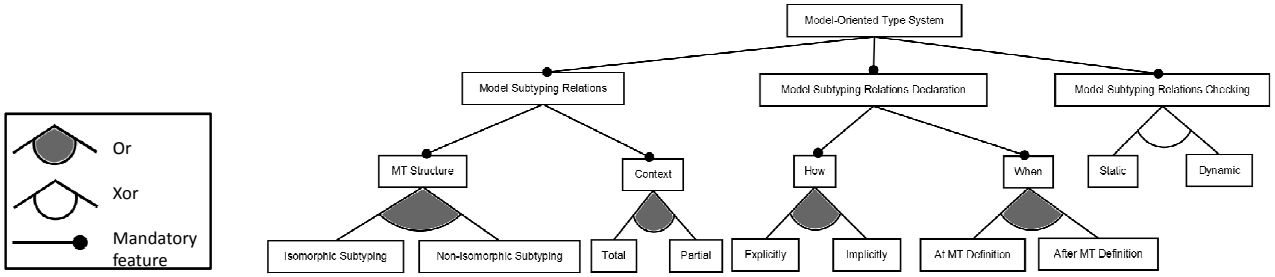
Model types for graphs

MODEL SUBTYPING RELATIONS

- How can we safely use model typed by A where a model typed by B is expected?

- Structure of the model types
  - *Isomorphic subtyping*: **Same** names, multiplicities…
  - *Non-isomorphic subtyping*: **Adaptation** from the subtype to its supertype

- Context of the subtyping relation
  - *Total subtyping*: Models typed by A can be safely used **everywhere** models typed by B are expected
  - *Partial subtyping*: Models typed by A can be safely used **in a given context** in which models typed by B are expected (e.g., a given model transformation)



A family of model-oriented type systems

	Total / Partial	Isomorphic/ ¬isomorphic	At / After definition	Explicit / Implicit	Checking	Legacy tool reuse
Varrò *et al.*	Total	Class renaming	After	Implicit	?	No
Cuccurru *et al.*	Total	Class renaming	After	Explicit	?	Yes
Steel *et al.*	Total	Class renaming multiplicities contraction	After	Implicit	At design time, with errors at runtime	Yes
Sanchez Cuadrado *et al.*	Total	Class renaming multiplicities contraction	After	Explicit	?	Yes
Sen *et al.*	Partial	Any adaptation	After	Explicit	At design time, with errors at runtime	Yes
De Lara *et al.*	Total	Class renaming navigation and filtering of properties, n-to-1 bindings	After (Binding) At (Specialization)	Explicit	?	No
Babau *et al.*	Total	Inclusive	After	Explicit	?	Yes

Classification of existing approaches for model manipulation reuse

CLASSIFICATION OF EXISTING APPROACHES

- Underused approaches:
  - Partial subtyping (i.e., subtyping wrt. a specific context)
  - Implicit declaration of a subtyping relation
  - Declaration at the definition of a type
  - Inclusive subtyping (too restrictive)
- Lack of information on subtyping relations checking

PERSPECTIVES

- Could we propose a type system providing each one of those features?

Supervisor: Jean-Marc Jézéquel
Co-advisors: Benoit Combemale & Steven Derrien

Université Joseph Fourier

TOBIAS

On-line tool for COMBINATORIAL SOFTWARE TESTING

L. du Bousquet, Y. Ledru, T. Triki, G. Vega
Laboratoire d'Informatique de Grenoble (LIG)
VASCO team, http://vasco.imag.fr

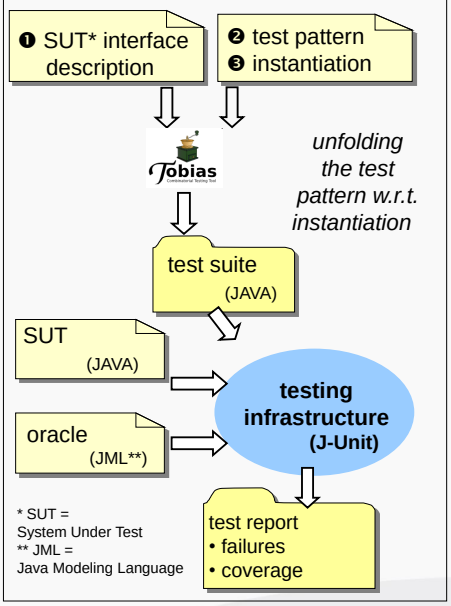
TOBIAS

- support of the scenario-based testing paradigm
- to produce easily lots of test sequences (massive testing)
- motivated by experimental industrial concerns (need for lots of similar test cases)
- for critical applications, information systems, …
- developed in the context of three projects funded by the French Research Agency[1,2,3]

Motivations

- automate the test generation, because testing is the main validation method for software development
- increase the confidence level thanks to lots of tests produced by a systematic approach
- improve the productivity: free the tester from repetitive tasks

TOBIAS outline



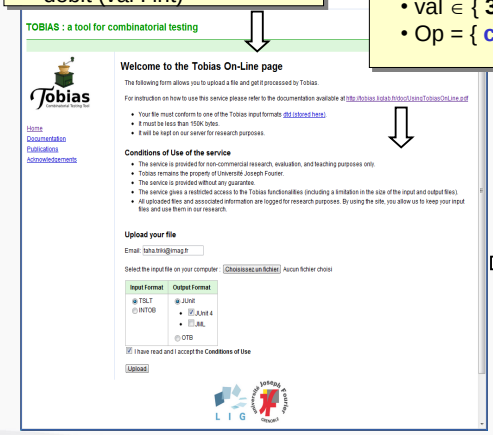
Simple example

An electronic purse

❶ SUT interface
- authenticate (pin : int)
  (correct pin = 1234 *)*
- credit (val : int)
- debit (val : int)

❷ A pattern
authenticate (pin**) ; Op ; Op**

❸ An instantiation
- pin ∈ { **1234**, **8989** }
- val ∈ { **3, 10, 0, -7** } *(* input value definition *)*
- Op = { **credit(**val**), debit(**val**)** } *(* set of methods *)*



authenticate(1234); credit(3); credit(3)
authenticate(1234); credit(3); credit(10)
authenticate(1234); credit(3); credit(0)
authenticate(1234); credit(3); credit(-7)
authenticate(1234); credit(3); debit(3)
authenticate(1234); credit(3); debit(10)
authenticate(1234); credit(3); debit(0)
…
authenticate(1234); debit(3); credit(3)
…
authenticate(8989); credit(10); debit(3)
…
authenticate(8989); debit(-7); debit(-7)

Tobias strengths

- Eclipse plug-in available
- improved productivity (automation & abstraction)
- improved methodology
- easier test suite maintenance
- traceability from patterns to test cases
- used for embedded systems, IHM, web site, smart home applications…

Try the tool on-line at http://tobias.liglab.fr

Features & performances

- massive testing: the tool is implemented to support the generation of more than one million test cases
- productivity: a 10 line textual description of a schema can be unfolded into thousands of executable tests in less than 5 mn
- concepts: bounded regular expressions, set abstraction (data, method names, objects, instructions), filtering and selection
- various target technologies: Java/JML, C++, VDM, B, UML/OCL, …

Related work

- Similar works
  - combinatorial testing (combining parameter values for one single test call)
  - KORAT tool of MIT (to produce complex data structures)
  - JML/J-UNIT of IOWA Univ. (similar principles of TOBIAS, with sequences restricted to one method call)
  - Yagg (combinatorial generation from a grammar)
- Approaches and tools based on Tobias
  - Overture Combinatorial Testing Plug-in, Aarhus, Denmark
  - TL CAT, Trusted Labs
  - JSynopsis, LIFC

VASCO team
Laboratoire LIG

BP 72,
38402 St Martin d'Hères cedex

Lydie.du-bousquet@imag.fr
Yves.Ledru@imag.fr

References

- Y. Ledru, L. du Bousquet, O. Maury, P. Bontron. Filtering Tobias combinatorial test suites. In 7th Int. Conf. FASE, vol. 2984 of LNCS, pp 281-294, 2004. Springer
- Y. Ledru, F. Dadeau, L. du Bousquet, S. Ville, E. Rose. Mastering combinatorial explosion with the Tobias-2 test generator. In 22nd IEEE/ACM ASE 2007, pp 535-536. ACM
- T. Triki, Y. Ledru, L. du Bousquet, F. Dadeau, J. Botella. Model-based filtering of combinatorial test suites. In FASE. LNCS, vol. 7212, pp. 439–454 (2012)

[1] RNTL COTE (2000-02) - [2] RNTL POSE (2006-07) - [3] ANR TASCCC (2009-12) #ANR-09-SEGI-014

Handles

Property-Propagating First Class Reference for Dynamically-Typed Languages.

Jean-Baptiste ARNAUD
jean-baptiste.arnaud@inria.fr
RMoD, INRIA Lille Nord Europe

Scientific challenge: Build a dynamic (but secure) language

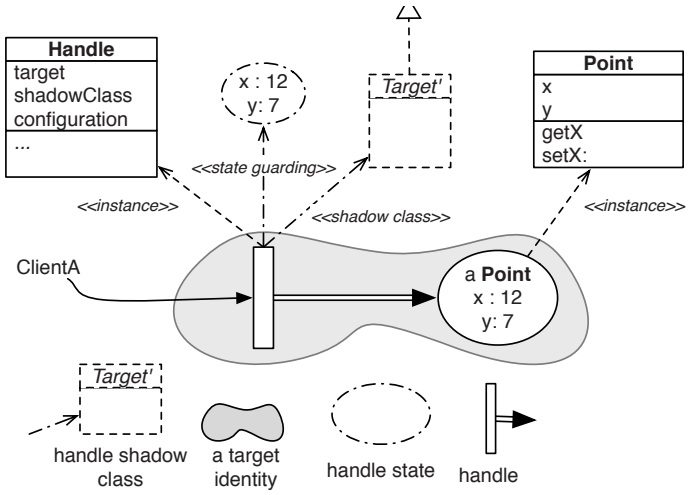
- Controlling references and giving specific semantics to single objects and to graphs of objects is essential to be able to build more secure systems, but is notoriously hard to achieve in absence of static type systems. This is due to tree constraints:
  - Open world, system can potentially change at runtime.
  - No type information until runtime.
  - Presence of reflection.

- Problem: what is the underlying mechanism that can support the definition of properties (such as revocable, read-only, lent...) at the reference level in the absence of a static type system?

Solution

- Our solution is the Handle:
  Handles are first class references (i.e., objects modelling a reference).
  The Handle provides:
  - Indistinguishable property: Handle represents another object (target object) of the system, from the system point of view it is considered as the target object.
  - State guarding: it retains its own version of the state of the target object.
  - Shadow class: it can change the behavior of the target object (only for this reference).
  - Handle propagates its own configuration to the target object graph, when accessed.

- Metahandle provides a real time control of Handles
  - Change the configuration of Handle at runtime (Behavior change, State guarding, etc.).
  - Install the Handle (remove the Handle; if state guarding is activate install the current state on the target object).
  - Can be created only at Handle creation time.



Result

- A working implementation with a specific virtual machine and API for using Handles, do on Pharo language.

- Implementation of complex security properties (such, Read-Only, Membrane, Mirror, Software Transactional Memory, etc.)
  - With a minimalist design (less than 5 classes and 30 methods).
  - Usable performance.

Do you want to know more?

- http://jeanbaptiste-arnaud.eu/handles

- Jean-Baptiste Arnaud, Marcus Denker, Stéphane Ducasse, Damien Pollet, Alexandre Bergel and Mathieu Suen. *Read-Only Execution for Dynamic Languages*. In Tools Europe. Springer Verlag, June 2010.

INVENTEURS DU MONDE NUMÉRIQUE

[!] EUROPEAN SMALLTALK USER GROUP

Using Architecture Models to Rapidly Prototype Feedback Control Systems

Filip Křikava and Philippe Collet

Université Nice Sophia Antipoli, I3S - CNRS UMR 7271
filip.krikava@i3s.unice.fr, philippe.collet@unice.fr

Introduction. Self-Adaptive systems are a promising research direction to address the ever growing complexity and uncertainty of the contemporary software systems. Such systems autonomously (i.e., without or with minimal human interactions) adjust their structure and behavior at runtime based on their perception of their state and the state of their environment in order to follow some higher-level objectives [1].

While there exist many different forms of self-adaptive systems they are usually organized in some form of *Feedback Control Loop* (FCL), where measurements of system's outputs are used to determine the control inputs which in turn should affect again the outputs converging towards the determined goal [2]. However, engineering feedback control systems in a predictive and effective way is a major engineering challenge. Not only it is difficult to find the right control model that drives the system adaptation, but also the process of building the necessary surrounding architecture around the control is tedious and far from being trivial.

Our long term goal is to provide researchers and engineers with a tooled approach for rapidly prototyping explicit adaptation strategies while abstracting them from the painful low-level implementation details. We focus on externalized control in which the core system logic is separated from the adaptation behavior. In order to facilitate the rapid prototyping we use model-driven engineering techniques to design, implement and verify several adaptation behaviors based on fine-grained architectures that are composed of reusable explicit control loop elements. The resulting architecture models can be then transformed into to a deployable running system.

Approach. The figure 1 gives an overview of the rapid prototyping process. The process starts with modeling the logic of the adaptation (i.e., the structure and the data/control flow of the FCLs) using a provided Domain-Specific Language FCDL that is then reified as an architectural model FCDM. The completed and validated model is used as an input into various tools such as a source code generator, which with some auxiliary information, can output a skeleton implementation of the system. A user is then required to only complete the adaptation specific code (e.g., data collection, control logic) while it takes care for all the necessary scaffolding in order to provide a runnable application. Other example is the support to perform an explicit verification of some assumptions about the behavior of the model. For instance we can check at design time which FCL elements will be activated by data emission of a given source. Concretely, we use LTL formulas to represent such assumptions that are then checked by the Spin[1] model checker against the Promela[1] model that is generated from the FCDM. The prototyping is designed to provide early feedback based on which the user can reiterate at any time and go back to the architecture description to make any necessary adjustments.

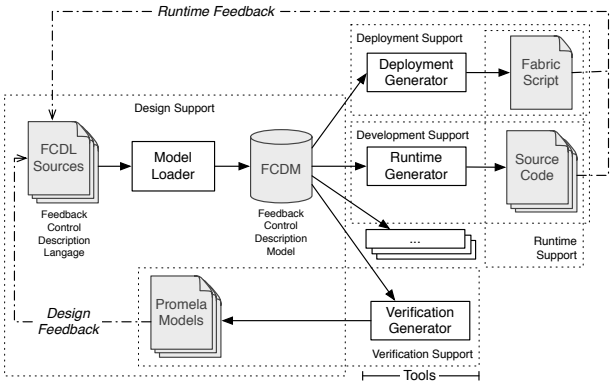


Fig. 1: Overview of the rapid prototyping process.

The architecture of the adaptation is described using a technologically agnostic meta-model that allows to support different degrees of separation with respect to the target system: from running in a separate runtime to getting integrated inside the system using aspect-oriented techniques

[1] http://spinroot.com/

or direct source code generation. The elements of this meta-model capture the main parts of typical FCLs that collect relevant context information from various sources *sensors*, process them through *filters* into *controllers* that are responsible for reasoning about the state of the system and for planing the appropriate actions to be carried out by the available *effectors* in order to get the system into some desired state.

One originality of our approach is in turning all these elements to be adaptable themselves. This means that any of the above presented elements can provide its own sensors and effectors that are tied to its context. This feature enables these *adaptive elements* to be introspected (meta-data, state) and modified in the very same way as any other part of the target controlled system. We can therefore hierarchically compose not only control on the top of the target system, but also on top of controllers, sensors, etc. The added adaptation capabilities then become self-adaptive as well and does that in an uniform way because the relation is merely a hint for the tooling support to handle them differently, but from the adaptation perspective they are the same.

Besides element definition, the architecture meta-model also supports *composition* allowing better reuse of elements assembly and *annotations* that are used to extend the meta-model to provide any additional information that might be required by the various tools. The distribution of the elements is explicitly supported via ports that proxy the communication to the remotely running objects. The ports are themselves also adaptive elements thus some of the remote communication properties such as timeouts, compression, encryption, etc. can be adapted in the very same manner.

Validation. In validating our approach we are concerned whether the proposed design-time and run-time representations are expressive and intuitive enough to support activities of feedback control system architects. In [3] we provide the first results of our approach on a use case that aims at improving throughput for workflow executions in Condor[2] environment. The excerpt of the resulting architecture model is presented in the figure 2 using our graphical notation showing two FCLs.

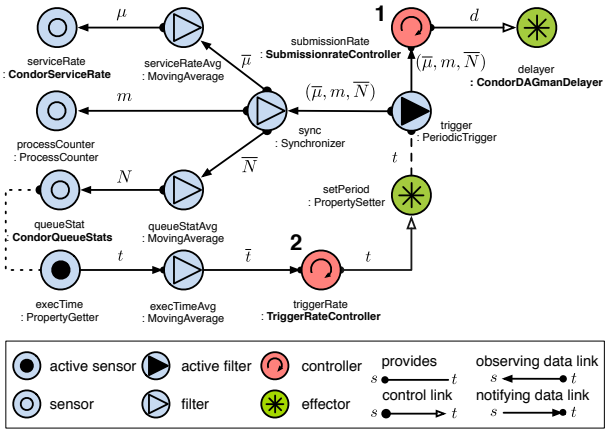


Fig. 2: Example of the architecture model.

The reference implementation of the runtime platform is based on a Scala message passing actor threading model where each adaptive element from the model (including links) is realized as an actor. From the preliminary results the expressiveness of the model seems to be sufficient, nevertheless, empirical evidence and wider usage are still missing. The proposed approach is currently under validation inside the ANR-funded SALTY project[3]. The outgoing work on its use-cases in different domains (e.g., controlling geo-tracking fleet management system or BPEL process adaptation) should help to complement validation and provide more thorough experience of applying the proposed approach. It should also allow us to derive a library of reusable FCL elements and patterns. Moreover, another mapping between the FCDM elements and component-based SCA runtime is under ongoing development in collaboration with the project partners.

References.

1. B. Cheng et al., "Software Engineering for Self-Adaptive Systems: A Research Roadmap," *Lecture Notes in Computer Science*, 2009, Volume 5525/2009, 1-26
2. J. Hellerstein, Y. Diao, S. Parekh, and D. Tilbury. "Feedback Control of Computing Systems," *Wiley-IEEE*, 2004.
3. F. Křikava and P. Collet, "A Reflective Model for Architecting Feedback Control Systems," *The 23rd International Conference on Software Engineering & Knowledge Engineering*, 2011

[2] `http://research.cs.wisc.edu/condor/`

[3] `http://salty.unice.fr` – ANR-09-SEGI-012

Un environnement pour la vérification des règles ajoutées de l'Atelier B

Mélanie Jacquel, Karim Berkani, David Delahaye, Catherine Dubois
Siemens IC-MOL, CEDRIC/CNAM/ENSIIE, INRIA/Deducteam

Problématique

Vérifier les règles automatiquement à l'aide de prouveurs tout en assurant leur correction vis-à-vis de la méthode B.

Contexte

. Siemens Mobility and Logistics

- VAL (Véhicule Automatique Léger)
- Automatismes d'aide à la conduite et intégral
- Guidage optique

Méthode B utilisée chez Siemens pour assurer le fonctionnement sûr des métros automatiques.

. Activité de preuve avec l'Atelier B

Génération des obligations de preuve
Preuves automatiques (pp)
Preuves interactives
-> Application de tactiques élémentaires
-> Possibilité d'ajouter des règles (comme axiomes)

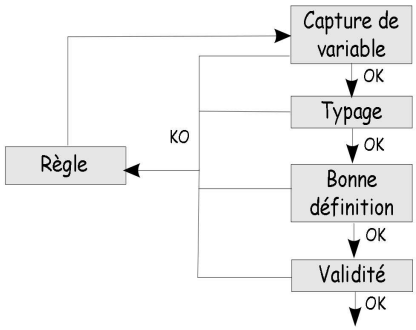
. Règles

- Formules de la théorie B avec des gardes qui permettent d'ajouter des conditions d'applicabilité
- Règle de déduction:

InSetXY:
$\text{binhyp}(f \in A \nrightarrow B) \wedge\ (a \in \text{dom}(f)) \wedge\ (f(a) \in u) \Rightarrow\ (a \in f^{-1}[u])$

Règle de réécriture:
Associativity: $a \cup (b \cup c) == a \cup b \cup c$

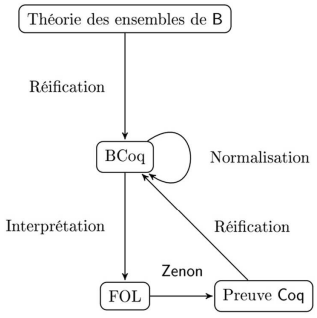
Processus de vérification d'une règle



Preuve de la validité

. Approche n°1

- Etape de pré-normalisation pour obtenir une formule du premier ordre à partir d'une formule ensembliste
- Implantée grâce au langage de tactique de Coq : Ltac
- Vérification de la preuve obtenue par Zenon dans un environnement de preuve B implanté en Coq appelé BCoq



. Approche n°2

Extension de Zenon avec la méthode B grâce à la superdeduction
- Etape de normalisation supprimée
- Principe de la superdeduction :

Axiome : $\forall a \forall b\ ((a \subseteq b) \Leftrightarrow (\forall x\ (x \in a \Rightarrow x \in b)))$

Construction de la régle de superdeduction :

$$\frac{\dfrac{\Gamma, x \in a \vdash x \in b, \Delta}{\Gamma \vdash x \in a \Rightarrow x \in b, \Delta} \Rightarrow_{\mathrm{R}}}{\Gamma \vdash \forall x\ (x \in a \Rightarrow x \in b), \Delta} \forall_{\mathrm{R}},\ x \notin \Gamma, \Delta$$

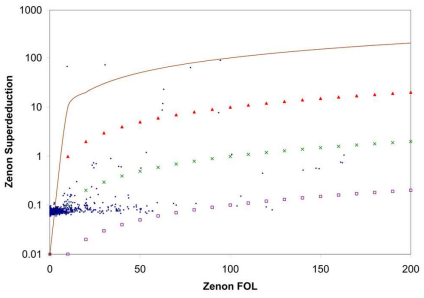
Règle de superdeduction obtenue :

$$\frac{\Gamma, x \in a \vdash x \in b, \Delta}{\Gamma \vdash a \subseteq b, \Delta}\ \mathrm{IncR},\ x \notin \Gamma, \Delta$$

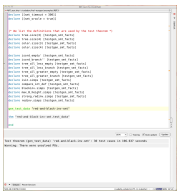
Conclusion

- Nombre de règles ajoutées démontrées automatiquement :
- Approche Zenon : 1145/1425 règles
- Approche superdeduction : 1340/1425 règles
- Temps de preuve avec superdeduction en moyenne 67 fois plus rapide (max. 1540 fois) que temps de preuve sans superdeduction. (cf. graphe)

Publications:
[1] Mélanie Jacquel, Karim Berkani, David Delahaye, and Catherine Dubois. Verifying B Proof Rules using Deep Embedding and Automated Theorem Proving. In Software Engineering and Formal Methods (SEFM), volume 7041 of LNCS, pages 253–268, Montevideo (Uruguay), November 2011. Springer.
[2] Mélanie Jacquel, Karim Berkani, David Delahaye, and Catherine Dubois. Tableaux Modulo Theories using Superdeduction. In The 6th International Joint Conference on Automated Reasoning (IJCAR), LNCS, Manchester (UK), June 2012. Springer. A paraître.



Comparaison en temps (s) des deux approches

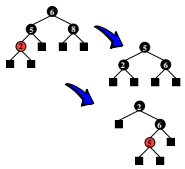


Interactive Testing with HOL-TestGen

Achim D. Brucker[1] and Burkhart Wolff[2]

[1] http://www.brucker.ch/ [2] Université Paris-Sud 11, LRI – Batiment 650, 91405 Orsay Cedex, France

E-mail: wolff@lri.fr



Abstract

HOL-TestGen is a test environment for specification-based testing built upon the proof assistant Isabelle/HOL. The environment provides support for the modeling of the problem domain, the formulation of the test specification, the generation of test-cases and tests, the test-execution and the test document generation. The environment has been used in significant case-studies requiring combined test and proof efforts. As such, HOL-TestGen can also be understood as a unifying technical and conceptual framework for presenting and investigating the variety of unit test techniques in a logically consistent way.

1. Vision of HOL-TestGen

Observation:

- Any formal testcase-generation method is based on the solution of *logical constraints*.
- Limits of constraint solvers hamper testcase-generation techniques.

Conclusion:

$\Longrightarrow$ Testing should be integrated as automatic tactic in an interactive theorem proving environment

$\Longrightarrow$ Testing and Verification may converge by using Explicit Test Hypothesis THYP's

$\Longrightarrow$ Test-tools can be provably correct tools (based on conservative embeddings and derived rules)

2. How to use HOL-TestGen

The abstract workflow is divided into five phases:

1. the *test theory* development,
2. the *test specification* development,
3. generation of *test cases* (and a *test theorem*),
4. generation of *test data* (TD), and
5. the *test execution* (*result verification*) phase.

Once a test theory is completed, documents can be generated that represent a formal test plan.

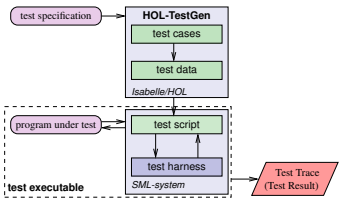


Figure 1: *Overview of the Standard Workflow*

The properties of the *program under test* are specified using higer-order logics (HOL) in the *test specification* (TS). The system will decompose the TS in the test case generation phase into a semantically equivalent *test theorem* containing explicit *test hypotheses* marked by the constant THYP (semantically: an identity).
A test theorem has the following meaning:

If the program under test passes the tests for all TC_i *successfully, and if it satisfies all test hypothesis, it conforms to the test specification.*

In this sense, a test theorem bridges the gap between test and verification.

3. *Black-Box Testing:* Foundation

We describe two test-specific phases in detail:

1. The test case generation phase for TS (over *program under test* prog) is essentially a normal-form-computation resulting in the test-theorem:

$$TC_1 \Longrightarrow \cdots \Longrightarrow TC_n \Longrightarrow THYP(H_1) \Longrightarrow \cdots \Longrightarrow THYP(H_m) \Longrightarrow TS$$

where testcases $TC_{i\in\{1..n\}}$ have the form

$$PRE_i(x) \Longrightarrow POST_i(x, prog(x))$$

and where THYP's are test-hypotheses, e.g. the uniformity hypothesis

$$THYP(\exists x \in PRE_i \bullet POST_i(x, prog(x))) \longrightarrow \forall x \in PRE_i \bullet POST_i(x, prog(x)))$$

2. The test-data generation ("selection"), e.g. finding solutions for the testcases by finding ground instances for x satisfying the PRE_i. This is essentially a constraint resolution problem (using random generators, SAT and SMT solvers . . .)

4. *Black-Box Testing:* An Example

Red-black trees store the balancing information in one additional bit per node, which is called the "color of a node". A valid (balanced) red-black tree must among others fulfill the invariant:

```
fun max_B_height :: "'a tree ⇒ nat" where
  "max_B_height E = 0"
| "max_B_height (T B a y b)
        = Suc(max (max_B_height a) (max_B_height b))"
| "max_B_height (T R a y b)
        = (max (max_B_height a) (max_B_height b))"

fun blackinv :: "'a tree ⇒ bool" where
  "blackinv E = True"
| "blackinv (T color a y b) = ((blackinv a) ∧ (blackinv b)
                             ∧ ((max_B_height a) = (max_B_height b)))"
```

For testing that a (functional) insertion or deletion (prog) fulfill the black invariant we write the following test specification:

```
test_spec "(isord t∧ isin y t ∧strong_redinv t∧ blackinv t)
           ⟶ (blackinv(prog(y,t ))) "
```

The user proceeds by configurable tactic procedures executing the test-case and data generation.

5. *Black-Box Sequence Testing:* Foundation

In many systems, the state σ of the system under test is not under direct control of the tester, except in the initialization phase. Test executions must therefore provide sequences of operation calls of the program under test, such that appropriate pre-states of a critical operation under test are constructed by their internal side-effects.
Based on HOL, test theories can formulate operations in the state-exception monad $(\sigma, \alpha)Mon_{SE}$ as partial state transition computations. The monad yields the usual infrastructure for *bind*: $x \leftarrow op; m\ x$ and *unit*: result (P).
Continuing the red-black-tree example above, this means that t is not an explicit argument, rather a state variable whose value must be appropriately set by a sequence of ins and del -operations. Test generation is reduced to synthesizing *valid* test-sequences not raising exceptions, for example:

$$E \models \_ \leftarrow ins(1); \_ \leftarrow ins(3); \_ \leftarrow del(1); result(is\_blackinv)$$

The technique is extended to classical IOCO-testing and applied in case studies, for example, for testing stateful firewalls [4].

6. *White-Box Testing:* Foundation

How can this be applied for white-box-testing of imperative programs?

- *Answer:* Take IMP and prove a path unwind theorem over the oper. semantics $\langle p, \sigma\rangle \rightarrow_c \sigma'$:
- *Consequence:*

$$\models \{P\}\ c\ \{Q\}$$
$$= \{P\}\ unwind(n,c)\ \{Q\}$$
$$= \forall\ \sigma\ \sigma'.\ \langle unwind(n,c),\sigma\rangle \rightarrow_c \sigma' \longrightarrow P\ \sigma \longrightarrow Q\ \sigma'$$

- *Testhypothesis:*

$$THYP(TC(\sigma) \longrightarrow \langle \texttt{WHILE ... DO ...}, \sigma\rangle \rightarrow_c \sigma' \longrightarrow post\ \sigma\ \sigma'$$

7. *White-Box Testing:* Example

Test theory. Computing the integer square root:

```
definition sqrt :: "[loc,loc,loc,loc] ⇒ com" where
 "sqrt tm sum i a ≡ ((  tm  :== λσ. 1);
                     ((  sum :== λσ. 1);
                      ((i    :== λσ. 0);
                        WHILE λσ. (σ sum) <= (σ a) DO
                          ((  i   :== λσ. (σ i) + 1);
                           ((tm   :== λσ. (σ tm) + 2);
                            (sum :== λσ. (σ tm) + (σ sum)))))))
```

Test Specification (as a Hoare-Triple):

$$\models \{\lambda\ \sigma.\ true\}\ sqrt\ tm\ sum\ i\ a\ \{post\ a\ i\}$$

Testcases:

1. $9 \leq \sigma\ a \Longrightarrow \langle$ WHILE $\lambda\sigma.\ \sigma sum \leq \sigma a$ DO i :== $\lambda\sigma.$ Suc (σ i) ; (tm :== $\lambda\sigma.$ Suc (Suc (σ tm)) ; sum :== $\lambda\sigma.\ \sigma tm + \sigma sum$), $\sigma(i:=3,\ tm:=7,\ sum:=16)\rangle \rightarrow_c \sigma'$
2. $[\![4 \leq \sigma\ a;\ 8 < \sigma\ a\ ]\!] \Longrightarrow \sigma' = \sigma\ (i:=2,\ tm:=5,\ sum:=9)$
3. $[\![1 \leq \sigma\ a;\ \sigma\ a < 4]\!] \Longrightarrow \sigma' = \sigma\ (i:=1,\ tm:=3,\ sum:=4)$
4. $\sigma\ a = 0 \Longrightarrow \sigma' = \sigma(tm:=1,\ sum:=1,\ i:=0)$

Remaining THYP*:*

1. $THYP(9 \leq \sigma a \longrightarrow \langle$ WHILE $\lambda\sigma.\ \sigma sum \leq \sigma a$ DO i :== $\lambda\sigma.$ Suc (σ i); (tm :== $\lambda\sigma.$ Suc (Suc (σ tm)); sum :== $\lambda\sigma.\ \sigma$ tm + σ sum), $\sigma(i:=3,\ tm:=7,\ sum:=16)\rangle \rightarrow_c \sigma'$ $\wedge$ post a i σ')

Note that no invariant has to be provided by the user of this method.

8. Conclusion

- Method for ITP based test generation approach
- Uniform framework for various testing techniques (black-box, sequence, and white-box)
- Controllable, explicit test hypothesis
- Combined test and proof techniques leveraged large case-studies [3,4].

References

[1] HOL-TestGen. http://www.brucker.ch/projects/hol-testgen/.

[2] Achim D. Brucker and Burkhart Wolff. On Theorem Prover-based Testing. In *Formal Aspects of Computing (FAOC)*, DOI: 10.1007/s00165-012-0222-y, 2011.

[3] Achim D. Brucker, Lukas Brügger, Paul Kearney, and Burkhart Wolff. An approach to modular and testable security models of real-world health-care applications. In *SACMAT 2011: 133-142*, DOI: 10.1145/1998441.1998461, ACM, 2011.

[4] Achim D. Brucker, Lukas Brügger, Paul Kearney, and Burkhart Wolff. Verified Firewall Policy Transformations for Test Case Generation. In *ICST 2010*, DOI: 10.1109/ICST.2010.50, IEEE Computer Society, 2010.

Choisir son Nuage à l'Aide des Modèles de Caractéristiques

Clément Quinton, Laurence Duchien and Romain Rouvoy

Inria Lille-Nord Europe
Université Lille 1
LILF UMR CNRS 8022
France
{clement.quinton,laurence.duchien}@inria.fr
romain.rouvoy@lifl.fr

1 État des Lieux

L'informatique dans les nuages (*Cloud Computing*) est une tendance majeure dans les environnements d'informatique répartie qui permet d'accéder à des ressources virtuelles délivrées à la demande par les fournisseurs de cloud [1]. Le client paie en fonction de l'utilisation de ces ressources, *i.e.*, des applications déployées et gérées en tant que service sur l'infrastructure de ces fournisseurs de cloud. Pour être accessible en tant que service (*i.e.*, *Software-as-a-Service*, SaaS) sur le cloud, une application peut être déployée soit sur l'infrastructure (*i.e.*, *Infrastructure-as-a-Service*, IaaS) du fournisseur de cloud, soit sur la plateforme qui va l'exécuter (*i.e.*, *Platform-as-a-Service*, PaaS), cette dernière solution étant la plus simple à mettre en place car elle permet d'éviter la gestion de l'infrastructure sous-jacente.

Dans les deux cas, il faut prendre en compte un grand nombre de ressources ayant des niveaux de fonctionnalité différents parmi les différentes solutions de cloud disponibles (Amazon EC2, Jelastic, CloudBees, etc.). Lors d'un déploiement d'application sur un PaaS, il faut typiquement sélectionner un serveur d'application, le langage de développement de l'application, la base de données à utiliser (si nécessaire), etc. Il en va de même pour le déploiement sur un IaaS qui nécessite la configuration de toute la pile logicielle (système d'exploitation, librairies, serveurs d'applications) qui supporte l'application et qui s'exécute sur l'infrastructure du fournisseur de cloud. Que ce soit pour l'un ou l'autre cas, cette configuration est généralement effectuée de manière ad hoc et est source d'erreurs lorsqu'elle est faite à la main. Ainsi, certains utilisateurs préfèrent déployer leur application vers une solution de cloud qu'ils ont déjà utilisée et qui correspond à peu près aux exigences de l'application à déployer -même si cette solution n'est pas optimale- plutôt que de se risquer à configurer un PaaS ou un IaaS pourtant plus adapté à leur priorités (prix, flexibilité, coût, etc.). Nous définissions donc deux challenges auxquels font face tous les acteurs concernés par un déploiement dans le cloud :

$\mathcal{C}_1$: identifier les solutions de cloud computing qui permettent de déployer l'application.

$\mathcal{C}_2$: choisir parmi les solutions possibles celle qui convient le mieux (*i.e.*, qui répond aux exigences de l'application et aux priorités définies par l'utilisateur).

2 Ébauche de Solution

Nous proposons d'utiliser la configuration de modèles de caractéristiques [2] pour développer un outil d'aide à la décision permettant de choisir la meilleure offre possible. Un modèle de caractéristiques (*Feature Model*, FM) permet de modéliser la partie commune et la partie variable d'une famille de produits en sélectionnant les caractéristiques souhaitées (*features*) lors du processus de configuration. Ainsi, chaque PaaS peut facilement être modélisé sous forme de modèle de caractéristiques pour décrire les fonctionnalités qu'il propose, comme montré par la Fig. 1 a). La fusion des modèles de caractéristiques de chaque PaaS fournit à l'utilisateur une vision d'ensemble de la variabilité des solutions PaaS (`FM_PaaS`). Par le biais des dépendances entre caractéristiques, la configuration de l'application à déployer entraîne la sélection de caractéristiques au niveau du `FM_PaaS` et propose ainsi un éventail des solutions PaaS dont les fonctionnalités correspondent aux

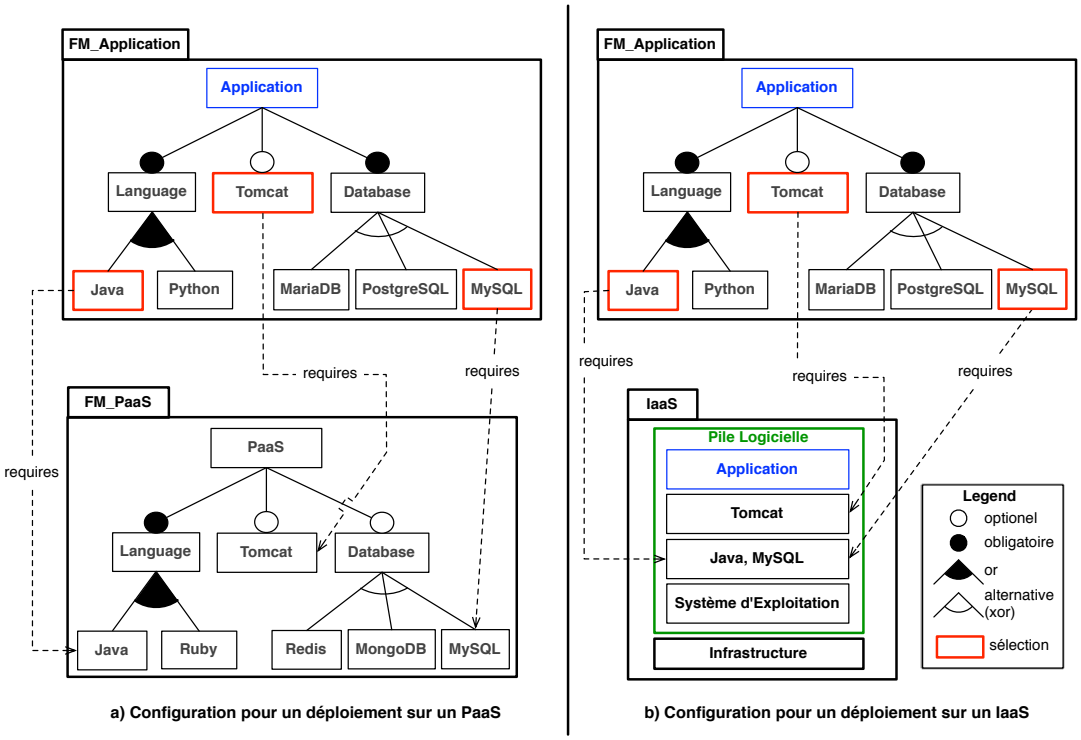


Figure 1. Différentes façon de configurer une application pour la déployer sur le cloud

exigences de l'application à déployer. La Fig. 1 b) décrit le même principe, s'appliquant cette fois au niveau de la pile logicielle nécessaire au déploiement sur un IaaS [3].

L'outil doit de plus prendre en compte les priorités de l'utilisateur. En effet, celui-ci doit pouvoir spécifier que son application soit flexible, sécurisée, fiable ou opérationnelle à faible coût et bien souvent une combinaison de ces priorités. Ces priorités, ou préoccupations, sont transverses aux fonctionnalités offertes par les fournisseurs de cloud (*i.e.* du modèle de caractéristiques du fournisseur). La définition des modèles de caractéristiques ayant attrait à plusieurs préoccupations est un problème de Séparation des Préoccupations.

Notre objectif est d'exprimer dans chaque modèle de caractéristiques les fonctionnalités liées au IaaS, au PaaS, à l'application, aux priorités définies par l'utilisateur, de choisir à l'aide de l'outil de configuration les éléments en cohérence permettant de construire l'environnement et l'application à déployer. Notre perspective sera ensuite de proposer à partir de ce configurateur, une dérivation sous forme de produits logiciels dans une ligne de production associée, de façon à automatiser la construction de l'application.

Références

1. P. Mell, T. Grance. The NIST Definition of Cloud Computing. Technical Report, 2009.
2. Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) - Feasibility Study. Technical Report, 1990.
3. Quinton Clément, Rouvoy Romain, Duchien Laurence. Leveraging Feature Models to Configure Virtual Appliances. In Proceedings of the 2nd International Workshop on Cloud Computing Platforms (CloudCP), 2012.

Distributed Complex Event Processing Engine

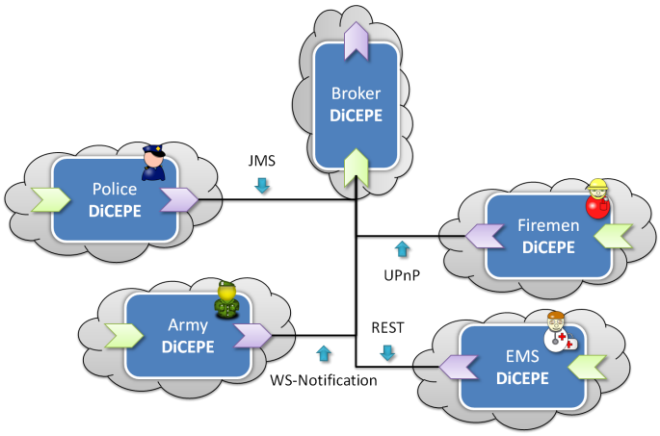
Fawaz PARAÏSO, Gabriel Hermosillo, Romain Rouvoy, Philippe Merle, Lionel Seinturier
INRIA Lille Nord Europe - University of Lille 1 - Laboratoire LIFL - CNRS UMR 8022

Introduction

- DiCEPE (Distributed Complex Event Processing) is a platform that focuses on the integration of CEP engines in distributed systems, and which is capable of using various communication protocols.
- It was built using a component-based approach, and inherits the flexibility and adaptation facilities provided by FraSCAti.
- The DiCEPE platform is used to federate complex event processing.

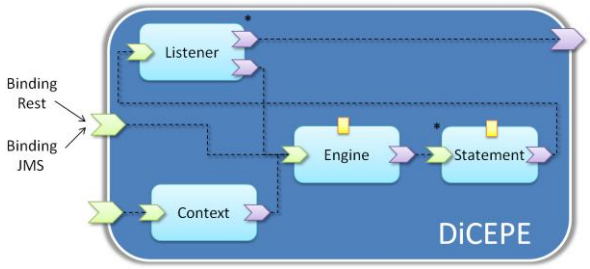
Experimental Deployment

- We developed a solution for a crisis management scenario using DiCEPE .
- This scenario was deployed in a cloud environment using CloudBees, a public Platform as a Service (PaaS) provider.
- Each actor of the scenario runs its own instance of DiCEPE.



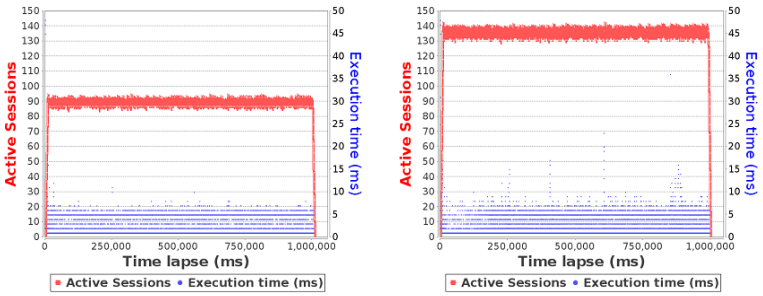
Platform Overview

- The architecture of DiCEPE is composed of four parts:
- **Engine:** This component acts as the engine instance, by which Statement components, events, and outputs (Listener component) are registered.
- **Statement:** This component is used for quering the inbound event streams. It is registered within the Engine component, which at the same time is connected to one or many Statement components.
- **Listener:** This component generates a new complex event when an action is detected.
- **Context:** This component collects information of the executing environment, like the number of statement rules deployed in the engine at run-time.



Scalability

- To evaluate the scalability, we used two different datasets of event generators: one with 10,000 and a second one with 15,000 which generated around 500,000 and 750,000 events respectively.
- As shown in the graphs, the processing time for each event remained stable and very low during both benchmarks (around a 10th of a millisecond), despite the fact that the average number of simultaneous sessions had a significant increase of about 50% (from 89 with the first dataset, to 135 with the second).



Integration

- The DiCEPE platform facilitates the integration of complex event processing engines.
- We integrated DiCEPE with the Esper and Etalis CEP engines.

Etalis EsperTech

Conclusion

- DiCEPE is a platform that offers interoperability for Distributed Complex Event Processing engines, via federation. It focuses on providing a very flexible component architecture, which supports the interaction of different complex event processing engines simultaneously, while enabling communication among them with a distributed execution and deployment system.

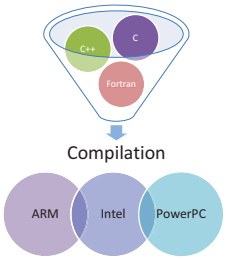
Acknowledgments. This work is partially funded by the French Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council and FEDER through the Contrat de Projets Etat Region Campus Intelligence Ambiante (CPER-CIA) 2007-2013, and the ANR (French National Research Agency) ARPEGE SocEDA project.

Marie KRUMPE GOLDSZTEJN* (PhD student), Yves LHUILLIER*,
Joel FALCOU** and Lionel LACASSAGNE*** (supervisors)
* CEA LIST, LCE (@cea.fr)
** Université Paris-Sud, LRI (@lri.fr)
*** Université Paris-Sud, IEF (@u-psud.fr)

energie atomique • energies alternatives

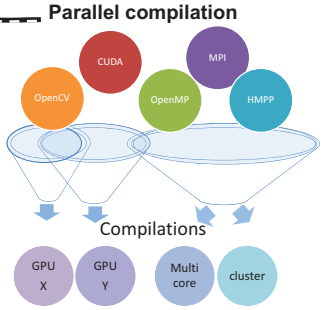
Automatic deployment on embedded parallel systems

Sequential compilation



Approach	Compiler	?	Library
Examples	OpenMP, CUDA, OpenCL, HMPP	?	MPI, pthreads, McAPI
Architecture adequacy	+	+	-
retargetable	-	+	+

Parallel compilation



Metaprogramming

- To adapt compilation to architecture
- To ease retargeting using flexible and modular back-ends
- ***Many targets*** platforms for a ***single user development***

Parallel programming using metaprogramming[1,2]

In order to be able to let developers write **efficient** and **portable** programs, we propose a programming approach based on **generative programming** and **algorithmic skeletons**.

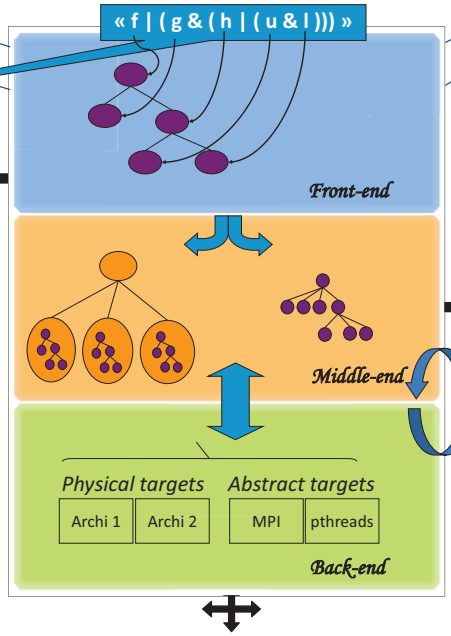
In our approach, generative programming (or metaprogramming) is used to separate architectural specific constructs from algorithmic description. Additionally, the algorithmic descriptions are done through intuitive, domain-specific, algorithmic skeletons. Skeletons are captured by the metaprogramming framework to deliver the best architecture-specific code.

Algorithmic skeletons

Generative programming

Code user expression:

```
Skel = f | ( g & ( h | ( u & l )));
run( Skel, <target> );
```



The Front-end is based on an algorithmic skeletons approach. Composable skeletons (map, pipe…) are provided to capture the application through a domain specific language (DSL). A domain specific language is thus associated to an adequate set of skeletons.
The user writes a skeleton and we use boost::proto that provides us the corresponding proto tree regarding the DSL-defined grammar.

The middle-end :

Architecture independent transformations to extract all possible semantic. This semantic will be later used to ease architecture adequacy and porting.

Typical middle-end transformations:

- Inversion
- Clustering
- Merging
- Splitting

Back-ends for platforms without c++ compiler : code generation

In this case we are performing code generation. We generate C-language but also some parameters files. It is the most important back-end for the embedded systems because they often use specific language.

Back-ends debug

We want to give the possibility to test :

- That each part of code is working
- That the skeleton is doing what is expected
- That the code, linked by the skeleton is working
- …

Back-ends c++

For systems that have standard gcc compilers, we can leverage common parallel programming environments such as OpenMP, MPI, pthreads…
The framework provides native back-ends that allow program skeletons to be parallelized and compiled in the same time.

[1] Jocelyn SEROT, Joel FALCOU, « Functional Meta-programming for Parallel Skeletons », Computational Sciences – ICCS, volume 5101, pages 154-163, 2008

[2] Joel FALCOU, Jocelyn SEROT, « Formal semantics applied to the implementation of a skeleton-based parallel programming library », Parallel Computing: Architectures, Algorithms and Applications, volume 38, pages 243-252, 2008

INSTITUT CARNOT CEA LIST

digiteo

FraSCAti Studio : création en ligne de services et déploiement dans les nuages

Michel Dirix, Antonio de Almeida Souza Neto et Philippe Merle

Inria Lille-Nord Europe, Université de Lille 1
{michel.dirix,antonio.souza,philippe.merle}@inria.fr

Résumé Nous présentons dans cette démonstration FraSCAti Studio, un environnement de création en ligne de services déployables automatiquement dans les nuages. Cet environnement s'appuie sur le modèle SCA et notre plate-forme FraSCAti d'exécution de services.

Keywords: Cloud Computing, SOC, SOA, SCA, FraSCAti

1 Contexte

L'Internet du Futur sera l'Internet des contenus, des objets et des services [6]. Ce ne sont pas moins de 2 milliards d'ordinateurs qui seront connectés en 2014 [2] et 10 milliards de smartphones en 2016 [5]. Cet Internet sera accessible de partout et par tout le monde. Les services prendront une place importante dans l'Internet du Futur et des milliards de services seront disponibles. Les individus ne seront plus que de simples utilisateurs de services mais deviendront des producteurs-consommateurs (prosumers) [3]. Pour cela, il faudra donner la capacité aux individus de produire des services, leur mettre à disposition des environnements de construction, d'hébergement et de configuration. De cette manière, les contraintes liées à la mise en place d'un serveur, la configuration, l'installation des logiciels, etc. seront cachés à l'utilisateur.

Depuis quelques années, de nombreuses plate-formes de *Cloud Computing* apparaissent rendant les ressources élastiques à grande échelle. De ce fait, que le service déployé soit populaire ou non, les ressources nécessaires s'adaptent en fonction de l'utilisation [1]. Les éditeurs de logiciels profitent de cette avancée, ainsi les logiciels ne sont plus directement installés sur le poste de l'utilisateur mais migrent dans les nuages. Le Cloud Computing permet de s'affranchir de l'hébergement, de la configuration, offrant à l'utilisateur une simplicité dans l'utilisation. En effet, l'environnement est prêt à être utilisé et les services peuvent être déployés presque immédiatement. De plus, avec l'engouement autour des réseaux sociaux, les services deviennent collaboratifs via le Web 2.0.

2 FraSCAti Studio

Nous présentons dans cette démonstration FraSCAti Studio, développé pour répondre à ce futur. Celui-ci est présent dans les nuages et propose un environnement de création rapide de services et de déploiement également dans les nuages tout en s'abstrayant de la problématique de l'hébergement et de la configuration. L'aspect social n'est pas oublié car l'utilisateur peut partager ses services et utiliser ceux de ses amis rendant l'environnement collaboratif.

FraSCAti Studio est une application dans les nuages, développé en composants SCA (Service Component Architecture) et s'exécutant au dessus de FraSCAti, notre plateforme d'exécution d'applications SCA [7][4]. Il permet de faciliter la création et la composition de services (indépendamment de leur implémentation) dans le cadre d'architectures orientées service (SOA). Cette composition de services se fait en utilisant un langage de description architectural (ADL) pour SCA au travers d'un ficher XML (appelé fichier composite).

Afin de construire facilement et rapidement des services, un **environnement de création** est mis à disposition des utilisateurs. Il leur permet de créer des applications de type SCA. Pour aider l'utilisateur, FraSCAti Studio crée des applications à partir de templates prédéfinis générant le fichier composite. Ensuite, la manipulation de l'architecture de l'application se fait au travers d'une

vue graphique arborescente représentant le fichier composite et de menus contextuels pour limiter au maximum les actions possibles. Le développement des applications se fait par l'ajout d'interfaces (Java, WSDL, etc.) sur les services et références et d'implémentations (Java, BPEL, Velocity, JavaScript, etc.) sur les composants. Ici, deux solutions s'offrent à l'utilisateur. La première est d'utiliser l'éditeur de texte intégré dans le studio pour créer le code source, cette notion d'éditeur est similaire à un IDE (Integrated Development Environment ou Environnement de Développement Intégré) simplifié. La seconde est de télécharger directement le code source.

Les nuages facilitent l'abstraction de **l'hébergement et de la configuration**. Ils sont utilisés pour déployer d'une part FraSCAti Studio et d'autres parts des plate-formes FraSCAti. Le studio peut alors déployer les applications sur ces FraSCAti distants. L'utilisateur n'a donc pas à se soucier du déploiement de ses services. Il peut ensuite consulter les services distants grâce au FraSCAti Web Explorer où il pourra les invoquer et les administrer.

Pour accélérer le processus de développement d'une application, la **création** peut être **collaborative**. En effet, FraSCAti Studio utilise les réseaux sociaux (Twitter et Facebook) pour fournir à l'utilisateur un catalogue de services des applications créées par ses amis.

FraSCAti Studio a donc pour but de fournir un environnement de création de services aux utilisateurs tout en leur masquant l'hébergement et la configuration des serveurs dans les nuages.

Remerciements

Ce travail est soutenu par le Consortium EasySOA [1], le Ministère Français de l'Education Nationale et de la Recherche, le Conseil Régional du Nord-Pas de Calais et le FEDER via le *Contrat de Projets Etat Region Campus Intelligence Ambiante* (CPER-CIA) 2007-2013.

Références

1. Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, and al. A view of cloud computing. *Commun. ACM*, April 2010.
2. Cisco visual networking index : Global mobile data traffic forecast update, 2011–2016, 2012. http ://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-520862.html, dernier accès au 18 avril 2012.
3. Papadimitriou D. Future internet-the cross-etp vision document, 2009.
4. Frascati. http ://frascati.ow2.org/.
5. Gartner says more than 1 billion pcs in use worldwide and headed to 2 billion units by 2014, 2008. http ://www.gartner.com/it/page.jsp ?id=703807, dernier accès au 18 avril 2012.
6. Valérie Issarny, Nikolaos Georgantas, Sara Hachem, and al. Service-Oriented Middleware for the Future Internet : State of the Art and Research Directions. *Journal of Internet Services and Applications*, 2(1) :23–45, May 2011.
7. Lionel Seinturier, Philippe Merle, Damien Fournier, Nicolas Dolet, Valerio Schiavoni, and Jean-Bernard Stefani. Reconfigurable SCA Applications with the FraSCAti Platform. In *6th IEEE International Conference on Service Computing (SCC'09)*, pages 268–275, Bangalore, Inde, 2009. IEEE.

1. www.easysoa.org

Mining Repair Actions for Automated Program Fixing

Matías Martínez, Martin Monperrus

Context

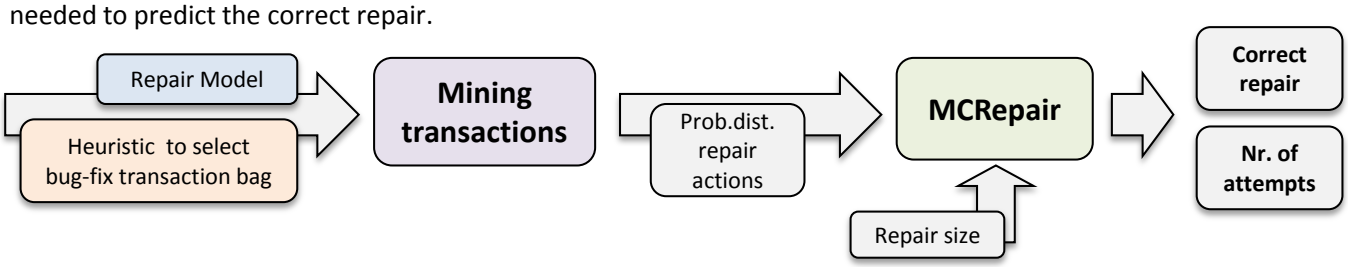
Repair actions are a kind of modification on source code that is made to fix a bug.
Automated program fixing consists of generating repair actions in order to fix bugs in an automated manner.

Objective

We present MCRepair, an automatic software repair process that minimizes the repair time.

Overview

We mine repair actions written by developers from bug-fix in software repositories (e.g. CVS, SVN or Git).
MCRepair uses the mined data to predict an unordered tuple of repairs actions and also counts the number of attempts needed to predict the correct repair.



Methodology

Repair models are sets of semantic repair actions
We define two repair models:

- Coarse-grain: 48 repairs actions.
  E.g. *Statement_Insert*
- Fine-grain: 186 repairs action. E.g. *Condition_Expresion_Changes_of_If_Statement*

Heuristics to select transaction bags T representative of software repair are based on:

- Commit text: contains words *fix, bug, path* (BFP).
- Syntactic features: T with N lines changes (N-LC).
- Semantic features: T with N semantic source code change (N-SC).

Small transactions are very likely to only contain a bug-fix and unlikely to contain a new feature.

MCRepair uses the probability distribution of repair actions from software repositories to maximize the likelihood of finding a correct repair action.

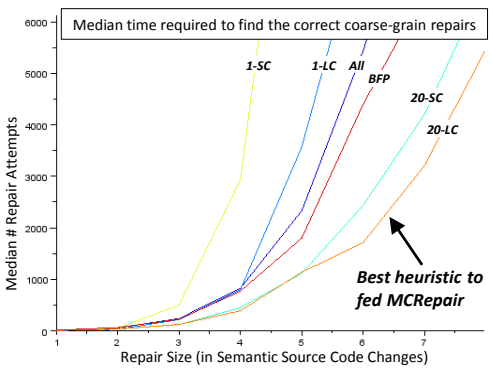
$$prob(repairAction_i) = \frac{cov(repairAction_i)}{\sum_j cov(repairAction_j)}$$

where $cov(repairAction_i)$ = percentage of transactions which include at least one $repairAction_i$

Mining transaction to get probabilities from repair actions based on the intuition: *Different definitions of "bug-fix transactions" yield different topologies for repair models.*

Evaluation

- Analyzed 14 open-source Java projects repositories: 89993 versioning transactions, 1377337 repair actions.
- Computed the median time required to find the correct repair of fix transactions for both repair model. MCRepair was fed with 6 heuristics.
- *Coarse grains*: bug-fix with 6 repair action < 2000 attempts and with 1 between 3 and 8.
- *Fine-grain*: bug-fix with 3 repair action < 22000 attempts, and with 1 between 3 and 16.



Future work

- Coupling MCRepair with fault localization approaches to reduce the time to find correct repairs
- Instantiating repairs actions to produce more precise repairs
- Predict higher order repair actions: co-occurring repair actions

Un processus de développement pour contrôler l'évolution des processus métiers en termes de QoS

Alexandre Feugas, Sébastien Mosser, and Laurence Duchien

Inria Lille-Nord Europe, Univ. Lille 1, LIFL (UMR CNRS 8022)
SINTEF IKT, Oslo, Norvège
{Alexandre.Feugas,Laurence.Duchien}@inria.fr
Sebastien.Mosser@sintef.no

1 Motivations

Les systèmes informatiques sont de plus en plus complexes. Leur développement doit être rapide et réactif afin de pouvoir répondre à des besoins utilisateurs en constante évolution. Les architectures orientées services offrent des éléments de solutions pour palier ce problème en proposant le concept de *service*, l'encapsulation d'une fonctionnalité donnée, et en offrant un couplage lâche des services constituants le système. Toutefois, si ces caractéristiques permettent de faire évoluer facilement le système, il n'en reste pas pour autant difficile de comprendre concrètement l'effet d'une évolution sur le système, notamment en terme de Qualité de Service (QoS) [2]. Nous présentons ici un canevas de développement nommé *Service Modeling for Impact of evoLution framEwork* (SMILE), qui, associé à un processus de développement, permet de contrôler l'évolution de processus métiers en terme de QoS.

2 SMILE : un Processus de Développement Outillé

SMILE est un processus de conception et de mise au point qui offre la possibilité d'étudier les valeurs de propriétés de QoS d'un système orienté service. L'étude du système s'effectue à la conception, en l'analysant pour déterminer les valeurs des propriétés, et à l'exécution, en calculant les informations manquantes qui n'ont pas pu être déterminées. En connaissant la valeur de QoS du système étudié, SMILE vérifie si le système remplit les exigences des utilisateurs. Enfin, notre approche donne la possibilité de contrôler l'évolution d'un système (en cas de changement dans les spécifications ou de violation de la QoS) par l'estimation de son effet sur la valeur de la propriété de QoS.

SMILE repose sur la coopération de deux profils d'acteurs, *l'expert QoS* en charge de la définition des propriétés de QoS du système, et *l'architecte du processus métier* en charge de la définition du processus global, tout au long des cinq étapes détaillées ci-dessous.

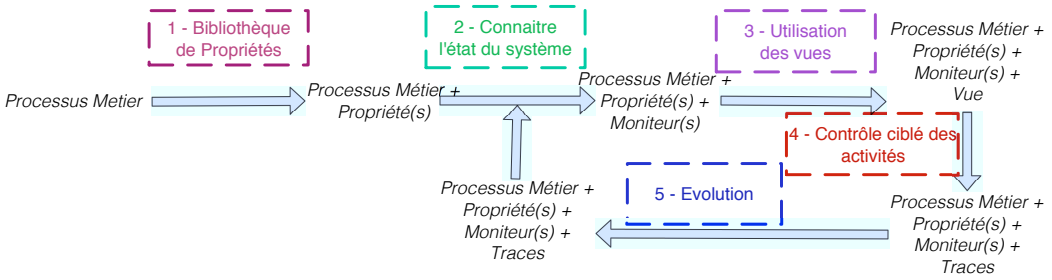


Figure 1. L'approche SMILE

1 - Une Bibliothèque de Propriétés - Le rôle de l'expert QoS est de définir les propriétés de QoS utilisée, de la conception à l'évolution du système. Cela se traduit par la description des analyses de processus métier pour (1) évaluer à la conception la valeur de QoS du processus métier et par composition du système complet, (2) mettre en place les informations de contrôle du système pour une propriété à l'exécution, et (3) gérer les effets de l'évolution. Ces outils sont utilisés dans les étapes suivantes.
2 - Connaître l'état du système en terme de Qualité de Service - La première étape nécessaire pour raisonner sur la QoS consiste à connaître la valeur de QoS du système étudié. Nous nous concentrons dans le cadre de SMILE sur les propriétés de QoS observables, pour lesquelles une formule est disponible pour calculer la valeur de QoS d'un système à partir de la valeur de QoS de ses composants [3]. Pour une propriété donnée, SMILE analyse le processus métier afin de calculer la formule associée. Nous pouvons alors appliquer cette formule afin d'obtenir les valeurs réelles des activités et du processus complet. Lorsque les informations nécessaires sont uniquement disponibles lors de l'exécution (comme par exemple le temps de réponse d'un service), SMILE enrichit le processus métier avec un mécanisme de contrôle pour capturer les informations manquantes à l'exécution et permettre le calcul a posteriori. Ainsi, après avoir déployé le processus métier, l'architecte du processus métier est en mesure de savoir si l'implémentation actuelle respecte les exigences utilisateurs.
3 - Utilisation de vues pour raisonner sur la qualité de service - Une vue [1] est une abstraction permettant de masquer les détails hors du champ d'étude de la propriété pour se concentrer sur les éléments du processus ayant un effet sur la valeur de la propriété. Cela permet notamment de s'abstraire de la granularité fine des valeurs de propriété des services pour pouvoir raisonner au niveau du système de manière plus simple. Cette séparation en termes de vues permet d'isoler la propriété et de mettre en évidence, lors d'une violation des exigences de l'utilisateur, quel est le facteur ayant causé cette violation.
4 - Contrôle ciblé des activités - SMILE essaie de déterminer lors de la conception les valeurs de propriétés. Si le calcul des valeurs nécessite des informations mesurables à l'exécution, SMILE va, de manière sélective, récupérer à l'exécution des informations de contrôle du processus pour capturer les informations manquantes. De cette manière, un sur-coût en terme de capture est évité.
5 - Calcul de l'effet d'une évolution sur la Qualité de Service - SMILE permet d'exprimer l'évolution sous formes d'opérations applicables aux processus métiers. Il analyse le résultat de l'évolution dans le but de déterminer pour une propriété donnée quelles seront les activités affectées directement ou par effet de bord. Par exemple, modifier une variable dans une activité peut entraîner des changements dans d'autres activités. Connaître ce sous-ensemble d'activités du processus métier permet de re-vérifier uniquement la valeur de QoS de ces activités, faisant gagner un temps non-négligeable dans la re-vérification. De plus, en identifiant l'ensemble des éléments affectés, l'architecte des processus métiers est en mesure de comprendre plus facilement quel a été l'effet de son évolution.

Notre approche permet d'avoir une analyse fine de la QoS d'un système de la conception à l'évolution, permettant de s'assurer du maintien de la QoS dans le temps. SMILE offre l'automatisation de l'identification des parties du système à re-vérifier, grâce à une modélisation fixe des propriétés et des systèmes étudiés. Dans un futur proche, nous souhaitons alimenter notre bibliothèque avec d'autres propriétés, faciliter l'expression des moniteurs dans la description des propriétés, et intégrer une approche auto-adaptative prenant en compte les informations de SMILE pour planifier une adaptation permettant de corriger une violation de la QoS.

Références

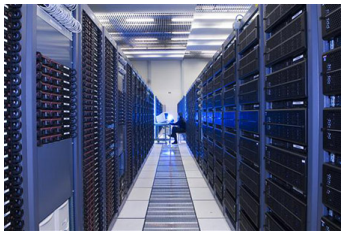
1. Clements, P., Garlan, D., Little, R., Nord, R. et Stafford, J. : Documenting software architectures : views and beyond. In : 25th International Conference on Software Engineering, 2003.
2. Cheng, B., de Lemos, R., Giese, H., Inverardi, P., Magee, J., et al. : Software Engineering for Self-Adaptive Systems : A Research Roadmap. In : Software Engineering for Self-Adaptive Systems,2009.
3. Crnkovic, I., Larsson, M., Preiss, O. : Concerning Predictability in Dependable Component-Based Systems : Classification of Quality Attributes, In : Dependable Component-Based Systems, 2005.

ACCESS-CONTROL MANAGEMENT @ RUNTIME

olivier-nathanael.ben_david@inria.fr

MOTIVATION

The boundary between development time and run time of eternal software intensive applications is fading. In particular, these systems are characterized by the necessity to evolve requirements continuously, even after the deployment.



PROBLEM

There are currently many solutions to enforce RBAC policies in web and mobile applications. However, these approaches require that running systems be stopped to incorporate changes in access control policies that may occur after the system is deployed.

CONTRIBUTION

Systematic causal link from RBAC policy to adaptive runtime environment.

SOLUTION

We rely on two metamodels : RBAC (Role Based Access-Control) and Kevoree. RBAC allows to define policy models, and Kevoree allows to model distributed component-oriented software architectures.
Policy model is verified and transformed into an architectural model enforcing it.
The architectural model is then used as the description of the target system state by the kevoree middleware platform for system adaptation.

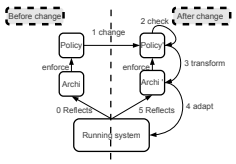
REFERENCES

[1] K. Yskout, and O-N. Ben David, R. Scandariato, B. Baudry. Requirements-driven Runtime Reconfiguration for Security In *EternalS '11*

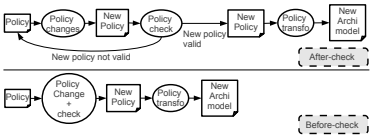
[2] B. Morin, T. Mouelhi, F. Fleurey, Y. Le Traon, O. Barais, and J.-M. Jézéquel. Security-driven Model-based Dynamic Adaptation In *ASE '10*

KEVOREE : `http://kevoree.org/`

OVERVIEW



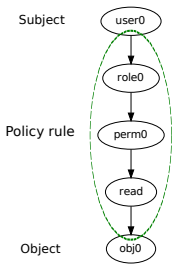
Our process to modify the policy without system interruption consists in (1) changing the policy of a system, (2) checking that the policy is valid against RBAC (Role-based access control) properties, (3) transforming the policy into an architecture model enforcing it.



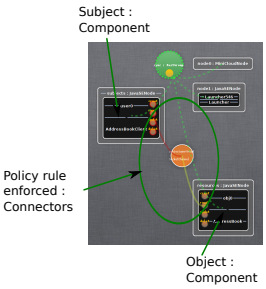
We propose two techniques for the policy verification, before-check and after-check, presented in the figure above, they intend to cope with two kind of policy changes (e.g. foreseeable and unforeseeable changes). A policy change consists in adding policy elements (e.g. adding roles, permissions ... or adding relationships between policy elements user-role assignment ...).

MAPPING

Mapping : $\{User = Component, Object = Component, Rule = Connectors\}$

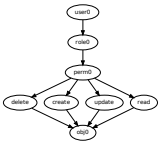
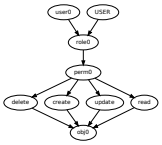
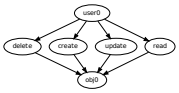
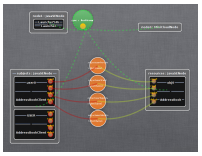
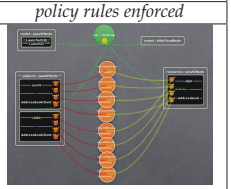


Policy model



Architecture model

INCREMENTAL TRANSFORMATION

Before change	Change	After Change
	addUserRole(USER,role0);	
policy model	*script*	*policy model*
		
policy rules enforced	*policy rules to be enforced*	*policy rules enforced*
	add channel subjectUSERcreate ... bind USER.create@subjects=>subjectUSERcreate ...	
architecture model	*script*	*architecture model*

ACKNOWLEDGEMENTS

This work is partially supported by the EU FP7-ICT-2009.1.4 Project Number : 256980, NESSoS: Network of Excellence on Engineering Secure Future Internet Software Services and Systems.

Security Testing in the TASCCC Project

K. Cabrera Castillos[1], F. Dadeau[1], Y. Ledru[2], L. du Bousquet[2], T. Triki[2], S. Taha[3], B. Legeard[1,4], J. Botella[4], B. Chetali[5], J. Bernet[5], D. Rouillard[6]

1 Femto-ST, 2 LIG, 3 Supélec, 4 Smartesting, 5 Trusted Labs, 6 Serma Technologies

Funded under grant ANR-09-SEGI-014

Project objectives

The TASCCC project aims at defining a methodology for assisting the Common Criteria evaluation. In particular, we focus on the validation phase for which it is mandatory to provide complete test cases and test reports explaining which test covers which security properties and how it has been designed. Figure 1 depicts the TASCCC process.

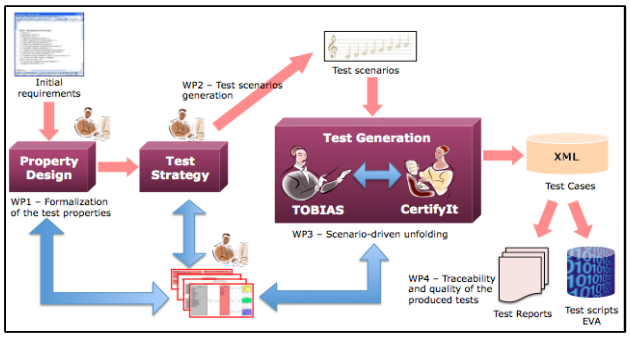


Fig. 1 – The TASCCC Process for Scenario-Based Test Generation

Property formalization

Properties are formalized using a dedicated formalism introduced in [CDJT2011], that is based on Dwyer's property patterns, in which a property is a combination of a **scope** and a **pattern**.

There are 5 scopes:

- **globally** for the whole execution;
- **before** *E*, representing the executions before the first occurrence of event *E*;
- **after** [**last**] *E*, representing the executions after the first (or last) occurrence of event *E*;
- **between** [**last**] *E1* **and** *E2*, representing the pieces of executions started by the first (or last) occurrence of event *E1* and ended by event *E2*;
- **after** [**last**] *E1* **until** *E2*, representing the pieces of executions started by the first (or the last) occurrence of event *E1* and possibly ended by event *E2*.

There are 5 patterns:

- **always** *P* means that predicate *P* is always satisfied;
- **never** *E* means that event *E* never appears ;
- **eventually** *E* [**at least**, **at most**] *k* **times** specifies the existence of event *E*;
- *E1* **precedes** *E2*, and *E1* **follows** *E2* specifying ordering of events.

There are 2 kinds of events: ***isCalled****(Op,Pre,Post,Tags)* which specifies that operation *Op* is called from a state satisfying *Pre* resulting in a state satisfying *Post* and activates one of the behaviors identified by *Tags*; ***becomesTrue****(P)* which specifies that *P* has become true by the last operation invocation.

Modeling for test generation

UML models for test generation are made using:

- a class diagram, which provides the data model
- an object diagram, which gives the initial state of the SUT
- additional OCL constraints that describe the expected behavior of the SUT.

Figure 2 shows an example of a class diagram and an excerpt of an OCL operation modeled with these guidelines.

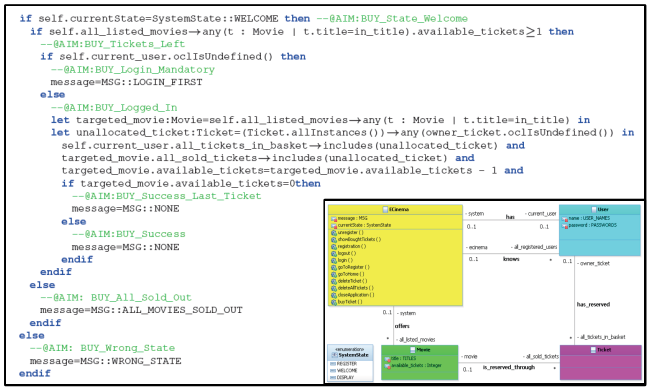
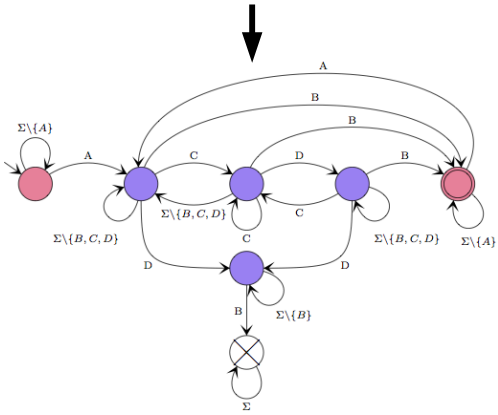


Fig. 2 – Operation specification with OCL constraints and traceability tags

Test Scenario Generation

In order to produce the test scenarios, each property is first translated into an automaton, which highlights relevant states and transitions expressed in the property. Test scenarios are then produced by extracting paths from these automata with respect to coverage criteria specifically defined for this kind of automata.

C **directly precedes** D **between** A **and** B



→(Σ\{A})*.**A.(Σ{B,C,D})*.C+.**Σ\{B,C,D})*.B
→Σ\{A})*.A.(Σ\{B,C,D})*.**C.C+**.B
→Σ\{A})*.A.(Σ\{B,C,D})*.**C .(Σ\{BCD})*.C+**.B
...

Test Scenario Unfolding

The scenarios are then unfolded with the TOBIAS tool designed for combinatorial testing. For this purpose, TOBIAS is coupled with the animation engine of the CertifyIt test generator from the Smartesting company. Also, additional test reduction mechanisms have been introduced in [TLdBDB]. Tests can be exported as test scripts or to produce test reports.

References

Web: http://lifc.univ-fcomte.fr/TASCCC

[CDJT11] K. Cabrera Castillos, F. Dadeau, J. Julliand and S. Taha. Measuring Test Properties Coverage for evaluating UML/OCL Model-Based Tests. ICTSS'2011.

[TLdBDB] T. Triki, Y. Ledru, L. du Bousquet, F. Dadeau, J. Botella, Model-based filtering of combinatorial test suites. FASE'2012.

EMF Models Transformations with Tom

jeanchristophe.bach@inria.fr, pierre-etienne.moreau@loria.fr
and marc.pantel@enseeiht.fr

Context

- Importance of MDE in software engineering: transformations are an essential part
- Need of safe software for critical systems: checking all steps of the development chain is mandatory. Therefore it is important to write qualified models transformations (*i.e.* transformations which ensure that interesting properties are preserved)
- Two main approaches to write models transformations:
  - using a General Purpose language (GPL) such as Java+EMF
  - using a DSL such as ATL, Kermeta, *etc.*
- Our approach: embedding a dedicated transformation language in a GPL by using Tom

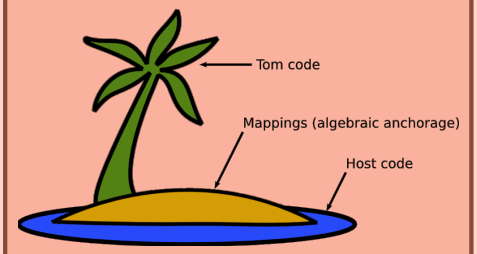
Tom language

Tom is a language which extends general purpose languages (C, C#, Caml, Java, Python, *etc.*) by addings features:

- pattern-matching
- rewriting
- strategies
- mappings

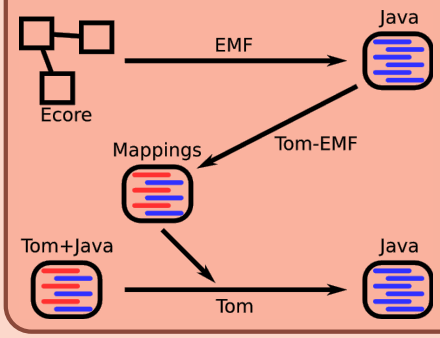
Tom code is written inside the Java program and a Tom block may contain Java code which may also contain Tom code. The Tom compiler compiles Tom constructs without parsing host code, and dissolves them into the host language.

Tom language implements the principle of *Formal Islands*.



http://tom.loria.fr

Tom-EMF: manipulating EMF models with Tom



Tom-EMF is a mappings generator: it takes a Java-EMF metamodel as input and generates mappings which allow to see an EMF model as a Tom term.

In addition, by using a dedicated tool called `EcoreIntrospector`, Tom strategies can be used to traverse and to rewrite the model.

Extension of Tom

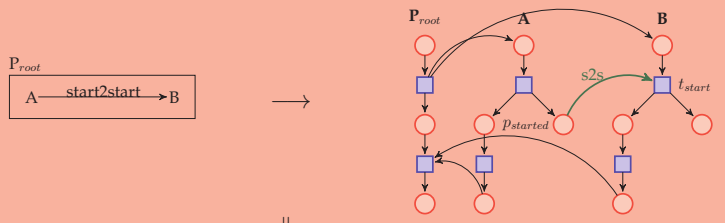
We propose to extend the Tom language to be able to write easily models transformations such as `SimplePDLToPetriNet` one:

- **%transformation**: the main construct dedicated to models transformations. It is composed of elementary transformations.
- **%tracelink**: this construct saves elements in the link model to reuse them during the transformation.
- **%resolve**: it creates intermediate elements representing elements which may not have already been created in other transformation steps.

Example of transformation code:

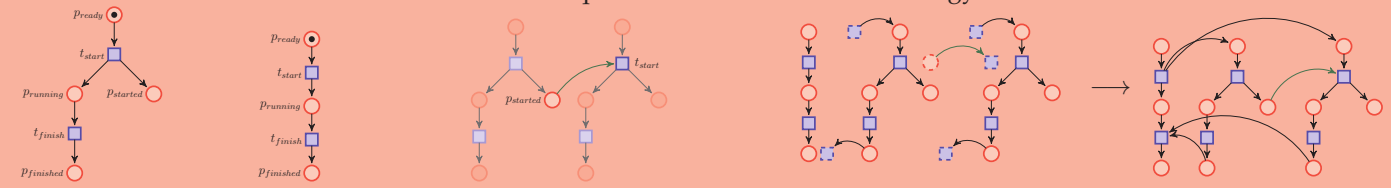
```
%transformation SimplePDLToPetrinet()
  with(Process,WorkDefinition,WorkSequence)
  to(Place,Transition,Arc) {
    definition P2PN traversal 'TopDown(P2PN()) {
      p@Process[name=n, from=f] -> {
        ...
        %tracelink(t_start: Transition, 'Transition[name='n]);
        ...
      }
    }
    definition WD2PN traversal 'BottomUp(WD2PN()) {
      wd@WorkDefinition[name=n, parent=p] -> {
        ...
        Transition source =
            %resolve('p:WorkDefinition,t_start:Transition);
      }
    }
    definition WS2PN traversal 'TopDown(WS2PN()) {
      ws@WorkSequence -> { ... }
    }
  }
```

Use case: SimplePDLToPetriNet transformation



- Decomposition of the transformation into three elementary transformations: ProcessToPetriNet, WorkDefinitionToPetriNet and WorkSequence-ToPetriNet
- Addition of intermediate elements to represent elements which may not have already been created
- Application of a Resolve strategy

WorkDefinition + Process + WorkSequence + Resolve strategy:



SMT solving for TLA$^+$ proof obligations

Stephan Merz[1] and Hernán Vanzetto[1,2]

[1] INRIA Nancy Grand-Est & LORIA, Nancy, France
[2] Microsoft Research-INRIA Joint Lab, Saclay, France

1 Introduction

TLA$^+$ [3] is a language for specifying and verifying systems, in particular concurrent and distributed algorithms. It is based on a variant of ZF set theory for specifying the data structures, and on the Temporal Logic of Actions (TLA) for describing the dynamic system behavior. It is supported by the TLA$^+$ toolbox, which provides an Eclipse-based GUI framework for TLA$^+$ tools. Recently, a first version of the TLA$^+$ proof system TLAPS [2] has been developed, in which users can deductively verify safety properties of TLA$^+$ specifications. TLA$^+$ contains a declarative language for writing hierarchical proofs, and TLAPS is built around a *proof manager*, which interprets this proof language, expands the necessary module and operator definitions, generates corresponding proof obligations, and passes them to backend verifiers. While TLAPS is an interactive proof environment that relies on users guiding the proof effort, it integrates automatic backends to discharge proof obligations that users consider trivial.

The two main backends of the current version of TLAPS are Zenon, a tableau prover for first-order logic and set theory, and Isabelle/TLA$^+$, a faithful encoding of TLA$^+$ in the Isabelle proof assistant, which provides automated proof methods based on first-order reasoning and rewriting. Beyond its use as a semi-automatic backend, Isabelle/TLA$^+$ can also be used to certify proof scripts produced by Zenon. Cooper's algorithm for Presburger arithmetic is also available as a backend.

TLA$^+$ heavily relies on modeling data using sets and functions. Assertions mixing arithmetic, sets and functions arise frequently in TLA$^+$ proofs. In the work reported here we present a new SMT-based backend for (non-temporal) TLA$^+$ formulas that encompasses set-theoretic expressions, functions, arithmetic, records, and tuples. The proposed Satisfiability Modulo Theories (SMT) approach fits our needs, as it combines first-order reasoning with decision procedures for theories such as equality, integer and real arithmetic and arrays.

2 From TLA$^+$ to SMT formats

The input languages of state-of-the-art SMT solvers are based on many-sorted first-order logic. This allows us to design a unique translation from TLA$^+$ expressions to an intermediate language, from which the translation to the actual target languages of particular SMT solvers is straightforward. The considered languages are: (i) SMT-LIB, the *de facto* standard input format for SMT solvers (in our experiments we use the CVC3 solver as a "baseline"), (ii) an extension of SMT-LIB for the solver Z3, and (iii) the native input language of the solver Yices. The considered TLA$^+$ formulas are translated to quantified first-order formulas over the theory of linear integer arithmetic, extended with free sort and function symbols. In particular, we make heavy use of uninterpreted functions and quantified formulas.

TLA$^+$ is an untyped language, which makes it very expressive, but also makes automated reasoning quite challenging. In contrast, our target languages have sorts, which are supported by dedicated decision procedures in SMT solvers. The first challenge is therefore to design a typing discipline that is compatible with the logics of SMT solvers but accommodates typical TLA$^+$ specifications. In a first step of the translation, a suitable type is assigned to every expression that appears in the proof obligation. We make use of this type assignment during the translation of expressions. For example, equality between integer expressions will be translated differently from equality between sets or functions.

Type inference may fail because not every set-theoretic expression is typable according to our typing discipline, and in this case the backend aborts. Otherwise, the proof obligation is translated to SMT formulas. Observe that type inference is relevant for the soundness of the SMT backend: a proof obligation that is unprovable according to the semantics of untyped TLA$^+$ must not become provable due to incorrect type annotation. As a trivial example, consider

the formula $x+0=x$, which should be provable only if x is known to be of arithmetic sort. Type inference essentially relies on assumptions that are present in the proof obligation and that constrain the values of symbols (variables or operators).

The type inference algorithm consists of three main steps: (1) Compute *safe* types, which broadly classify the universe of TLA$^+$ values, distinguishing between atomic values, sets, and functions. (2) Infer types for subexpressions from *typing hypotheses*, which are available facts of the form $x \approx exp$ and $\forall y \in S : x(y) \approx exp$, where $\approx \in \{=, \in, \subseteq\}$ and exp is an expression with known types. (3) Type-check the assigned values.

Once a type assignment is determined for the symbols in a TLA$^+$ proof obligation, it can be translated to the input languages of SMT solvers. This is done in two steps. In a first phase, the proof obligation is pre-processed to eliminate expressions that are not directly available in SMT, such as set operators or function expressions. The resulting formula will contain only TLA$^+$ expressions that have a direct representation in the first-order logic of SMTs, namely, the logical and arithmetic operators and the IF/THEN/ELSE construct. The translation to our target languages is then just a syntactic rewriting.

We omit the formal, technical definition of type assignment and of the translation. The SMT backend has been described in detail in [1].

3 Results

We have used our new backend with good success on several examples that had previously been proved interactively using TLAPS. In particular, we show in the following table results for two invariant proofs: (a) the N-process Bakery algorithm for mutual exclusion, and (b) the Memoir security architecture. For each benchmark, we indicate the size (number of lines) of the original interactive proof, the time (in seconds) required to verify that proof on a standard laptop, as well as the corresponding figures when parts of the proof are performed using the SMT backend, in its three flavors.

	Original		CVC3		Yices		Z3	
	size	time	size	time	size	time	size	time
Bakery	398	180	7	33	76	63	7	5
Memoir	2381	53	208	7	208	5	208	7

We can see that the interactive proof size and the time to prove them were reduced significantly. These encouraging results show that significant automation can be gained by using SMT solvers for the verification of standard TLA$^+$ models. In particular, the typing discipline is well adapted for a direct translation to SMT formats. The SMT backend can handle a useful fragment of TLA$^+$, including first-order logic, elementary sets, functions, linear arithmetic, records, tuples, although it cannot handle sets of sets. We observe that it also scales well for large formulas. In practice, the SMT method has replaced the backend based on Cooper's algorithm. In some cases, however, type inference may fail for logically valid obligations, implying that some obligations may require logically unnecessary typing hypotheses to be proved.

We are currently working on an encoding of TLA$^+$ formulas that requires less type information. We are also interested in certifying proofs produced by the SMT solvers in Isabelle/TLA$^+$. The SMT backend is available at [4] as part of the current release of TLAPS.

References

1. Stephan Merz and Hernán Vanzetto *Automatic Verification of* TLA$^+$ *Proof Obligations with SMT Solvers*. LPAR, 2012.
2. Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. *Verifying Safety Properties with the* TLA$^+$ *Proof System.*. IJCAR, 2010.
3. Leslie Lamport. *Specifying Systems*. Addison-Wesley, 2002.
4. TLAPS website. `http://www.msr-inria.inria.fr/~doligez/tlaps/`.

Génération de code fonctionnel à partir de spécifications inductives dans Coq

Pierre-Nicolas Tollitte, David Delahaye, Catherine Dubois

Écrire du code ou bien des spécifications ?

Dans l'assistant à la preuve Coq, il est possible d'écrire des spécifications logiques et du code fonctionnel. Les deux approches couvrent des besoins différents.

Spécification logique

```
Inductive nat : Set := O : nat | S : nat → nat.
Inductive add : nat → nat → nat → Prop :=
| addZero : forall n, add n O n
| addSucc : forall n m p, add n m p →
                          add n (S m) (S p).
```

La relation *add n m p* est définie quand *p* est la somme des entiers naturels *n* et *n*.

Code fonctionnel

Cette fonction Coq calcule le dernier argument de la spécification précédente à partir des deux premiers :

```
Fixpoint add (p1 : nat) (p2 : nat) : nat :=
  match (p1, p2) with
    | (n, O) ⇒ n
    | (n, (S m)) ⇒ S (add n m)
  end.
```

Problématique

Coq propose un mécanisme d'extraction, mais il n'effectue aucun traitement pour les spécifications logiques. Pourtant, on préfère souvent écrire une spécification logique plutôt qu'une fonction :

- Plus simple à exprimer qu'une fonction (pas de problème de terminaison, on n'écrit que les cas vrais)
- Extraction possible de plusieurs fonctions à partir d'une seule spécification
- Impossibilité de faire le chemin inverse (fonction → spécification)

Solution

Compléter l'extraction classique de Coq pour extraire automatiquement des fonctions ML à partir des spécifications logiques. Deux types d'extraction :

- Extraction complète : Extraction de la fonction booléenne définie par la relation inductive (*add* : *nat* → *nat* → *nat* → *bool*)
- Extraction partielle : Extraction de fonctions qui calculent certains arguments de cette relation

Pour pouvoir réaliser une extraction partielle, on attribue un mode d'extraction à la spécification qui contient les indices désignant les arguments qui seront des entrées pour la fonction extraite. A partir de *add*, on peut générer la fonction d'addition grâce au mode **{1, 2}** et la fonction soustraction grâce au mode **{1, 3}**.
Avant d'extraire une fonction, on vérifie que la spécification est déterministe.

Plugin d'extraction

- Distribution dans les contributions de Coq depuis la version 8.4
- Implantation des deux types d'extractions complète et partielle
- Extraction de fonctions vers ML, ainsi que toutes leurs dépendances, dans un fichier séparé
- Extraction de fonctions mutuellement récursives

Travail en cours : extraction vers Coq

Le plugin sera étendu pour générer des fonctions Coq et fournir automatiquement les preuves de correction des fonctions générées par rapport à la spécification.

Résultats

Les commandes d'extraction suivantes :

```
Extraction Relation add [1 2].
Extraction Relation add [1 3].
Extraction Relation add [1 2 3].
```

permettent d'obtenir le fichier ML suivant :

```
type nat = O | S of nat

let rec add12 p1 p2 =
  match (p1, p2) with
  | (n, O) → n
  | (n, S m) → S (add12 n m)

let rec add23 p1 p2 =
  match (p1, p2) with
  | (O, n) → n
  | (S m, S p) → add23 m p
  | _ → assert false

let rec add_full p1 p2 p3 =
  match (p1, p2, p3) with
  | (n_1, O, n) → n = n_1
  | (n, S m, S p) → add_full n m p
  | _ → false
```

Modélisation explicite de l'adaptation sémantique entre modèles de calcul

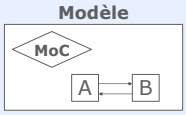
ÉCOLE SUPÉRIEURE D'ÉLECTRICITÉ
Supélec

Ayman Dogui, Frédéric Boulanger, Christophe Jacquet, Cécile Hardebolle

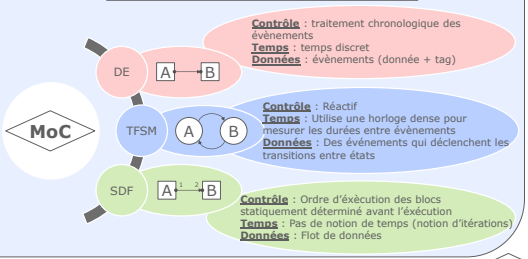
1 Bases de l'approche

Modèle sous forme de diagramme boîte-flèches

Modèle

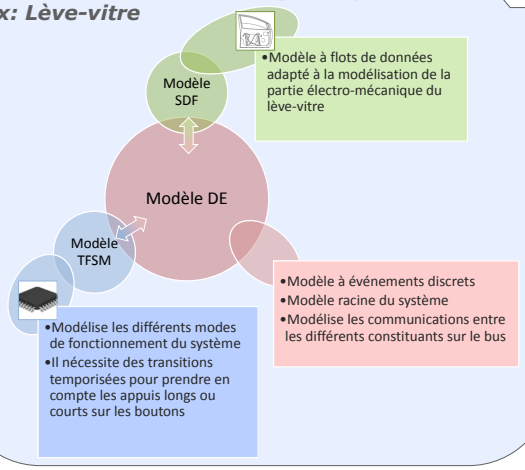


Le MoC définit la nature du modèle

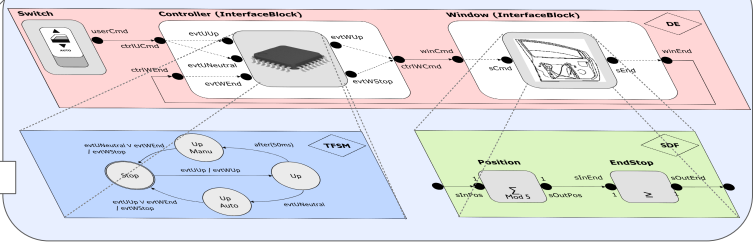


2 Modèle multi-paradigme

Ex: Lève-vitre



3 Composition hiérarchique de modèles



4 Problématique

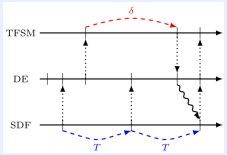
- Hétérogénéité des modèles
- Composition de comportements décrits dans des formalismes différents
- Adaptation sémantique entre les différents modèles

5

Adaptation sémantique du contrôle

-Contraintes d'horloge CCSL-

- Clock Constraint Specification Language (CCSL) :
  - Langage déclaratif basé sur les horloges (Clock)
  - Une horloge est un ensemble d'occurrences d'évènements discrets
  - Une occurrence d'évènement est représentée par un tick sur l'horloge
- CCSL est suffisant pour spécifier l'adaptation sémantique du contrôle

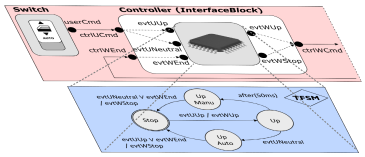


: Ligne de temps
: Adaptation du contrôle
: 'Tick' d'horloge – les instants de contrôle
: Délais de temps TFSM (pour une transition temporisée)
: Période d'itération SDF

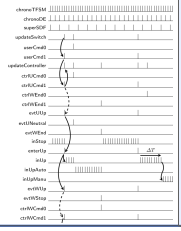
Semantic Adaptation using CCSL Clock Constraints
Frédéric Boulanger, Ayman DOGUI, Cécile Hardebolle, Christophe Jacquet, Dominique Marcadet, Iuliana Prodan
5th International Workshop on Multi-Paradigm Modeling (MPM 2011) at MODELS 2011

Adaptation sémantique du temps

-Algorithme de résolution de contraintes-



- Les ticks d'horloges CCSL ne portent pas d'étiquettes (tag)
- CCSL est insuffisant pour spécifier l'adaptation sémantique du temps :
  - Deux ticks qui coincident ne possèdent pas forcément le même tag
- Approche :
  - Rajouter la notion de Tag aux ticks d'horloge (Tagged Signal Model)
  - Rajouter aux contraintes CCSL des relations entre Tags



Adaptation sémantique des données

- Associer des ticks d'horloge à la présence de données sur les pins?
- Langage de modélisation de l'adaptation sémantique des données?
- ...

Relations entre horloges

- Le temps s'écoule 3x plus vite dans TFSM que dans DE **(TagRelation)**
- L'entrée dans UpManu est retardée de 50ms par rapport à l'entrée dans Up **(DelayFor)**
- L'horloge TFSM est une sous horloge de DE **(SubClock)**
- ... (FilterBy, SustainUpTo)

Collecte des contraintes

- Parcourir le modèle à partir de la racine (modèle hiérarchique)
- Vérifier les contraintes des blocs
- Créer des ticks d'horloge correspondants aux contraintes (avec ou sans étiquettes)

Algorithme de résolution de contraintes

- While(!done)
  - Trouver les ticks à l'instant courant
  - Trouver les tags des nouveaux ticks
  - Done = plus de nouveaux ticks

Ayman DOGUI <ayman.dogui@supelec.fr>
Supélec – Département Informatique

Generation of operational transformation rules from examples

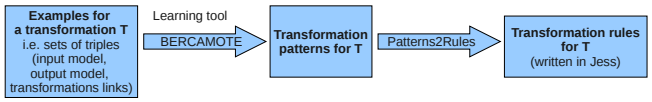
Hajer Saada [1], Xavier Dolques [2], Marianne Huchard [1], Clémentine Nebut [1], Houari Sahraoui [3]

[1]LIRMM (Université de Montpellier 2 et CNRS), [2]INRIA (Rennes), [3]DIRO (Montréal)

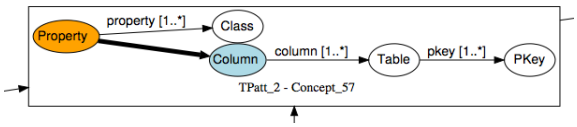
Introduction

- Model Transformation By Example
  - key component of Model Driven Engineering
  - consists in learning transformation programs/rules from examples

Overview of the rules generation and execution

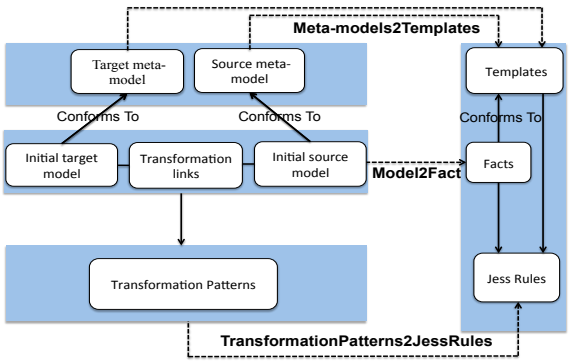


- Bercamote: a tool based on *Relational Concept Analysis* to generate transformation patterns
- Relational Concept Analysis to generate transformation patterns
  - classification of the elements of source and target models of the transformation
  - classification of the links that show how elements are connected by the transformation
  - transformation of the resulting concepts into transformation patterns
  - a transformation pattern is composed of premise (source elements) and conclusion (corresponding target elements)
  - an example of a generated transformation pattern:

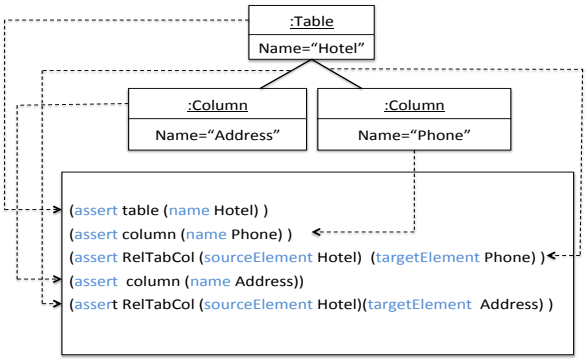


  - interpretation: a property linked to a class (premise) is transformed to a column linked to a table. This table is also linked to a primary key (conclusion)
- Objective: making the transformation patterns operational
  - converting transformation patterns into operational rules using *Java Expert System Shell*
- Java Expert System Shell
  - a rule engine integrated in the Java platform
  - a JESS program is composed of *facts* (data) and *rules* which are conform to *templates*
  - properties of facts are defined using the *slots*
  - a JESS rule contains conditions, called left-hand-side (LHS) and actions, called right-hand-side (RHS)

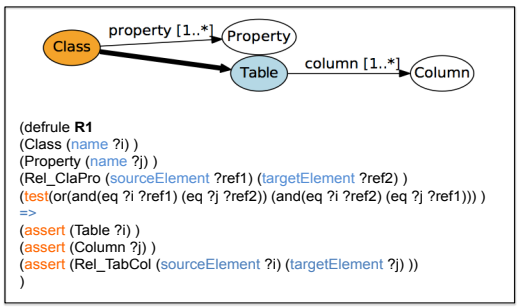
Generation operational rules process



- **Meta_models2Templates**
  - each meta-class of the meta-model is transformed into a template keeping the same name
  - each meta-attribute is transformed into a *slot*. The type of the slot is the type of the meta-attribute
  - each meta-reference is transformed into a template which contains source element and target element names
- **Models2Facts**
  - each instance of a meta-class is transformed into a fact
  - each instance of a meta-attribute is transformed into a fact
  - each instance of a meta-reference between two instances of meta classes is transformed into a fact
  - example of transformation of a simple relational model:



- **TransformationPatterns2JessRules**
  - premise is equivalent to the LHS: both of them describe condition to check to apply the rule
  - conclusion is equivalent to the RHS: they describe the action to perform if the first part is satisfied
  - transformation of the premise
    - each element in the premise is transformed into a JESS condition with a slot for the name
    - a relation between two elements is transformed to: a JESS condition with two slots containing the relation elements names, and a test to verify that this relation links the two elements.
  - transformation of the conclusion
    - each element in the conclusion is transformed into a JESS fact assertion with a slot for the name
    - relation between two elements is also converted to fact assertion linking the two elements
  - example of pattern transformation into JESS rule:



  - interpretation
    - the premise of the pattern is transformed to four JESS conditions: the existence of a class i, the existence of a property j and the existence of a relation from a class to a property and that the existing relation links i to j
    - the conclusion of the pattern is transformed to three fact assertions: a table i, a column j and relation between i and j

Tools and Reference

- JESS Rule Engine, http ://herzberg.ca.sandia.gov/jess
- Dolques, X., Huchard, M., Nebut, C. : From transformation traces to transformation rules : Assisting Model Driven Engineering approach with formal concept analysis. In : Supplementary Proceedings of ICCS'09. pp. 15-29 (2009)
- Ganter, B., Wille, R.: Formal Concept Analysis, Mathematical Foundations. Springer (1999)

Hajer.Saada@lirmm.fr, Xavier.Dolques@inria.fr, Marianne.Huchard@lirmm.fr, Clementine.Nebut@lirmm.fr, sahraouh@iro.umontreal.ca

Campagne de collecte de données et vie privée

Nicolas Haderer[1], Miguel Núñez, del Prado Cortez[2,3], Romain Rouvoy[1], Marc-Olivier Killijian[2], and Matthieu Roy[2,3]

[1] INRIA Lille – Nord Europe, Project-team ADAM
University Lille 1, LIFL – CNRS UMR 8022, France
{nicolas.haderer,romain.rouvoy}@inria.fr
[2] LAAS-CNRS, France
[3] Université de Toulouse ; UPS, INSA, INP, ISAE ; LAAS ; F-31077 Toulouse, France
{mnunezde,mkilliji,mroy}@laas.fr

Résumé Les communautés scientifiques ont souvent recours à la simulation dans le but de valider leurs théories. Cependant, la pertinence des résultats obtenus est fortement dépendante de la qualité des traces générées par les simulateurs. Ce phénomène est particulièrement vrai lorsque l'on considère les traces de mobilité humaine qui sont difficilement prévisibles. Dans ce contexte, la popularité des nouvelles générations de smartphones, équipés d'une grande variété de capteurs (GPS, bluetooth, accéléromètre, etc.), offre de nouvelles perspectives pour la collecte de données réalistes au sein d'une population. Cependant, la nature sensible, du point de vue de la vie privée, des informations collectées représente un des principaux obstacles aux déploiement généralisé d'une application de collecte de données et à son adoption auprès des utilisateurs.

C'est pourquoi nous présentons UBILAB , une nouvelle plate-forme permettant aux scientifiques de mettre en place facilement des campagnes de collecte de données et d'inférer automatiquement différentes attaques sur les données partagées par les utilisateurs mobiles afin de les avertir d'un risque potentiel d'atteinte à leurs informations privées.

1 Introduction

La nouvelle génération de smartphones (Android, iPhone), maintenant équipée d'une grande variété de capteurs (GPS, bluetooth, accéléromètre, etc.), offre de nouvelles perspectives à diverses communautés scientifiques afin de réaliser différentes campagnes de collectes de données massives d'une population et de son environnement. Ces données peuvent ainsi être exploitées pour mieux comprendre les mouvements d'une population, de mettre au point de nouveaux protocoles de communication, d'analyser les interactions sociales des utilisateurs, etc. La nature sensible des données collectées, généralement couplant des informations temporelles et géographiques, peuvent révéler des informations critiques sur la vie privée d'un utilisateur (résidence privée, opinion politique ou réseau social), même si celles-ci ont été préalablement anonymisées. Ce risque potentiel représente un des principaux obstacles aux déploiement généralisé d'une application de collecte de données et à son adoption auprès des utilisateurs.

Dans ce contexte, nous présentons UBILAB , une plate-forme dédié à la gestion de campagnes de collecte de données auprès d'utilisateurs de téléphones mobiles. UBILAB est le résultat de la l'association de deux plate-formes : ANTDROID [2] et GEPETO (GeoPrivacy Enhanced TOolkit)[1]. Ce système profite ainsi de l'architecture de ANTDROID pour rapidement mettre en place une campagne de collecte de données, et des algorithmes de GEPETO pour inférer automatiquement différentes attaques sur les données partagées par les utilisateurs mobiles afin de les avertir d'un risque potentiel d'atteinte à leurs vies privées.

2 UBILAB

La plate-forme est composée de deux parties — chacune est destinée aux différents acteurs évoluant dans la plate-forme : les scientifiques et les cobayes. Le serveur d'application dédié, destiné aux scientifiques, repose sur le style architectural REST (*REpresentational State Transfer*) fournissant l'ensemble des services pour la définition, la diffusion et l'exploitation d'une expérience de collecte de données. L'application cliente est une application Android téléchargeable, utilisée par une communauté d'utilisateurs pour partager leurs traces d'activités. Pour ce faire, l'utilisateur s'abonne à une ou plusieurs campagnes publiées par des scientifiques. Ces campagnes correspondent à des scripts de collecte automatique des informations requises par le scientifique. Ces scripts sont téléchargés via le serveur d'application dédié puis interprétés par un moteur de script intégré dans l'application cliente. Afin de maîtriser toute diffusion d'information, l'application cliente dispose de différents contrôles permettant aux utilisateurs d'autoriser ou non la collecte de certaines informations jugées trop sensibles (par ex., sa position). Les données collectées sont ensuite envoyées automatiquement sur le serveur d'application lorsque le téléphone est alimenté en courant pour limiter sa consommation énergétique. Ces données sont ensuite soumises à analyses par la plate-forme GEPETO, essayant d'inférer le pattern de mobilité de l'utilisateur en identifiant ses points d'intérêt (*POI*) et comment l'utilisateur transite entre ses différents *POI*. Le résultat est ensuite renvoyée à l'utilisateur afin qu'il puisse évaluer si les données collectées peuvent compromettre sa vie privée. L'utilisateur peut alors ensuite décider de valider les données pour les rendre accessibles pour les scientifiques, de les supprimer, ou d'appliquer des algorithmes d'assainissement (distorsion aléatoire, sous échantillonnage, etc..) fourni par GEPETO pour ajouter du bruit sur les traces de mobilité.

Références

1. S. Gambs, M.-O. Killijian, and M. N. del Prado Cortez. Gepeto : a geoprivacy-enhancing toolkit. *AINA'09 Workshop on Advances in Mobile Computing and Applications : Security, Privacy and Trust, Perth, Australia.*, August 2009.
2. N. Haderer, R. Rouvoy, and L. Seinturier. AntDroid : A distributed platform for mobile sensing. Rapport de recherche RR-7885, INRIA, Lille, France., February 2012.

Résumé

Ce document contient les actes des Quatrièmes journées nationales du Groupement De Recherche CNRS du Génie de la Programmation et du Logiciel (GDR GPL) s'étant déroulées à l'Université de Rennes I du 20 au 22 Juin 2012.

Les contributions présentées dans ce document ont été sélectionnées par les différents groupes de travail du GDR. Il s'agit de résumés, de nouvelles versions et de posters qui correspondent à des travaux qui ont déjà été validés par les comités de programmes d'autres conférences et revues et dont les droits appartiennent exclusivement à leurs auteurs.

GDR·GPL 2012