Défis pour le Génie Logiciel et la Programmation à l'échéance 2020

Ce document est un extrait des
« Actes des deuxièmes journées nationales du Groupement De Recherche CNRS du Génie de la Programmation et du Logiciel »,
Edités par Eric Cariou, Yves Ledru et Laurence Duchien,
Université de Pau
10 au 12 Mars 2010

Les actes des deuxièmes journées nationales sont disponibles sur le site web du GDR GPL

http://gdr-gpl.cnrs.fr/

Cet extrait regroupe les 5 textes qui ont été sélectionnés suite à notre appel à défis

Appel à Défis pour le Génie de la Programmation et du Logiciel à échéance de 2020
Organisée dans le cadre des journées du GDR CNRS GPL
11 mars 2010
Pau

L'omniprésence de l'informatique dans notre quotidien à l'échelle de l'embarqué et de l'intelligence ambiante, l'extension du web au niveau de la planète, mais également dans les objets du quotidien, le développement de grandes infrastructures de calcul ou des centres de traitement de grandes masses de données soulèvent de nombreuses questions pour le génie de la programmation et du logiciel. Parmi ces questions, quelles sont celles qui correspondent à des défis que devront relever les chercheurs dans le domaine du GPL? Cet appel vise à proposer une vision prospective de notre domaine et à souligner les points qui nous paraissent particulièrement cruciaux.

Ainsi, de nouveaux paradigmes, de nouveaux langages, de nouvelles approches de modélisation, de vérification, de tests et de nouveaux outils dans le domaine de la programmation et du logiciel devraient voir le jour dans les 5 à 10 ans à venir, que ce soit pour faciliter la vie des concepteurs de logiciel, pour modéliser et fiabiliser les logiciels ou encore pour devancer l'évolution technologique. Pour arriver à dégager de nouvelles visions dans notre domaine, des défis doivent être posés. L'objectif de cet appel est d'encourager tous les groupes du GDR GPL, mais aussi les équipes des laboratoires, des groupes informels ou des individus à proposer des défis dans leur domaine. Ces défis pourront ensuite déboucher sur la création de nouveaux réseaux, de création d'ateliers, de proposition de projets, etc.

Idéalement, un défi devra proposer une rupture par rapport à l'usage. Il pourra contenir une description du défi, les fondements scientifiques sur lesquels il repose, les verrous scientifiques et techniques à lever, les usages et impacts sociétaux qu'il adresse, ainsi qu'un ensemble de jalons qui correspondra aux étapes pour le réaliser et le valider et qui pourra servir d'agenda. Des domaines applicatifs pourront accompagner ce défi.

Un défi prendra la forme d'un texte de deux à quatre pages.

Les défis seront soumis sur le site http://gdr-gpl.cnrs.fr. Ils seront sélectionnés par le comité de programme en vue d'être inclus dans les actes des journées du GDR GPL et présentés au cours d'un panel lors des journées du GDR GPL du 8 au 12 mars 2010 à Pau.

Suite à ce panel, il est envisagé de publier ces défis sous la forme d'un livre ou d'un numéro spécial de revue. Des discussions sont en cours avec des éditeurs.

Dates importantes

- 10 janvier 2010 : soumission en ligne (2 à 4 pages)
- 25 janvier 2010 : notification aux auteurs
- 1 février 2010 : réception de la version à inclure dans les actes du GDR GPL
- 11 mars 2010 : panel aux journées du GDR à Pau

- 31 mai 2010 : soumission étendue en vue d'une publication dans une revue/livre (20 pages)
- 15 septembre 2010 : notification aux auteurs
- fin 2010 : publication

Comité de programme (provisoire)

- Laurence Duchien (LIFL/INRIA, Lille) (présidente du comité)
- Yamine Ait Ameur (LISI/ENSMA, Poitiers)
- Jean-Pierre Banâtre (INRIA, Rennes)
- Mireille Blay-Fornarino (I3S, Université de Nice-Sophia Antipolis)
- Pierre Castéran (LABRI, Bordeaux)
- Charles Consel (INRIA/LABRI, Bordeaux)
- Jean-Michel Couvreur (LIFO, Université d'Orléans)
- Catherine Dubois (CEDRIC, Evry)
- Hubert Dubois (CEA LIST, Saclay)
- Jean-Louis Giavitto (IBISC, Evry)
- Gaétan Hains (LACL, Créteil)
- Valérie Issarny (INRIA, Paris-Rocquencourt)
- Olga Kouchnarenko (LIFC, Université Franche-Comté)
- Philippe Lahire (I3S, Nice)
- Yves Ledru (LIG, Grenoble)
- Frédéric Loulergue (LIFO, Université d'Orléans)
- Mourad Oussalah (LINA, Nantes)
- Marie-Laure Potet (Vérimag, Grenoble)
- Salah Sadou (Valoria, Université Bretagne-Sud)
- Christel Seguin (ONERA, Toulouse)
- Mikal Ziane (LIP6, Université Paris Descartes)

Défis et table ronde pour le Génie de la Programmation et du Logiciel à échéance de 2020

L'omniprésence de l'informatique dans notre quotidien à l'échelle de l'embarqué et de l'intelligence ambiante, l'extension du web au niveau de la planète, mais également dans les objets du quotidien, le développement de grandes infrastructures de calcul ou des centres de traitement de grandes masses de données soulèvent de nombreuses questions pour le génie de la programmation et du logiciel. Parmi ces questions, quelles sont celles qui correspondent à des défis que devront relever les chercheurs dans le domaine du génie de la programmation et du logiciel à échéance de 5 à 10 ans ?

De nouveaux paradigmes, de nouveaux langages, de nouvelles approches de modélisation, de vérification, de tests et de nouveaux outils dans le domaine de la programmation et du logiciel devraient voir le jour dans les 5 à 10 ans à venir, que ce soit pour faciliter la vie des concepteurs de logiciel, pour modéliser et fiabiliser les logiciels ou encore pour devancer l'évolution technologique, mais également pour prendre en compte de nouveaux enjeux de société tels que le développement durable et les économies d'énergie.

Les cinq défis seront présentés lors de deux sessions. Ils portent sur la (re)modularisation du logiciel, la mise en place de fermes de composants et de services, l'évaporation des langages, la modélisation pour l'utilisateur final et la réification de l'énergie au niveau des systèmes et des langages. Une table ronde permettra ensuite de débattre de ces sujets. Lors de cette table ronde, Bertrand Braunschweig fera le point sur la perception de notre domaine au sein de l'ANR.

Laurence Duchien

Software (re)modularization: Fight against the structure erosion and migration preparation

N. Anquetil, S. Denier, S. Ducasse, J. Laval, D. Pollet
RMoD INRIA
stephane.ducasse@inria.fr

R. Ducournau, R. Giroudeau, M. Huchard, J.C. König, D. Seriai
LIRMM - CNRS UMR 5506 - Université de Montpellier II - Montpellier (France)
huchard@lirmm.fr

I. CONTEXT

Software systems, and in particular, Object-Oriented systems are models of the real world that manipulate representations of its entities through models of its processes. The real world is not static: new laws are created, concurrents offer new functionalities, users have renewed expectation toward what a computer should offer them, memory constraints are added, etc. As a result, software systems must be continuously updated or face the risk of becoming gradually out-dated and irrelevant [34]. In the meantime, details and multiple abstraction levels result in a high level of complexity, and completely analyzing real software systems is impractical. For example, the Windows operating system consists of more than 60 millions lines of code (500,000 pages printed double-face, about 16 times the Encyclopedia Universalis). Maintaining such large applications is a trade-off between having to change a model that nobody can understand in details and limiting the impact of possible changes. Beyond maintenance, a good structure gives to the software systems good qualities for migration towards modern paradigms as web services or components, and the problem of architecture extraction is very close to the classical remodularization problem.

II. A SIGNIFICANT PROBLEM

Most of the effort while developing and maintaining a software system is spent in supporting its evolution [50]. It is well-known that up to 80% of the total cost of software development project is spent in maintenance and evolution of existing applications [15], [19]. Large IT applications including applications running central and critical business (*e.g.,* command tracking, banking, railway) have to run and evolve over decades.

Such situations are crystallized in the following two laws of Lehman and Belady. These laws, known as the laws of software evolution, stress the fact that software *must* continuously evolve to stay useful and that this evolution is accompanied by an increase of complexity.

> **Continuous Changes.** *"an E-type program—i.e., a software system that solves a problem or implements a computer application in the real world—that is used must be continually adapted else it becomes progressively less satisfactory"* [34]
>
> **Increasing Complexity.** *"As a program is evolved its complexity increases unless work is done to maintain or reduce it."*[34]

From a business perspective, maintenance is mandatory and vital. It is worth to note that the Year 2000 Bug was also revealing some business occasions. For example in France SOPRA which was mainly an SSII, has since the year 2000 developed its TMA (Tierce Maintenance Applicative) branch to get into the maintenance business. SOPRA TMA now represents 2500 developers working on maintenance on a total of 11000 developers.

If many organizations use third party software (e.g. COTS) which they don't have to maintain, this is less true for their core business applications where their market differential must be implemented in the applications that help running day to day activities. As highlighted by the law of Lehman and Belady, these in-house applications must evolve and in this process will extend far beyond their initial structure, in many independent (and sometimes even conflicting) directions. After some time, it becomes mandatory to restructure the application to federate these evolutions inside a more general and renewed architecture to foster possible future evolutions.

Supporting evolution and prepare migration of applications will be *always* mandatory. Different programming paradigms have been invented to cope with changes: late-binding, the cornerstone of object-oriented programming, is a typical illustration. But we think that no paradigm will eradicate the need for evolution and changes and that the only possible approach is to guide evolution or to repair the damages caused by an inevitable erosion.

III. THE FAILURE OF CURRENT SOLUTIONS

Whereas software (re-)modularization is a relatively old research field in the context of C or Cobol, it is still really important and requires *innovative approaches* to deal with the complexity of modern systems especially those

developed in OOP languages. Our analysis — also confirmed in the recent literature (e.g., [1], [7], [47]) — is that this failure is a direct consequence of: (1) the *complexity of the manipulated concepts and the variety of modular abstractions* (subsystems, packages, classes, class hierarchies, late-binding, aspects, various import relationships....) as well as (2) a monolithic approach: the use of *one* type of algorithms (clustering) and *one* kind of system representation (software components interactions found in the source code).

A. Modular Abstractions

Modular constructs have been the focus of a large body of research. Here we give a non-exhaustive list. A lot of work is currently underway in the context of aspect-oriented programming. Module and package systems have been the focus on a large amount of work. The recent work on Units [22], Jiazzi [42], Mixjuice [28], MJ [14], JAM [52], mixin layers [48] shows that this topic is a crucial research area. Bergel *et al.* conducted a survey on modular language constructs that reflects such a diversity [6].

Modular constructs have also been considered at a lower level than classes, *e.g.,* with mixins [9], traits [17]. These constructs denote, here, sets of properties that somewhat represent the *differentia* in the Aristotelian *genus-differentia* definition. Overall, remodularization must address both modular construct levels, by clustering related properties for defining classes and related classes for defining packages. Recursively nested class models have also been proposed [20], [44]—however they cannot be considered as long as the point with non-recursive models is not fixed.

Component-based software approach proposed to build software systems by assembling prefabricated reusable components [54]. Assembly consists in connecting matching interfaces of the components: in a connection, a required interface (describing the services a component needs) is connected to a provided interface (describing the services another component offers). Extracting a component architecture from an object-oriented software has many in common with remodularization because classes need to be grouped based on their dependencies to form the components [11].

Current Issues – *The class notion is the only universally accepted modular abstraction. Higher and lower level abstractions are often still in flux. Different languages use different concepts such as modules, packages, namespaces. Hence, current issues involve both identifying adequate abstractions and adapting remodularization algorithms to the various alternative abstractions. Moreover, assessing the modularity of software requires specific tools (e.g., metrics, visualization) that must be adapted to each modular abstraction.*

B. Remodularization Approaches

Class hierarchy analysis: Class hierarchy analysis has been largely investigated, for restructuring purposes or finding separate concerns (aspects, traits). As it has been shown in [26], most of the refactorization approaches use explicitly or implicitly substructures of those obtained by Formal Concept Analysis (FCA).

The application of clustering algorithms for software remodularization has been intensively studied [2]. Thousands of experiments were conducted to compare different clustering algorithms, different representation schemes and different coupling metrics between files. Although the experiments used procedural systems, many conclusions may be applied to OO systems as well. The extraction of class views based on Formal Concept Analysis has been proposed in [3]. They evaluated how FCA supports the identification of traits in existing hierarchies [8], [35]. Godin [24] developed FCA algorithms to infer a non-redundant form for implementation and interface hierarchies and carried out experiments on several Smalltalk applications. For dealing with UML models, Relational Concept Analysis, an extension of FCA, takes the relations into account [27]. Other approaches analyze class hierarchies using access or usage information. [49], [51] analyze the *usage* of the hierarchy by a set of client programs. Mining aspects has been considered in the context of FCA [10].

Current Issues – *Factorization is by nature a combinatorial process. Recent studies [21] show that Relational Concept Analysis, applied to rich UML descriptions including references between concepts, produces a huge number of artefacts which is quite impossible to analyze by hand. Execution time can become a problem for large size software, but the actual difficulty is the result size.*

Other modular construct discovery: Clustering approaches are, by far, the preferred algorithmic approach to the problem. They have been proposed to identify modules in applications that are not specifically object-oriented (e.g. [29], [30], [38], [43]).

Finding components in object-oriented software is proposed in [12]. Simulated annealing is used to gather classes into components by optimizing metrics measuring cohesion and coupling. The problem has similarity with package mining.

It is a well-known practice to layer applications with bottom layers being more stable that top layers [41]. Until now few works have been done in practice to identify layers: Mudpie [55] is a first cut at identifying cycles between packages as well as package groups potentially representing layers. DSM (dependency structure matrix) [53], [46] are adapted for such a task but there is a lack of detail information. From the side of remodularization algorithms, a lot of them were defined for procedural languages [31]. However object-oriented programming languages bring some specific problems linked with late-binding and the fact that a package does not have to be systematically cohesive since it can be an extension of another one [18], [57].

Current Issues – *These approaches are often not customized for object-oriented applications. Existing solutions propose modules at a very low level of abstraction that do not reduce enough the size of the system comprehension problem. Solutions that may offer larger (potentially more abstract) modules, result in modules that have no meaning for the software engineers.*

C. Techniques for module assessment

Software Metrics: Re-modularization of software systems is geared toward producing highly cohesive and loosely coupled modules. Many different cohesion/coupling metrics were proposed (including a study by [2]). In the more specific case of object-oriented programming, assessing cohesion and coupling has been the focus of several metrics. However their success is rather mitigated as the number of critics raised. For example, LCOM [13] has been highly criticized [4]. Other approaches have been proposed such as RFC and CBO [13] to assess coupling between classes. However, many other metrics have not been the subject of careful analysis such as Data Abstraction Coupling (DAC) and Message Passing Coupling (MPC) [5], or some metrics are not clearly specified (MCX, CCO, CCP, CRE) [36]. New cohesion measures were proposed [40], [45] taking class usage into account. The Cohesion/Coupling dogma, however, started to receive critics in recent times [1], [47]. People argue that software engineers do not base clustering on this criterion but rather use more semantical approaches.

Software Visualization: There is a significant effort to create efficient software visualizations to support the understanding and analyses of applications [32], [37], [39], [56]. Lanza and Ducasse worked on system level understanding combining metrics and visualization [33] and class understanding support [16].

Current Issues – *Existing cohesion and coupling metric resulting values are difficult to map back to the actual situation, they lead to packages that seem artificial and are not understood by experts of the systems. There is a lack for package cohesion and coupling software metrics in presence of late-binding promoted by object-oriented programming. There is a need for program visualization to support the understanding of packages and procedural code. In addition, there is a need for new metrics that would yield more "natural" packages.*

IV. SCIENTIFIC CHALLENGES AND TRACKS OF RESEARCH

The main scientific challenges are as follows.

Abstraction Diversity: One of the major problems to solve when tackling remodularization of object-oriented systems is the choice of good abstractions and the appropriate relations between them.

Complementary Remodularization Algorithms: There is a need for a global remodularization infrastructure in terms of analyses (algorithms, information presentation, metrics) that can take into account the diversity of the abstractions in presence (different module semantics, different abstractions and relationships including different levels, functions, classes, packages, etc.).

Complexity and approximation: In the point of view of graph theory, the central problem seems to be close to the classic k-cuts problem [23], [25]. Nevertheless, we must add several new criteria, like the *quality* of the proposed solution. The first step of the research will be the characterization of an optimization criterion. We also think useful to study and analyse the sensibility of the problem with respect to the operations: add/delete of vertices/edges. It is a major challenge to be able to propose robust approximations.

Scalability: Computational complexity of algorithms has a limit, already known for Formal Concept Analysis (FCA/RCA) and foreseeable for exact methods. Checking the scalability of these algorithms is thus an additional challenge. Another issue is the combinatorial explosion which may occur in the remodularization results. Because of the size of current applications, presenting these results to the engineers and guiding them to take a decision is an additional challenge.

Reengineer inputs and quality of the solution: Engineers should drive the remodularization. Fully automated approaches are applicable only to a very limited context. In reality, external constraints have to be specified and taken into account by the remodularization algorithms (such as the inclusion of a class in a specific package). Software engineers should guide the process possibly confronted to different solutions and their relative impacts. Often favoring minimum impact on existing code has to be considered. Finally the quality of the resulting modularizations has to be taken into account.

As a conclusion: The problems of software evolution are many and varied, in this proposal we plan to consider one of these problems: software remodularization. We think urgent to drive such a complete study of the problem, both "vertically" by studying all the aspects of the modularization problem (modeling of the software, modularization quality metrics, modularization algorithms, presentation of the results), and, "horizontally", by considering different modularization approaches. The solution will not apply one single method, but a combination of various skills in different research domains. Such a research would also be guided by platforms for testing ideas on real-world applications.

REFERENCES

[1] F. B. Abreu and M. Goulao. Coupling and cohesion as modularization drivers: are we being over-persuaded? In *Fifth European Conference on Software Maintenance and Reengineering*, pages 47–57, Mar. 2001.

[2] N. Anquetil and T. Lethbridge. Comparative study of clustering algorithms and abstract representations for software remodularization. *IEE Proceedings - Software*, 150(3):185–201, 2003.

[3] G. Arévalo, S. Ducasse, and O. Nierstrasz. X-Ray views: Understanding the internals of classes. In *Proceedings of 18th Conference on Automated Software Engineering (ASE'03)*, pages 267–270. IEEE Computer Society, Oct. 2003. Short paper.

[4] E. Arisholm, L. C. Briand, and A. Foyen. Dynamic coupling measurement for object-oriented software. *IEEE Transactions on Software Engineering*, 30(8):491–506, 2004.

[5] J. Bansiya, L. Etzkorn, C. Davis, and W. Li. A class cohesion metric for object-oriented designs. *Journal of Object-Oriented Programming*, 11(8):47–52, Jan. 1999.

[6] A. Bergel, S. Ducasse, and O. Nierstrasz. Analyzing module diversity. *Journal of Universal Computer Science*, 11(10):1613–1644, Nov. 2005.

[7] P. Bhatia and Y. Singh. Quantification criteria for optimization of modules in oo design. In *Software Engineering Research and Practice*, pages 972–979, 2006.

[8] A. P. Black, N. Schärli, and S. Ducasse. Applying traits to the Smalltalk collection hierarchy. In *Proceedings of 17th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'03)*, volume 38, pages 47–64, Oct. 2003.

[9] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices*, volume 25, pages 303–311, Oct. 1990.

[10] M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonella, and T. Tourwé. Applying and combining three different aspect mining techniques. *Software Quality Journal*, 14(3):209–231, 2006.

[11] S. Chardigny, A. Seriai, M. Oussalah, and D. Tamzalit. Extraction of component-based architecture from object-oriented systems. In *WICSA*, pages 285–288. IEEE Computer Society, 2008.

[12] S. Chardigny, A. Seriai, M. C. Oussalah, and D. Tamzalit. Extraction d'Architecture à Base de Composants d'un Système Orienté Objet. In *INFORSID*, pages 487–502, 2007.

[13] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[14] J. Corwin, D. F. Bacon, D. Grove, and C. Murthy. MJ: a rational module system for Java and its applications. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 241–254. ACM Press, 2003.

[15] A. M. Davis. *201 Principles of Software Development*. McGraw-Hill, 1995.

[16] S. Ducasse and M. Lanza. The Class Blueprint: Visually supporting the understanding of classes. *Transactions on Software Engineering (TSE)*, 31(1):75–90, Jan. 2005.

[17] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. P. Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):331–388, Mar. 2006.

[18] A. Dunsmore, M. Roper, and M. Wood. Object-oriented inspection in the face of delocalisation. In *Proceedings of ICSE '00 (22nd International Conference on Software Engineering)*, pages 467–476. ACM Press, 2000.

[19] L. Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000.

[20] E. Ernst. Higher-order hierarchies. In *Proceedings European Conference on Object-Oriented Programming (ECOOP 2003)*, LNCS, pages 303–329, Heidelberg, July 2003. Springer Verlag.

[21] J.-R. Falleri, M. Huchard, and C. Nebut. A generic approach for class model normalization (short paper). In *ASE 2008: 23th IEEE/ACM International Conference on Automated Software Engineering*, 2008.

[22] M. Flatt and M. Felleisen. Units: Cool modules for hot languages. In *Proceedings of PLDI '98 Conference on Programming Language Design and Implementation*, pages 236–248. ACM Press, 1998.

[23] A. Freize and M. Jerrum. Improved approximation algorithm for $\max -k$-cut and $\max$ bisection. *Algorithmica*, 18:67–81, 1997.

[24] R. Godin, H. Mili, G. W. Mineau, R. Missaoui, A. Arfi, and T.-T. Chau. Design of Class Hierarchies based on Concept (Galois) Lattices. *Theory and Application of Object Systems*, 4(2):117–134, 1998.

[25] O. Goldschmidt and D. Hochbaum. Polynomial algorithm for the k-cut problem. In I. C. Society, editor, *Proc. 29th Annual IEEE Symposium on Foundations of Computer Science*, pages 444–451, 1988.

[26] M. Huchard, H. Dicky, and H. Leblanc. Galois lattice as a framework to specify building class hierarchies algorithms. *ITA*, 34(6):521–548, 2000.

[27] M. Huchard, M. R. Hacene, C. Roume, and P. Valtchev. Relational concept discovery in structured datasets. *Ann. Math. Artif. Intell.*, 49(1-4):39–76, 2007.

[28] Y. Ichisugi and A. Tanaka. Difference-based modules: A class independent module mechanism. In *Proceedings ECOOP 2002*, volume 2374 of *LNCS*, Malaga, Spain, June 2002. Springer Verlag.

[29] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice Hall, Englewood Cliffs, 1988.

[30] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[31] R. Koschke. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. PhD thesis, Universität Stuttgart, 2000.

[32] G. Langelier, H. A. Sahraoui, and P. Poulin. Visualization-based analysis of quality for large-scale software systems. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 214–223, New York, NY, USA, 2005. ACM.

[33] M. Lanza and S. Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)*, 29(9):782–795, Sept. 2003.

[34] M. Lehman. Laws of software evolution revisited. In *European Workshop on Software Process Technology*, pages 108–124, Berlin, 1996. Springer.

[35] A. Lienhard, S. Ducasse, and G. Arévalo. Identifying traits with formal concept analysis. In *Proceedings of 20th Conference on Automated Software Engineering (ASE'05)*, pages 66–75. IEEE Computer Society, Nov. 2005.

[36] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics: A Practical Guide*. Prentice-Hall, 1994.

[37] J. I. Maletic and A. Marcus. Supporting program comprehension using semantic and structural information. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001)*, pages 103–112, May 2001.

[38] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner. Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures. In *Proceedings of ICSM '99 (International Conference on Software Maintenance)*, Oxford, England, 1999. IEEE Computer Society Press.

[39] A. Marcus, L. Feng, and J. I. Maletic. 3D representations for software visualization. In *Proceedings of the ACM Symposium on Software Visualization*, pages 27–ff. IEEE, 2003.

[40] A. Marcus and D. Poshyvanyk. The conceptual cohesion of classes. In *Proceedings International Conference on Software Maintenance (ICSM 2005)*, pages 133–142, Los Alamitos CA, 2005. IEEE Computer Society Press.

[41] R. C. Martin. *Agile Software Development. Principles, Patterns, and Practices*. Prentice-Hall, 2002.

[42] S. McDirmid, M. Flatt, and W. Hsieh. Jiazzi: New age components for old fashioned Java. In *Proceedings OOPSLA 2001, ACM SIGPLAN Notices*, pages 211–222, Oct. 2001.

[43] B. S. Mitchell and S. Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, 32(3):193–208, 2006.

[44] N. Nystrom, X. Qi, and A. C. Myers. J&: nested intersection for scalable software composition. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 21–36, New York, NY, USA, 2006. ACM.

[45] L. Ponisio and O. Nierstrasz. Using context information to re-architect a system. In *Proceedings of the 3rd Software Measurement European Forum 2006 (SMEF'06)*, pages 91–103, 2006.

[46] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using dependency models to manage complex software architecture. In *Proceedings of OOPSLA'05*, pages 167–176, 2005.

[47] R. Sindhgatta and K. Pooloth. Identifying software decompositions by applying transaction clustering on source code. In *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference*, pages 317–326, Washington, DC, USA, 2007. IEEE Computer Society.

[48] Y. Smaragdakis and D. Batory. Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Soft. Eng. Meth.*, 11(2):215–255, 2002.

[49] G. Snelting and F. Tip. Reengineering Class Hierarchies using Concept Analysis. In *ACM Trans. Programming Languages and Systems*, 1998.

[50] I. Sommerville. *Software Engineering*. Addison Wesley, fifth edition, 1996.

[51] M. Streckenbach and G. Snelting. Refactoring class hierarchies with KABA. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 315–330, New York, NY, USA, 2004. ACM Press.

[52] R. Strniša, P. Sewell, and M. Parkinson. The java module system: core design and semantic definition. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 499–514, New York, NY, USA, 2007. ACM.

[53] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen. The structure and value of modularity in software design. In *ESEC/FSE 2001*, 2001.

[54] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[55] D. Vainsencher. Mudpie: layers in the ball of mud. *Computer Languages, Systems & Structures*, 30(1-2):5–19, 2004.

[56] R. Wettel and M. Lanza. Program comprehension through software habitability. In *Proceedings of ICPC 2007 (15th International Conference on Program Comprehension)*, pages 231–240. IEEE CS Press, 2007.

[57] N. Wilde and R. Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, SE-18(12):1038–1044, Dec. 1992.

Component and Service Farms

Gabriela Arévalo
LIFIA - Facultad de Informática (UNLP) - La Plata (Argentina)
garevalo@sol.info.unlp.edu.ar

Zeina Azmeh, Marianne Huchard, Chouki Tibermacine
LIRMM - CNRS UMR 5506 - Université de Montpellier II - Montpellier (France)
{*azmeh, huchard, tibermacin*}*@lirmm.fr*

Christelle Urtado, Sylvain Vauttier
LGI2P - Ecole des Mines d'Alès - Nîmes (France)
{*Christelle.Urtado, Sylvain.Vauttier*}*@mines-ales.fr*

I. CONTEXT

Software components and web services are software building blocks that are used in the composition of modern software applications. They both provide functionalities that require to be advertised by registries in order to be discovered and reused during software building processes[1] [1], [2], [3], [4], [5]. Building or evolving existing software implies assembling software components. This task is not trivial because it requires to select the adequate component or service that provides some part of the desired application functionality and connects easily (with minimum adaptations) to other selected components.

Within this context, we identified two main issues: (1) finding appropriate components from huge databases, and (2) creating and maintaining distributed applications.

Finding appropriate components from huge databases: A huge number of components already exist. For example, the Seekda[2] web service search engine has a catalog of more than 28.000 references, and the OW2 consortium[3] groups around 40 open-source projects in the domain of middleware technology. In Seekda, functionalities are rather directly exposed, but in projects, components are sometimes buried into application code repositories and should be properly extracted before being publicly advertised. However, in both cases the task of finding and adapting an appropriate component is hard and time-consuming, and this will surely worsen when more components become available.

Creating and maintaining distributed applications: There is a growing need for being able to create and maintain distributed applications assembled from third party components as network, middleware and deployment technologies now are mature enough. Reduced cost, increased quality and decreased dependence to a given provider also contribute to popularize this trend. Paradigms that are based on this principle are numerous: component-based development, web 2.0, mashups, cloud computing, software as a service, etc. State-of-the-practice technologies such as OSGi[4], that enables the deployment and redeployment of software (for example, in remotely administered internet boxes), or upcoming technologies such as Google Chrome OS[5], that aims to put web-apps at the center of net-books' operating system, also illustrate this trend.

II. INNOVATIVE REGISTRIES

In this context, it is necessary to design innovative software component registries to advertise components and assist users when they need to search, select, adapt and connect a component to others [6], [7]. We even could think of the automation of these tasks as much as possible, in order to adapt to open and dynamic contexts (remotely administered embedded devices, pervasive computing, open and extensible applications, mobile computing, etc.).

Then, the challenge consists in proposing an online architecture for a component service registry, with two functionalities (a) gives efficient access to adequate components, and (b) provides life-cycle long support to developers and applications (in automatic mode). This registry will be a platform for developers to share their knowledge and experiences on the components they use (what runs and what does not, what are ideal contexts for running some given component, what adaptations have already been performed or tested on some component, what are user ratings on components, etc.), and improve development efficiency as well as running software reliability or liveliness.

As an extension of the component search engine, we think of a *component farm*, where components could be either physically located (component repository model) or solely referenced in an adequately organized component

[1] In the following, we will often use the term *components* as a generic name for both web services and software components.

[2] http://webservices.seekda.com/

[3] http://www.ow2.org/

[4] http://www.osgi.org

[5] http://googleblog.blogspot.com/2009/07/introducing-google-chrome-os.html

advertisement directory). The component farm (*cf.* Fig. 1) should offer multiple views on components and tailored efficient retrieval mechanisms.

Developed software applications that use components from the farm could either simply use the farm's public facilities or register to benefit from increased community-related capabilities. In the latter case, the added value would be provided by the exchange of component-related data between registered software applications and the farm, in both directions:

- *from software applications to the farm.* Applications could register components developed for their own purposes into the farm directory or share usage information on components they have previously retrieved from the farm, etc. This means that developers should contribute as in collaborative web.
- *from the farm to software applications.* The farm should provide several operations supported by efficient tools for searching and selecting components, providing usage feedback on components, informing the developer of other developers' experiences regarding new products, found problems with components, problem solving issues, possible adaptations, etc.

Our proposal focuses on two challenging issues, each of which is developped in one of the followig sections: (1) the adequacy and efficiency of the organization of the component directory, (2) the proposal of tailored support to applications.

III. Efficient organization

Many components exist but are not always available in public servers/directories. Providing abstract views and adequate services, a global organization could help to stimulate their sharing and be profitable increasing cross-fertilization between projects. The first aim of the registry is to efficiently organize components for several development and maintenance tasks. The components can be physically contained in the repository or just accessible from the directory, through hypertext links, from other external locations. An efficient organization is based on an efficient classification of the components. For example, some approaches [8], [9], [10], [11], [12], [13] have already investigated formal concept analysis (FCA) [14], [15], an approach that rigorously classifies data in structures that have strong mathematical properties. Other approaches should nevertheless be studied.

Several aspects can be used to classify components: syntactical, semantic and pragmatic [16], [17]. All of them could be used complementarily. At the syntactical level, component external descriptions include ports, interfaces, functionality signatures and parameter types. At the semantic level, components can be documented (with plain or structured text, with interaction protocols, etc.), described by keywords (either normalized through ontologies, for example, or not), and by any information that conveys the meaning of the component (for what purpose it has been developed, by whom, etc.). At the pragmatic level, components are documented by information that provides feedback from its usage, in assemblies, choreographies or orchestrations: which functionalities are used, which functionalities are never used, how components are connected (with which adaptations), good and bad experiences, etc.

IV. Application support

The second aim of the registry is software application support. Applications using the components from the registry are invited to register so as the directory can receive feedback. Thus, the directory stores information on which application uses which components and documentation about this use, in order to share usage information among developers.

The registry memorizes the manual or automatic adaptations that are necessary when a faulty component is replaced by another. This information is used to optimize future replacements and is capitalized to be used by other registered applications.

The registry prepares backups (sets of possible substitutes) for each used component in order to ensure rapid repairing of a software application in case one of its components fails [18]. These backups are organized in small classifications for efficiency purposes. They can be used manually or sometimes automatically in restricted cases where either exact matching between the used component and its potential substitute exists, or automatic adaptations are known and can be applied. These backups can be stored inside the repository or uploaded on demand by applications. Backups can also be used by designers to make the software evolve when the application designer wants to add extra functionalities or improve the software quality attributes.

Candidate backup components and possible adaptations are dynamically updated, as components become unavailable or are upgraded. Software applications that use them are also dynamically informed of these updates.

V. Starting point

In previous work, we have started to imagine partial solutions to the issues identified here.

- We have developed an automated process for classifying components from their external descriptions. This process is based on type-theory (we only use syntactic information) and uses FCA to iteratively build lattices that provide functionality signature classifications, interface classifications and component classifications [8], [9], [10].
- We have prototyped the CoCoLa tool [10] that implements the aforementioned process. Thanks to a pivot meta-model, component descriptions from various formats are translated into comparable models (instances of the common meta-model). These descriptions are then processed to build context tables and lattices.

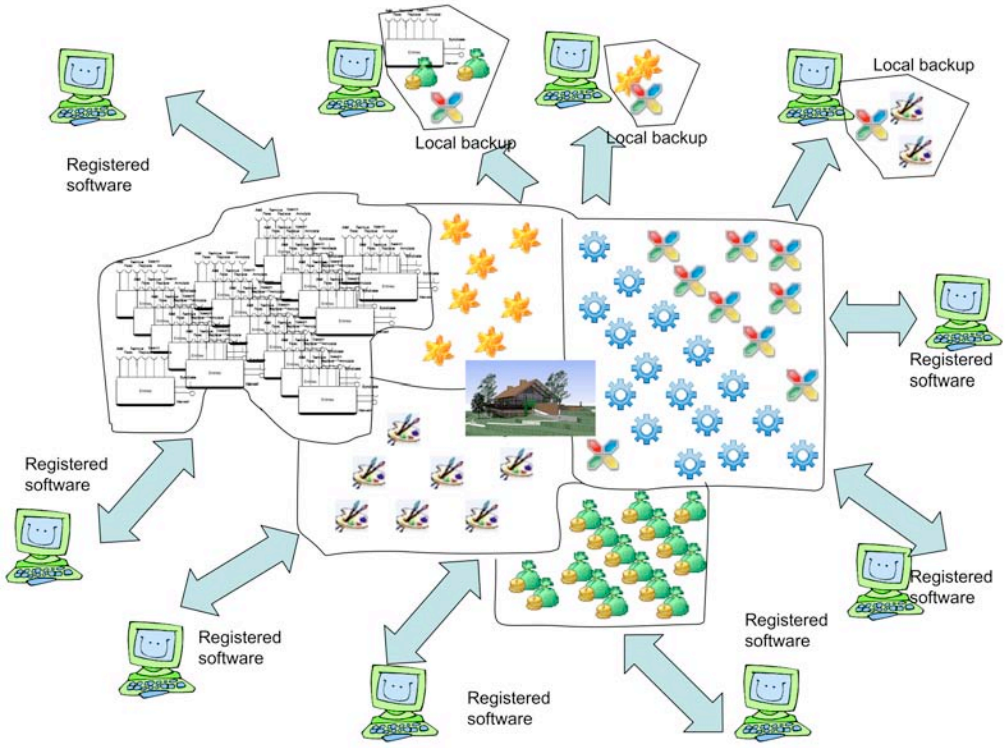

Figure 1. Overview of the component farm

Experiments have been run on the Dream[6] component library (that comes from a real-world component-based framework). They show the feasibility of our approach as the produced lattices enables us to identify possible component substitutions and provides a readable component classification.

- We also have proposed an approach based on formal concept analysis (FCA) for classifying web services [19], [18]. A web service lattice reveals the invisible relations between web services in a certain domain, showing the services that are able to replace other ones. This facilitates service browsing, selection and identification of possible substitutions. We explained how to exploit the resulting lattices to build web services orchestrations and support them with backup services.

VI. MAIN CHALLENGES

There still are numerous challenges to overcome in order to build such a component farm:

- combine syntactical, semantic and pragmatic views of components in order to be able to efficiently search and select appropriated components (pertinence of the classification).
- automate component indexing and classification as well as feedback collection in order not to put the burden on component developers or component users. This task could need to exploit automatically all the available semantic information available (such as functionality names, documentations, interaction protocols, etc.) using text mining techniques [20].
- try and capitalize coarser grained software parts by classifying whole component assemblies,
- think about building techniques and browsing techniques for each of the provided classifications but also inter-classifications.
- propose replacement scenarios and tools,
- define the services that would motivate users in collaboratively providing usage feedback information,
- find means to conduct early experimentations of the component farm. This is not easy as very few component directories are available online, as for now.

As a response to the globalization challenge, software engineering has the capability to positively react in providing software component catalogs, thus increasing software quality while decreasing costs and time-to-market. As an immaterial product market-place, the software component industry benefits from great assets: software components must not be stocked, they can be transported at no cost and instantaneously, they are not perishable. To take advantage of this, efforts should be focused not only on development,

[6]http://dream.ow2.org/

as traditionally done, but also on better diffusion, sharing, deployment and maintenance. We believe that component farms could contribute to this objective.

REFERENCES

[1] W. Hoschek, "The web service discovery architecture," in *Int'l. IEEE/ACM Supercomputing Conference (SC 2002)*, I. C. S. Press, Ed., 2002.

[2] OMG, "Trading Object Service Specification (TOSS) v1.0," 2000, http://www.omg.org/cgi-bin/doc?formal/2000-06-27.

[3] *ebXML Registry Services Specification (RS) v3.0*, http://www.oasis-open.org/, May 2005.

[4] L. Clement, A. Hately, C. von Riegen, and T. Rogers, "Uddi version 3.0.2. uddi spec technical committee draft , dated 20041019. http://uddi.org/pubs/uddi_v3.htm," Tech. Rep. [Online]. Available: http://uddi.org/pubs/uddiv3.htm

[5] Information Technology Open Distributed Processing, "ODP Trading Function Specification ISO/IEC 13235-1:1998(E)," December 1998, http://webstore.iec.ch/preview/info_isoiec13235-1%7Bed1.0%7Den.pdf.

[6] L. Iribarne, J. M. Troya, and A. Vallecillo, "A trading service for COTS components," *The Computer Journal*, vol. 47, no. 3, pp. 342–357, 2004.

[7] S. Dustdar and M. Treiber, "A view based analysis on web service registries," *Distributed and Parallel Databases*, vol. 18, pp. 147–171, 2005.

[8] G. Arévalo, N. Desnos, M. Huchard, C. Urtado, and S. Vauttier, "Precalculating component interface compatibility using FCA," in *Proceedings of the 5^{th} international conference on Concept Lattices and their Applications (CLA 2007), CEUR Workshop Proceedings Vol. 331*, J. Diatta, P. Eklund, and M. Liquière, Eds., Montpellier, France, October 2007, pp. 241–252. [Online]. Available: http://ceur-ws.org/Vol-331/

[9] ——, "FCA-based service classification to dynamically build efficient software component directories," *Int. Journ. of General Systems*, vol. 38, no. 4, pp. 427–453, May 2009.

[10] N. A. Aboud, G. Arévalo, J.-R. Falleri, M. Huchard, C. Tibermacine, C. Urtado, and S. Vauttier, "Automated architectural component classification using concept lattices," in *In proceedings of the Joint Working IEEE/IFIP Conference on Software Architecture 2009 (WICSA/ECSA'09)*. Cambridge, UK: IEEE Computer Society Press, September 2009.

[11] A. M. Zaremski and J. M. Wing, "Specification matching of software components," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 6, no. 4, pp. 333–369, 1997.

[12] B. Fischer, "Specification-based browsing of software component libraries," in *Proc. of the 13^{th} IEEE int. conf. on Automated Software Engineering (ASE'98)*, 1998, pp. 74–83.

[13] B. Sigonneau and O. Ridoux, "Indexation multiple et automatisée de composants logiciels orientés objet," in *AFADL — Approches Formelles dans l'Assistance au Développement de Logiciels*, J. Julliand, Ed. Besançon, France: RTSI, Lavoisier, Juin 2004.

[14] M. Barbut and B. Monjardet, *Ordre et Classification*. Hachette, 1970.

[15] B. Ganter and R. Wille, *Formal Concept Analysis: Mathematical Foundations*. Springer, 1999.

[16] M. Á. Corella and P. Castells, "Semi-automatic semantic-based web service classification," in *Business Process Management Workshops*, ser. LNCS 4103, J. Eder and S. Dustdar, Eds. Springer, 2006, pp. 459–470.

[17] M. Bruno, G. Canfora, M. D. Penta, and R. Scognamiglio, "An approach to support web service classification and annotation," in *Proc. of the IEEE Int. Conf. on e-Technology, e-Commerce and e-Service (EEE'05)*, 2005, pp. 138–143.

[18] Z. Azmeh, M. Huchard, C. Tibermacine, C. Urtado, and S. Vauttier, "Using concept lattices to support web service compositions with backup services," in *To appear in Proceedings of the 5^{th} International Conference on Internet and Web Applications and Services (ICIW 2010)*, Barcelona, Spain, May 2010.

[19] ——, "WSPAB: A tool for automatic classification & selection of web services using formal concept analysis," in *Proceedings of the 6^{th} IEEE European Conference on Web Services (ECOWS 2008)*. Dublin, Ireland: IEEE, November 2008, pp. 31–40.

[20] J.-R. Falleri, Z. Azmeh, M. Huchard, and C. Tibermacine, "Automatic tag identification in web service descriptions," in *To appear in proceedings of the International Conference on Web Information Systems and Technology (WEBIST'10)*, Valencia, Spain, April 2010.

End-User Modelling

Albert Patrick *(IBM, Paris)*, Blay-Fornarino Mireille *(I3S, Sophia-Antipolis)*, Collet Philippe *(I3S, Sophia-Antipolis)*, Combemale Benoit *(IRISA, Rennes)*, Dupuy-Chessa Sophie *(LIG, Grenoble)*, Front Agnès *(LIG, Grenoble)*, Grost Anthony *(ATOS, Lille)*, Lahire Philippe *(I3S, Sophia-Antipolis)*, Le Pallec Xavier *(LIFL, Lille)*, Ledrich Lionel *(ALTEN Nord, Lille)*, Nodenot Thierry *(LIUPPA, Pau)* et Pinna-Dery Anne-Marie *(I3S, Sophia-Antipolis)* et Rusinek Stéphane *(Psitec, Lille)*

1 Contexte et domaines applicatifs

En 2006, le *Standish Group* indiquait que l'amélioration du taux de réussite des projets informatiques (passant de 16 à 32%) provenait de différents facteurs, dont le plus important était l'implication grandissante des utilisateurs finaux. Le mouvement syndicaliste scandinave (début 70) nommé “conception participative” avait déjà influencé l'ingénierie logicielle dans les années 80 dans ce sens: l'adhésion des employés pour un nouvel outil peut s'obtenir par leur participation à sa conception. Ce sont les démarches agiles, officialisées en 2001, qui ont popularisé ce principe d'implication des utilisateurs dans la conception logicielle. Parallèlement, l'augmentation de la production logicielle et de la taille des applications informatiques a accentué la préoccupation de réutilisation logicielle. Une réponse à celle-ci a été d'accorder plus d'importance à la modélisation dans la conception logicielle : un modèle fournit un meilleur support pour la discussion, la compréhension d'une architecture ou d'un composant et demeure moins impacté par les évolutions technologiques.

L’utilisation de plus en plus intensive des « wiki » et la mise à disposition d’outils google témoignent de l’intérêt de donner aux utilisateurs les moyens de « programmer » leurs propres outils. Cette thématique se retrouve au sein d’éditeurs d'applications *mash up* comme Yahoo Pipes, MS Pop Fly. Ces éditeurs, très en vogue actuellement, sont souvent basés sur des éditeurs de modèles graphiques très accessibles et en principe au moins lisible par un grand nombre d'utilisateurs “avertis”. De manière plus industrielle, la construction d’usines logicielles et les environnements de modélisation dédiés à des domaines spécifiques entrent également dans ce cadre, en donnant à l’utilisateur des langages de modélisation adaptés à son métier. Le défi proposé ici est donc : *comment donner aux experts d’un domaine les moyens de construire leur propre système informatique en utilisant des techniques de modélisation ?*

Les enjeux sont une plus grande productivité et une meilleure adéquation des produits aux problèmes. En particulier en permettant aux utilisateurs d'adapter le logiciel, une plus grande réactivité/agilité est obtenue. Ces enjeux ne seront atteints que si les artefacts de modélisation donnés sont adéquats et si les produits résultants sont fiables et opérationnels.

Les risques sont la définition de méga-métamodèles non exploitables, l’obtention de produits inutiles, non performants, non réutilisables.

Nous proposons d’aborder ce défi par la construction d’environnements de modélisation dédiés à des domaines spécifiques. Des systèmes tels que CENTAUR [1] dans les années 80-90 avaient des objectifs similaires mais étaient orientés langages de programmation. Le défi que nous proposons en s’appuyant sur ces acquis envisage une appréhension de la “programmation” dirigée non pas par la syntaxe mais par les concepts, les usages et les contraintes du métier. Le défi est de faciliter la construction de ces environnements en utilisant des techniques de modélisation, en particulier la modélisation sous la forme de diagrammes.

2 Verrous

Les verrous identifiés se situent au niveau (i) des représentations à donner aux multiples profiles d'utilisateurs qui doivent être en adéquation avec leur usage, (ii) de l’interprétation de ces notations par le système pour assurer l’aide, la validation et la collaboration et l’exécution éventuelle des systèmes produits, (iii) de l’impact sur les démarches usuelles de développement.

1. Représentation métier :
   - comment éviter la surcharge cognitive au niveau des syntaxes concrètes dédiées aux utilisateurs ?
   - comment supporter la montée en compétences (connaissances du système) d'un utilisateur ?
   - comment supporter les multi-représentations lors de travaux collaboratifs ?
2. Interprétation des modélisations métier :
   - comment supporter plusieurs représentations d'un même système (multi-utilisateurs, multi-niveaux) ?
   - comment rendre opérationnelles ces représentations en limitant les coûts de développements et en assurant la cohérence des systèmes construits ? (intégration de l'existant, incrément, agilité des développements ?
   - comment permettre à l'utilisateur final d'interagir avec le système de manière à en adapter le comportement (e.g., systèmes adaptables) ?
   - Un des verrous à lever est le juste appariement entre des approches dîtes "formelles" indispensables à ancrer le procédé dans une logique de fiabilité, et des approches de modélisation plus flexibles qui supportent une expression plus "naturelle" des domaines métiers.
3. Démarches de développement :
   - comment supporter l'agilité de la construction, la robustesse des environnements construits tout en acceptant les incohérences nécessaires au développement collaboratif, …

3 Fondements

L'IDM apporte un ensemble de fondements permettant d'aborder la problématique soulevée par ce défi [2]. En effet, la montée en abstraction supportée par l'IDM pour la construction de systèmes complexes réduit l'écart entre la spécification et la conception/développement d'un système. En premier lieu, la prise en compte des différents profils d'utilisateurs finaux nécessitent de définir des représentations adéquates pour chacun d'entre eux. Cette adéquation nécessite une étude des propriétés cognitives des formalismes utilisés. Pour cela, nous pourrons nous baser sur les synthèses de Bernard Morand [16] sur les propriétés cognitives des diagrammes (du point de vue de la psychologie cognitive, de la linguistique…) comme sur les travaux de Gerald Lohse [17] qui sont plus spécifiques aux diagrammes représentant des systèmes informatiques. D'autres travaux ont porté sur la compréhensibilité globale d'une notation et ont permis d'aboutir à des protocoles expérimentaux réutilisables et utilisées dans le monde industriel [18]. Dans tous ces cas, l'approche utilisée est celle des expériences utilisateurs.

Mais une autre approche, plus automatisable, est possible pour évaluer la qualité d'un modèle ou d'un langage : elle se base sur l'utilisation de métriques qui peuvent être définies sur les modèles [19] ou les métamodèles [20]. Il existent donc de nombreuses possibilités pour appréhender ce qui a été définie dans les frameworks de qualité [21][22] comme la qualité pragmatique, c'est-à-dire la qualité dépendant de l'interprétation des utilisateurs et des concepteurs.

La définition d'un langage dédié (*Domain Specific Language*, ou DSL), et plus récemment de langage de modélisation dédié (*Domain Specific Modeling Language*, ou DSML) a été étudié au travers de techniques telles que les profils UML (consistant à spécialiser UML pour des besoins particuliers), ou la métamodélisation offrant toute l'ingénierie pour la définition d'un nouveau langage [9]. L'outillage de celui-ci peut par ailleurs être facilité à l'aide d'approches génératives ou génériques, telle que par exemple GMF[1] qui permet de générer automatiquement des outils de modélisation métier graphique pour un langage dédié donné. Le traitement automatique, à base de modèles, de langue naturelle ou de règles métiers est également un axe de recherche intéressant pour la résolution de ce défi. La publication récente par l'OMG de la norme *Semantics for Business Vocabulary and Rules* (SBVR)[2] permet par exemple d'interpréter formellement sous la forme de modèle un anglais structuré [23]

Les travaux sur les approches par points de vues devront également être pris en considération de manière à bien préciser l'intention de chaque type d'utilisateurs [15], de même que les travaux sur les usines logicielles [8]. D'autres travaux sur l'extension des métamodèles, tant syntaxique que comportementale (e.g., la modélisation orientée aspect [3,4]) offre également des moyens d'adapter ou de préciser un langage existant pour des besoins spécifiques.

[1] Cf. http://www.eclipse.org/gmf/

[2] *Semantics of Business Vocabulary and Business Rules* 1.0 specification (2008), http://www.omg.org/spec/SBVR/1.0/

D'autre part, des techniques de composition [5,6,10] et de transformation de modèles [7] permettent de prendre en compte les modèles issus de différentes préoccupations de manière à obtenir un modèle unique du système complet. Ces techniques permettent ainsi de rendre productif les modèles métiers et de les considérer comme des artefacts terminaux du processus de développement. Le couplage transparent des représentations métiers avec les méthodes formelles [13] devra également être pris en considération afin d'assurer la fiabilité des systèmes construits.
L'utilisation de ces techniques doit toutefois s'intégrer dans une démarche agile favorisant un cycle court de manière à permettre à l'utilisateur de voir très rapidement le résultat de son travail. Des adaptations dynamiques de l'environnement doivent également être envisagées [24,25]. La nécessité d'intégrer un processus itératif est donc indispensable et doit s'appuyer pour cela sur la traçabilité entre les modèles métiers et le système final obtenu. Cette traçabilité peut par exemple être utilisée pour indiquer les impacts sur le système final de chaque modification d'un modèle métier.

Enfin, l'assistance à la modélisation peut s'appuyer d'une part sur les modèles paramétrés pour permettre une modélisation par la réutilisation de briques génériques [14], et d'autre part sur le typage de modèle [11] pour permettre la généricité de certain traitement [12].

4 Usages et impacts sociétaux

- La programmation des applications est donnée aux experts des domaines, la construction des environnements est donnée aux experts logiciels. Mais les environnements eux-mêmes doivent pouvoir évoluer par des cycles courts.
- Les services informatiques représentent une activité économique importante (> 700Md $ en 2008). C'est pourtant un secteur dont la majorité des projets sont des échecs. L'implication des utilisateurs est le facteur de réussite préconisé mais sa mise en œuvre reste difficile. Un objectif majeur du défi est de diminuer voir supprimer cette difficulté.

5 Jalons, démarche

- Circonscrire la complexité par des études de cas :
  - Expérimentations sur des classes d'applications pour lesquelles les langages/formalismes sont maîtrisables par les "personnes métier" (ex. : Business Rules, modélisation de workflows en analyse d'images médicales, modélisation de systèmes de diffusions d'informations en milieu scolaire, domotique),
- Définition des artefacts de modélisation (formalismes) cognitivement adaptés aux différents rôles (ou personnes) impliquées
  - Processus de construction attendu, validations, tests : les artefacts se définissent relativement à leur usage.
  - Les experts d'un domaine peuvent adapter ou faire évoluer l'application uniquement sur la partie métier.
- Corrélation entre modélisation métier et mise en œuvre au niveau des plates-formes :
  - Dans un premier temps, des experts en programmation définissent les méthodologies, processus, outils de tests et de validation qui permettent de construire, modifier, valider des "modèles métiers" et d'assurer la cohérence des systèmes construits : "*Controlled Agility*".
  - Dans un second temps, des patrons (transformations, mécanismes de profiling) sont utilisés pour, sur la base de la définition de métamodèles métiers, obtenir les outils associés (validation, tests, …). Partiellement atteint dans le cadre des langages de programmation, cet objectif devrait bénéficier des avancées en matière de métamodélisation.
  - En obtenant une certaine automatisation de la production des outils, les utilisateurs finaux devraient pouvoir faire évoluer leur propre environnement de modélisation dans la limite des contraintes nécessaires pour assurer la cohérence des environnements et des systèmes construits.

References

1. P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual, *Centaur: the system* , Proceedings of SIGSOFT'88, Third Annual Symposium on Software Development Environments (SDE3), Boston, USA, 1988.
2. Jean-Marie Favre, Jacky Establier, and Mireille Blay-Fornarino, editors. L'ingénierie dirigée par les modèles : au-delà du MDA. Hermes-Lavoisier, Cachan, France, February 2006.
3. Jacques Klein, Franck Fleurey, and Jean Marc Jézéquel. Weaving multiple aspects in sequence diagrams. Transactions on Aspect-Oriented Software Development (TAOSD), LNCS 4620 :167–199, 2007.
4. Jean-Marc Jézéquel. – Model driven design and aspect weaving. – Journal of Software and Systems Modeling (SoSyM), 7(2):209–218, may 2008.
5. Robert France, Franck Fleurey, Raghu Reddy, Benoit Baudry, and Sudipto Ghosh. Providing support for model composition in metamodels. In EDOC'07 (Entreprise Distributed Object Computing Conference), Annapolis, MD, USA, 2007.
6. Olivier Barais, Jacques Klein, Benoit Baudry, Andrew Jackson, Siobhan Clarke, *Composing Multi-View Aspect Models*, 7th IEEE International Conference on Composition-Based Software Systems (ICCBSS) (2008)
7. Jean Bézivin, Salim Bouzitouna, Marcos Didonet Del Fabro, Marie-Pierre Gervais, Frédéric Jouault, Dimitrios S. Kolovos, Ivan Kurtev, and Richard F. Paige. A canonical scheme for model composition. In Arend Rensink and Jos Warmer, editors, ECMDA-FA, volume 4066 of Lecture Notes in Computer Science, pages 346–360. Springer, 2006.
8. Carlos Parra , Rafael Leano , Xavier Blanc, Laurence Duchien, Nicolas Pessemier, Chantal Taconet and Zakia Kaziaoul, *Dynamic Software Product Lines for Context-Aware Web Services*., in Enabling Context-Aware Web Services: Methods, Architectures, and Technologies. Chapman and all, July 2009
9. F. Barbier, *UML 2 et MDE - Ingénierie des modèles avec études de cas*, Dunod, 2005
10. Mireille Blay-Fornarino, *"Interprétations de la composition d'activités"*, HDR Thesis University of Nice-Sophia Antipolis, 1-201 pages, Sophia Antipolis, France, apr 2009
11. Jim Steel and Jean-Marc Jézéquel. – On model typing. – Journal of Software and Systems Modeling (SoSyM), 6(4):401–414, December 2007.
12. Naouel Moha, Vincent Mahé, Olivier Barais, and Jean-Marc Jézéquel. – Generic Model Refactorings. – In ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems (MODELS'09), Denver, Colorado, USA, Oct 2009.
13. B. Combemale, X. Crégut, P.-L. Garoche and X. Thirioux – Essay on Semantics Definition in MDE. An Instrumented Approach for Model Verification – Journal of Software (JSW), 4(6), December 2009
14. Alexis Muller, Olivier Caron, Bernard Carré, Gilles Vanwormhoudt, *On Some Properties of Parameterized Model Application* in Proceedings of ECMDA-FA'2005: First European Conference on Model Driven Architecture - Foundations and Applications, Nuremberg, November 2005, LNCS 3748, A. Hartman and D. Kreische Eds.
15. Adil Anwar, Sophie Ebersold, Bernard Coulette, Mahmoud Nassar, Abdelaziz Kriouile. A Rule-Driven Approach for composing Viewpoint-oriented Models. Dans : Journal of Object Technology, ETH Swiss Federal Institute of Technology, p. 1-26, 2010
16. B. Morand, Logique de la Conception, Figures de sémiotique générale d'après Charles S. Peirce, L'Harmattan, Collection L'Ouverture Philosophique, 294 p.
17. G. L. Lohse, D. Minb and J. R. Olsonc, Cognitive evaluation of system representation diagrams, Information & Management, Volume 29, Issue 2, August 1995, Pages 79-94
18. (Patig 2008) : S. Patig, A practical guide to testing the understandability of notations, Conferences in Research and Practice in Information Technology Series; Vol. 325, Proceedings of the fifth on Asia-Pacific conference on conceptual modelling - Volume 7, 2008, pp 49-58
19. (Lange 2005) : C. Lange, M. Chaudron, Managing Model Quality in UML-Based Sotware Delevopment, Proc. of the 13th workshop on software Tachnology and Engineering Practice (STEP'05), 2005, pp. 7-16.
20. (Rossi 1996) M. Rossi et S. Brinkkemper, Complexity metrics for systems development methods and techniques, Information Systems, Vol. 21, num 2, 1996, pp. 209-227.
21. (Linbdland 1994) O. Lindland, G. Sindre, A. Solvberg, Understanding Quality in Conceptual Modeling, IEEE Software, Vol. 11, 1994, pp. 42-49.
22. (Krogstie 1998) J. Krogstie, Integrating the Understanding of Quality in Requirements Specification and Concptual Modeling, Software Engineering Notes, ACM SIGSOFT, Vol 23, num 1, 1996, pp. 86-91
23. Mathias Kleiner, Patrick Albert and Jean Bézivin, Parsing SBVR-based controlled languages, In ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems (MODELS'09), empirical track, Denver, Colorado, USA, Oct 2009, pages 122-136.
24. Reza Razavi, Noury Bouraqadi, Joseph W. Yoder, Jean-François Perrot, Ralph E. Johnson: *Language support for adaptive object-models using metaclasses*. Computer Languages, Systems & Structures 31(3-4): 199-218 (2005)
25. Razavi, Reza, Kirill Mechitov, Gul Agha, Jean-Francois Perrot. "Ambiance: A Mobile Agent Platform for End-User Programmable Ambient Systems," J.C. Augusto and D. Shapiro (eds.), Advances in Ambient Intelligence, Frontiers in Artificial Intelligence and Applications (FAIA), vol. 164, IOS Press, 2007

Towards Disappearing Languages

Charles Consel,
INRIA / University of Bordeaux
Charles.Consel@inria.fr

This paper explores challenges that need to be addressed to make the Domain-Specific Language (DSL) approach successful. In particular, we argue that a DSL should be blended with a domain process and used by domain experts in support of their job. We call such languages *disappearing languages*.

The DSL approach has long been used with great success in both historical domains, such as telephony, and recent ones, such as Web application development. And yet, from software engineering to programming languages, there is a shared feeling that there is still much work to do to make the DSL approach successful.

Unlike General-Purpose programming Languages (GPLs) that target trained programmers, a DSL revolves around a domain: it originates from a domain and targets members of this domain. Thus, a successful DSL should be some kind of a *disappearing language*; that is, one that is blended with some domain process. In doing so, programming expands its scope to reach end users. That is, users for which writing programs come in support of achieving their primary job function. Well-known examples include Excel and MatLab.

Creating a disappearing language critically relies on an analysis phase of the target domain. The result of this analysis then fuels a design phase of the language. Practical experience shows that these two phases are time consuming, human intensive and high risk.

- What kind of tools could assist in performing these two phases?
- How much improvement can we expect from a tool-assisted process?

A successful DSL is above all one that is being used. To achieve this goal, the designer may need to downgrade, simplify and customize a language. In doing so, DSL development contrasts with programming language research where generality, expressivity and power should characterize any new language. As a consequence, programming language experts may not be the perfect match for developing a DSL.

- Does this mean that, for a given domain, its members should be developing their own DSL?
- Or, should there be a new community of *language engineers* that bridge the gap between programming language experts and members of a domain?

In many respects, a DSL is often an over-simplified version of a GPL: customized syntactic constructs, simple semantics, and by-design verifiable properties. These key differences may raise concerns about a lack of tool support for DSL development. Yet, there are many program manipulation tools (parser generators, editors, IDEs) that can be easily customized for new languages, whether textual or graphical. Furthermore, for a large class of DSLs,

compilation amounts to producing code for a domain-specific programming framework, and enabling the use of high-level transformation tools. Lastly, properties can often be checked by generic verification tools.

- Then, what is missing to develop DSLs?
- Do we need to have an integrated environment for DSL development, orchestrating a library of tools?
- Should there be a new breed of compiler and analysis generators matching the requirements of DSLs?

The research community in programming languages has always overwhelmingly focused on the design and implementation of languages. Very little effort has been devoted to understanding how humans use programming languages. This avenue of research is a key to expanding programming beyond computer scientists.

- How do we design languages that are easy to use?
- How do we measure the productivity benefit of the use of a language?
- How do we assess the software quality of DSL programs?

Vers une réification de l'énergie dans le domaine du logiciel

L'énergie comme ressource de première classe

Jean-Marc Menaud, Adrien Lèbre, Thomas Ledoux, Jacques Noyé
Pierre Cointe, Rémi Douence, Mario Südholt

Équipe ASCOLA (EMNantes-INRIA, LINA)
prénom.nom@emn.fr

Résumé

En quelques années, le problème de la gestion de l'énergie est devenu un enjeu de société. En informatique, les principaux travaux se sont concentrés sur des mécanismes permettant de maîtriser l'énergie au niveau du matériel. Le renforcement du rôle de l'informatique dans notre société (développement des centres de données, prolifération des objets numériques du quotidien) conduit à traiter ces problèmes aussi au niveau du logiciel.

Dans ce papier, nous nous posons la question de la réification de l'énergie comme fut posée en son temps celle de la réification de la mémoire (l'espace) et de l'interpréteur (la machine d'exécution). Le défi est d'abord de sensibiliser l'utilisateur final au problème de la consommation énergétique en visualisant, grâce à des mécanismes d'introspection, sa consommation, à l'image de ce qui se fait aujourd'hui dans le domaine automobile (consommation instantanée d'essence). Il s'agira ensuite de proposer aux développeurs des mécanismes d'intercession leur permettant de contrôler cette consommation énergétique. Ces mécanismes réflexifs devront concerner l'ensemble du cycle de vie du logiciel.

1 L'énergie comme nouvelle préoccupation ?

En 2008, 2% de la production électrique mondiale a été consommée par les centres de données utilisés pour l'hébergement de l'Internet. La problématique est d'importance car, d'un point de vue mondial, et selon l'*Environmental Protection Agency* (EPA), il a été estimé qu'entre 2007 et 2012 les serveurs et les centres de données vont doubler leurs besoins en énergie pour atteindre 100 milliards de kWh. Cette augmentation des besoins est liée à l'accroissement du nombre d'internautes [1], des terminaux d'accès [2], des données disponibles [3], et à l'apparition de nouveaux usages comme le *Cloud Computing*, ou l'Internet des Objets [4]. À ce rythme, dans 25 ans, Internet consommera autant d'énergie que l'humanité tout entière en 2008.

D'un point de vue plus global, l'impact des TIC

1. Principalement dans les pays du BRIC : Brésil, Russie, Inde, Chine.

2. Le nombre d'ordinateurs est estimé à 1 milliard en 2008, 2 milliards en 2013, sans compter le nouveau marché des téléphones intelligents (77 millions vendus sur les trois premiers mois de 2009 soit 14 par seconde).

3. 161 exaoctets de données ont été créées en 2006, soit approximativement 3 millions de fois l'information contenue dans tous les livres jamais écrits.

4. D'ici à quelques années, le nombre d'appareils communiquants pourrait être de plusieurs dizaines de milliards, soit plus que le nombre d'êtres humains.

dans la consommation électrique est de l'ordre de 10 à 15% dans les pays développés. Pour la France, l'estimation de la consommation électrique est comprise entre 55 et 60 TWh par an, soit 13,5% de la consommation française. Au niveau des particuliers, les TIC dans leur globalité représentent 30% [5] de l'électricité consommée. Depuis 10 ans, cette consommation a triplé. Cette progression risque d'augmenter fortement dans les prochaines années avec l'apparition d'équipements toujours plus « énergivores » (écrans LCD, passerelles domestiques, etc.).

Par conséquent, une rupture technologique est essentielle pour relever le défi majeur de la maîtrise énergétique des équipements informatiques. Les fondeurs, constructeurs et installateurs de matériels informatiques ont déjà travaillé sur ce problème. Par exemple, dans les centres de données, l'unité d'accueil de nouvelles machines n'est pas le nombre de serveurs à installer ou leur type, mais la puissance électrique consommée. Si des modèles de consommation d'énergie au niveau matériel sont aujourd'hui disponibles, il n'existe pas l'équivalent au niveau logiciel.

Dans ce papier, nous explorons les possibilités d'intégrer l'énergie en tant que ressource de première classe pour le logiciel, et donc de disposer de tels modèles dans le domaine du logiciel, manipulables par le logiciel. Cette réification permettra de maîtriser l'empreinte énergétique d'une application tout au long de son cycle de vie.

2 L'énergie, nouvelle ressource physique ?

En matière de gestion explicite des ressources physiques et depuis la fin des années 40, l'informatique est passée d'un extrême à l'autre. Pendant longtemps, le programmeur a été comptable de toutes les ressources qu'il utilisait : mémoire, nombre de cycles du processeur, nombre d'accès au disque, etc. Ainsi sous DOS, la mémoire était organisée en blocs de 64 Ko et cela expliquait la taille maximale d'un tableau dans des langages comme Pascal. Pour le langage Lisp, les programmeurs attachaient une grande importance au contrôle de la consommation de la mémoire qui se traduisait dans le nombre de doublets alloués et la taille de la pile. D'où l'utilisation des fonctions de chirurgie (par exemple `nconc` versus `append` pour la concaténation) permettant d'altérer physiquement les listes. « *Ces fonctions sont d'un maniement délicat faisant passer LISP d'un langage d'expressions à un langage de manipulation de pointeurs où les programmes peuvent s'automodifier* » (chapitre 2 de [2]). Avec l'usage des macros, ces fonctions ont donné lieu aux travaux sur la métaprogrammation et la réflexion, dont 3LISP, dans le but initial d'autodécrire des mécanismes de base de l'interpréteur dont la gestion de la mémoire et celle des continuations [4].

À partir des années 80, la mise en œuvre de mécanismes de gestion des ressources, comme les ramasse-miettes, a permis de décharger les développeurs de la contrainte d'une gestion fine des ressources. Ce détachement du programmeur vis-à-vis des ressources a été amplifié par la montée en puissance des capacités des machines suivant la loi de Moore. Les méthodes de programmation et de génie logiciel se sont affranchies (presque) complètement des notions physiques de la machine pour traiter de la rapidité de développement (objets, programmation agile, patrons de conception, usine logicielle) et la facilité de maintenance et d'évolution (architectures à base de composants, d'aspects et de services).

Grâce à ces techniques logicielles, et à la loi de Moore, nous avons pu construire des applications complexes à forte fonctionnalité ajoutée. Mais, alors que la demande continue à croître, il est impératif d'enrayer l'envolée de la consommation énergétique associée. On va donc rapidement se retrouver dans la situation des programmeurs des années 70, avec des contraintes énergétiques plutôt que des contraintes sur la consommation de la mémoire. Peut-on appliquer à l'énergie les mêmes stratégies que celles utilisées pour gérer la consommation de la mémoire ? Contrairement à la mémoire, la consommation énergétique propre d'une application n'est pas directement accessible. Le premier verrou à lever consiste à réifier cette ressource particulière.

5. Étude REMODECE 2007, ADEME, électricité spécifique consommée.

3 Comment réifier l'énergie ?

Pour réifier l'énergie, il faut dans un premier temps la quantifier, donc disposer d'outils logiciels ou matériels permettant de mesurer la consommation électrique d'un quantum d'exécution (instruction de la machine, pseudo-code, exécution d'un service, suivant la granularité choisie). Cependant mesurer la consommation énergétique d'un quantum d'exécution complexe, comme un service, est une tâche ardue. En effet, l'énergie consommée par un quantum d'exécution est la somme des énergies consommées par les ressources physiques utilisées par ce quantum d'exécution (CPU, mémoire, disque, réseau, etc.).

Dans un deuxième temps, il est nécessaire de construire un modèle théorique de la consommation énergétique qui permette d'attacher des quanta d'énergie aux quanta d'exécution. Pour être utilisable, ce modèle doit être compositionnel et réaliste, c'est-à-dire qu'il doit être capable de prédire avec suffisamment de précision la consommation d'un programme par composition simultanée des quanta d'exécution et des quanta d'énergie. Ces besoins peuvent conduire à choisir des quanta de grain assez gros. Travailler à un niveau de pseudo-code semble prometteur [3]. Une première difficulté peut être, suivant le modèle de programmation, la prise en compte de la composition parallèle de programmes. Certains modèles de programmation, comme le modèle de programmation synchrone, peuvent être plus favorable que d'autres. Une deuxième difficulté est, dans un contexte distribué, de prendre en compte les coûts de communication et plus généralement d'infrastructure (assurant la disponibilité des services).

Cette quantification énergétique des applications sera utilisée à la fois statiquement par le programmeur et le compilateur, assistés de systèmes d'analyse statique quantitative (voir, par exemple, [5]), et dynamiquement par le système d'exécution.

Réifier l'énergie au niveau langage permettra à un programmeur d'introspecter son code sous l'angle énergétique, et de le modifier en changeant ses fonctionnalités ou son implémentation, par exemple en développant des algorithmes de type *anytime* [6]. Les algorithmes *anytime* sont des algorithmes dont la qualité des résultats augmente avec le temps de calcul. Ce type d'algorithme permet de contrôler l'énergie affectée à un calcul en interrompant ce calcul lorsque son quota est épuisé.

Réifier l'énergie au niveau du système d'exécution permettra d'attribuer l'énergie disponible aux applications et d'arbitrer ces allocations en cas de pénurie (énergétique) comme cela est fait pour d'autres ressources [1]. Des stratégies d'allocations spécifiques à la gestion énergétique seront probablement à développer.

4 Conclusion

L'informatique au sens large a et aura un impact significatif sur la consommation électrique mondiale. La poursuite de son développement passe par une prise en charge généralisée de la contrainte énergétique au-delà des domaines habituels du matériel et de certains secteurs de l'informatique embarquée. Pour ce faire, il est nécessaire de réifier l'énergie en faisant de celle-ci une ressource accessible statiquement et dynamiquement au niveau de chaque composant applicatif. Il sera alors possible, dans un premier temps, de sensibiliser de manière efficace à la fois les utilisateurs et les programmeurs finaux aux coûts énergétiques et, dans un deuxième temps, de fournir des concepts [6] et des outils de développement ainsi que des systèmes d'exécution susceptibles de significativement réduire ces coûts.

Références

[1] Luc Morau and Christian Queinnec. Resource aware programming. *ACM Transactions on Programming Languages and Systems*, 27(3) :441–476, May 2005.

[2] Christian Queinnec. *LISP Mode d'emploi.* Eyrolles, 1984.

[3] Chiyoung Seo, Sam Malek, and Nenad Medvidovic. Component-Level Energy Consumption Esti-

6. On peut par exemple penser au développement, dans la continuité des algorithmes *anytime*, d'une algorithmique ≪ verte ≫.

mation for Distributed Java-Based Software Systems. In *Component-Based Software Engineering*, pages 97–113. Springer-Verlag, 2008.

[4] Brian Cantwel Smith. Reflection and semantics in LISP. Technical report, Palo Alto Research Center, 1984.

[5] Pascal Sotin, David Cachera, and Thomas Jensen. Quantitative static analysis over semirings : Analysing cache behaviour for Java Card. In Alessandra Di Pierro and Herbert Wiklicky, editors, *Quantitative Aspects of Programming Languages (QAPL '06)*, 2006.

[6] Shlomo Zilberstein. Using anytime algorithms in intelligent systems. *AI Magazine*, 17(3) :73–83, 1996.