

Platform-independent modeling of explicitly parallel programs

Christoph W. Kessler, PELAB, IDA, Linköping University, Sweden
 Wladimir Schamai, EADS Innovation Works, Hamburg, Germany
 Peter Fritzson, PELAB, IDA, Linköping University, Sweden

Abstract

We propose a model-driven approach to parallel programming of SPMD-style, explicitly parallel computations. We define an executable, platform-independent modeling language with explicitly parallel control and data flow for an abstract parallel computer with shared address space, and implement it as an extension of UML2 activity diagrams and a generator for Fork source code that can be compiled and executed on a high-level abstract parallel machine simulator. We also sketch how to refine the modeling language to address more realistic parallel platforms.

1 Introduction

The current transition to many-core systems in all IT domains, including desktop and embedded computing, means a huge challenge both for programmers and programming tool providers. With the notable exception of high-performance computing platforms, the sequential von-Neumann model has been dominating computer instruction sets and industry-relevant higher-level programming environments for more than half a century. We have trained (and in many sites, we still do today) many generations of students mainly in sequential thinking about algorithms, control and data structures. This will have to change, as sequential performance of computers will not increase as in the past, and tomorrow's programmers will face hundreds of hardware threads that need to be coordinated properly to speed up a single application.

However, there is a problem with the transition to many-core programming for the masses. The von-Neumann model of sequential computing is a universal programming model, as it fits virtually all sequential programming languages and also the instruction sets of most processor architectures. A program written in a sequential language such as C needs, in general, only recompilation to be run on a new processor type. Many-core platforms, in contrast, do not have a uniform parallel programming model: Some multicore systems are symmetric multiprocessors with shared memory, others have a NUMA or distributed memory, or hybrid forms; some heavily rely on SIMD computing, others feature hundreds of hardware threads to hide memory access latency; some have hardware-managed memory hierarchies, others require explicit data layout and manually orchestrated bulk data transfer to and from memory.

The lack of a universal parallel architecture model is reflected in the lack of a universal high-level parallel programming model. For instance, we have threading libraries like pthreads to extend sequential languages, mes-

sage passing constructs, or explicitly parallel languages like Fork, Cilk, OpenMP, NestStep, X10 or UPC, that each work well for certain kinds of parallel architectures but are not appropriate for others. Porting parallel algorithms, data structures, and entire programs from one model to another is a tedious, error-prone manual task. Given this situation where source code of high-level (parallel) programming languages is no longer portable, how can we write future-proof parallel programs?

In this paper, we propose model-driven software development as a solution to this dilemma. *Model-driven development (MDD)* comprises a hierarchy of modeling languages of decreasing level of application domain specificity and increasing level of platform specificity down to source code. Along the lines of the OMG's Model-Driven Architecture®, MDD starts with a high-level *platform-independent model (PIM)*, which is often expressed in a domain-specific modeling language and focuses on the essence of a (parallel) algorithm. Subsequent refinement steps adapt and add details on data structures, data layout, etc., resulting in more *platform-specific models (PSMs)*. The PIM-to-PSM and PSM-to-PSM transitions are partly automated model rewrite transformations that capture typical specializations in a domain or for a given platform. From the final PSM, source code is generated. MDD increases programmer productivity, as the PIM and possibly also higher-level PSM survive changes in the platform, and there are less bugs because much code is generated automatically. Also, models may allow for better communication of design concepts and for simulation and verification at a higher level.

While often using semi-formal modeling languages such as UML, we advocate in this paper to use, at each level of the modeling hierarchy, a completely executable modeling language that contains a complete specification of behavior, such that it could be simulated on an abstract machine corresponding to its level of abstraction. In most model-driven approaches, high-level executable models are very

restrictive, such as state machines for modeling sequential reactive behavior, work flow for modeling control flow with simple concurrency, or synchronous data flow, which usually works well for the given application domain. In contrast, we target (also) general-purpose application domains, therefore we provide general control and data flow modeling.

In this paper, we describe our proposal for a top-level modeling language, which we call *ParML*, for *explicitly parallel modeling language*. ParML provides a shared address space and full-fledged modeling of both sequential and explicitly parallel control and data flow. For practical reasons, namely in order to use existing UML editors and tools, we base ParML on UML2 activity diagrams and give its implementation as a UML2 profile. ParML also has a textual modeling view, which corresponds to the high-level shared-memory parallel programming language Fork [13, 8].

ParML is executable: It could, for instance, be simulated by a model interpreter, and we also describe how we generate Fork source code from it that can be compiled and executed on the PRAM simulator `pramsim` [8]. But this is only a first step. The main purpose of ParML will be to serve as a starting point for further refinement to PSMs e.g. to platforms with advanced memory hierarchies, explicit communication and/or relaxed synchronization. We sketch a first refinement step towards a PSM for a bulk-synchronous parallel (BSP), partitioned global address space (PGAS) platform.

In this way, executable models and model transformation specifications could become the new "source code" of the next decades.

The remainder of this paper is organized as follows: Section 2 introduces our new modeling elements for SPMD-parallel computations. Section 3 presents an example. Section 4 shows how to extend the basic, shared-address-space parallel modeling language for more platform-specific modeling. Section 5 describes how the new modeling elements are realized as UML metamodel extensions and presents our Fork code generator. Section 6 discusses related work, and Section 7 concludes.

2 Modeling elements for SPMD-parallel computations

Activity diagrams in UML 2.0 are based on Petri nets [19], i.e., they are concurrent control and data flow graphs at various levels of detail—they may even be broken down to elementary operations in a program, although it is not necessary and not common to model everything graphically down to the statement level; in order to obtain an executable model, it is sufficient to encapsulate pseudocode or source code in action nodes to describe their behavior.

The activity diagram is, roughly speaking, the dual to a UML statechart diagram: Nodes correspond to *actions*,

such as computations, data access, or communication, thus state transitions, while edges correspond to control and data flow. *Activities* are composite actions that fold multiple actions or activities that are single-entry, single-exit regions of control flow (ignoring exception and error exits for now). For brevity, we will refer to "actions or activities" (thus explicitly including folded activities) by the term *acts*¹. The semantics of activity diagrams is defined by token passing along edges, as known from Petri nets. While data flow edges alone usually imply only a partial execution order among the acts, the control flow edges can additionally constrain the execution order. For more information, we refer to the UML 2.0 Superstructure document on OMG's web site (www.omg.org) and the book by Rumbaugh et al. [20].

2.1 Parallel control flow

In the following subsections, we define *extended activity diagrams* where we statically classify control flow edges into sequential (**Figure 1(a)**) and parallel (**Figure 1(b)**) edges. We draw *parallel control flow edges* by double arrows to visually distinguish them from sequential control flow. Whereas a sequential control flow edge is traversed by a single token at execution, a parallel edge will be traversed by a *group* of p tokens (threads). A parallel control flow edge could thus be regarded as a "flat-band cable", a bundle of p wires each corresponding to an ordinary sequential control flow edge. This *width* p of a parallel control flow edge, which may symbolically annotate the edge as shown e.g. in **Figure 1(f)**, needs not be static or explicitly given but it will be determined from the run-time context in the constructs where parallel control flow originates, that is, where a parallel activity starts (parallel initial node, **Figure 1(c)**) or a thread group is split into subgroups (parallel fork node, **Figure 1(f)**). We require that an extended activity diagram only have one (parallel) initial node and one (parallel) final node (**Figure 1(d)**).

The only constructs that affect the width of parallel control flow edges are parallel forks and parallel joins, see **Figures 1(f) and (g)**. In these cases, the accumulated widths of ingoing parallel edges must equal the accumulated widths of outgoing parallel edges. In contrast to sequential fork and join nodes, no new threads are spawned at a parallel fork and none disappear at a parallel join; instead, the existing threads are simply regrouped.

Tokens (threads) may proceed across a barrier synchronization point (**Figure 1(e)**) only if all tokens of a group have arrived there. Note that parallel fork and join (**Figures 1(f–g)**) and the exit of a parallel activity (**Figure 1(d)**) imply barriers as well.

In order to reduce model complexity especially where we only have standard UML editor support, we require that

¹The corresponding term is apparently missing in the UML2 specification; terms for equivalent entities at programming language level such as "region" have other meanings or restrictions in UML2.

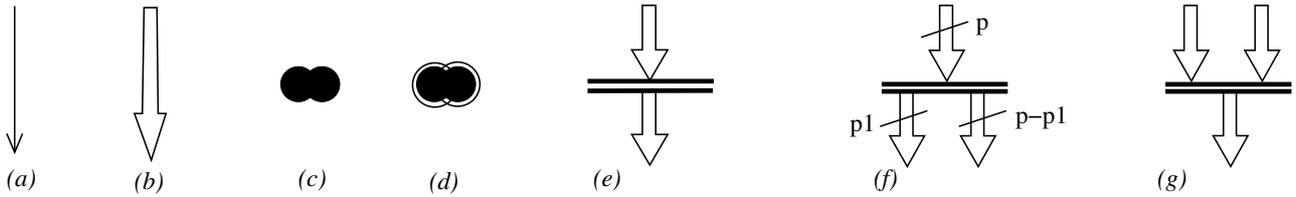


Figure 1: Constructs for parallel control flow in extended activity diagrams: (a) Sequential control flow. (b) Parallel control flow. (c) Entry of parallel activity. (d) Exit of parallel activity. (e) Barrier synchronisation point. (f) Parallel fork (split thread group). (g) Parallel merge (merge thread groups).

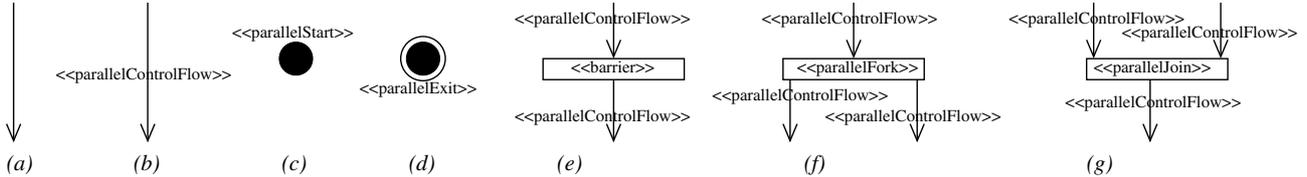


Figure 2: The parallel control flow constructs of **Figure 1**, current implementation using basic UML2 notation only.

extended activity diagrams defining serial activities contain only serial control flow edges and that those defining parallel activities only contain parallel control flow edges. Hence, if a serial activity is to be integrated in a parallel control flow (which means that it will be executed by only one of the threads of an executing group while the others wait), it has to be defined out-of-line in a separate extended activity diagram or source code function.

2.2 Parallel actions and activities

Likewise, acts are classified into sequential and parallel ones, where the parallel acts are shown as doubly framed rounded boxes or as `<<parallel>>` stereotyped act boxes with standard UML2 editors, see **Figure 4(a)**.

Sequential control flow entering a parallel act results in multiple tokens (threads) appearing in the parallel act's start node; the number is usually derived from the run-time context. Likewise, sequential control flow leaving a parallel act means that all tokens of the parallel act are waited for to reach its exit node, and all but one thread is continuing along the edge while the others are suppressed. Parallel control flow entering and leaving a sequential act means that all but one token skip the act and wait at its end for the one executing it. Apart from these switching cases, the parallel acts nested within a parallel activity have one entering and one leaving parallel control flow edge each.

As in plain UML, *opaque actions* can contain arbitrary source code, in our case plain C code for sequential opaque actions and Fork [8, 13] source code for parallel opaque actions. This allows to switch to textual modeling whenever the use of graphical elements would be too cumbersome.

We also introduce parallel conditions, shown as doubly-framed rhombs to distinguish from their sequential counterparts. It may happen that the same condition evaluates to different boolean values for different threads, which implies group splitting. The subgroups will be reunited again

at the corresponding parallel merge node, proper nesting assumed.

Parallel acts can be composed serially and in parallel, see **Figure 3(a)** and **(b)**. For *serial composition*, the widths of the composed acts and of the control flow arrows must be the same. *Parallel composition* includes a parallel fork and a parallel join; the accumulated number of threads executing the parallel sub-acts equals the number of threads that enter and leave the compound parallel activity.

There are two basic ways of breaking down a parallel act into its constituent sequential tasks. One method is to construct the task graph explicitly. For a small, statically known constant number of threads an activity could be unfolded at the model level using swim lanes, see **Figure 3(c)**. To unfold a parallel activity for an arbitrary number of threads, we suggest the notation in **Figure 3(d)**, a parallel iterator over equally defined sequential tasks that are mapped to the executing thread group (work sharing).

Generally, the modeler has lots of design choices in the spectrum between two primary styles: (1) to decompose a parallel act early into its sequential threads and then mainly use sequential modeling for each thread, or (2) to keep activities and flows bundled as long as possible and therefore mainly use parallel acts. While the former variant is more convenient to integrate sequential legacy models, we consider the latter one a better modeling style because it tends to reduce model complexity and improve analyzability e.g. of performance.

2.3 Data flow

Acts have *input pins* and *output pins* for data read and written in the act. They are shown as rectangular symbols attached to the act box, see **Figure 4(b)**. An act can be executed when there is a control token and there are (data) tokens on all input pins. Execution removes these tokens and issues tokens on all output pins.

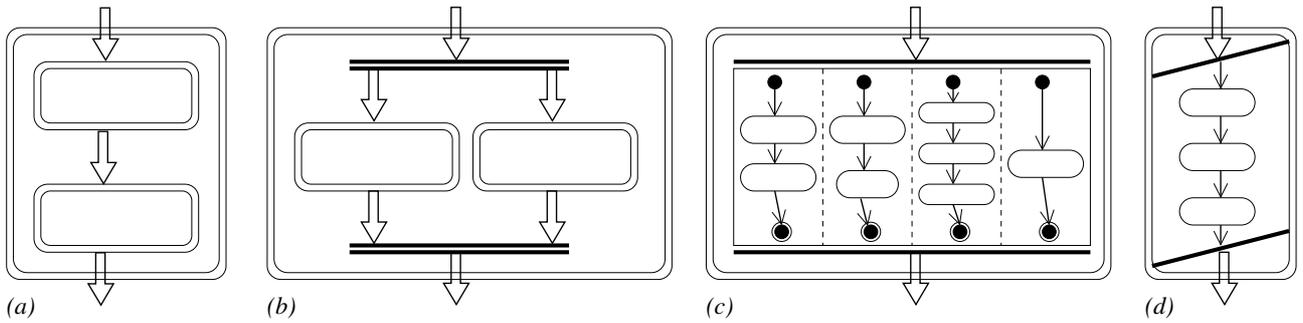


Figure 3: Composing and decomposing parallel acts in extended activity diagrams: (a) Serial composition. (b) Parallel composition. (c) Unfolding a parallel act by its constituent sequential threads. (d) Unfolding a parallel loop.

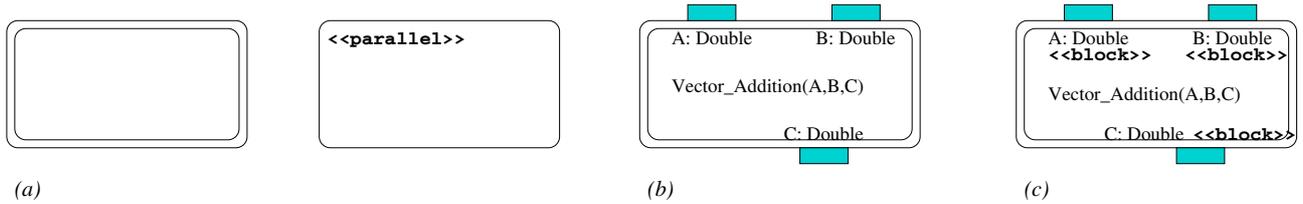


Figure 4: Elementary parallel activities (a) Parallel act, enhanced graphic symbol and UML stereotype notation. (b) We use a simplified symbol (shaded rectangle) for parallel (array) pins. (c) Extending the pins by distribution constraints for a PSM.

Note that the control flow entry and exit points of an action can be seen as a special sort of data flow, a triggering input that fully serializes all acts for a thread. To distinguish these from data flow entries and exits (data pins) of action boxes, we do not show pins for control flow edges. Data pins will bind to values (much like formal parameters in procedures called by value) and can thus be connected by data flow edges.

We mark data flow edges that bundle vector-valued data flow (i.e., arrays, array sections, or other iterable collections of data that allow parallel random access) between acts with blue wide arrows, see **Figure 5(d)**, and call them *parallel data flow edges*.

Pins may be bundled in a similar way as data flow edges; parallel pins (blue) represent vector-valued operands that are accessible by all threads of a thread group executing the act, see **Figure 4(b)**. The width of a pin must match the width of the edge(s) attached to it.

When modeling under the static single assignment (SSA) requirement, only one single *value* can flow into any input pin. Where necessary, multiply assigned values must be combined in a SSA phi-node first (see **Figure 5(c)**). This also holds accordingly for parallel edges, where we provide *parallel phi nodes* that merge values stemming from (potentially) different definitions on an element-by-element basis (see **Figure 5(d)**). The resulting array data flow is thus very similar to Array-SSA form [14]. Pseudocode or source code specified within a (parallel) opaque action may locally deviate from Array SSA form, by accessing shared variables, arrays, and objects implicitly by their name, as long as such multi-assignment code is prop-

erly encapsulated. Array concatenation, shown in **Figure 5(f)**, is a special case of an array-phi with a unique static predecessor for each section of an array.

In order to better distinguish between control and data flow edges, we reserve black color for control flow edges while using shaded or colored arrows for data flow edges, where the choice of any specific color (default: blue) is up to the modeler—it serves as a kind of comment to enhance readability.

Usually, ParML modeling starts with specifying control flow, and data flow is added afterwards, but also the opposite order is possible. The control flow view and the data flow view of ParML models can be considered separately, in order to reduce complexity.

3 Example: Parallel merge sort

Figure 6 shows the control flow view of the extended activity diagram for a parallel merge sort implementation. It is well suited to explain the basic structure of the algorithm, because the two parallel recursive calls to `ParMergeSort` are shown side-by-side, and not in some serial order as in almost all parallel programming languages with text-based syntax, as e.g. in **Figure 7**. The integrated view with data flow can be seen in **Figure 8** (there using standard UML notation, though).

Given the complete model as in **Figure 8**, generating source code e.g. in the shared memory language Fork [13, 8] (an extension of C for SPMD shared-memory parallel computing), such as the one shown in **Figure 7**, is straightforward, because the functionality in ParML is a subset of

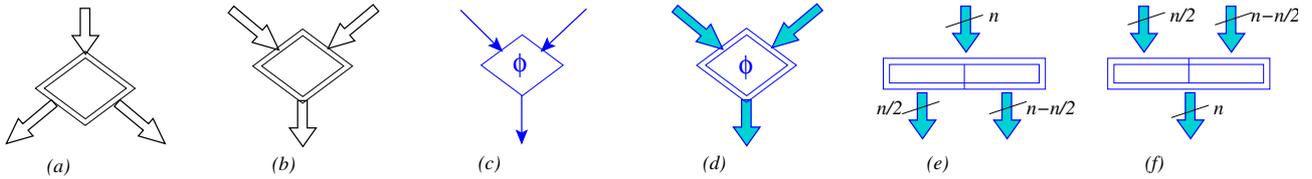


Figure 5: Constructs for parallel control flow (black) and data flow (blue) in extended activity diagrams: (a) Conditional branch of parallel control flow. (b) Merge of parallel control flow. (c) Scalar phi-node for merging scalar data flows. (d) Array phi-node for (element-wise) merging parallel data flows. (e) Array splitting. (f) Array concatenation.

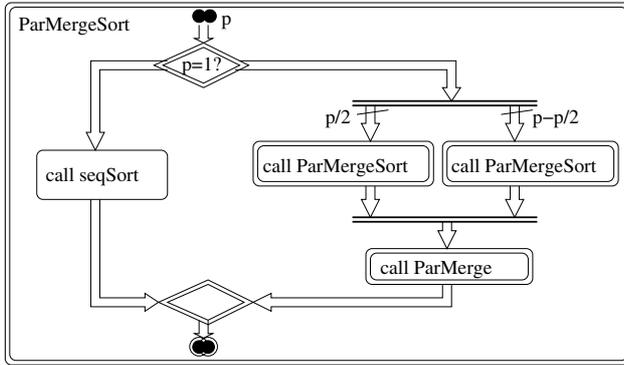


Figure 6: Extended activity diagram for Parallel Merge Sort, showing the control flow view only.

what Fork supports at the language level. Using the Fork compiler and SBPRAM simulator [8], the correctness of the ParML model can be tested immediately on the generated executable code.

4 Towards platform-specific parallel models

In an approach to platform-specific modeling for NUMA or distributed memory systems, which are likely architectures for future massively parallel many-core chips, we introduce *array distributions* as annotations to edges and pins, see **Figures 9(a)** and **4(c)**. The distribution of flowing data is thus a static property of a modeling element (a kind of type extension). Explicit data redistribution actions are marked with the notation in **Figure 9(c)**. The array partition owned by a thread can be extracted as shown in **Figure 9(b)**. Also, explicit collective communication operations such as *reductions* in **Figure 9(d)** can be specified. Acts may be grouped into a BSP (bulk-synchronous parallel) *superstep*, with global communication/redistribution happening at its end, using the notation in **Figure 9(f)**.

Adding distribution, either manually or by semiautomatic transformation rules, to a ParML PIM leads to a PSM, which could be optimized separately (e.g. by aligning data distributions or by merging independent subsequent BSP supersteps) and from which source code in a partitioned global address space (PGAS) language such as NestStep, X10 or UPC, or even message passing or hybrid code could be generated. We leave the details for future work and

only briefly motivate the idea for the case of the equivalent transition from the PIM level (which basically corresponds to the PRAM programming model, represented e.g. by the Fork language) to the parallel programming language NestStep [12, 9] for the bulk-synchronous parallel (BSP) model. Due to similarity of language elements, it is (up to a certain degree) possible to extend/adapt a given Fork source program to obtain a NestStep source program for the same problem. As an example, **Figure 10** shows NestStep source code for bulk-synchronous parallel merge sort, corresponding to the Fork source code in **Figure 7**. The major new constructs to be added are: BSP supersteps (marked by `step` and `neststep` statements) that control both synchronicity and memory consistency; array distribution qualifiers (such as `</>` for the block-wise distribution of a shared array across the declaring group of threads/processes); and update qualifiers for shared variables written concurrently in BSP supersteps (such as `<+>` for combining of concurrent writes by global summation, or `<=>` for no combining of equal-valued concurrent writes). The remaining constructs such as group splitting (`neststep`) work similarly to those in Fork and differ mainly in syntax.

A similar refinement step could have been done using the corresponding graphical elements in ParML, such as adding elements to array data flow ports and arrows that indicate array distribution. **Figure 11** gives an overview of the different model elaboration and transformation steps discussed.

A more general treatment of the transition from PIM to PSM models in ParML or its textual language counterparts is beyond the scope of this paper and left for future work.

5 Realization as UML meta-model extensions and Fork code generator

As UML editor and modeling tool, we use Papyrus [4], which is an Eclipse plug-in. Papyrus is an open-source tool with restricted graphical features. We have implemented the above described new elements for extended activity diagrams as a UML profile, see **Figure 12**. An editor supporting ParML's extended graphical notation such as wide or shaded arrows, doubly framed boxes etc., is an issue of future work. For now, we use UML standard graphical

```

extern void seqSort( float A[], float B[], int n );
extern sync void ParMerge( sh float A[], sh int n );

sync void ParMergeSort( sh float A[], sh float B[], sh int n )
{
  sh int p = groupsize(); // query run-time context to assign p
  if (p==1) {
    seq {
      seqSort( A, B, n );
    }
  }
  else {
    int me = get_rank();
    if (me < p/2) { // parallel fork with 2 subgroups
      ParMergeSort( A, B, n/2 );
    } else {
      ParMergeSort( A+n/2, B+n/2, n-n/2 );
    }
    ParMerge( B, n/2, B+n/2, n-n/2 );
  }
}
}

```

Figure 7: Parallel Merge Sort in Fork source code.

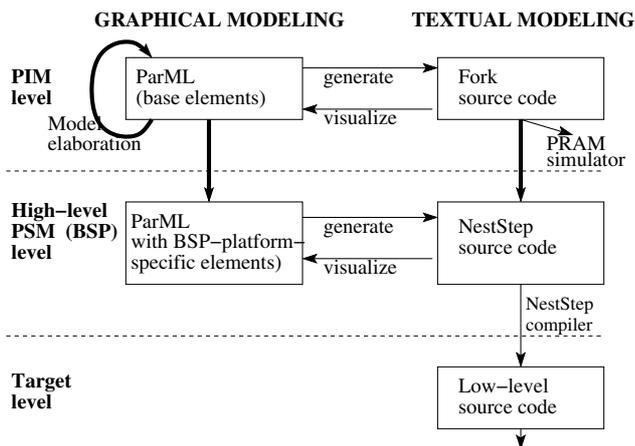


Figure 11: Overview of graphical and textual modeling languages discussed in this paper. Model elaboration and transitions between PIM and PSM level require manual work (bold arrows) while the remaining transitions can be fully automated.

notation with stereotypes as concrete syntax. Switching locally from graphical to textual notation is always possible by using OpaqueAction elements that can contain arbitrary source code snippets. Vice versa, switching from textual to graphical notation can be done via Fork function calls whose target is defined as an extended activity diagram.

As a proof of concept we implemented a simple prototype that generates Fork source code as shown in Figure 7 from the ParML model. Generating Fork source code from ParML (i.e., its base modeling language without platform-specific annotations) is relatively straightforward because of the strong similarity between ParML modeling elements

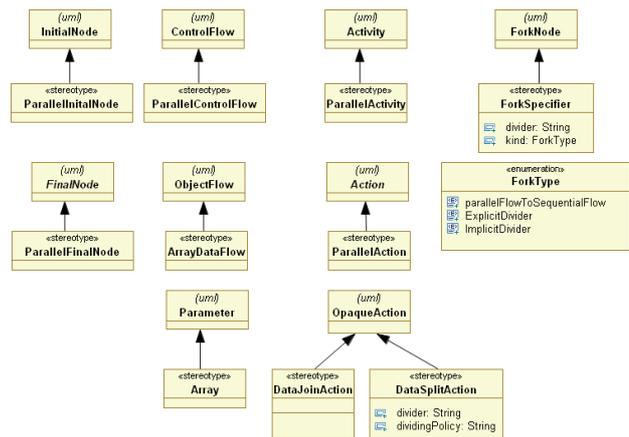


Figure 12: Excerpt of the implemented UML metamodel extensions.

and Fork language constructs. For instance, ParML acts correspond to Fork statements, pins to operands and results of operations or calls, and data flow arrows to variables and temporaries. Likewise, it would not be difficult to generate ParML diagrams from Fork source code; for instance, this could be realized by running a graphical visualizer on a Fork compiler’s high-level intermediate representation².

The code generator is implemented using the Acceleo [2] Eclipse plug-in. It generates code based on the control flow defined in the ParML model (up to now, we use dataflow only for descriptive modeling in our example).

²The existing Fork compiler [8, 13] is a one-pass compiler that exposes only a low-level intermediate representation but no abstract syntax tree or high-level IR and no SSA form.

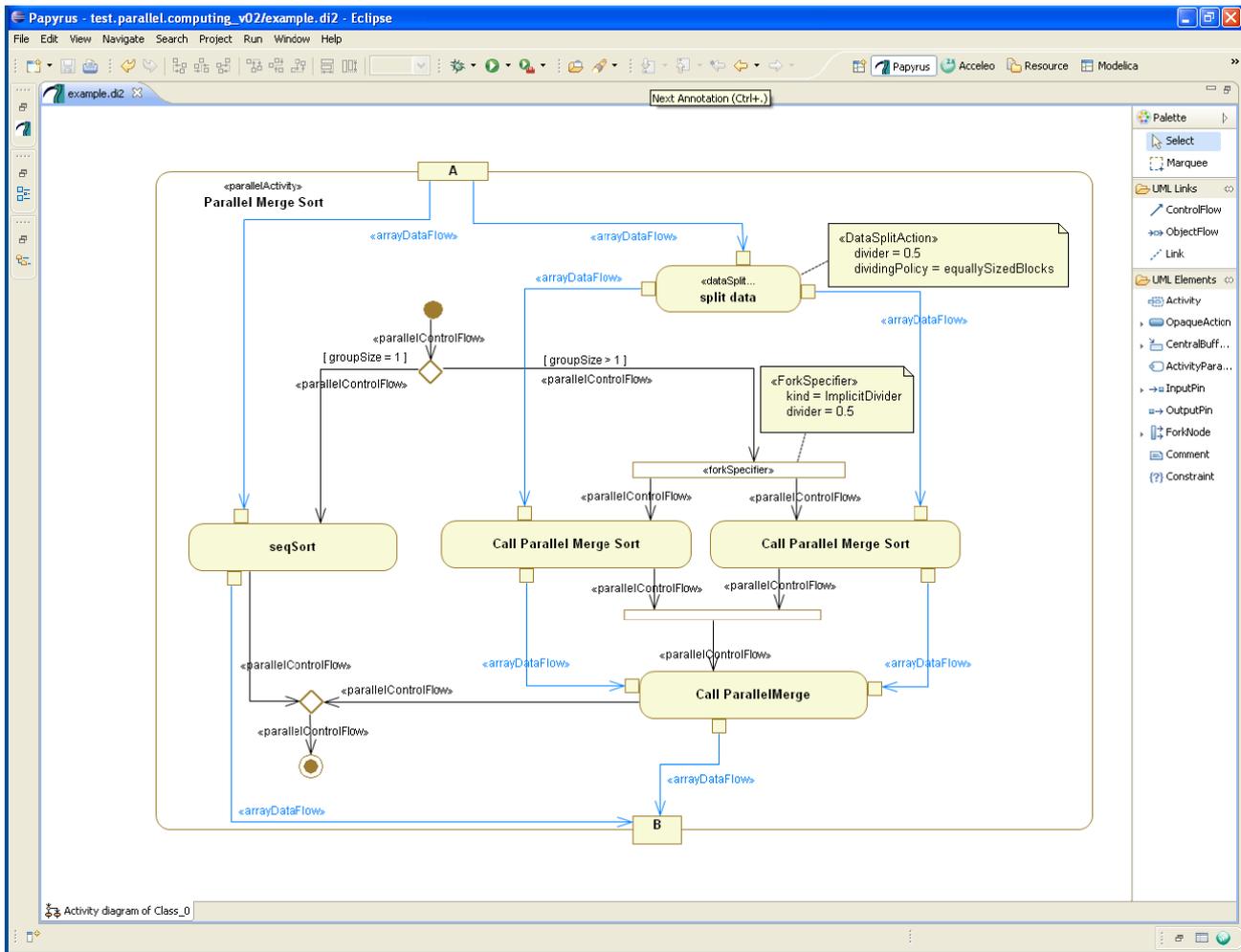


Figure 8: Screenshot of the Papyrus tool, showing the ParMergeSort activity, with data flow arrows in blue color. The model is not in SSA mode, as there is no array phi-node explicitly merging the array values arriving for output parameter B along different control flow paths.

The code generator prototype assumes a valid model that respects the following constraints (typically implemented in a UML profile using OCL):

- An activity has exactly one initial node with exactly one outgoing control flow.
- Any OpaqueAction has exactly one outgoing control flow.
- Decision and merge nodes, as well as fork and join nodes, are properly nested (i.e., the counterpart can be found by following the control flow).
- One of the outgoing control flows of a decision node has `else` as guard (i.e., condition).
- OpaqueActions have Fork source code (statements or calls) in their bodies
- Any other text annotations in the model (such as the guards of control flows) are based on Fork syntax.

The basic algorithm for generating structured code (i.e., with properly nested if/else blocks etc.) is as follows:

1. Generate a Fork function header for the activity defined in the given model.
2. Find the initial node of the activity.
3. Follow the outgoing control flows and recursively generate code for the next act, depending on its type (such as decision node, fork, OpaqueAction, etc.) based on templates, respectively:
 - (a) OpaqueActions: output the bodies (Fork code).
 - (b) Decision nodes / merge nodes are translated into if/else code. The conditions are based on the specified guards (Fork code). The number of if/elseif blocks is determined by the number of the outgoing control flows. The corresponding merge node needs to be found in order to determine the conditional blocks and to later continue with code generation after the if/else block (this avoids replication of code following the merge

node). Then, code for the if/else blocks can be generated recursively.

- (c) Fork/join nodes with two subgroups are translated into if/else code with a private branch condition (for more than two subgroups, the `fork` statement of Fork should be used). The number of if/elseif blocks is determined by the number of the outgoing control flows. The conditions of the if/elseif blocks are deduced from the fork specification (kind, divider). After the if/else block is generated, the corresponding join node is found and the code generation is resumed.
- (d) Final node: do nothing (end of the control flow).

6 Related work

Pllana and Fahringer [18] model the control flow of shared memory and message passing computations by UML stereotypes (metamodel extensions) that are very close to the respective API's predominant in high-performance parallel computing, OpenMP and MPI. Hence, their resulting models can be viewed as platform-specific models. Their main area of application is performance analysis. They use the activity diagram for modeling control flow, but do not graphically distinguish between parallel and sequential control flow. Sets of activities are composed to processes using replication (for SPMD) or swim lines in activity diagrams (for general MIMD systems). The collaboration diagram is used to model logical process topologies such as meshes and trees. Finally, processes are mapped to physical topologies using the deployment diagram. Parallel data flow is not modeled explicitly; e.g. the type of communication pattern for data distribution and the volume of communicated data must be specified explicitly by the modeler. Activities for shared memory and message passing can be mixed, which allows to model multi-(programming)model applications for hybrid parallel platforms such as SMP clusters.

Labani et al. [15] describe the introduction of explicit control flow (state charts) and data parallel data flow (component diagrams) in the Gaspard2 application UML metamodel, which targets synchronous reactive systems for signal and image processing applications. A code generator for OpenMP is described by Taillard et al. [22].

Scherger et al. [21] describe the BSP model in UML, using the sequence diagram to model individual parallel processes and their message passing interactions, i.e., the number of executing processes is hardcoded in the model.

The Nimrod/K system [1] enables graphic specification of massively parallel dynamic grid workflows based on the graphic dynamic dataflow tool in Ptolemy [6].

Workflows or business process modeling can be done graphically at a high abstraction level by domain-specific tools such as [16], based on the high level Modelica modeling language [7], which can be compiled to parallel archi-

tectures e.g. at a coarse grained level using an annotation-based approach [17], or a parallel algorithmic approach to a Modelica dialect of NestStep [10].

7 Conclusion

We have presented ParML, a platform-independent graphical modeling language for explicitly parallel SPMD computations based on UML activity diagrams. A prototype of ParML is implemented as a UML2 profile extending modeling elements of UML2s activity diagrams. ParML models can be used to generate Fork source code for early testing and as a starting point for platform-specific modeling, e.g. towards distributed memory systems.

While the proposed graphical syntax may be a matter of taste, we strongly advocate the idea of model-driven parallel programming, starting from a high-level but executable and explicitly parallel modeling language for an abstract parallel platform model such as the PRAM at the highest modeling level (PIM). Due to the semantic equivalence of ParML's modeling elements and corresponding Fork language constructs, the PRAM language Fork can be considered as just an alternative (text-based) syntax to ParML and thus as another PIM-level modeling language for parallel programs. Indeed, graphical modeling is not suitable for specifying every aspect of a parallel program's behavior. In practice, graphical and textual modeling will complement each other. For instance, it is possible (and often most reasonable) to model only a part of a program's behavior, for instance its control flow as in the example of Figure 6, graphically with ParML and fill in the remaining aspects in the generated source code skeleton.

The (nearly) one-to-one correspondence between ParML elements and Fork language constructs would even allow for automated round-trip engineering (see e.g. [5]), such that also manual edits or transformations that are applied to the generated source code are consistently propagated back to the modeling level.

The modeling layer also allows for PIM-level model transformations such as model elaboration before source code or PSM generation. An example is the conversion from a dataflow model to a model explicitly expressing in-place computation, as we had shown in Figure 13.

The implementation of the suggested graphical modeling elements in ParML with existing UML syntax was motivated by convenience to have a quick prototype; for a better leveraging of the graphical elements, we would need model editors with richer graphical capabilities.

Future work will focus on model transformations for mapping ParML (semi-)automatically to more platform-specific models, on combining ParML with both textual and graphical domain-specific languages, by adding modeling support for parallel component frameworks (e.g. skeletons [3] or self-tunable components [11]), on model-

level optimizations, and on automatic code generation from platform-specific ParML.

Acknowledgements: This research was partially funded by SSF ePUMA and Itea2 OPENPROD.

References

- [1] David Abramson, Colin Enticott, and Ilkay Altinas. Nimrod/K: towards massively parallel dynamic grid workflows. In *Proc. Supercomputing Conf., Austin, Texas*. IEEE, November 2008.
- [2] Acceleo. Acceleo code generator for MDA. <http://www.acceleo.org/pages/home/en>, 2009.
- [3] Markus Ålind, Mattias Eriksson, and Christoph Kessler. Blocklib: A skeleton library for Cell Broadband Engine. In *Proc. ACM Int. Workshop on Multi-core Software Engineering (IWMSE-2008) at ICSE-2008, Leipzig, Germany*, May 2008.
- [4] CEA. Papyrus for UML, web site. www.papyrusuml.org, 2009.
- [5] Mikhail Chalabine and Christoph Kessler. A formal framework for automated round-trip software engineering in static aspect weaving and transformations. In *Proc. ACM SIGSOFT/IEEE 29th Int. Conf. on Software Engineering (ICSE-2007)*, May 2007.
- [6] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE*, 91(2), January 2003.
- [7] Peter Fritzson. *Principles of Object Oriented Modeling and Simulation with Modelica 2.1*. Wiley-IEEE Press, February 2004.
- [8] Jörg Keller, Christoph Kessler, and Jesper Träff. *Practical PRAM Programming*. Wiley, New York, 2001.
- [9] Christoph Kessler. Managing distributed shared arrays in a bulk-synchronous parallel environment. *Concurrency – Pract. Exp.*, 16:133–153, 2004.
- [10] Christoph Kessler, Peter Fritzson, and Mattias Eriksson. NestStepModelica – mathematical modeling and bulk-synchronous parallel simulation. In Bo Kågström, Erik Elmroth, Jack Dongarra, and Jerzy Wasniewski, editors, *Applied Parallel Computing – State-of-the-Art in Scientific and Parallel Computing (PARA 2006, Umea, Sweden, June 2006)*, pages 1006–1015. Springer LNCS vol. 4699, 2007.
- [11] Christoph Kessler and Welf Löwe. A framework for performance-aware composition of explicitly parallel components. In *Proc. ParCo-2007 conference, Jülich/Aachen, Germany, Sep. 2007*. In C. Bischof et al. (eds.): *Parallel Computing: Architectures, Algorithms and Applications, Advances in Parallel Computing Series, Volume 15, IOS Press*, pages 227–234, February 2008. Also published as: NIC Series Vol. 38, Dec. 2007.
- [12] Christoph W. Keßler. NestStep: Nested Parallelism and Virtual Shared Memory for the BSP model. *The J. of Supercomputing*, 17:245–262, 2000.
- [13] Christoph W. Keßler and Helmut Seidl. The Fork95 Parallel Programming Language: Design, Implementation, Application. *Int. J. Parallel Programming*, 25(1):17–50, February 1997.
- [14] Kathleen Knobe and Vivek Sarkar. Array SSA form and its use in parallelization. In *Proc. 25th ACM SIGPLAN Symp. Principles of Programming Languages*, pages 107–120, 1998.
- [15] Ouassila Labbani, Jean-Luc Dekeyser, Pierre Boulet, and Eric Rutten. Introducing control in the Gaspard2 data-parallel metamodel: Synchronous approach. In *Proc. Int. Workshop MARTES: Modelling and Analysis of Real-Time and Embedded Systems (with Models/UML 2005), Montego Bay, Jamaica, October 2005*.
- [16] Hannu Niemistö, Teemu Lempinen, and Tommi Karhela. System dynamic business process modelling and simulation tool based on OpenModelica. In *Proc. 2nd OpenModelica Annual Workshop, Linköping, Sweden, Feb. 8, 2010 (www.openmodelica.org)*, 2010.
- [17] Kaj Nyström and Peter Fritzson. Parallel simulation with transmission lines in Modelica. In *Proc. 5th Int. Modelica Conference (Modelica’2006), Vienna, Austria, September 2006*.
- [18] Sabri Pllana and Thomas Fahringer. UML based modeling of performance oriented parallel and distributed applications. In *Proc. 2002 Winter Simulation Conference*, pages 497–505, 2002.
- [19] Wolfgang Reisig. *Petri Nets, An Introduction*. Springer, 1985.
- [20] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Manual, Second Edition*. Addison-Wesley, 2004.
- [21] M. Scherger, J. Baker, and J. Potter. Using the UML to describe the BSP model of parallel computation. In *Proc. Int. Conf. Parallel and Distributed Processing Technology and Applications*, 2002.
- [22] Julien Taillard, Frederic Guyomarch, and Jean-Luc Dekeyser. OpenMP code generation based on a model driven engineering approach. In *Proc. High-Performance Computing and Simulation Conference*, 2008.

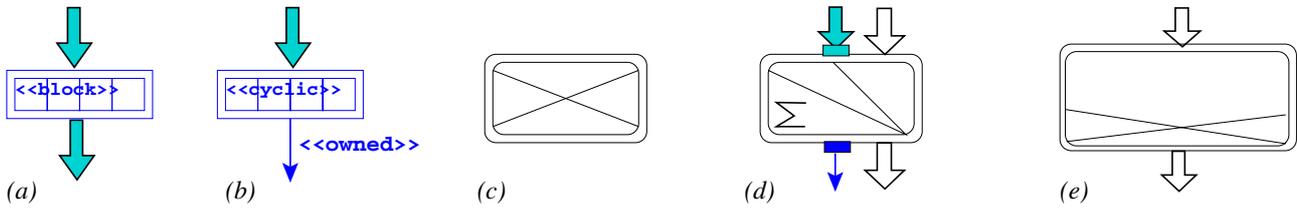


Figure 9: Constructs for parallel data flow and collective communication in extended activity diagrams, tailored towards distributed memory platforms and generation of code in a partitioned global address space language: (a) Annotation of a parallel data flow edge, indicating storage in a block-wise distributed shared array. (b) Extraction of the thread-owned (local) partition of a cyclically distributed shared array. (c) Data redistribution. (d) Parallel reduction of a shared array (global sum). (e) BSP superstep. For (a)–(c), parallel control flow arrows are not shown. For (c) and (e), data flow pins and arrows are not shown.

```

sh<=> float weight[] = { 0.5, 0.5 };

extern void seqSort( float A[], float B[], int n );
extern void BSParMergeInPlace( sh float B[]</>, sh<=> int n );

void BSParMergeSort( sh float A[]</>, sh float B[]</>, sh<=> int n )
    // A, B are block-wise distributed shared arrays
{
    if (thisgroup_size()==1)
        seqSort ( owned(A), owned(B), n ); // works on local arrays
    else {
        neststep ( 2, weight ) { // nested superstep, group splitting
            if (@==0) BSParMergeSort( ); // first subgroup
            else      BSParMergeSort( ); // second subgroup
        }
        step
            BSParMergeInPlace( B, n );
    }
}

```

Figure 10: Bulk-Synchronous Parallel Merge Sort in NestStep source code.

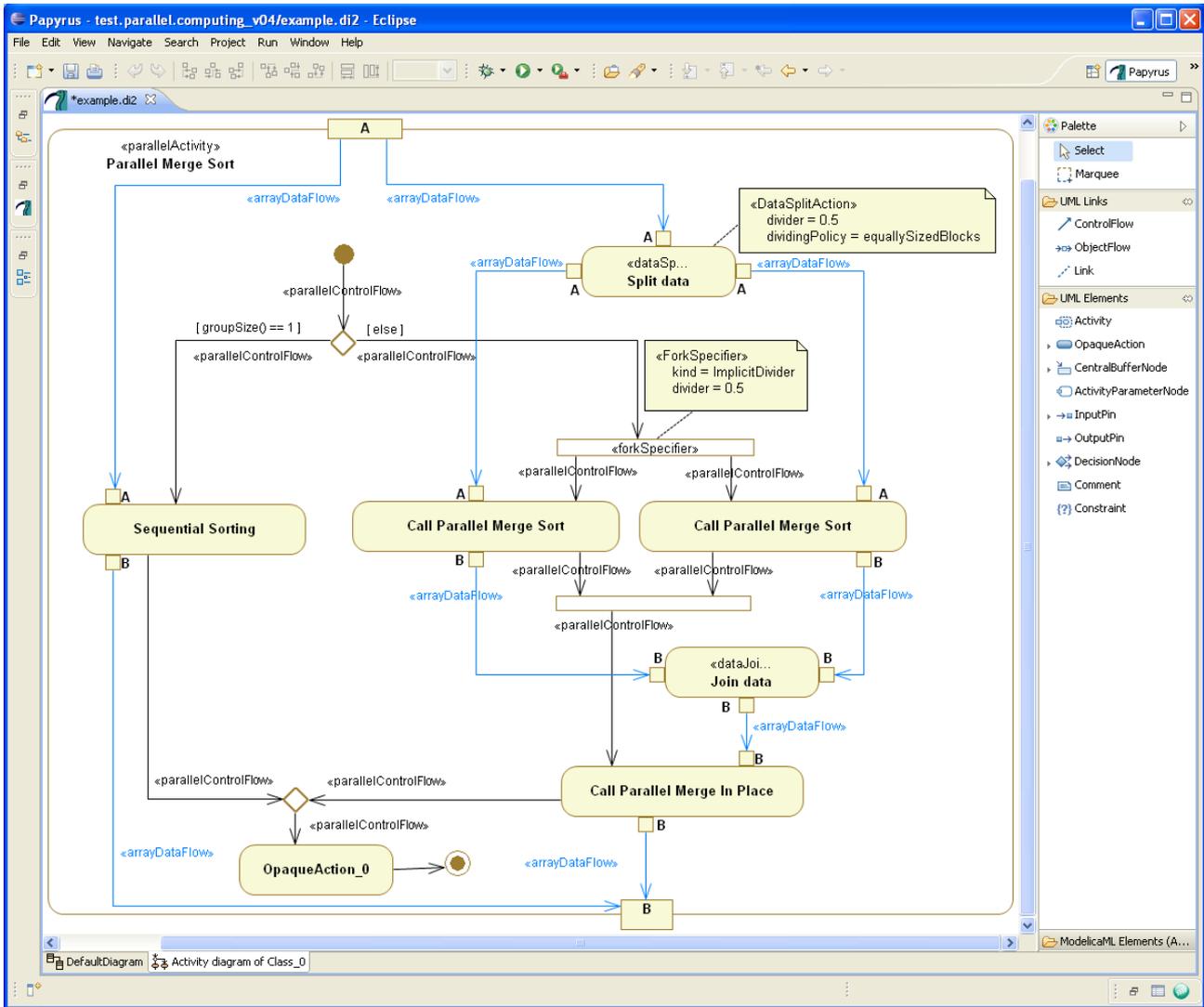


Figure 13: Screenshot of the Papyrus tool, showing the ParMergeSort activity after manually adapting the data flow towards the ParMerge call activity for in-place merging. The array data flow join element enforces that both subarrays are placed in a common array that becomes input to ParMerge. Explicit copying for this purpose can be avoided if the placement constraint is backwards propagated to make the dataflow predecessor activities write their result directly into the right destination locations.