Automatic Synthesis of Heterogeneous CPU-GPU Embedded Applications from a UML Profile

Federico Ciccozzi

School of Innovation, Design and Engineering - IDT
Mälardalen University, Västerås, Sweden
federico.ciccozzi@mdh.se

Abstract. Modern embedded systems present an ever increasing complexity and model-driven engineering has been shown to be helpful in mitigating it. In our previous works we exploited the power of model-driven engineering to develop a round-trip approach for aiding the evaluation and assessment of extra-functional properties preservation from models to code.
In addition, we showed how the round-trip approach could be employed to evaluate different deployment strategies, and the focus was on homogeneous CPU-based platforms. Due to the fact that the assortment of target-platforms in the embedded domain is inevitably shifting to heterogeneous solutions, our goal is to broaden the scope of the round-trip approach towards mixed CPU-GPU configurations. In this work we focus on the modelling of heterogeneous deployment and the enhancement of the current automatic code generator to synthesize code targeting such heterogeneous configurations.

Keywords: model-driven engineering, code synthesis, heterogeneous systems, embedded systems, CHESS-ML, UML, MARTE, ALF

1 Introduction

The complexity of embedded systems is increasing at high pace and therefore development processes based on code-centric approaches tend to become highly complex and error-prone thus demanding the adoption of more powerful and automated mechanisms. Model-Driven Engineering (MDE) has been proven capable of reducing development complexity through the ability of abstracting from details not needed at design level and that are typical of code-centric approaches. More specifically, the focus shifts from hand-written code to models by which the system can be early analysed and validated; furthermore the application is meant to be automatically generated from them through the employment of model transformation mechanisms.

Automating the code generation phase is a task considered inescapable in order to make of MDE an eligible approach to substitute code-centric approaches, especially in industry. Powerful code generation mechanisms can improve quality and maintainability of the final application as well as enforce its consistency to the source models. In this way, results from analysis performed at model level are more likely to be valid at code level too (and the other way around). Development-wise, effective code generation can positively affect time-to-market as well as overall costs and risks. Additionally, generated code is meant to achieve higher and more consistent quality than hand-written

code with respect to errors, maintainability and readability.

In [1] we proposed an automated round-trip engineering support for MDE of embedded systems with focus on the preservation of extra-functional properties (EFPs). The round-trip support is made of four core steps. The first step consists in modelling the system through a structural design in terms of components, a behavioural description by means of state-machines and action code, as well as a deployment model describing the allocation of software components to operating system's processes. Then, from the information contained in the design model, we automatically generate full functional code to be run as a singleprocess[1] application on singlecore CPU-based platforms. When the application is generated, we monitor its execution on the target platform and measure selected EFPs. Then gathered values are back-propagated to the design model and, after their evaluation, the models can be manually tuned to generate more resource-efficient code.

The round-trip support has been validated in industrial settings where the necessity to extend the generation of code to account more complex platforms arose [2]. Therefore, we proposed and developed preliminary extensions in order to enable the generation of multiprocess applications on CPUs, that we employed for deployment assessment in [3]. Nevertheless, the expectation for embedded systems to be able to process vast amounts of data, even in real-time, is spreading among several different domains and a possible solution to make embedded systems fulfil this expectation is the adoption of hardware technologies based on heterogeneous configurations [4]. A common scenario is represented by mixed CPU-GPU configurations where, e.g., input data comes into a CPU, which in turn may exploit one or more GPUs as coprocessors for parallel processing of large blocks of data.

The crossover from homogeneous to heterogeneous platforms brings along new research challenges ranging from modelling to coding of the embedded system. In our case, while generally increasing performances of the resulting applications, the introduction of heterogeneity adds additional complexity in modelling and generating the application. In fact, since different processing units (e.g., CPUs and GPUs) usually employ different formalisms and mechanisms for code execution, the design models should contain the information needed by the generation process to map model entities to code artefacts written in different target languages as well as to generate the communication code needed for the interaction between CPUs and GPUs.

Pursuing this direction, the contributions of this work are (i) the identification of the modelling means to specify heterogeneous deployment especially regarding the allocation of single component functions to GPU cores, and (ii) the extension of the code synthesis to generate heterogeneous applications to be run on mixed CPU-GPU heterogeneous platforms. Moreover, we aim at maintaining a deployment-agnostic specification of the functional characteristics of the systems, while modelling the platform and deployment specific details as extra-functional annotations that drive the generation of the heterogeneous application.

The remainder of the paper is organised as follows. Section 2 describes the scope of the work and its contextual delimitation. The relation of our contribution to the state

[1] We refer to *process* as an independent execution unit that only interacts with other processes via interprocess communication mechanisms (managed by the operating system).

of the art is given in Section 3. Section 4 depicts the means we identified for modelling heterogeneous allocation and deployment as well as for enabling the synthesis of applications to be run on mixed CPU-GPU platforms. The paper is concluded in Section 5 with a discussion of the means for the proposed solution as well as current limitations and planned future work.

2 Context

In our work we employ the CHESS Modelling Language (CHESS-ML) [5], defined within the CHESS project as a UML profile [6], including subsets of the MARTE [7] and SysML profiles. The CHESS-ML is part of the CHESS framework[2] which leverages the Papyrus [8] Project [9], an open-source environment for editing Eclipse Modeling Framework (EMF) models and particularly supporting UML and related profiles such as SysML and MARTE, on the Eclipse platform. CHESS-ML allows the specification of a system together with relevant EFPs such as predictability, dependability and security. Moreover, it supports a development methodology expressly based on separation of concerns; distinct design views address distinct concerns. In addition, CHESS-ML supports component-based development as prescribed by the UML Superstructure [10].

For the functional definition of the system in CHESS, UML component and composite component diagrams are employed to model the structural aspects while statemachines are used to express functional behaviour. Action Language for Foundational UML (ALF) [11] is used to define the actual behaviour of the component operations (also addressed in this paper as functions). In this way, we reach the necessary expressive power to be able to generate the full implementation directly from the functional models with no need for manual fine-tuning of the code after its generation. In compliance with the principle of separation of concerns adopted in CHESS-ML, the functional models are decorated with extra-functional information thereby ensuring that the definition of the functional entities is not altered.

The target languages are C++, for code portions running on CPU, and CUDA C/C++ [12] for code portions to be deployed on GPU. The application is run on OSE, a commercial and industrial real-time operating system developed by Enea [13], which provides the concept of direct and asynchronous message passing for communication and synchronisation between tasks. This allows tasks to run on different processors or cores, utilising the same message-based communication model as on a single processor. This programming model provides the advantage of avoiding the use of shared memory among tasks. In OSE, the runnable real-time entity equivalent to a task is called *process*, and the messages that are passed between processes are referred to as *signals* (thus, the terms *process* and *task* in this paper can be considered synonyms).

3 Related Work

Overall, a number of different approaches have been proposed for the generation of multicore systems, starting from different abstraction levels, such as in [14], [15], [16].

[2] Download at `http://www.math.unipd.it/~azovi/CHESS/CHESS_3.2/`.

Nevertheless, the input needed for these approaches is at a very low abstraction level and the output is meant to complement elsewhere generated or already existing code artefacts. In our solution the whole implementation is meant to be generated from the design models in one single transformation process.

Different approaches aiming at achieving code generation for embedded systems can be found in the literature but despite the numerous attempts, this still represents an open research issue especially when it comes to the generation of code to be run on heterogeneous platforms. The most concrete attempt to heterogeneous code generation for CPU-GPU configurations has been proposed by Rodrigues et al. in [17] where the authors define a code generation process from UML–MARTE models to OpenCL. While similar to ours in terms of the underlying idea, the approach proposed in this work assumes a more detailed modelled platform information and it targets only GPU-related code. In the approach we propose the aim is to provide an environment that allows end-users to freely model systems and thereby allocate components and their functions to either CPUs or GPUs leaving the burden of communication code between CPU and GPU to the code generation process.

In [18] the authors aim at automating the task of determining the appropriate memory usage as well as the coding of data transfer between memories. This work could be exploited in the next steps of our research work towards possible optimizations of the code to be generated. Additionally, attempts to automatically generate C from CUDA have also been proposed, as in [19, 20], and they could represent a useful guidance for definition and implementation of our ALF to CUDA C/C++ transformation chain.

4 Modelling and Synthesizing the Heterogeneous Application

In this section we define the means we identified for modelling heterogeneous deployment and thereby enabling the generation of applications to be run on mixed CPU-GPU platforms from the design models.

4.1 Modelling Heterogeneous Deployment

As previously mentioned, the functional definition of the system is modelled in CHESS-ML by means of UML components as well as state-machines. Moreover ALF is leveraged to implement the actual behaviour of the components' operations. Note that the functional specification of the system is meant to remain deployment-agnostic, thus leaving platform and deployment details to be modelled as extra-functional decorations as described later in this section.

The first step to enhance the code generation from models to target heterogeneous platforms is to identify the modelling means for those points of variability that cannot be embedded in the transformation process. More specifically, while we already allow the allocation of components to CPU cores via OSE processes, we want to enable the modeller to define the allocation of single component functions to GPU cores as well.

In Fig. 1 we show a simplified version of the vision system from an Autonomous Underwater Vehicle (AUV), focusing on functional definition and allocation to hardware resources of components and functions. The system is represented by the composite component `Vision_System`, containing components `VisionManager_impl`

of type VisionManager, FrontCamera and BottomCamera of type StereoCamera and representing the two camera systems of the AUV, and the Filter component of type StereoMatcher. The components are allocated to different OSE processes, i.e., Process_A, Process_B, Process_C, Process_D, of type OSE_Process and stereotyped with MARTE's ≪*MemoryPartition*≫. The processes are then allocated to the two-cores CPU_chip defined as MARTE's ≪*hwProcessor*≫.

The allocation is modelled by means of MARTE's ≪*allocated*≫ (on allocated components and resources) and ≪*allocate*≫ (dotted arrows between elements with allocation relationship). Moreover, the core ID on which the process is meant to be allocated is specified through a MARTE's ≪*nfpConstraint*≫ called Core_ID. Let us suppose that we want to allocate Filter's function *f_sum()* to the core with $ID = 1$ of the GPU chip. This is done by:

- Modelling an ≪*allocate*≫ link between Filter and the ≪*hwProcessor*≫ GPU_chip;
- Specifying the function (i.e., *f_sum()*) to be allocated to the GPU core through a decoration of the ≪*allocate*≫ link with MARTE's ≪*assign*≫;
- Decorate the ≪*allocate*≫ with a ≪*nfpConstraint*≫ called Core_ID for specifying the core on which to allocate the function, a ≪*nfpConstraint*≫ called GridD for the definition of the grid dimension, and a ≪*nfpConstraint*≫ called BlockN for the definition of the thread block.

Note that, while in Fig. 1 all the details (functional, extra-functional and deployment) are exposed in a single view, in the actual CHESS-ML model they are placed in separated views (i.e., functional, extra-functional, deployment) to enforce separation of concerns.

4.2 Generating the Application

The generation process is constituted by a set of model transformations. Starting from the CHESS-ML model of the system under development, we translate the structural definition from component, composite component and state-machine diagrams through a model-to-model transformation chain[3]. Regarding the translation of state-machines, our approach resembles the *state design pattern*, as defined in [21], and considers the component owning the state-machine as the *context* for the related states.

As prescribed in its specification [11], the execution semantics for ALF is specified by a formal mapping to foundational UML (fUML) [22]. There are three prescribed ways in which ALF execution semantics may be implemented [11], namely *interpretive execution*, *compilative execution* and *translational execution*. In our code generation, we provided a solution towards the ***translational execution*** of ALF, focusing on the *minimum conformance* level (as defined in [11]), by means of model-to-model transformations which are introduced in [23].

In the followings we describe the generation principles to achieve the needed communication code to call functions allocated to GPUs (i.e., *f_sum()*) from functions allocated to CPUs, as well as the code for *f_sum()*, specified in the model in terms of ALF, to CUDA C/C++ code. Let us suppose that the function *caller()* in FrontCamera

[3] More details on the transformation process can be found in [1]; the complete description of generation process and involved artefacts for multiprocess applications is currently under submission.

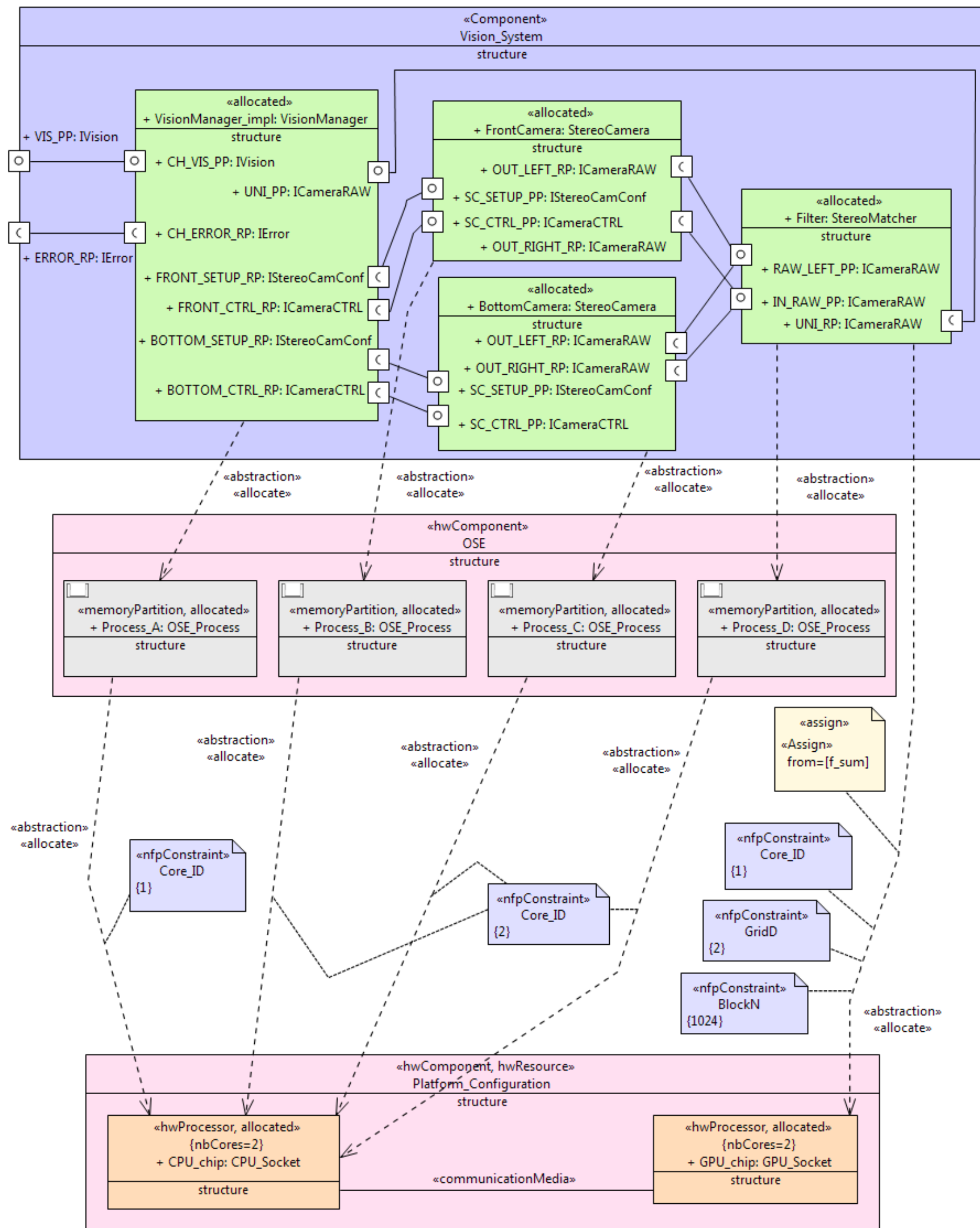

Fig. 1: Modelling of Heterogeneous Deployment in Papyrus

calls `Filter`'s *f_sum()* which is allocated to the GPU core with $Core_ID = 1$; the code related to the two functions is depicted in the following ALF-like code snippet.

```
1 // caller in ALF-like
2 public caller(in p1[], in p2[]){
3         int N = 10000;
4         int res[N];
5         Filter.f_sum(p1, p2, N, res);
6 }
7
8 // f_sum in ALF-like
9 public f_sum(in a[], in b[], in N, out result[]){
10        int i = 0;
11        for( i in N )
12                result[i] = a[i] + b[i];
13 }
```

Code 1.1: caller() and f_sum() functions in ALF-like

As depicted in the code snippet, function *f_sum()* computes the sum of the arrays given as input parameters (*in a[]*, *in b[]*) and put the result in the output parameter array (*out result[]*); for simplicity reasons we statically define the arrays' length as *N*. On the one hand, the function *caller()* has to be generated as standard C++ function, and therefore can be handled by the code generator in [1]. On the other hand, *f_sum()*, which is deployed on a GPU core, needs to be translated into CUDA C/C++ and communication code has to be generated in order to allow *caller()* to call it.

The idea is to enhance the generation process to produce (i) communication code whenever the code generator runs into a call (*caller()*) to a function (*f_sum()*) allocated to a GPU core, and (ii) the parallel code which corresponds to the sequential ALF code defined for it (*f_sum()*). Code 1.2 depicts the generated C++ *caller()* function as well as the generated CUDA C/C++ code in terms of the kernel *f_sum()*, representing the translated body of *f_sum()*, and *f_sum_caller()*, which represents the function implementing the communication code needed to call the kernel.

```
1 ——————— .cpp file ———————
2 // caller in C++ (.cpp file)
3 void caller(int *p1, int *p2){
4         int N = 10000;
5         int res[N];
6         Filter.f_sum_caller(p1, p2, N, res);
7 }
8
9 ——————— .cu file ———————
10 // f_sum() in CUDA C++
11 __global__ void f_sum(int *a, int *b, int *result, int N) {
12        // determine in which thread we are
13        int i = threadIdx.x + (blockIdx.x * blockDim.x);
14        // parallel sum
15        if (i < N) result[i] = a[i] + b[i];
16 }
17
18 // f_sum_caller to enable calls to f_sum()
```

```
19 void f_sum_caller(int *p1, int *p2, int N, int *res)
20 {
21         // pointers to device parameters
22         int *p1_d, *p2_d, *res_d;
23         // size, in bytes, of each array
24         size_t bytes = N*sizeof(int);
25         // allocate memory for parameters
26         cudaMalloc(&p1_d, bytes);
27         cudaMalloc(&p2_d, bytes);
28         cudaMalloc(&res_d, bytes);
29         // copy host memory to device memory
30         cudaMemcpyToSymbol(p1_d, p1, bytes);
31         cudaMemcpyToSymbol(p2_d, p2, bytes);
32         // init grid and block dimensions
33         dim3 dimGrid(2);
34         dim3 dimBlock(1024);
35         // select GPU device
36         cudaSetDevice(1);
37         // call kernel function
38         f_sum<<<dimGrid,dimBlock>>>(p1_d, p2_d, res_d, N);
39         // copy device memory to host memory
40         cudaMemcpyFromSymbol(res, res_d, bytes);
41         // deallocate
42         cudaFree(res_d);
43         cudaFree(p1_d);
44         cudaFree(p2_d);
45 }
```

Code 1.2: Generated Functions in C++ and CUDA C/C++

The following steps are performed to generate the kernel *f_sum()* and the communication code for it to be called. Firstly, since the body of *f_sum()* is defined in terms of sequential computation, we parallelize it by substituting the iterating `for` loop with a multithread parallel sum, in which each thread in the block sums the respective *i-th* arrays element (lines 10-16). This step is currently meant to be provided only in a semi-automatic fashion, hence requiring manual fine-tuning in more complex cases.

The next step is to create a communication function called *f_sum_caller()* (lines 18-45) which would be called by *caller()* and that provides the CUDA-related operations needed to call the kernel to *f_sum()*. In order to do this, a pointer for each of the parameters (both *in* and *out*) of *f_sum()* is declared and given memory through the `cudaMalloc()` API (lines 22-28). They will be used for exchanging data between CPU and GPU via host and device memories. The pointers are then made to point to the values carried by the parameters by copying host memory to device memory through the `cudaMemcpyToSymbol()` API (lines 30-31).

As depicted in Fig. 1, the modeller defines the number of grid dimensions (i.e., 2) by ≪*nfpConstraint*≫ `GridD` as well as the thread block (i.e., 1024) by ≪*nfpConstraint*≫ `BlockN` as decorations of the allocation of *f_sum()* on the GPU core with $Core_ID = 1$. The generation process will locate this information in the model and use it to declare the actual dimensions of both grid and block (lines 33-34); the GPU core's ID is used

to assign the device to be used, through the `cudaSetDevice(ID)` API (line 36).

At this point we generate the call to the kernel *f_sum()* using the CUDA-specific syntax (line 38). When the computation is completed, we move the result hold in the device memory back to the host memory through the `cudaMemcpyFromSymbol()` API (line 40). Finally, we can release the allocated resources through the `cudaFree()` API (lines 42-44) and end the parallel computation.

5 Discussion and Conclusion

The actual development of the proposed approach is carried out by (i) identifying the means to model deployment on mixed CPU-GPU configurations, (ii) improving the intermediate artefacts employed by the code generator, and described in [1], to host CUDA-related information, (iii) defining model-to-model and model-to-text transformations to carry out the actual code generation. Currently, the actual parallelisation of ALF code is only provided in a semi-automatic manner, thus able to translate rather simple cases (e.g., `for` loops both simple and nested) of parallelizable code. Future enhancements of the approach will therefore focus on broadening the set of covered cases. Nevertheless, as it can be noticed in the proposed example, in order to call *f_sum()* from *caller()* (less than 20 code lines together), we would have needed to manually code more than 25 lines of communication code (per call). This gives a hint on the usefulness of automating the generation of communication code and therefore relieving the end-user of an error-prone and time consuming burden. Moreover, once we will have finalized the necessary information to model heterogeneous allocation (e.g., CoreID, GridD, BlockN), we intend to produce custom stereotypes, concentrating constraints and allocations in a single place, that would be folded into the CHESS-ML profile.

In this work we focused on the allocation of entire ALF functions to GPU cores for parallel computation. Since ALF allows to specify possible parallelization at finer-grained level, as for the statements `block` and `for`, through the `parallel` annotation, we will introduce the possibility of allocating only such specific portions to the GPU core. According to the fUML semantics [22], the presence of a `parallel` annotation does not imply the implementation of actual parallelism on the execution platform, therefore the deployment-independence of the system's functional description would not be jeorpardised. The `parallel` annotation will in fact be taken into account only if the owning function would be allocated to a GPU core, and in that case, instead of computing the entire function in parallel, only the annotated statements would.

Possible future directions could target the definition of (semi-)automatic allocation of components to processes and functions to either CPU or GPU cores, in order to optimize performance and/or to decrease communication overhead. In order to achieve this, a first step would be the definition of a more detailed memory model, both in terms of the actual hardware resource as well as the allocation of components and functions/statements to it. Moreover, enhancements of the monitoring features as well as the back-propagation capabilities would be required for exploiting the round-trip approach in [1, 3]. Finally, it is important to remark that, even if applied in the context of CHESS-ML as enhancement of the round-trip support, the solution described in this work does not

depend on any CHESS-specific stereotype, and that makes it more generally applicable to approaches leveraging on UML, MARTE and ALF.

References

1. F. Ciccozzi, A. Cicchetti, and M. Sjödin. Round-Trip Support for Extra-functional Property Management in Model-Driven Engineering of Embedded Systems. *Information and Software Technology*, 2012.
2. F. Ciccozzi and M. Sjödin. Enhancing the Generation of Correct-by-construction Code from Design Models for Complex Embedded Systems. In *ETFA*. IEEE, July 2012.
3. F. Ciccozzi, M. Saadatmand, A. Cicchetti, and M. Sjödin. An Automated Round-trip Support Towards Deployment Assessment in Component-based Embedded Systems. In *CBSE*, 2013.
4. D. Hallmans, M. Asberg, and T. Nolte. Towards using the Graphics Processing Unit (GPU) for embedded systems. In *ETFA*, pages 1–4, 2012.
5. A. Cicchetti, F. Ciccozzi, S. Mazzini, S. Puri, M. Panunzio, A. Zovi, and T. Vardanega. CHESS: a model-driven engineering tool environment for aiding the development of complex industrial systems. In *ASE*, pages 362–365, 2012.
6. Bran Selic. Unified Modeling Language (UML). In *Wiley Encyclopedia of Computer Science and Engineering*. 2008.
7. S. Taha, A. Radermacher, S. Gérard, and J.-L. Dekeyser. MARTE: UML-based Hardware Design from Modelling to Simulation. In *FDL*, pages 274–279, 2007.
8. S. Gérard, C. Dumoulin, P. Tessier, and B. Selic. Papyrus: A UML2 Tool for Domain-Specific Language Modeling. In *Model-Based Engineering of Embedded Real-Time Systems*, pages 361–368, 2007.
9. Eclipse Projects. Papyrus. `http://www.eclipse.org/papyrus/`, Last Accessed: July 2013.
10. Object Management Group (OMG). UML Superstructure Specification V2.3. `http://www.omg.org/spec/UML/2.3/Superstructure/PDF/`, Last Accessed: July 2013.
11. OMG. Action Language For FoundationalUML - ALF. `http://www.omg.org/spec/ALF/`, Last Accessed: July 2013.
12. Nvidia. Get Started with CUDA C/C++. `https://developer.nvidia.com/get-started-cuda-cc`, Last Accessed: July 2013.
13. Enea. The architectural advantages of enea ose in telecom applications. `http://www.enea.com/software/products/rtos/ose/`, Last Accessed: February 2012.
14. J. Piat, S. S. Bhattacharyya, M. Pelcat, and M. Raulet. Multicore code generation from interface based hierarchy. In *DASIP '09*.
15. M. Cha, K. H. Kim, C. J. Lee, D. Ha, and B. S. Kim. Deriving High-Performance Real-Time Multicore Systems Based on Simulink Applications. In *DASC'11*.
16. R. L. Collins, B. Vellore, and L. P. Carloni. Recursion-driven parallel code generation for multi-core platforms. In *DATE'10*.
17. A.W.O. Rodrigues, F. Guyomarc'h, and J.-L. Dekeyser. An MDE Approach for Automatic Code Generation from UML/MARTE to OpenCL. *Computing in Science Engineering*, 15:46–55, 2013.
18. S.-Z. Ueng, M. Lathara, S. S. Baghsorkhi, and W.-M. W. Hwu. Languages and Compilers for Parallel Computing. chapter CUDA-Lite: Reducing GPU Programming Complexity, pages 1–15. Springer-Verlag, 2008.
19. M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA Code Generation for Affine Programs. In *Compiler Construction*, volume 6011 of *LNCS*, pages 244–263. Springer, 2010.
20. M. Amini, B. Creusillet, S. Even, R. Keryell, O. Goubier, S. Guelton, J. O. Mcmahon, F.-X. Pasquier, G. Péan, and P. Villalon. Par4All: From Convex Array Regions to Heterogeneous Computing. 2012.
21. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
22. OMG. Foundational Subset For Executable UML Models (FUML). `http://www.omg.org/spec/FUML/1.1/`, Last Accessed: July 2013.
23. F. Ciccozzi, A. Cicchetti, and M. Sjödin. Towards Translational Execution of Action Language for Foundational UML. In *SEAA*, September 2013.