A Platform-Independent Tool for Modeling Parallel Programs

Ferosh Jacob, Jeff Gray
Department of Computer Science
University of Alabama
Tuscaloosa, AL 35401
fjacob@crimson.ua.edu, gray@cs.ua.edu

Yu Sun, Purushotham Bangalore
Department of Computer and Information Sciences
University of Alabama at Birmingham
Birmingham, AL 35205
{yusun, puri}@cis.uab.edu

ABSTRACT
Programming languages that can utilize the underlying parallel architecture in shared memory, distributed memory or Graphics Processing Units (GPUs) are used extensively for solving scientific problems. However, from our observation of studying multiple parallel programs from various domains, such programming languages have a substantial amount of sequential code mixed with the parallel code. When rewriting the parallel code for another platform, the same sequential code is often reused without much modification. Although this is a common occurrence, existing tools and programming environments do not offer much support for this process. In this paper, we introduce a tool named PPmodel, which was designed and implemented to assist programmers in separating the core computation from the details of a specific parallel architecture. Using PPmodel, a programmer can identify and retarget the parallel section of a program to execute in a different platform. With PPmodel, a programmer is better enabled to focus on the parallel section of interest, while ignoring other parallel and sequential sections in a program. The tool is explained by example execution of the parallel section of an OpenMP program for the circuit satisfiability problem in a cluster using the Message Passing Interface (MPI).

Categories and Subject Descriptors
F.1.2 [**Modes of Computation**]: Parallelism and concurrency

General Terms
Algorithms, Performance, Languages.

Keywords
Parallel programming, model-driven engineering.

1. INTRODUCTION
With the popularity of multi-cores and multi-processors, the need to understand parallel programming techniques will continually emerge as a necessary skill needed by future software engineers. For next generation applications, programmers will be required to adapt to a new style of programming to utilize the parallelism in the processors available to them. In addition to the platform in which code is executed, the execution time of parallel programs can vary based on the logic of the solution as well as the problem size. This gives rise to the need for creating and maintaining multiple versions of the same program intended for different problem sizes. The existing programming styles involve creating programs that have parallel and sequential sections. The parallel sections are either platform-specific or architecture-specific; these details often make parallel programming challenging for average programmers. Often, the parallel section is deeply tangled with the sequential section, which can affect the productivity of the programmer. Creating a new version for an existing program targeted to a new platform requires copying the sequential section, rewriting the parallel section, and making necessary modifications to bridge the new parallel code with the existing sequential code. A parallel programming style that is void of any machine-specific details, yet can aid programmers in bridging the parallel and sequential sections of code, has the potential to offer much benefit to future software engineers.

A survey of general-purpose computation on graphics hardware reveals that General-Purpose GPU (GPGPU) algorithms continue to be developed for a wide range of problems [1]. To use GPGPUs outside of their intended context, much work is required to make such algorithms accessible to a broader range of software developers. Abstractions in parallel programming languages and directives or annotations in sequential code have shown initial promise in reducing some of the burdens of parallel programming. However, even with all of these advances, parallel programming still requires skill beyond that possessed by an average programmer. There are several challenges that emerge when designing parallel programs.

1.1 Why are parallel programs long?
A programmer writing code for an alternate parallel solution should focus on defining the parallel block representing the code to be executed by each thread or process. In an ideal situation, he or she should not have to delve into the sequential part of the program. Instead, a programmer should be given the flexibility to port their program into another language by just rewriting the parallel section of the program. Currently, many long parallel programs have short parallel sections and long sequential sections as revealed by our analysis. By separating the short parallel sections from the long sequential sections, programmers are freed from the additional task of understanding the complete code, and allowed to focus on the core parallel computation.

An analysis was conducted on ten OpenMP programs collected from various domains. In an OpenMP program, a parallel block is defined by a compiler directive starting with `#pragma omp parallel`. The details of the analysis are shown in Table 1. The first column of the table shows the name of the program, second column shows the total Lines Of Code (LOC), third column

49th ACM Southeast Conference, March 24-26, 2011, Kennesaw, GA, USA. Copyright 2011 ACM 978-1-4503-0686-7 $5.00.

shows the total LOC of the parallel block, and the last column shows the number of parallel blocks in each program. The LOC of the parallel blocks to the total LOC of the program ranges from 2% to 57%, with an average of 19% for the selected ten OpenMP programs. To create a different execution environment for any of these programs, more than 50% of the total LOC would need to be rewritten for most of the programs. Currently, programmers manually copy/paste or rewrite the sequential section in the parallel program. To the best of our knowledge, there is no current support for creating or maintaining the sequential section while rewriting the parallel program for a new platform. In this paper we explore the possibilities to automate this process.

Table 1. Parallel sections in OpenMP programs

No	Program Name	Total LOC	Parallel LOC	No. of blocks
1	2D Integral with Quadrature rule	601	11 (2%)	1
2	Linear algebra routine	557	28 (5%)	4
3	Random number generator	80	9 (11%)	1
4	Logical circuit satisfiability	157	37 (18%)	1
5	Dijkstra's shortest path	201	37 (18%)	1
6	Fast Fourier Transform	278	51 (18%)	3
7	Integral with Quadrature rule	41	8 (19%)	1
8	Molecular dynamics	215	48 (22%)	4
9	Prime numbers	65	17 (26%)	1
10	Steady state heat equation	98	56 (57%)	3

1.2 Which programming model to use?

In the current state of practice, in order to write a program that will execute a block of code in parallel, a programmer must learn a parallel programming language and supporting libraries to describe the computation. After the program is executed, the programmer must compare the results with some other baseline representation of the computation in order to optimize performance.

As an example, the shared memory (OpenMP) solution may perform better for small problem sizes compared to using a GPU (CUDA), which has a high threshold because of the expensive data transfer operations. As the problem size increases, the GPU programs become faster than shared memory programs. The problem size for which a GPU performs better than a CPU differs with each application. In the current practice, programmers have to manually create the new version which may share a substantial amount of code with the original version. Usually the programming models require some additional code to setup the execution in addition to actually specifying the execution. In the case of OpenMP programs, additional code is required for the library declarations; for MPI, the library declarations are initialization and finalization code for process instance and process size variables.

1.3 Solution: Modeling parallel programs

Models are often created as a higher level abstraction of some system design [2]. Our research has led us to the realization of the benefits of adopting a modeling approach to address the challenges of parallel programming. The result of our work is a modeling environment called PPmodel, which has two goals: 1) to separate the parallel sections from the sequential parts of a program, which allows a programmer to focus more on the parallelism, and 2) to define a new execution strategy for the computation intensive part of the program without changing the flow of the program. Using PPmodel, the parallel part of the program can be separated from the sequential part of the program, re-designed, and then regenerated. With our approach, programmers can switch between technical solution spaces (e.g., MPI, OpenMP, CUDA and OpenCL) without actually changing the core sequential part of a program. Our approach allows a programmer to concentrate more on the essence of the parallelization, rather than focusing on the accidental complexities of language-specific details. Section 2 describes PPmodel and the design that led to the approach introduced in Section 3. Related works are overviewed in Section 4 and the paper is concluded in Section 5 by enumerating some possible extensions of the current work.

2. WORKING WITH PPMODEL

In this section, PPmodel is explained from a user's perspective with the Satisfy problem example. The Satisfy problem has only one parallel block, with each block determining whether the current value satisfies the given circuit. The parallel program written in OpenMP can be executed in a cluster using MPI by rewriting the parallel part of the program while keeping the sequential part of the program untouched. The following sections introduce the three stages of using PPmodel.

2.1 Model creation for Satisfy problem

PPmodel is implemented as a modeling editor in Eclipse (please see Section 3 for implementation details). As shown in Figure 1, the model is created through a two-step process: 1) Model representation of the program is generated by right-clicking the program "Satisfy.c" in "Project Explorer" and selecting "ModelMe," and 2) From the model representation, a visual representation of the program for a generated model is created by right-clicking the generated file, "_satisfy.parallelsystem" and selecting "DrawMe." The model representation illustrates the parallel blocks of the program and the visual representation links the program to the target environment. It is possible to have different visualizations for the same program. The visualization model is a representation of the program in a particular configuration, specifying the target platform for each block. The two stages are explained in more detail in the following subsections.

2.1.1 Creating a model from an existing program

On selecting the option "ModelMe," two folders are created: 1) "model," which is a folder for model related files, and 2) "generated," which is a folder for generated source files. From the selected program, the parallel blocks are identified and the information is stored in the file "_satisfy.parallelsystem" in the model folder. A copy of the program is placed in the "generated" folder and renamed as "_satisfy.c". The modifications that occur as the result of the modeling activity are applied to this file.

2.1.2 Creating a visual representation of model

On selecting the option "DrawMe," a new file named "_satisfy.parallelsystem_diagram" is created. Upon opening the file, a view similar to Figure 2 is presented to the user ("MPI cluster" node and the connecting link can be modeled).

2.2 Modeling the Circuit Satisfy problem

Modeling helps the programmer specify the automatically detected blocks to execute in a different platform. The modeling environment as shown in Figure 2 has a palette that consists of Objects and Connections. The Objects represent the set of nodes for modeling and Connections link a parallel block node with any of the execution devices. In Figure 2, an execution device can be a GPU device, MPI nodes, or even an unknown device (e.g., Xdevice). After creating a link between an execution device and parallel block, a new file is created in the "generated" folder. The name of the file is formed from the first four characters of the parallel block name and first three characters of the execution device name.

2.3 Code generation for Satisfy problem

In the editor, upon selecting "satisfy.parallelsystem_diagram" there is an option for code generation. This integrates the code written in "main_MPI.c" with the program in "_satisfy.c". Code integration involves replacing the OpenMP code with the newly added MPI code, adding libraries to execute MPI code, and some code to initialize MPI-specific variables (e.g., process identifier, number of processes). An overview of the generated code using the tool is shown in Listing 1. The execution plot for MPI and OpenMP implementations is shown in Figure 3.

3. IMPLEMENTATION DETAILS OF PPMODEL

A high-level diagram of PPmodel is shown in Figure 4. Domain-Specific Modeling (DSM) [3] is applied to facilitate the tool implementation for the modeling environment. DSM is a Model-driven Engineering [4] methodology that uses a Domain-Specific Modeling Language (DSML) [5] to declaratively define a system using specific domain concepts, directly compute and analyze the domain through model interpreters, and automatically generate the

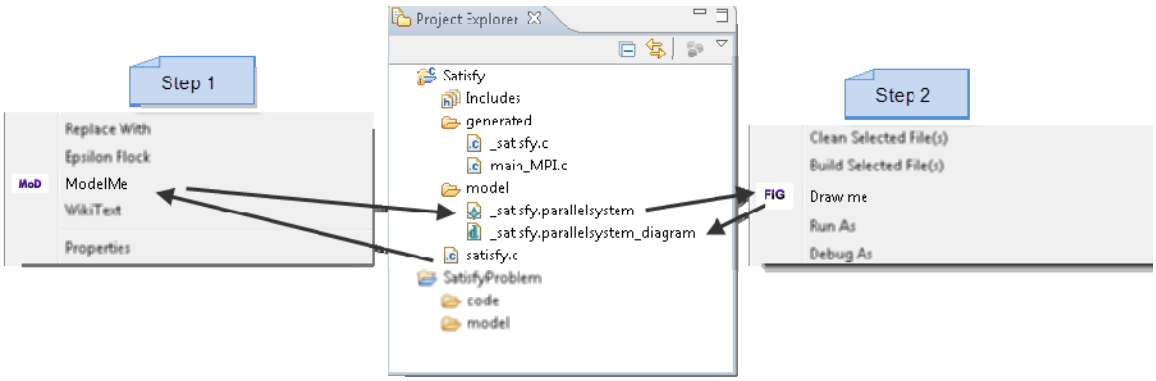


Figure 1 Modeling the Satisfy problem

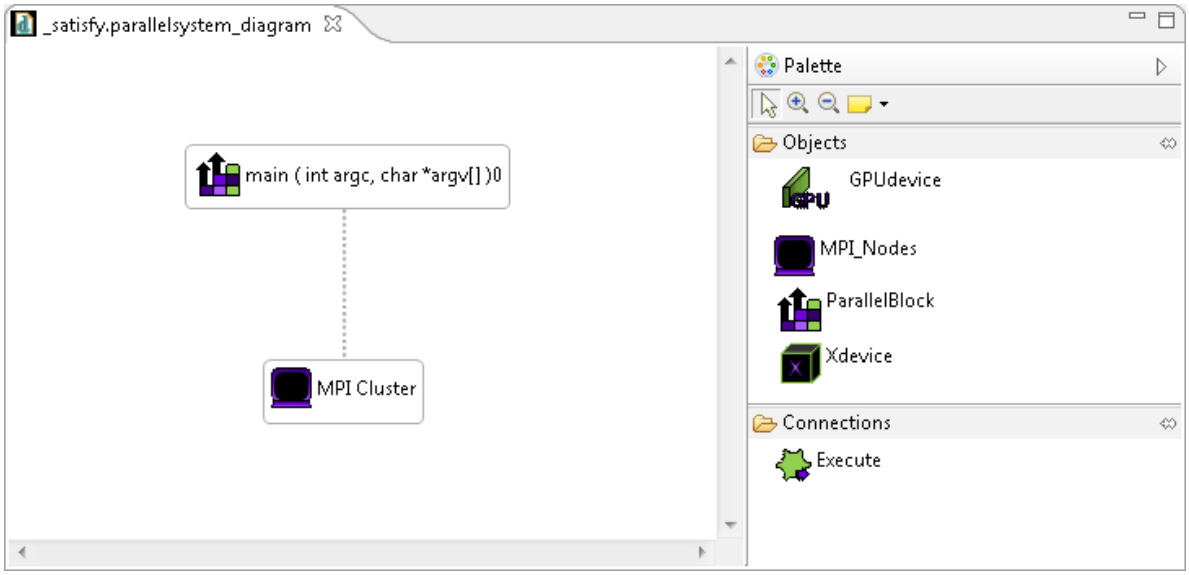
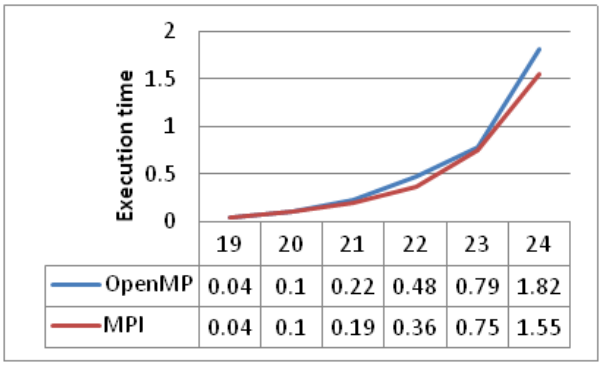


Figure 2 Modeling environment for the Satisfy program

desired software artifacts by model transformation engines and code generators. Considering the description about the structure of parallel programs as a specific application domain, the formal specification for this domain – the metamodel - must be defined first.



	19	20	21	22	23	24
OpenMP	0.04	0.1	0.22	0.48	0.79	1.82
MPI	0.04	0.1	0.19	0.36	0.75	1.55

Figure 3 Comparison of OpenMP/MPI wth problem size

The metamodel [6] specifies the entities, associations and constraints for the specific domain, which can be used to generate a modeling environment, enabling users to build concrete models and specify the structure of parallel programs. The models conform to the definition of the metamodel and can be used in computation, analysis and generation of other software artifacts.

We used the Graphical Modeling Framework (GMF), a powerful DSM tool in Eclipse, to support the implementation. The metamodel for GMF consists of three components: 1) the abstract syntax of the structure of parallel programs is captured in the domain model, 2) the concrete syntax (i.e., the visualization with icons) is specified in the graphical model, and 3) the tooling model defines the functions of the editing environment (e.g., the palette, creation buttons, actions). This metamodel is applied to create the PPmodel modeling editor automatically. The separation of domain model, graphical model and tooling model realizes an extensible framework, so that any changes to the visualization will not affect the domain concept definition and the editing environment, and any alteration of the domain model or tooling model will not force the other two to change.

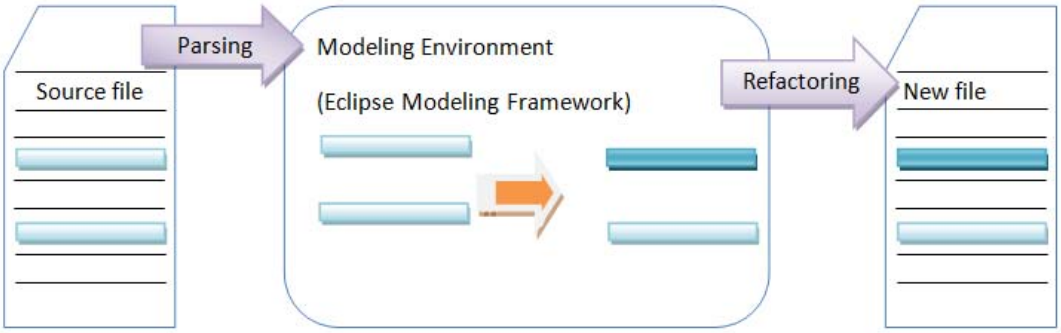


Figure 4 High level diagram of PPmodel

Listing 1. Final MPI program generated using PPmodel

```
# include <stdlib.h>
# include <mpi.h>                                   /* Include MPI library.*

int main ( int argc, char *argv[] ){
  int id, p;                                        /* Initialize MPI variables with other variables.*/
  MPI_Init ( &argc, &argv );
                                                    /* Determine the rank and number of processors.*/
  MPI_Comm_rank ( MPI_COMM_WORLD, &id );
  MPI_Comm_size ( MPI_COMM_WORLD, &p );
                                                    /* Sequential code from OpenMP Satisfy version.*/
……
if ( id == 0 )                                      /* Newly inserted code
  ilo = 0;
...
.
……
                                                    /* Sequential code from OpenMP Satisfy version.
                                                    /* Terminate MPI.*/
  MPI_Finalize ( );
}
```

In PPmodel, the initial creation of models is not completed by performing the basic editing operations from scratch. Instead, a C-parser has been developed to separate the OpenMP preprocessor statements from the rest of the C program and a data structure is constructed with the information regarding the variables and location of the preprocessor statements. Each block can be uniquely identified with the function name and an identifier representing the order of the block in that function. Using the generated APIs to access and manipulate models in GMF, the corresponding model elements and links are created based on the collected information.

The initialized model can then be edited by users in order to replace and modify the parallel components. Based on the data in the modified models, the original code will be refactored to replace the previous implementation with newly generated code.

4. RELATED WORKS

There have been a few modeling efforts in the parallel programming domain. The CODE[1] [7] programming language is based on a generalized dependency graph to express the computation in a unified parallel computation model without any implementation details. In comparison with PPmodel, CODE is a graphical programming environment, but PPmodel is a complete modeling tool to create parallel programs from sequential or parallel programs written for another target platform. GASPARD [8] is another visual parallel programming environment supporting task and data parallelism.

Rather than providing abstraction for a language from one parallel programming model to the other, modeling the parallel part of the program makes this work unique. OpenMP to GPGPU [9] converts OpenMP programs to CUDA code. However, the goal of our work is to express the parallel part of a program in a way that is separate from the sequential part to allow the programmers to focus more on the parallel problem than the program as a whole. Other related works include program transformations from sequential to parallel and abstractions in parallel programs. Many of the sequential to parallel converters [10] use data dependency and refactoring approaches that are similar to our current implementation. Many efforts [11], [12], [13], [14] were done on the abstraction of GPU programs. Most of the work was concentrated on a particular device or language; [11],[12], and [13] all target CUDA. CGiS [14] provides support for multiple devices. Some of the features include parallel control structures and special vector operators.

PPmodel is also motivated by the numerous successful DSM applications, which involve creating models for specific domains and generating low-level software artifacts from the models automatically. For instance, a DSML is defined in [15] to specify a home automation system and improve its quality and portability by generating low-level control implementation from the models; [16] focuses on the physical workflow domain and models the specific working processes to enable model-based computation and analysis. Our work is different from these typical DSM applications in that the creation of our models is based on a reverse engineering approach, which captures the essence of the domain from the program source code. Although performing reverse engineering on text has also been applied in some other model-based applications, such as transforming the SPL (a telephony language) code to CPL code (another telephony language) by model transformation [17], and homogenizing different textual code clone analysis results into a uniform format [18], their main purpose is to realize data interoperability between two different domains, rather than supporting the modification of the parallel programs that occur in the same domain.

[1] http://www.cs.utexas.edu/users/code

5. CONCLUSION

In this paper, we presented a tool named PPmodel that can be to separate the parallel part from the sequential part of a program. Using the modeling framework, programmers can execute the parallel blocks in a different platform without actually rewriting the program. The approach is independent of any platform or language and hence it can be extended to any language. In this paper, an OpenMP program written to solve the Circuit Satisfiability problem was redesigned to execute in multiple nodes using MPI.

The tool currently can model only C OpenMP programs and generate target code for the MPI library. PPmodel can be extended to support a GPU programming language like CUDA. Similar implementations can be created for other programming languages and platforms. The programming language determines the refactoring framework to use and the platform decides the code to be inserted or refactored.

6. REFERENCES

[1] Owens, John D., Luebke, David, Govindaraju, Naga, Harris, Mark, Krüger, Jens, Lefohn, Aaron E., and Purcell, Timothy J. A survey of general-purpose computation on graphics hardware. Computer Graphics Forum, 26 (March 2007), 80–113.

[2] Atlee, Joanne M., France, Robert, Georg, Geri, Moreira, Ana, Rumpe, Bernhard, and Zschaler, Steffen. Modeling in Software Engineering. In 29th International Conference on Software Engineering (Minneapolis, MN May 2007), 113-114.

[3] Gray, Jeff, Tolvanen, Juha-Pekka, Kelly, Steven, Gokhale, Aniruddha, Neema, Sandeep, and Sprinkle, Jonathan. In Domain-specific modeling: Handbook of dynamic system modeling. CRC Press, 2007.

[4] Schmidt, Douglas. Model-driven engineering. IEEE Computer, 39 (2006), 25-32.

[5] Lédeczi, Ákos, Bakay, Árpád, Maróti, Miklós, Völgyesi, Péter, Nordstrom, Greg, Sprinkle, Jonathan, and Karsai, Gábor. Composing domain-specific design environments. IEEE Computer, 34 (2001), 44-51.

[6] Atkinson, Colin and Kuhne, Thomas. Model-driven development: A metamodeling foundation. IEEE Software, 20 (2003), 36-41.

[7] Browne, J.C., Azam, Muhammed, and Sobek, Stephen. CODE: A unified approach to parallel programming. IEEE Software, 6 (July 1989), 10-18.

[8] Devin, Florent, Boulet, Pierre, Dekeyser, Jean-Luc, and Marquet, Philippe. GASPARD: A visual parallel programming environment. In International Conference on Parallel Computing in Electrical Engineering (Warsaw, Poland September 2002), 145.

[9] Lee, Seyong, Min, Seung-Jai, and Eigenmann, Rudolf. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. SIGPLAN Notes, 44 (February 2009), 101-110.

[10] Allen, Randy and Kennedy, Ken. Automatic Translation of FORTRAN Programs to Vector Form. ACM Transactions on Programming Languages and Systems, 9 (1987), 491--542.

[11] Ueng, Sain-Zee, Lathara, Melvin, Baghsorkhi, Sara S., and Hwu, Wen-Mei W. CUDA-Lite: reducing GPU programming complexity. In Proceedings of the International Workshop on Languages and Compilers for Parallel Computing (Edmonton, Canada July 2008), 1-15.

[12] Breitbart, Jens. CuPP - A framework for easy CUDA integration. In Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (Rome, Italy May 2009), 1-8.

[13] Han, Tianyi David and Abdelrahman, Tarek S. hiCUDA: a high-level directive-based language for GPU programming. In Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units (Washington, D.C March 2009), 52-61.

[14] Fritz, Nicolas, Lucas, Philipp, and Slusallek, Philipp. CGiS: a new Language for data-parallel GPU programming. In In Proceedings of the 9th International Workshop on Vision, Modeling, and Visualization (Stanford, CA November 2004), 241–248.

[15] Buendía, Manuel Jiménez, Rosique, Francisca, Sánchez, Pedro, Álvarez, Bárbara, and Iborra, Andrés. Habitation: A domain-specific language for home automation. IEEE Software, 26 (2009), 30-38.

[16] Mathe, Janos L., Martin, Jason B., Miller, Peter et al. A model-integrated, guideline-driven, clinical decision-support system. IEEE Software, 26 (2009).

[17] Jouault, Frederik, Bézivin, Jean, Consel, Charles, Kurtev, Ivan, and Latry, Fabien. Building DSLs with AMMA/ATL, a case study on SPL and CPL telephony languages. In In Proceedings of the 1st ECOOP Workshop on Domain-Specific Program Development (DSPD) (Nantes, France July 2006), 4.

[18] Sun, Yu, Demirezen, Zekai, Jouault, Frédéric, Tairas, Robert, and Gray, Jeff. A model engineering approach to tool interoperability. In In 1st International Conference on Software Language Engineering (SLE), Tool Demonstration (Toulouse, France September 2008), 178-187.

[19] Atkinson, Colin and Kuhne, Thomas. Model-driven development: A metamodeling foundation. IEEE Software, 20 (2003), 36-41.