Fostering metamodels and grammars within a dedicated environment for HPC: the NabLab environment (tool demo)

Benoît Lelandais, Marie-Pierre Oudot, Benoit Combemale

▶ **To cite this version:**

Benoît Lelandais, Marie-Pierre Oudot, Benoit Combemale. Fostering metamodels and grammars within a dedicated environment for HPC: the NabLab environment (tool demo). SLE 2018 - International Conference on Software Language Engineering, Nov 2018, Boston, United States. pp.1-9, 10.1145/3276604.3276620 . hal-01910139

HAL Id: hal-01910139
https://inria.hal.science/hal-01910139v1

Submitted on 31 Oct 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Fostering Metamodels and Grammars within a Dedicated Environment for HPC

The NabLab Environment (Tool Demo)

Benoît Lelandais
CEA/DAM/DIF, France
benoit.lelandais@cea.fr

Marie-Pierre Oudot
CEA/DAM/DIF, France
marie-pierre.oudot@cea.fr

Benoit Combemale
Univ. Toulouse & Inria, France
benoit.combemale@inria.fr

Abstract

Advanced and mature language workbenches have been proposed in the past decades to develop Domain-Specific Languages (DSL) and rich associated environments. They all come in various flavors, mostly depending on the underlying technological space (*e.g.*, grammarware or modelware). However, when the time comes to start a new DSL project, it often comes with the choice of a unique technological space which later bounds the possible expected features.

In this tool paper, we introduce NabLab, a full-fledged industrial environment for scientific computing and High Performance Computing (HPC), involving several metamodels and grammars. Beyond the description of an industrial experience of the development and use of tool-supported DSLs, we report in this paper our lessons learned, and demonstrate the benefits from usefully combining metamodels and grammars in an integrated environment.

CCS Concepts • **Software and its engineering** → **Domain specific languages**; *Model-driven software engineering*; • **Computing methodologies** → Massively parallel and high-performance simulations;

Keywords HPC, DSL, Grammar, Metamodel, Compilers

ACM Reference Format:
Benoît Lelandais, Marie-Pierre Oudot, and Benoit Combemale. 2018. Fostering Metamodels and Grammars within a Dedicated Environment for HPC: The NabLab Environment (Tool Demo). In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering (SLE '18), November 5–6, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3276604.3276620

ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

SLE '18, November 5–6, 2018, Boston, MA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6029-6/18/11…$15.00
https://doi.org/10.1145/3276604.3276620

1 Introduction

In the field of High Performance Computing (HPC), addressing the major challenges of software productivity and performance portability is becoming necessary to take advantage of emerging extreme-scale computing architectures. There is a growing demand for new programming environments to improve scientific productivity, facilitate design and implementation, and optimize large production codes with higher-level programming abstraction. In this context, the numerical-analysis specific language NabLab improves applied mathematicians productivity and enables new algorithmic developments for the construction of hierarchical and modular high-performance scientific applications.

NabLab results from a longstanding effort at CEA (French Atomic Energy Commission) to develop advanced tools and methods to support the development of ever more complex simulation codes. To this end, software engineers have studied the use of workbenches proposed in recent decades to develop Domain-Specific Languages (DSLs) and rich associated environments (*e.g.*, editor, analysis and optimization tools, or compiler). They all come in various flavors, mostly depending on the underlying technological space (*e.g.*, grammarware or modelware). Grammar-based language workbenches have been first proposed, together with associated frameworks supporting efficient language manipulation [1]. More recently, metamodel-based language workbenches have emerged with a rich ecosystem to develop modeling environments including tools directly usable by domain experts [10]. NabLab takes advantage of all of these approaches, first to develop compilers optimized for different low-level architectures, and more recently to address the intrinsic complexity of simulation codes themselves by supporting the design and validation activities.

Beyond the description of an industrial experience of the development and use of tool-supported DSLs, this tool demonstration paper also reflects our main lessons learned from developments involving different technological spaces. In particular, we discuss the benefits from usefully combining metamodels and grammars in an integrated environment to leverage complementary ecosystems.

The rest of the paper is structured as follows. Section 2 provides an overview of the modeling environment, which is described later in Section 3 with the technical architectures

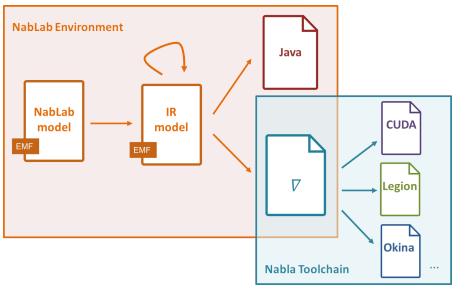

Figure 1. Seamless integration of NabLab and Nabla

and the experience report of both developers and users. In Section 4, we discuss several design and technology choices, leading to open questions for the SLE community. Section 5 reflects on related work and Section 6 concludes the paper and outline prospects for the future.

2 Overview of the NabLab Environment

In this section we introduce both the NabLab and the Nabla open-source Domain Specific Language (DSL), and the corresponding development environment for numerical-analysis.

Nabla[1] aims at writing out numerical-analysis algorithmic sources in order to generate optimized code for different runtimes and architectures [8] called backends (cf. Fig. 1, right). Nabla is dedicated to applied mathematicians and physicists, who are the language *users* in our context. The DSL allows the conception of multi-physics applications, and is based on different concepts: no central main function, a multi-tasks based parallelism model and a hierarchical logical time-triggered scheduling. A Nabla source file is divided into two parts: on the one side the declarations of options, functions and variables (cf. Fig. 2, left) and, on the other, the definition of small unit functions called jobs (cf. Fig. 2, right). One can observe that jobs are tagged with a '@' directive and a number representing a hierarchical logical time (HLT), their relative triggering time. Jobs with a higher logical time wait for a previous one to be finished before being scheduled. Two jobs with the same logical time are triggered in parallel. This new dimension to parallelism is explicitly expressed to go beyond classical programming models. Control and data concurrencies are combined consistently to achieve statically analyzable transformations and efficient code generation. The Nabla compiler holds these effective generation stages for different architectures (*e.g.*, CUDA[2], Legion [4] and Okina in Fig. 1). While the Nabla syntax is close to numerical-analysis concepts, the concept of HLT is related to the optimization domain thus bringing the DSL closer to the solution space, i.e., requiring the DSL user to map a given problem in terms of the way it is eventually implemented.

With the increasing complexity of simulation codes, there is a pressing need for high-level abstractions with which

```
options {
  ℝ option_δt_ini = 1.0e-4;
}

nodes {
  ℝ m, ρ, V;    // mass, density & volume
  ℝ³ X;         // vertice coords
}

cells {
  ℝ³ C[nodes];  // flux vector
}

global {
  ℝ t, t_n_plus_1, δt;
}
```

```
iniDeltaT @ -9.0 {
  δt = option_δt_ini;
}

init_t @ -3.0 {
  t = 0.0;
}

∀ cells computeV @ 7.0 {
  ℝ tmp = 0;
  ∀ nodes tmp += dot(C, X);
  V = 0.5 * fabs(tmp);
}

∀ cells computeDensity @ 9.0 {
  ρ = m / V;
}
```

Figure 2. Example of Nabla source file

language users can directly reason about their problems, and generative approaches dealing with implementation details (*e.g.*, HLT). NabLab has been developed in this context, with the main objective of providing abstraction and tools that help language users to design and validate their simulation codes and to translate them into Nabla programs to be further efficiently compiled for different architectures. The syntax of NabLab is strongly inspired by Nabla, but has been developed as a separate external DSL with its own high-level abstractions, and comes with a rich development environment that supports DSL users in the design and validation of their specifications. A textual editor (cf. Fig. 3, center) proposes contextual code completion, code folding, syntax highlighting, error detection, quick fixes, variable scoping, and type checking. The environment also provides a model explorer, a dedicated outline view (cf. Fig. 3, left), and a contextual LaTeX view (cf. Fig. 3, bottom). A debugging environment under construction will provide variable inspection, plot display and 2D/3D visualization. When the code has no error, the NabLab compiler can automatically generate a LaTeX report, and efficient implementations thanks to the associated compilation chain.

To bridge the abstraction gap between NabLab and Nabla, an Intermediate Representation (cf. IR in Fig. 1) has been introduced into the compilation chain to offer the relevant structure for performing analysis and optimisations. For instance, a NabLab program is composed of jobs whose execution does not sequentially start at the beginning of the program. To highlight the execution flow of the program and detect unintended cycles, a data flow graph is computed and displayed on a graphical diagram (cf. Fig. 3, right). This data flow graph is also used to automatically infer the '@' directives required to generate an optimized Nabla program, compiled further to the different low-level backends supported by Nabla. Alternatively, the IR has recently been supporting the implementation of additional code generators to high level execution platforms (*e.g.*, JVM).

3 Implementation and Retrospective

This section focuses on design and technology choices for the development of the NabLab environment, and reports on the experiences of language users and designers.

[1]Numerical Analysis Based LAnguage, https://www.nabla-lang.org
[2]NVIDIA CUDA, https://docs.nvidia.com/cuda

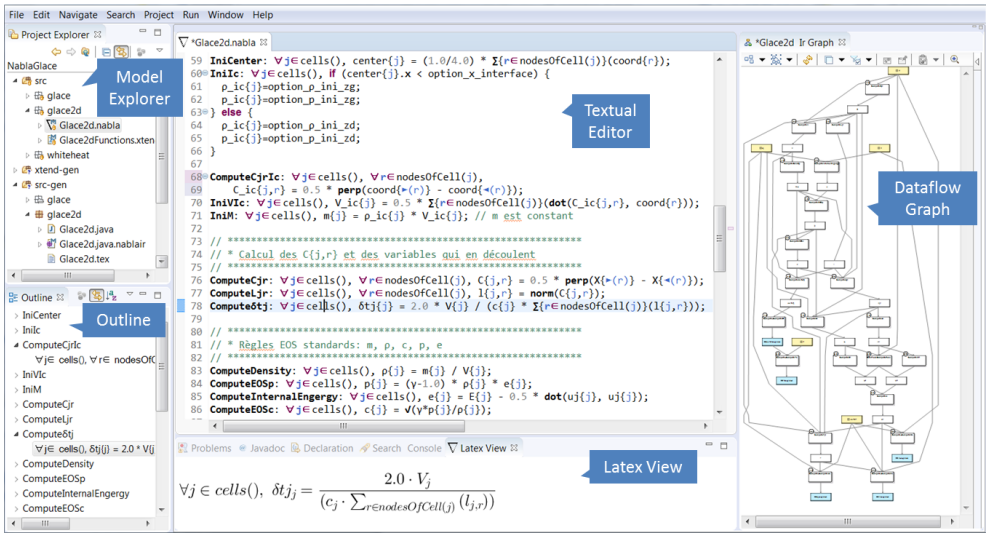

Figure 3. NabLab Integrated Development Environment

3.1 The Nabla Toolchain

The Nabla toolchain is composed of three main parts: *the frontend*, a Flex[3]/Bison[4] parser that reads a set of input files, *the middle-end* that provides a collection of steps *e.g.*, instrumentation, performance, analysis, validation and verification or data layout optimizations, and *the backends* that holds the effective generation stages for different targets or architectures like Arcane [11], CUDA, Legion or Okina, a standalone backend which generates fully-vectorized C/C++ source files.

A dozen hydrodynamic applications have been written with Nabla (structured/unstructured meshes, explicit/implicit schemes), and the users help to gradually evolve the syntax of the language to better match the different algorithms.

Nabla users appreciate the language for prototyping new numerical schemes and test new software stacks. They also appreciate the Open-Source nature of the project to collaborate with students and universities. However, while they like the compactness and mathematical rendering of the UTF8 syntax, they express the need for an advanced dedicated editor. They also identify the complexity to compose big applications in providing the HLT, particularly the difficulty to extend an existing program and to detect unintended job cycles. They point out that this task should be hidden for applied mathematicians. On a more general level, they say that they can develop, debug and test a simple mini-application with a basic editor, but it becomes tricky to follow the same development process on larger applications.

Nabla language designers are used to working with the grammar-based Flex/Bison tools and C/C++ language. Many tools and code examples are available. This grammar-based approach is broadly used in the field of optimizing compilers. However, language designers point out that providing tools and editors with this technology stack is not easy.

[3]Flex: the fast lexical parser, https://github.com/westes/flex
[4]GNU Bison, https://www.gnu.org/software/bison

3.2 The NabLab Environment

NabLab is based on the *Eclipse Modeling Framework*[5] (EMF) [18] and its ecosystem: the textual editor and the outline are based on Xtext[6], and the data flow graph is displayed through a Sirius diagram[7]. NabLab corresponds to 5 KLOC of Xtend, including both all editor capabilities (*e.g.*, validation, scoping, typing and quick fixes) as well as the graph construction.

NabLab brings some changes to Nabla: explicit declaration of external functions and required connectivities, mesh variables definition on connectivities, explicit naming of iterators and generalization of reductions (*i.e.*, integration of reductions, like sums, inside a loop). These modifications avoid any ambiguity on iterators and variables, enabling syntactic validation rules (*e.g.*, loop patterns correctness). NabLab also increases the level of abstraction to bring the language closer to the problem space (*e.g.*, the metamodel no longer requires users to explicitly provide HLTs on jobs).

To bridge the gap between NabLab and Nabla, we introduce a numerical analysis specific Intermediate Representation (IR). This IR is implemented as an Ecore metamodel, automatically instantiated when a valid NabLab file is saved (cf. Fig. 1). As NabLab metamodel hides the tricky notion of HLT to mathematicians, this data is automatically inferred by synthesizing *in* and *out* variables for each job. This data flow is presented in a Sirius diagram (cf. right view in Fig. 3). Once the data flow is correct and does not present cycles anymore, the shortest path to each job is computed and its value stored in the IR model. Several transformation and optimization passes are defined on the IR (*e.g.*, variable default values replaced by initialization jobs, inner reductions transformed in loop patterns, or jobs tagged with a HLT). After these transformation steps, the IR is very close to the

[5]EMF, https://www.eclipse.org/modeling/emf
[6]Xtext, https://www.eclipse.org/Xtext/
[7]Sirius, https://www.eclipse.org/sirius

solution space and it becomes the source of the last *Model To Text* transformation to generate Nabla code (or alternatively multi-threaded Java programs). Thanks to this IR, the code generation is easier, the compilation chain is modular and the concerns are well separated.

Thanks to the Java code generator, users get directly executable code in the Eclipse environment, including validating mathematical algorithms without having to connect to a calculator and independently of the performance. A debugging environment is still under development. The goal is to be able to execute the code step by step by examining not the generated code but the initial NabLab code owned by the user. For such a purpose, information is accumulated throughout the compilation chain into a trace model (*i.e.*, using annotations).

Language designers appreciate Xtend for its high level functions, support for model manipulation (*e.g.*, lambda) and code generation (*e.g.*, template). Xtext greatly facilitates the creation of a DSL, providing architecture and design patterns to develop various required tools. Language designers also appreciate the rapidly evolving EMF ecosystem, always providing new cutting-edge technologies.

4 Discussions

4.1 Fostering Metamodels and Grammars

The Nabla grammar-based approach uses the C++ language and time-honored technologies like Flex/Bison. This approach is well known and appreciated by the optimization community for its proximity to execution architectures (*e.g.*, processor co-design). The NabLab metamodel-based approach uses the EMF ecosystem that offers several tools to build a user-friendly environment with rich textual editors and graphical diagrams. The NabLab metamodel provides an extra abstraction level to stay close to the problem space. To avoid mixing the concerns, notions related to optimization are deliberately omitted and the metamodel remains close to the mathematical expression.

The numerical analysis IR has been created to bridge the gap between the mathematicians needs and the optimization needs. Additionally, the IR helps in separating the stakeholders (*e.g.*, teams working on software engineering tools and those working on optimization) and software life cycles. The NabLab metamodel can evolve independently to satisfy mathematicians and physicists and take into account new numerical formulations while the Nabla compilation chain evolves towards new optimization strategies.

4.2 Looking Ahead

The seamless integration of NabLab and Nabla allows us both to generate Java code using the same technology stack (Java API) and to target lower level backends like CUDA or FPGA. In the past decade, the HPC community explored various programming models providing abstractions for both parallel execution of code and data management, such as Kokkos[8]. Regarding such frameworks, it appears relevant to raise the generation at the level of the IR, since the gap is limited. We are currently working on such a proof of concept, to check if optimization stakeholders can directly interact with the IR. Hence, two issues remain. The first is to identify up to what level of abstraction of the backend we are able to generate from the IR and what criteria determine whether the generation is more relevant from the IR or from the Nabla code. The second is to determine which changes in the technology stack are ready to accept the HPC optimization stakeholders (*e.g.*, should we offer them a C++ API for IR, for example using EMF4CPP[9] or is it conceivable that they use the traditional EMF technology stack with the Java API?).

5 Related Work

The use of DSLs for scientific computing and computational science has a long-standing history [15], especially to address various specific concerns (*e.g.*, architecture [2], stencil codes [14], deployment on parallel platforms [3] or cloud platforms [7], etc.), or even specific application domains (*e.g.*, fluid dynamics [13]). More recently, problem-specific DSLs for HPC has been explored in the context of MDE [6, 16], incl. applications in the web and the cloud [17].

In this area of application particularly relevant for DSLs, as far as we know, this tool demonstration paper is the first to report an industrial experience on the combination of grammars and metamodels in a complete modeling environment for numerical analysis and HPC. This demonstration takes a language engineering point of view, to document the pros and cons of the different technological spaces involved.

6 Conclusion and Perspectives

The development of NabLab over Nabla, usefully combining metamodels and grammars, is an interesting experience from both a technical and a social point of view. While the respective complementary ecosystems offer advanced features for developing industrial-grade environments, this integration facilitates collaboration between optimization experts and development team.

Future work includes debugging and results visualization capabilities (*e.g.*, step-by-step execution, variables inspection, curves, mesh visualization). We also plan to investigate using the LLVM compiler infrastructure[10] for low-level source code generation. Finally, another challenge consists in coupling Modane [12] and NabLab to describe both the architectural and behavioral aspects of simulation codes. Indeed, Modane is a DSL developed to design the static part of a numerical simulation code while NabLab focuses on the dynamic part. Thus, the modeling toolchain could meet our need to facilitate adaptation to the continuous hardware evolution.

[8] Kokkos, https://github.com/kokkos
[9] EMF for C++, https://code.google.com/archive/p/emf4cpp
[10] The LLVM Compiler Infrastructure, https://llvm.org

References

[1] A. Aho, M. Lam, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, second edition, 2007.

[2] B. A. Allan, R. Armstrong, D. E. Bernholdt, F. Bertrand, K. Chiu, T. L. Dahlgren, K. Damevski, W. R. Elwasif, T. G. W. Epperly, M. Govindaraju, D. S. Katz, J. A. Kohl, M. Krishnan, G. Kumfert, J. W. Larson, S. Lefantzi, M. J. Lewis, A. D. Malony, L. C. Mclnnes, J. Nieplocha, B. Norris, S. G. Parker, J. Ray, S. Shende, T. L. Windus, and S. Zhou. A component architecture for high-performance scientific computing. *Int. J. High Perform. Comput. Appl.*, 20(2):163–202, May 2006. ISSN 1094-3420.

[3] E. Arkin, B. Tekinerdogan, and K. M. Imre. Model-driven approach for supporting the mapping of parallel algorithms to parallel computing platforms. In *Proceedings of the 16th International Conference on Model-Driven Engineering Languages and Systems - Volume 8107*, pages 757–773. Springer-Verlag New York, Inc., 2013. ISBN 978-3-642-41532-6.

[4] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 66:1–66:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press. ISBN 978-1-4673-0804-5. URL http://dl.acm.org/citation.cfm?id=2388996.2389086.

[5] L. Bettini. *Implementing Domain Specific Languages with Xtext and Xtend - Second Edition*. Packt Publishing, 2nd edition, 2016. ISBN 1786464969, 9781786464965.

[6] J.-M. Bruel, B. Combemale, I. Ober, and H. Raynal. MDE in Practice for Computational Science. In *International Conferene on Computational Science (ICCS)*, June 2015. URL https://hal.inria.fr/hal-01141393.

[7] C. Bunch, N. Chohan, C. Krintz, and K. Shams. Neptune: A domain specific language for deploying hpc software on cloud platforms. In *Proceedings of the 2Nd International Workshop on Scientific Cloud Computing*, ScienceCloud '11, pages 59–68. ACM, 2011.

[8] J.-S. Camier. Improving performance portability and exascale software productivity with the nabla numerical programming language. In *Proceedings of the 3rd International Conference on Exascale Applications and Software*, EASC '15, pages 126–131, Edinburgh, Scotland, UK, 2015. University of Edinburgh. ISBN 978-0-9926615-1-9. URL http://dl.acm.org/citation.cfm?id=2820083.2820107.

[9] G. Carré, S. D. Pino, B. Després, and E. Labourasse. A cell-centered lagrangian hydrodynamics scheme on general unstructured meshes in arbitrary dimension. *Journal of Computational Physics*, 228(14): 5160 – 5183, 2009. ISSN 0021-9991. doi: https://doi.org/10.1016/j.jcp.2009.04.015. URL http://www.sciencedirect.com/science/article/pii/S002199910900196X.

[10] B. Combemale, R. France, J.-M. Jézéquel, B. Rumpe, J. R. Steel, and D. Vojtisek. *Engineering Modeling Languages*. Chapman and Hall/CRC, Nov. 2016. URL http://mdebook.irisa.fr/.

[11] G. Grospellier and B. Lelandais. The Arcane development framework. In *Proceedings of the 8th Workshop on Parallel/High-Performance Object-Oriented Scientific Computing*, POOSC '09, pages 4:1–4:11. ACM, 2009.

[12] B. Lelandais and M.-P. Oudot. Modane: A design support tool for numerical simulation codes. *Oil Gas Sci. Technol. - Rev. IFP Energies Nouvelles*, 71(4):57, 2016.

[13] S. Macià, S. Mateo, P. J. Martínez-Ferrer, V. Beltran, D. Mira, and E. Ayguadé. Saiph: Towards a dsl for high-performance computational fluid dynamics. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*, RWDSL2018. ACM, 2018.

[14] R. Membarth, F. Hannig, J. Teich, and H. Köstler. Towards domain-specific computing for stencil codes in hpc. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 1133–1138, Nov 2012.

[15] D. Orchard and A. Rice. A computational science agenda for programming language research. *Procedia Computer Science*, 29(0):713 – 727, 2014. ISSN 1877-0509. doi: http://dx.doi.org/10.1016/j.procs.2014.05.064. URL http://www.sciencedirect.com/science/article/pii/S1877050914002415.

[16] M. Palyart, D. Lugato, I. Ober, and J. Bruel. Improving scalability and maintenance of software for high-performance scientific computing by combining MDE and frameworks. In *MODELS 2011*, LNCS, pages 213–227, 2011. doi: 10.1007/978-3-642-24485-8_16. URL http://dx.doi.org/10.1007/978-3-642-24485-8_16.

[17] W. Simm, F. Samreen, R. Bassett, M. Ferrario, G. Blair, P. Young, and J. Whittle. Se in es: Opportunities for software engineering and cloud computing in environmental science. In *ICSE-SEIS'18*. ACM, 12 2018. ISBN 9781450356619. doi: 10.1145/3183428.3183430.

[18] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: eclipse modeling framework*. Addison-Wesley Professional, 2008.

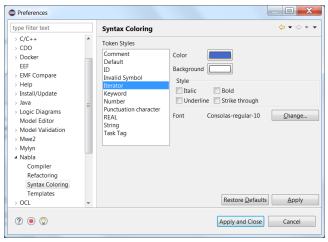

Figure 5. Nabla Syntax Coloring parameter window

A Demonstration Outline

During this demonstration, we will exhibit the capabilities provided by the Nabla programming language and the edition, design and generation services integrated in the NabLab environment.

The demonstration will be divided into two parts: first we will present the resulting environment, with a demonstration of a program example, and all the facilities provided by the tooling. Second, we will review the architecture of the NabLab environment with a focus on building the modular compilation chain around the IR.

A.1 NabLab Runtime Environment

For this demonstration, we will take as an example the use case of a resolution of a 2D hydrodynamics problem using Glace [9], a cell-centered lagrangian scheme.

We will show the possibilities offered by the Eclipse environment to help the user write his numerical code model. We will also show the graph deduced from the model and Java and Nabla codes generated. Then we will present the result of the Java execution. Concerning Nabla, we will limit ourselves to the generated code and show the results obtained on the HPC cluster. Finally, we will show a use case based on an increased complexity model to discuss the contributions but also the limits of the graphical interface.

The working environment for mathematicians using NabLab consists of an Eclipse configured with the following components: a model explorer, a textual editor, an outline and a latex view as shown in Fig. 4.

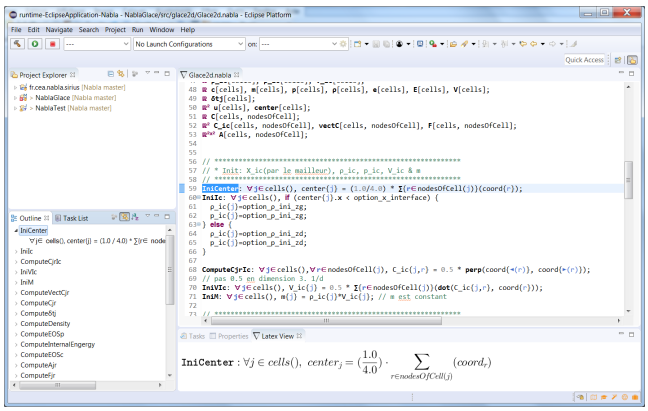

Figure 4. NabLab environment

A.1.1 Model Explorer

The model explorer enables users to move in the directories of their projects and to open inter alia, NabLab files. They are identified by a specific icon (∇).

A.1.2 Textual Editor

The textual editor is a specific editor whose syntax highlighting can be defined in a dedicated setting menu (cf. Fig. 5).

NabLab editor offers several useful facilities for end-users:

- Contextual completion proposal. This proposal simplifies the writing for end-users. Only valid instructions are proposed and they do not have to write UTF8 characters (cf. Fig 6).

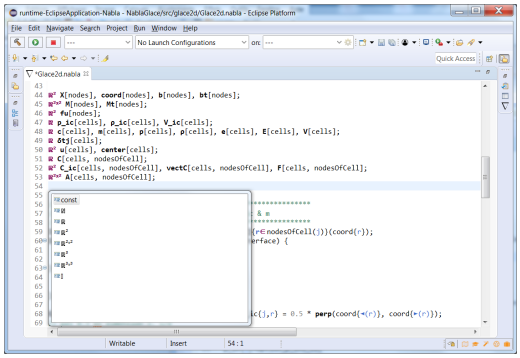

Figure 6. NabLab Editor Contextual proposal

- Syntactic errors highlighting and quick-fixes. Errors are marked with a red cross and an explanatory message is displayed. Eclipse quick-fixes functionality has been used to help users to solve the problem when it is possible. For example, in the following example, the iterator used is misfit (we loop over cells with a j iterator and the r iterator is used as reference for the volume variable). A message is displayed to warn about the unresolve reference to iterator r (cf. Fig. 7) and a quick-fix proposes to replace this reference by one of the available iterators (cf. Fig. 7). This solves the problem immediately.

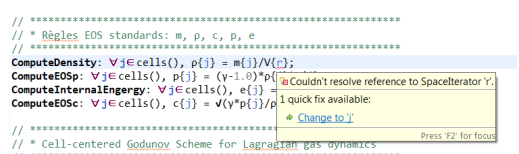

Figure 7. NabLab Editor Error Highlighting and Quick-fix mechanism

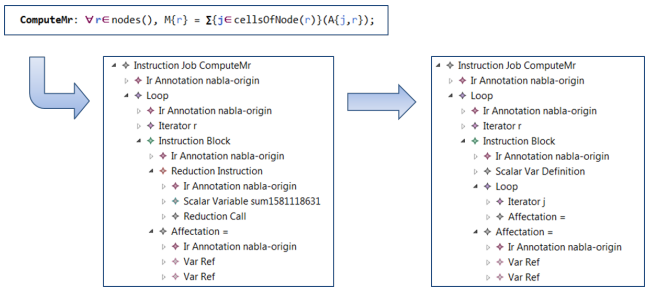

Figure 9. Example of IR transformation

The editor is contextually linked with:

- an outline which offers a synthetic view of the jobs in the NabLab file and allows you to simply move around them (cf. window at the bottom left in Fig. 4).
- a LaTeX view which displays a representation in the form of a latex equation for each selected expression (cf. bottom view in Fig. 4).
- a tooltip which exposes for the selected expression, the inferred type (cf. Fig. 8).

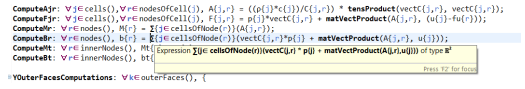

Figure 8. NabLab Editor tooltype

The NabLab metamodel does not require the user to enter the logical times (@) but deduces them from the calculation of a data flow graph. This data flow is derived from input / output variables for each job.

A.1.3 IR Metamodel

The IR metamodel is defined within the EMF framework in the Ecore format. When a valid file is saved in the textual editor, Xtext automatically triggers the generator. An IR model corresponding to the source code is then created.

During IR instantiation, the generator picks up the position of each model element in the source file and stores it as an annotation. Thus, it is possible to keep the link between IR and Nabla elements.

Each code generator stores a list of transformation steps to apply to the IR model before triggering model-to-text generation steps. Some of those steps are shared between Nabla and Java generators while others are dedicated to the Nabla language. The steps can be very simple, like UTF8 characters replacement (for languages that do not tolerate them), and others quite complex, like the inner reductions removals. Reductions are mathematical reduction operations on collections like sum or min/max. Inner reductions mean reductions inside reductions or loops. This step consists of replacing these inner reductions by loops. Figure 9 shows an example of IR model before and after this transformation step.

The last transformation step is in charge of calculating the HLT of each job. NabLab provides services for jobs to get input and output variables. They are used to build a data flow graph. The shortest path for each job is calculated using the JGraphT library [11] and stored in the IR model

When all transformation steps have been applied, the textual generator starts from the last IR model, i.e. the model resulting from the last transformation step. Code generators are written with Xtend using templates.

[11] JGraphT, https://jgrapht.org

A.1.4 Data Flow Graph

The corresponding graph is obtained using Sirius (cf. Fig. 10). An outline (cf. window at the bottom left in Fig. 10) shows the entire graph and allows the user to zoom in on a specific part in the main window. For more legibility, filters are used to hide/show the layers presenting the initialization jobs of the variables or those which correspond to the copies of variables of a time n at time n + 1. On this graph, all the cycles are highlighted.

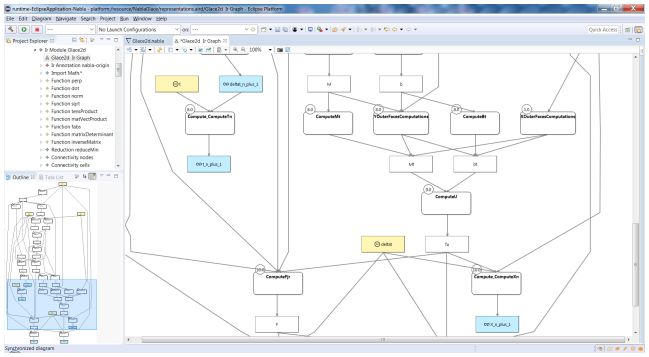

Figure 10. NabLab Data Flow Graph

Once the user has solved these potential cycles (usually by correcting his code or introducing intermediate variables), the system can compute logical times by browsing the graph (a logical time per level of depth as shown in Fig. 10).

A.1.5 Java Code Generator

Through the Xtext generation mechanisms, a Java code generator is automatically triggered when saving a valid NabLab file. For this processing, an intermediate model (IR) is calculated and different model conversion stages are applied on this IR model (cf. Fig. 1).

Java code produced is multithreaded using ParallelStreams (cf. Fig. 11). A main program is also generated which repeats a certain number of iterations of a loop in which the jobs are called iteratively depending on their computed logical time. So this code is directly executable in Eclipse with the Run As Option.

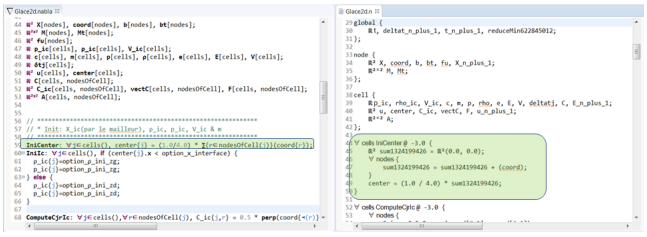

Figure 13. Example of Nabla Code Generation

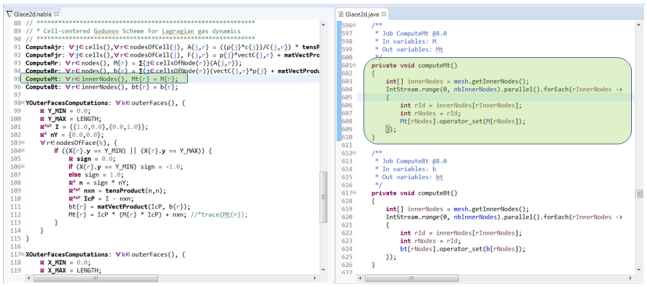

Figure 11. Example of Java Code Generation

The generated Java code contains VTK calls to record the state of each variable as the simulation goes on. For the moment, ParaView[12] and gnuplot [13] are used to visualize execution results and we plan to have an integrated visualization environment in the near future. Similarly, an upcoming version of the environment will allow the user to specify (probably using an execution model attached to the numerical model) which variables he wishes to dump.

A.1.6 Simple LaTeX Generator

A very simple generator is provided to allow users to get the LaTeX formula of the jobs (cf. Fig. 12) using JLatexMath library.

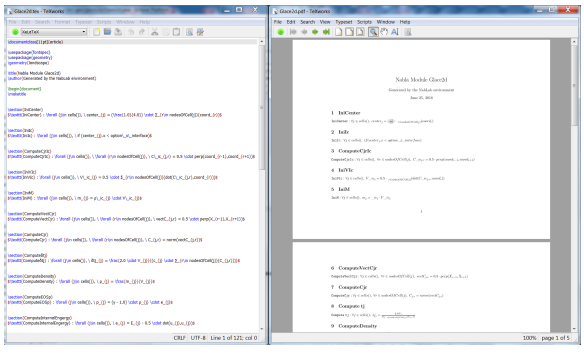

Figure 12. Simple LaTeX Generation

A.1.7 Nabla Code Generator

In the same way as for the Java generation, a Nabla code generator is triggered. Having a look at Nabla file to compare with the equivalent NabLab model, we can see that logical times have been computed thanks to the data flow (cf. Fig. 13). We can also observe the creation of all initialization jobs as it is not possible in the Nabla language for scheduling reasons to specify an initial value for a variable.

The compilation chain that allows to generate code for different backends from the Nabla file has not yet been integrated into the Eclipse environment. However, it is possible to explicitly call the Nabla compilation chain with the Nabla file.

[12] ParaView, https://www.paraview.org

[13] Gnuplot, http://www.gnuplot.info

A.1.8 Monai Use Case

For the last part of this demonstration of Eclipse NabLab environment, we will briefly show an industrial use case: the 1993 tsunami runup onto Monai Valley, Japan, benchmark from the NOAA (National Oceanic and Atmospheric Administration) center for tsunami research [14].

The numerical model is much more complex and the graph is quite difficult to read. But in the end, we were able to reproduce exactly the results of the real simulation code as shown in Fig. 14.

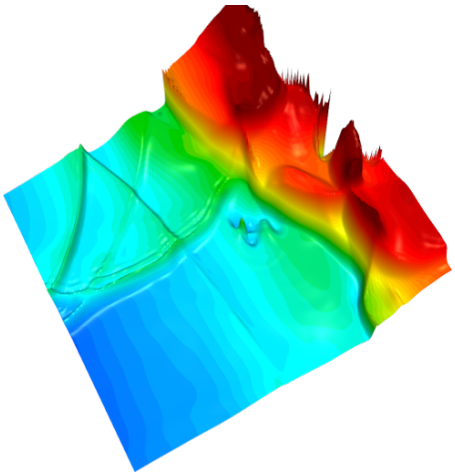

Figure 14. Simulation of tsunami runup onto Monai Valley

A.2 NabLab Development Environment Architecture

This second part of the demonstration will review the architecture of the NabLab environment. The NabLab Development Environment is composed of 3 main parts.

The first part is in charge of the NabLab language and is composed of the *fr.cea.nabla* project, the Xtext project containing the grammar, the validation rules, the type inference, and the LaTeX and IR generators. The validation rules are

[14] Cf. https://nctr.pmel.noaa.gov/benchmark/Laboratory/Laboratory_MonaiValley

Env.	Tools	Languages	Generation
Emacs	Flex, Bison	C, C++, ASM	A lot !
Eclipse	EMF Xtext, Sirius JGraphT, JLaTeXMath	Ecore, Xtext, Xtend	C++ Kokkos
			Java Stream API

Table 1. Used technologies

implemented in Xtend for a seamless integration in the Xtext generated code. The type inference follows the type provider pattern as explained in book [5].

The second part consists of 2 projects in charge of the IR and the language generators. The *fr.cea.nabla.ir* project contains the IR Ecore metamodel, the IR transformation rules and the Java and Nabla generators. The Java generated code is based on basic types and mesh capabilities defined in the *fr.cea.nabla.javalib* project.

The third part is in charge of the environment and is composed of 2 main projects. The first one *fr.cea.nabla.ui* is initialized by Xtext and is in charge of the textual editor and the LaTeX view. It contains the tooltips management (hovers), the content assist, the quickfixes, the outline and the syntax coloring. The second one *fr.cea.nabla.sirius* holds the Sirius graph description and its Xtend services. The other projects of this part are just generated by EMF and Xtext.

Table 1 presents the overall technologies used in the NabLab environment. The first line is dedicated to technologies and tools used by optimization stakeholders while the second line represents those used by tool developers. The C++ Kokkos generation is obtained both by the Nabla tool-chain and by the NabLab direct generation.