Executable Modelling for Highly Parallel Accelerators

Lorenzo Addazi, Federico Ciccozzi, Björn Lisper
School of Innovation, Design, and Engineering
Mälardalen University - Västerås, Sweden
{lorenzo.addazi, federico.ciccozzi, bjorn.lisper}@mdh.se

***Abstract*—High-performance embedded computing is developing rapidly since applications in most domains require a large and increasing amount of computing power. On the hardware side, this requirement is met by the introduction of heterogeneous systems, with highly parallel accelerators that are designed to take care of the computation-heavy parts of an application. There is today a plethora of accelerator architectures, including GPUs, many-cores, FPGAs, and domain-specific architectures such as AI accelerators. They all have their own programming models, which are typically complex, low-level, and involve explicit parallelism. This yields error-prone software that puts the functional safety at risk, unacceptable for safety-critical embedded applications. In this position paper we argue that high-level executable modelling languages tailored for parallel computing can help in the software design for high performance embedded applications. In particular, we consider the data-parallel model to be a suitable candidate, since it allows very abstract parallel algorithm specifications free from race conditions. Moreover, we promote the Action Language for fUML (and thereby fUML) as suitable host language.**

***Index Terms*—parallel programming, fUML, Alf, UML, modelling languages, high-performance computing, data-parallelism, executable models**

I. Introduction

There is an ever-growing need for computational power. Recent advances in autonomous driving applications, for example, integrate complex machine learning, signal processing and computer vision algorithms into resource-constrained embedded systems [1]. The response from the hardware industry has been to develop increasingly integrated heterogeneous hardware where *computational accelerators* are placed on the chip or board to offload computationally heavy tasks from the main processor. In this way, large computational resources can be provided at low cost. Today we see a large proliferation of accelerator architectures: General-Purpose Graphical Processing Units (GPGPUs), many-cores, solutions involving Field Programmable Gate Arrays (FPGAs), and even Application Specific Integration Circuits (ASICs). Although these architectures are quite different, they have in common that they typically rely on massive parallelism to boost performance. Besides embedded systems, these kinds of accelerators are also increasingly being used in traditional High-Performance Computing (HPC) as well as cloud computing: an example of the latter is Microsoft's Catapult project that integrates FPGAs into cloud servers[1]. However, languages and tools for software development are lagging behind. Programming accelerators needs device-specific expertise in computer architecture and low-level parallel programming for the accelerator at hand. Programming is error-prone and debugging can be very difficult due to potential race conditions: this is particularly disturbing as many embedded applications, like autonomous vehicles, are safety-critical, meaning that failures may have lethal consequences. Furthermore software becomes hard to port between different kinds of accelerators.

A convenient solution is to adopt a model-driven approach, where the computation-intense parts of the application are expressed in a high-level, accelerator-agnostic executable modelling language that is apt for modelling massively parallel computations. This allows flexibility in the choice of accelerator and it enables early analysis of the chosen solution. Later in the software development process these models can also be used for verifying the functionality of hand-coded solutions for the accelerator at hand; a more intriguing option would be to generate accelerator code directly from them.

In this position paper we advocate the use of *data-parallel* programming for this purpose. Data-parallel programming languages originally arose as means to program Single Instruction, Multiple Data (SIMD) and vector machines [3]. Nevertheless, they are not necessarily tied to specific processor architectures, but they should rather be seen as general-purpose implicitly parallel languages. They can be designed to express massively parallel computations at a high level, hiding implementation details. We therefore believe that they are apt for modelling massively parallel computing on accelerators. The modelling language identified to potentially host a data-parallel programming paradigm is the Action Language for Foundational UML (Alf) and the underlying Foundational UML Subset (fUML); in the remainder of the paper we will reason on why the combination fUML/Alf is a suitable candidate and what challenges and benefits may arise in providing implicit data-parallel facilities for it.

II. The data-parallel Programming Model

In the basic data-parallel programming model there is a single control flow. The model specifies a number of collective operations on homogeneous data structures such as arrays or lists, similar to higher order functions such as "map", or "fold" in functional languages (see Table I for a more complete list). These operations are inherently very parallel,

This research was funded by the KK-foundation through the HERO project, under grant no. 20180039.

[1]`microsoft.com/research/project/project-catapult`

TABLE I
DATA-PARALLEL PRIMITIVES THAT WE WANT TO PROVIDE IN fUML/ALF

Primitive	Description	Example
Element-wise scalar operations	Take one (or several) data structures, and apply a scalar operation to the respective elements in each position k. The result is a new data structure	Add two arrays $A[k]$, $B[k]$ in a resulting array $C[k]$
Parallel read	A parallel read operation, where each processor k, in parallel, reads the element of a data structure from some other processor $G[k]$	Receive elements k by $X[k]$ from $Y[G[k]]$
Parallel write	A parallel write operation, where each processor k, in parallel, sends the element of a data structure to some target processor $G[k]$	Send elements k from $Y[k]$ from $X[G[k]]$
Replication	It consists of duplication of a single piece of data to many processors (a special case of parallel read)	Replication of a 1-D vector into a matrix
Masking	It consists of selecting part of a data structure for some data-parallel operation, usually done with respect to some boolean mask or guard	Selection of a submatrix from a matrix
Reduce	Let $\mathtt{op}$ be a binary, associative operation (like +, *, min max, ..), and X be a data structure with positions $0, ..., n-1$ (e.g., an array): $reduce(\mathtt{op}, X) = X[0]\ \mathtt{op}\ ...\ \mathtt{op}\ X[n-1]$	Computation of pairwise sums in an array of integers
Scan	Let $\mathtt{op}$ be a binary, associative operation (like +, *, min max, ..), and X be a data structure with positions $0, ..., n-1$ (e.g., an array), it computes an array of all partial $\mathtt{op}$: $scan(\mathtt{op}, X) = X[0], X[0]\ \mathtt{op}\ X[1], ..., X[0]\ \mathtt{op}\ ...\ \mathtt{op}\ X[n-1]$	Computation of all partial sums in an array of integers

and all the parallelism in a data-parallel language is implicitly present in them rather than in explicit threads or processes. This has some consequences. If the data-parallel operations are properly designed and implemented then there will be no race conditions, due to the single flow of control. Similarly there will be no deadlocks. If costs (like execution times, or energy consumption) can be assigned to the data-parallel operations, then cost analyses can be done in the same fashion as for sequential programs. Thus, verifying functional correctness as well as deciding non-functional properties such as resource consumption has the potential to be be much simpler than for general explicitly parallel software, where features such as race conditions can make the software extremely hard to debug. Another observation is that the use of data-parallel primitives can lead to very clear and succinct code. Again this is similar to the use of higher order functions such as map, fold, and filter in functional languages, which often yields very succinct code that is easy to understand for someone who is familiar with these functions. Some data-parallel languages also offer additional syntactic conveniences such as the MATLAB style overloading of arithmetic operators to also work on matrices, or the advanced array selection statements in ZPL [6]. Such features also help writing code that is easy to understand and maintain. Based on the above observations, we believe that executable models for accelerator software can preferably be based on the data-parallel programming paradigm.

III. INTEGRATION OF THE DATA-PARALLEL MODEL IN fUML/ALF

If executable models for accelerators are to be based on the data-parallel model, then a crucial issue is how to integrate this model with existing modelling languages. The data-parallel model is close to the functional paradigm, and indeed there are several examples of data-parallel functional languages [4], [11], [12]. However most modelling languages are object-oriented. Thus, the question is how to integrate the data-parallel and object-oriented paradigms in such a way that the beneficial features of the data-parallel model are not lost. For instance, the determinacy of the data-parallel primitives relies on that the functions involved are side-effect free. The object-oriented paradigm aims to encapsulate state, which naturally yields side-effects that can be hard to find. Static program analyses that can find all possible side-effects of functions exist [20], but these analyses are formulated for higher order functional languages such as the Meta Language (ML) [19]. We are not aware of any similar analyses for object-oriented languages. Nevertheless, some attempts have been done to combine object-orientation and data-parallelism [15], [23]. We are currently investigating how these works can guide the integration of the data-parallel model into the object-oriented modelling language that we are targeting, Alf (together with fUML).

A. *Why fUML/Alf?*

Among the many modelling languages currently available in the MDE landscape, we consider UML to be the most suitable for our purposes. UML is a general-purpose and multi-faceted language, a de-facto standard in software industry [13] and an ISO/IEC (19505) standard[2]. The formalisation of (i) fUML, which gives a precise execution semantics to a subset of UML limited to composite structures, classes, activities and state-machines (application models designed with fUML are executable by definition) [26], and (ii) the Alf textual action language to express complex execution behaviours, has made UML a full-fledged implementation quality language [22], hence an excellent candidate as host language for our data-parallel paradigm. We have identified Alf (and thereby fUML) as an attractive host language for the following reasons.

Standard. Alf, as part of the UML family of languages,

[2]https://www.iso.org/standard/52854.html

has been formalised by the OMG[3]. Its reference specification represents a solid and unequivocal source for its syntactical and semantic conformance.

Platform-independent. Despite being an object-oriented textual language, Alf inherits the high-level and, most importantly, platform-independent essence of UML. Since we believe that a data-parallel modelling language should permit to describe parallel applications for multiple different target architectures, platform-independence represents a core characteristic of the host language.

Flexible. Being a modelling language that offers a textual concrete syntax, which is seamlessly integrated with the diagrammatic concrete syntax of UML, Alf provides a much more flexible "programming" style than common programming languages. Moreover, Alf can be used in combination with UML profiles addressing domain-specific aspects (e.g., modelling of parallel hardware), making it apt to also entail structural and deployment needs of modelled applications [24].

Executable. According to its specification, Alf has three prescribed ways to achieve *semantic conformance*, i.e. how execution semantics is implemented, summarised as follows:

- **Interpretive**: Alf is directly interpreted and executed;
- **Compilative**: Alf is translated into a UML model conforming to fUML and executed on the actual target platform according to fUML semantics;
- **Translational**: Alf and all surrounding UML concepts in the model are translated into an executable for a non-UML target platform, and executed on it.

The variety of execution possibilities entailed in the specification permits to use Alf models for virtually any development activity, from simulation and debugging (through interpretive and compilative execution) [21], to transformation for deployment to and execution on the actual target platform (through translational execution) [7].

Analysable. Thanks to a convenient trade-off between expressiveness and abstraction, Alf can be fruitfully used for early model-based investigations, both to identify possible flaws in the model through, e.g., static model-based analysis [17] or to explore for possible optimisations of the modelled applications, e.g. for parallelisable model portions [18].

B. Challenges

Currently, Alf provides a concept for enabling parallel execution of potentially parallel algorithms, the annotation `@parallel`, which can be used in combination with `block` and `for` statements. The execution semantics is different depending on the type of statement, but it can avoid race conditions in both cases. In a parallel `block` statement for instance, names assigned in one statement of the block cannot be assigned in any subsequent statement in the same block.

Interestingly, the `@parallel` annotation enables both data- and task-parallelism, depending on the statement it annotates. When used in combination with a `block` statement it enables a parallel execution of a set of statements (task-parallelism). On the other hand, when used with a `for` statement it enables the parallel execution of identical operations on a set of complex data structures (data-parallelism).

[3]https://www.omg.org/spec/ALF/About-ALF/

The latter is a powerful mechanism for implementing a coarse-grained and rather *explicit* data-parallelism. Our aim is to investigate whether Alf can support a complete set of *implicit* data-parallel primitives, as introduced in Table I. Unlike explicitly parallel languages, an implicit one does not provide special primitives, but rather relies on the compiler/interpreter to exploit the parallelism inherent to computations expressed by some of the language primitives when applied to suitable data structures, e.g. arrays. Implicit parallelism would imply that the modeller is relieved from expressing when and where to enforce parallelism and how, such task is instead left to the compiler/interpreter. This is desirable for various reasons, including:

- **Usability**: it may be more or less intricate to "program" explicit parallelism for someone who is not used to parallel programming. Implicit parallel procedures hide this complexity from the modeller, who can continue modelling in a "iterative" fashion (e.g., `for` statements).
- **Platform-independency**: the possibility to enforce parallelism depends on the underlying hardware configuration. At the level of an fUML/Alf model, the modeller is not expected to be fully aware of such a configuration, and in general should keep functional models detached and agnostic of the target hardware configuration.
- **Correctness**: leaving the burden of identifying when implicitly parallel procedures can actually be run in parallel to an automated mechanism, being it a compiler or an interpreter, avoids potential human errors that may arise when explicitly specifying parallelism.

Alf is particularly interesting for implicit parallelism since, as per specification, when the semantics of some computations is specified as concurrent, there is no strict requirement on their actual execution, meaning that they be executed in any particular sequential order by the execution tool. Concepts provided by Alf, such as `@parallel`, `forAll`, `for`, `select`, will be the basis for integrating the implicitly parallel primitives introduced in Table I. Note that, as prescribed in the Alf specification, "the portion of the execution corresponding to an Alf input text must have the equivalent effect to mapping that text to fUML per the Alf specification and executing the resulting model per the semantics specified in the fUML specification". An example is the Alf parallel `for` statement, which is mapped to an fUML expansion region, tagged with mode *parallel*.

This means that, besides exploiting Alf's syntax to express the aforementioned concepts, we will most importantly have to investigate how to suitably map them to the fUML specification without disrupting its execution semantics, which, notably, is already inherently concurrent for activities.

IV. Related Work

A limited number of approaches supporting the design and development of parallel embedded systems has been proposed in the literature. The GASPARD framework is the one that comes closest to our objectives [9]. There, high-level descriptions of the system are defined using the MARTE UML standard profile. Software and hardware parallelism is described using RMoC, an extension of the Array-OL domain-specific language for multidimensional signal processing [10]. Complex behaviours are represented as compositions of elementary specifications written in C. The main drawbacks of integrating general-purpose code snippets in models result from the fact these are at a different semantic level than model elements, hence their correct mapping and consistency management is delegated to the users. Furthermore, behavioural specifications expressed using general-purpose programming languages are often not platform-independent, hence clashing with a core objective of modelling. The HOE modelling language introduces an action language supporting data-parallelism and operations over compound data in Hierarchical State Machines (HSMs) [16]. Automated code generation is provided to produce efficient, low-level code using OpenCL [25] and C. However, the modelling language does not provide support for neither hardware nor software allocation modelling. Consequently, platform-specific refinements of the generated code are not supported.

High-level, portable data-parallel languages were around long before the term "data-parallel" was coined. Already APL [14] provided a large set of advanced array operations. A strand of development has been to introduce explicit array operations into languages for numerical processing to facilitate automatic vectorization, as in ZPL [6] for example. The set of array operations is quite restricted in the first, whereas the second also has support for strided and sparse arrays. A recent heir to ZPL is Chapel [5], which also provides means for more explicit parallel programming. Data-parallelism and functional programming languages are a good match. Examples of functional data-parallel languages are data-parallel Haskell [4], which extends Haskell with data-parallel arrays, and Data Field Haskell [12] which uses *data fields*, an abstract array data type that encompasses both sparse and dense arrays. NESL [2] uses nested sequences as the parallel data structure and comes with a cost model for the data-parallel operations. A recent addition is Futhark [11], a high-level functional array language for GPU programming. We are currently synthesising the results of a systematic literature review on parallel programming and modelling languages. Besides confirming that what we are outlining in this paper is not yet available in the literature, the review results will be used to take the best of existing parallel languages and avoid known mistakes.

V. Outlook

To be able to freely experiment with Alf, we have re-implemented the specification using Xtext in Eclipse and we are currently investigating how to integrate the implicit data-parallel paradigm described in this paper. For the actual integration we will exploit the review results especially in relation to previous attempts of integration of data-parallel models into object-oriented languages. To have control over the management of implicit parallelism and related optimisations, we are also developing a Low-Level Virtual Machine (LLVM) front-end for our Alf implementation [8]. At the same time, we are deepening an orthogonal investigation [24] on how to properly model heterogeneous massive parallel architectures and software to hardware allocations using (f)UML and MARTE.

References

[1] S. Aldegheri, S. Manzato, and N. Bombieri. Enhancing performance of computer vision applications on low-power embedded systems through heterogeneous parallel programming. In *Proc. VLSI-SoC*, 2018.

[2] G. E. Blelloch. Programming parallel algorithms. *Comm. ACM*, 1996.

[3] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *J. Parallel Distrib. Comput.*, 1994.

[4] M. M. T. Chakravarty, R. Leshchinskiy, S. Peyton Jones, G. Keller, and S. Marlow. Data Parallel Haskell: A status report. In *Proc. DAMP*. ACM, 2007.

[5] B. L. Chamberlain. Chapel. In *Programming Models for Parallel Computing*. MIT Press, 2015.

[6] B. L. Chamberlain, S.-E. Choi, E. C. Lewis, C. Lin, L. Snyder, and W. D. Weathersby. The case for high level parallel programming in ZPL. *IEEE Computational Science and Engineering*, 1998.

[7] F. Ciccozzi. On the automated translational execution of the action language for foundational UML. *Software & Systems Modeling*, 2018.

[8] F. Ciccozzi. UniComp: a semantics-aware model compiler for optimised predictable software. In *ICSE - NIER*, 2018.

[9] A. Gamatié, S. Le Beux, E. Piel, R. Ben Atitallah, A. Etien, P. Marquet, and J.-L. Dekeyser. A Model-Driven Design Framework for Massively Parallel Embedded Systems. *ACM Trans. Embed. Comput. Syst.*, 2011.

[10] C. Glitia, P. Dumont, and P. Boulet. Array-OL with delays, a domain specific specification language for multidimensional intensive signal processing. *Multidimensional Systems and Signal Processing*, 2010.

[11] T. Henriksen, N. G. W. Serup, M. Elsman, F. Henglein, and C. E. Oancea. Futhark: Purely functional GPU-programming with nested parallelism and in-place array updates. In *Proc. PLDI*. ACM, 2017.

[12] J. Holmerin and B. Lisper. Data Field Haskell. In G. Hutton, editor, *Proc. Fourth Haskell Workshop*, 2000.

[13] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. Empirical Assessment of MDE in Industry. In *Proc. ICSE*. ACM, 2011.

[14] K. E. Iverson. *A Programming Language*. Wiley, 1962.

[15] J. Larus. C**: A large-grain, object-oriented, data-parallel programming language. In *Proc. LCPC*. Springer, 1993.

[16] I. Llopard, C. Fabre, and A. Cohen. From a Formalized Parallel Action Language to Its Efficient Code Generation. *ACM Trans. Embed. Comput. Syst.*, 2017.

[17] J. Malm, F. Ciccozzi, J. Gustafsson, B. Lisper, and J. Skoog. Static flow analysis of the Action Language for Foundational UML. In *Proc. ETFA*. IEEE, 2018.

[18] A. N. Masud, B. Lisper, and F. Ciccozzi. Automatic inference of task parallelism in task-graph-based actor models. *IEEE Access*, 2018.

[19] R. Milner, M. Tofte, and R. Harper. Definition of standard ML. 1990.

[20] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis, 2^{nd} edition*. Springer, 2005.

[21] E. Seidewitz. UML with meaning: executable modeling in foundational UML and the Alf action language. In *SIGAda Ada Letters*. ACM, 2014.

[22] B. Selic. The Less Well Known UML. *Formal Methods for Model-Driven Engineering*, 7320:1–20, 2012.

[23] T. J. Sheffler and S. Chatterjee. An object-oriented approach to nested data parallelism. In *Proc. Frontiers*, 1995.

[24] V. Stoico. A Model-Driven Approach for modeling Heterogeneous Embedded Systems, 2019. Master thesis - Mälardalen University.

[25] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science Engineering*, 2010.

[26] J. Tatibouët, A. Cuccuru, S. Gérard, and F. Terrier. Formalizing Execution Semantics of UML Profiles with fUML. In *Proc. MoDELS*. 2014.