ReFlO: An Interactive Tool for Pipe-And-Filter Domain Specification and Program Generation

Rui C. Gonçalves, Don Batory, João L. Sobral

Abstract—ReFlO is a framework and interactive tool to record and systematize domain knowledge used by experts to derive complex *pipe-and-filter (PnF)* applications. Domain knowledge is encoded as transformations that alter PnF graphs by *refinement* (adding more details), *flattening* (removing modular boundaries), and *optimization* (substituting inefficient PnF graphs with more efficient ones). All three kinds of transformations arise in reverse-engineering legacy PnF applications. We present the conceptual foundation and tool capabilities of ReFlO, illustrate how parallel PnF applications are designed and generated, and how domain-specific libraries of transformations are developed.

Index Terms—MDE, Tools, Software Architectures, Design by Transformation, Refinement, Optimization, Graph Transformations

1 Introduction

Component Based Software Engineering (CBSE) promotes the development of software by graphically wiring together reusable components. CBSE tools foster a circuit analogy to software development and, like actual circuit design tools, can express hierarchical systems by levels of abstraction: a component at level i is defined in terms of a circuit of more primitive components at level $i+1$, recursively. CBSE is an early example of *Model Driven Engineering (MDE)* where models (*ie* hierarchical circuit diagrams) are transformed into executables.

Pipe-and-filter (PnF) or *streaming* systems are among the fundamental architecture styles used in CBSE [1], [2], where components are functions that process data that is transmitted through wires [3], [4], [5], [6]. Some time ago, we were given the task to re-engineer expert-created legacy PnF applications: a parallel database query processor and a crash-fault-tolerant server. The PnF graphs of these systems were spaghetti diagrams; our understanding of how these systems worked was minimal. We could not explain their PnF graphs nor did we know if they were correct.

Step-wise development provided an answer. We start with an elementary PnF graph that cleanly and abstractly describes the system to be reverse-engineered. In MDE terms, this is a *Platform Independent Model (PIM)*: a model that does not constrain the implementation or target platform and is an abstract specification of what to build. We then *derived* the target PnF graph (a *Platform Specific Model (PSM)* [7]) by applying a series of transformations that are well-known to engineers in that domain. Further, each transformation was simple enough to be demonstratably correct (by proof or other means). Our derivations were *correct by construction* [8].

We had to depart from contemporary CBSE tools to admit *architectural optimizations*—the ability to replace a PnF subgraph with another PnF subgraph that implements the same functionality but in a different way (to yield improved quality metrics, like performance). Optimizations were essential to our re-engineering tasks; *we could not derive legacy PnF graphs without them*. With *Model-to-Text (M2T)* transformations, we reproduced these legacy applications from our models.

Our critical insight was to recognize that *the transformations used to derive a PnF graph are building-blocks just as important as the components used in the application itself*.

This paper presents ReFlO, an interactive tool that embodies a derivational approach to PnF graphs. Initially we used ReFlO to reverse-engineer the design of legacy applications—an example of which we illustrate in this paper. Over time, the library of transformations that are used in deriving a family or domain of similar applications becomes extensive enough for forward-engineering. That is, given a PIM of an application, cataloged transformations can be used to mechanically derive the space of all PSMs and automatically select the most efficient. Our work on forward-engineering is not the focus of this paper and is detailed elsewhere [9]. Nonetheless, the strong connection of ReFlO to forward engineering demonstrates the significance of derivational approaches.

The contributions of this paper are:

- a simple way to encode domain knowledge of PnF graph construction as transformations,
- how ReFlO can be used as an interactive design tool to derive custom PnF graphs,
- an explanation why the Perry Substitution Principle, rather than the Liskov Substitution Principle, is central to derivational development of optimized PnF programs,
- how ReFlO provides a framework to allow different interpretations of PnF graphs to compute properties about them (besides producing executables), and
- how multiple derivations of a PSM can expose new transformation rules of a domain.

2 Foundational Concepts: Part I

2.1 PnF Graphs, Refinements, and Optimizations

A *pipe-and-filter (PnF) graph* [1] is a directed multigraph, where boxes (components) process data that is passed to other boxes via connectors (pipes). Boxes may receive inputs from different sources and compute zero or more outputs. Input ports are drawn as nubs on the left-side of boxes; output ports are drawn as nubs on the right. A connector links an input port to an output port. Figure 1 shows a PnF graph modeling a program,

- *R. Gonçalves and J. Sobral are with the Departamento de Informática, Universidade do Minho, Braga, Portugal*
Email: {rgoncalves, jls}@di.uminho.pt
- *D. Batory is with the Department of Computer Science, The University of Texas at Austin, Austin, TX, USA*
Email: batory@cs.utexas.edu

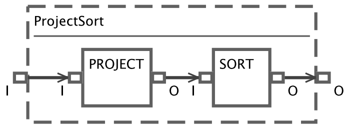


Fig. 1. The PnF graph `ProjectSort`.

called `ProjectSort`, that projects (eliminates) attributes of the tuples of its input stream and then sorts them.

We call boxes `PROJECT` and `SORT` *interfaces* as they specify only abstract behavior (their inputs and outputs, and, informally, their semantics). Besides input ports, boxes may have other inputs that are not shown graphically, such as the sort key for the `SORT` box or the list of attributes to remove for the `PROJECT` box. We call the former *essential parameters* and the latter *additional parameters* [10].

Figure 1 is a PIM as it makes no reference to or demands on its concrete implementation. It is a high-level specification that can be adapted to a particular platform or for particular inputs. Adaptation is accomplished in `ReFlO` by applying transformations.

A transformation can map an interface directly to a *primitive* box, representing a concrete code implementation. Besides primitives, there are other implementations of an interface that are expressed as a PnF graph, called *algorithms*. Algorithms may reference interfaces. Figure 2 is an algorithm. It shows the PnF graph called `parallel_sort` of a map-reduce implementation of `SORT`. Each box inside Figure 2, namely `SPLIT`, `SORT` and `SMERGE` (sorted merge), is an interface which can be subsequently elaborated.

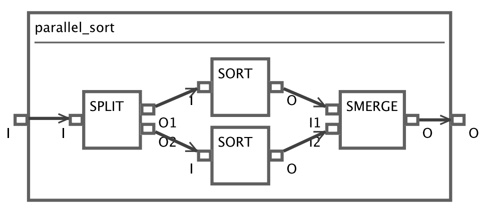


Fig. 2. `parallel_sort` implements `SORT` by map-reduce.

Refinement [11] is the replacement of an interface with one of its implementations (primitive or algorithm). By repeated refinements, eventually a graph of wired primitives is produced.

Figure 1 can be refined by replacing `SORT` with its `parallel_sort` algorithm and `PROJECT` with a similar map-reduce algorithm. Doing so yields the graph of Figure 3(a), or equivalently the graph of Figure 3(b), obtained by removing modular boundaries. Removing modular boundaries is called *flattening*.

Refinements alone are insufficient to derive complex PnF graphs. Look at Figure 3(b). We see a `MERGE` followed by the `SPLIT` operation, that is, two streams are merged and the resulting stream is immediately split again. Let interface `IMERGESPLIT` be the operation that receives two input streams, and produces two other streams, with the requirement that the union of the input streams is equal to the union of the output streams (see Figure 4). `ms_mergesplit` is one of its implementations. However, the `ms_identity` algorithm provides an alternative implementation that is obviously more efficient than `ms_mergesplit` as it does not require `MERGE`

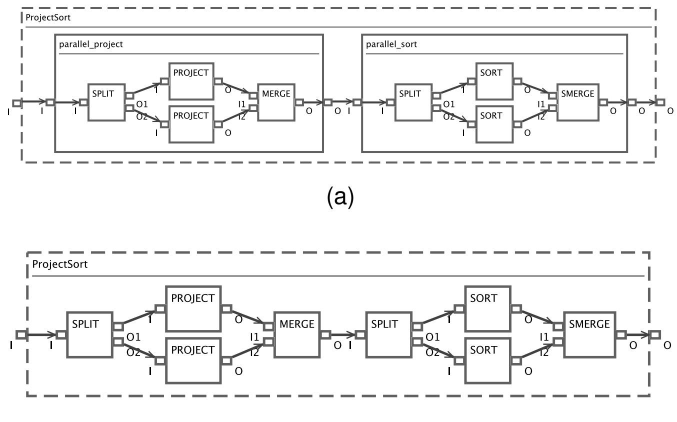


Fig. 3. Parallel version of `ProjectSort`.

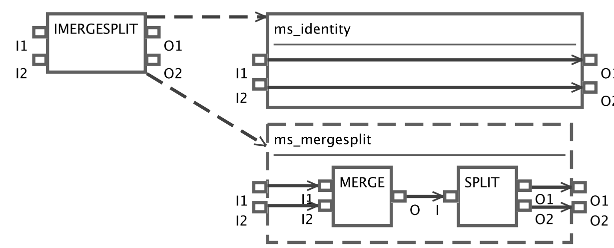


Fig. 4. Two implementations of the `IMERGESPLIT` interface.

and `SPLIT` computations.[1]

We can use `ms_identity` to optimize `ProjectSort`. The first step is to *abstract* Figure 3(b) with the `IMERGESPLIT` interface, obtaining Figure 5(a). Then, we refine `IMERGESPLIT` to its `ms_identity` algorithm, to obtain the optimized graph for `ProjectSort` (Figure 5(b)). We call the action of abstracting an (inefficient) composition of boxes to an interface and then refining it to an alternative implementation an *optimization*.[2]

2.2 Perry Substitution Principle

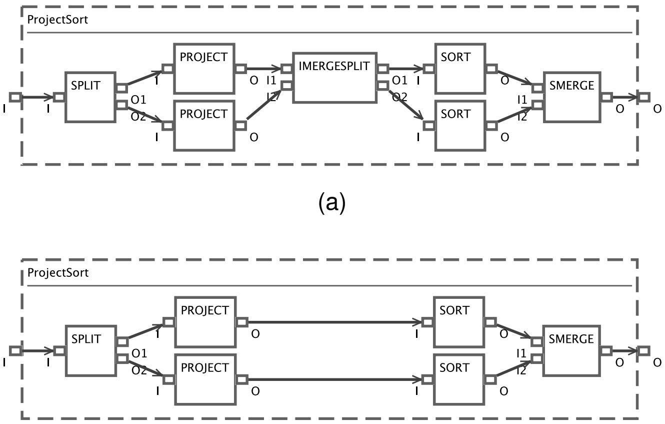
By studying several legacy applications from the same domain, it becomes obvious that there is a set of transformations that are commonly used in derivations. The collected set of transformations contains interface/implementation pairs (`I`, `A`), that we call *rewrite rules*, and specifies two distinct kinds of transformations:

- **Refinement** `I` $\rightsquigarrow$ `A`: An interface `I` is replaced by a graph `A` which represents a primitive or algorithm, and
- **Abstraction** `A` $\rightsquigarrow$ `I`: A graph `A` is replaced by interface `I`.

Under what circumstances is a rewrite rule permitted? A possible answer is based on the *Liskov Substitution Principle (LSP)* [12], which is a foundation of object-oriented design. LSP states that if `A` is a subtype of `I`, then objects of type `A` can be substituted for objects of type `I` without altering the correctness properties of a program. Substituting an interface

1. Readers may notice that algorithms `ms_mergesplit` and `ms_identity` do not necessarily produce the same result. However, both implement the semantics specified by `IMERGESPLIT`, and the result of `ms_identy` is one of the possible results of `ms_mergesplit`, *ie* `ms_identity` removes non-determinism.

2. Although called *optimizations*, they do not necessarily improve performance, but combinations of them typically do.



(a)

(b)

Fig. 5. Optimization of `ProjectSort`.

with an implementing object (component) is standard fare today, and is a way to realize refinement in LSP [13], [14]. The technical rationale behind LSP is that preconditions for using subtype `A` can *not* be stronger than preconditions for type `I`, and postconditions for `A` are *not* weaker than that for `I` [12]:

$$\mathtt{pre(I)} \Rightarrow \mathtt{pre(A)}$$
$$\mathtt{post(A)} \Rightarrow \mathtt{post(I)}$$

To our surprise, LSP is too restrictive for `ReFlO` graph rewrite rules, as *implementations are often accompanied by preconditions that are not required by their interfaces*. Such implementations are often more efficient than those that are not as specialized [15].

Example: Figure 6 shows three implementations of the `SORT` interface: a map-reduce algorithm, a `quicksort` primitive, and a `do_nothing` algorithm. `do_nothing` says: if the input stream is already in sorted order (a precondition not present in `SORT` but definitely present for `do_nothing`), then there is no need to sort. The (`SORT`, `do_nothing`) rewrite rule violates LSP: `do_nothing` implementation has *stronger* preconditions than its `SORT` interface. This is a common situation in graph rewrites.

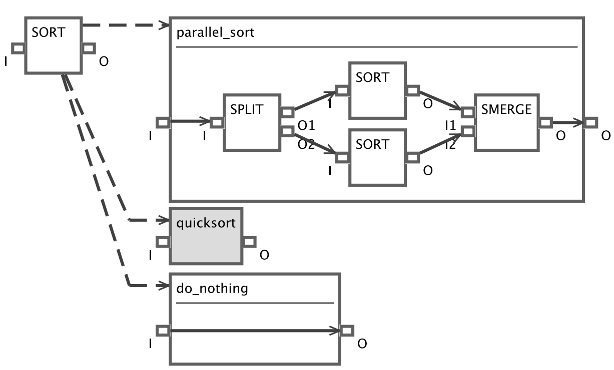


Fig. 6. Two algorithms and a primitive implementation of `SORT`.

Forcing our rewrite rules to comply with LSP, the standard notion of substitutability for object-oriented interface/implementation refinements, *we would not be able to derive the optimized programs that domain experts created manually*. When looking for alternative notions of substitutability, we found an existing precedence for a solution. Let `A` and `I` be boxes, and `pre` and `post` denote the pre- and postconditions of a box. Perry [16] defined that `A` is *upward compatible* with `I` if:

$$\mathtt{pre(A)} \Rightarrow \mathtt{pre(I)} \tag{1}$$
$$\mathtt{post(A)} \Rightarrow \mathtt{post(I)} \tag{2}$$

ie `A` requires and provides at least the same as `I`. We call this the *Perry Substitution Principle (PSP)*. It allows the specification of implementations specialized for certain inputs, essential during the derivation of optimized program implementations.

Not requiring rewrite rules to conform to LSP, and allowing an interface to be replaced with an implementation with stronger preconditions, means that a rewrite rule is *not always applicable* (it depends on the PnF graph we are refining). To guarantee that the behavior of the PnF graph is preserved when replacing interface `I` with implementation `A`, we must guarantee that the preconditions of `A` are met (in the context of PnF graph being transformed). If not, `ReFlO` disallows it.

Consider the `do_nothing` implementation of `SORT` and `ProjectSort` of Figure 1. Algorithm `do_nothing` has a precondition that requires its input to be sorted in an appropriate order (*eg* on ascending values of field `F`). We can use this rewrite rule in `ProjectSort` (to replace `SORT`) only if this precondition is met, *ie* if `PROJECT` has a postcondition specifying its output is sorted in ascending `F` order. Typically, `PROJECT` provides no such postcondition, thus `ReFlO` disallows `do_nothing` algorithm for `ProjectSort`. However, if `PROJECT` exported a postcondition specifying the sort order of its output, the input of `SORT` was in ascending `F` order, `do_nothing` would be a valid replacement of the `SORT` interface. In this scenario, even though `do_nothing` has stronger preconditions than `SORT`, it can be used, and the behavior of `ProjectSort` would be preserved.

If we assure the preconditions of the implementation being added (`A`) are met in the PnF graph being transformed (taking into account the postconditions of the boxes that compute the inputs of `A`), we guarantee that the transformation preserves the behavior of the PnF graph being transformed (*ie* no precondition is added to the PnF graph, and the postconditions are preserved).

Rewrite rules used in abstraction transformations $\mathtt{A} \rightsquigarrow \mathtt{I}$ have stronger constraints. An abstraction implies that a graph `A` must implement `I`, *ie* $\mathtt{I} \rightsquigarrow \mathtt{A}$. For both constraints to hold, the pre- and postconditions of `A` and `I` must be equivalent:

$$\mathtt{pre(I)} \Leftrightarrow \mathtt{pre(A)} \tag{3}$$
$$\mathtt{post(I)} \Leftrightarrow \mathtt{post(A)} \tag{4}$$

To summarize, refinement is a general concept [17]. In object-oriented designs, refinement is satisfied by LSP where an interface can be substituted with an implementing object. In MDE, a refinement corresponds to mapping of a model of of one type (metamodel) to that of another. In the world of `ReFlO` graph rewrite rules, refinement satisfies PSP.

3 Domain Model Specification

`ReFlO` (**Re**fine, **Fl**atten, and **O**ptimize) is an interactive tool to draw and derive PnF graphs, built upon the ideas of *Design by Transformation (DxT)* [18]. The rewrites that `ReFlO` applies are taken from a *domain model*—a library of graph transformations whose structure we explained in Section 2. ReFlO provides support for experts build such models.

3.1 Basic Features of a Domain Model

A *ReFlO Domain Model (RDM)* is a set of ordered pairs that associate an interface with an implementing algorithm or primitive. That is, a RDM encodes a library of transformations that can be applied to programs in a given domain. ReFlO provides the following objects to create RDMs: interfaces, primitives, algorithms, input/output ports, connectors, implementation links, and patterns. The UML class diagram of the RDM metamodel is Figure 7.

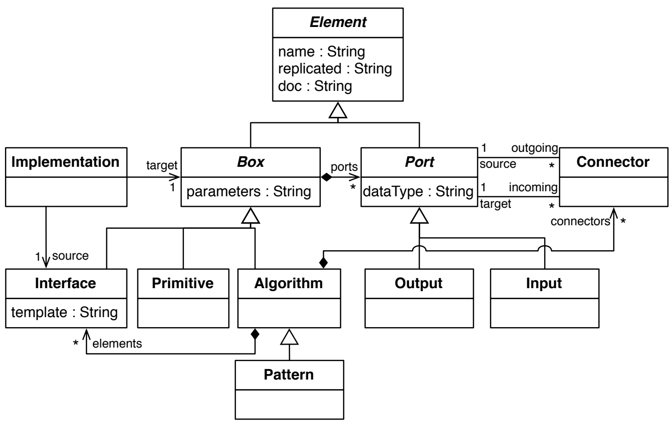


Fig. 7. RDM UML class diagram.

An *interface* is a named box with input and output ports. A *primitive* is drawn identically, except that primitives have a gray background whereas interfaces are white (Figure 6). Every *port* of a box has a unique name (to distinguish it from other ports) and a data type. A *connector* links a source port to a target port.

An *algorithm* is a named box with I/O ports that encloses a PnF graph.[3] A *pattern* is a special algorithm that not only implements its interface, but also specifies that its graph can be replaced with (or abstracted to) an interface, as part of an optimization. ReFlO graphically distinguishes patterns as dashed-line boxes from algorithms that are solid-line boxes (see Figure 4).

A domain model is specified in ReFlO by defining each interface, primitive, and algorithm. A *rewrite rule* is an ordered pair (interface, primitive) or (interface, algorithm) which is drawn/specified by an *implementation* link (a dashed arrow) connecting an interface to an implementation.

> **Example:** Figure 6 defined three implementations of the SORT interface: the parallel_sort algorithm, the quicksort primitive, and the do_nothing algorithm.

> **Example:** Figure 4 specified that pattern ms_mergesplit can be abstracted to the IMERGESPLIT interface, which can then be refined to the ms_identity algorithm. This compound rewrite was the optimization that we used earlier.

3. We refer to the interfaces (boxes) contained inside an algorithm as *internal interfaces (boxes)*, and to the algorithm as the *parent box* of those interfaces.

3.2 Advanced Features

3.2.1 Additional Parameters

Every box has a *parameters* attribute which holds a comma-separated list of names, data types and values, that specify the box's additional parameters. The value of an additional parameter may be a constant or the value of a parameter of its parent box. Additional parameters keep ReFlO diagrams simpler, allowing developers to focus on the essential parts of the model.

3.2.2 RDM Documentation

Transformation rules must be documented, so that others who inspect PnF graphs can understand the rules that were used to derive it. ReFlO boxes and ports have the *doc* attribute, where designers can place a textual description of model elements. ReFlO generates HTML documentation that contains the figures of boxes and their descriptions. This allows users to reference HTML pages for rule definitions. The HTML documentation for the rules that we use later in our case study is at http://www.cs.utexas.edu/users/schwartz/DxT/case-studies/gamma/models/databases.html.

3.2.3 Templates

Many rewrite rules are parameterized clones of each other. ReFlO was designed so that any rewrite rule could be used as a template. Every rewrite rule has a *template* attribute; if its value is null, the rule is not a template. A non-null value specifies (template box name, concrete box name) bindings to create a new instance of the rewrite rule. Typically a non-null value specifies multiple groups of bindings, one binding for every new instance of a rule. Details are given in [19].

> **Example:** The rewrite rules of Figure 8 define an optimization. Whenever box x_1 is followed by box x_2, where $x_2 = x_1^{-1}$ (the inverse operation of x1), box x_2 can be removed, yielding algorithm idx1.

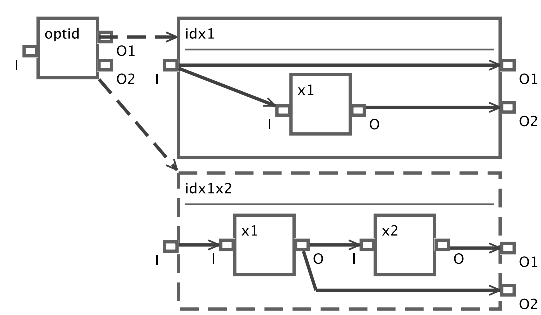


Fig. 8. A template with parameters optid, x1 and x2.

Figure 8 is a template for stamping out customized copies of itself. Using these bindings {(optid,OptIdF), (x1,F1), (x2,F2)} where $F2 = F1^{-1}$, ReFlO produces the customized rewrite rules of Figure 9. Additional bindings can produce other instances. Templates provide an elementary form of high-order transformations that reduce modeling effort [20].

3.2.4 Replicated Elements

Figure 2 showed the parallel_sort algorithm where two instances of SORT are performed in parallel. We want to specify a rewrite with an arbitrary number of instances. We use *replicated elements*. Ports and boxes have a [bracketed attribute]

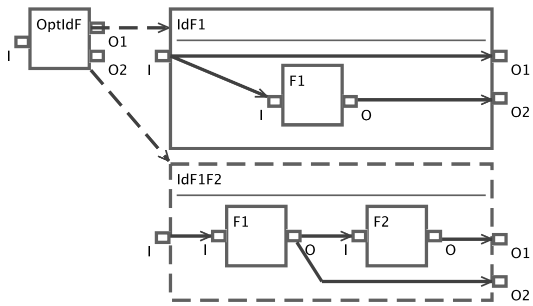


Fig. 9. A template instance.

that specifies replication. If brackets are absent, the element is not replicated. If a bracket contains an upper case letter, that is interpreted as a *replication variable* that specifies how many times the element is replicated.[4] Thus, box B[N] means that there are N instances of box B (B_i, for $i \in \{1 \ldots N\}$). Similarly for ports.

Example: Figure 10 expresses parallel_sort in a more general way. SPLIT has N output ports $\{O_1 \ldots O_N\}$. There are N SORT boxes $\{SORT_1 \ldots SORT_N\}$. SPLIT output port O_i is connected to input port I of $SORT_i$. Finally, the input port I of SMERGE is replicated $\{I_1 \ldots I_N\}$. The output of $SORT_i$ is connected to SMERGE input port I_i. Figure 2 is produced by setting $N = 2$.

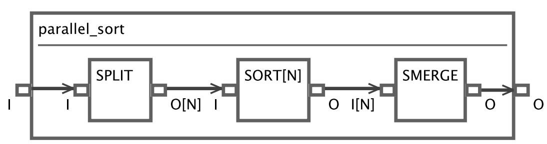


Fig. 10. parallel_sort with replicated elements.

Example: Figure 11 defines transformations where elements can be replicated a different number of times. The interface has N inputs and M outputs. Each pattern replicates some elements N times and others M times.

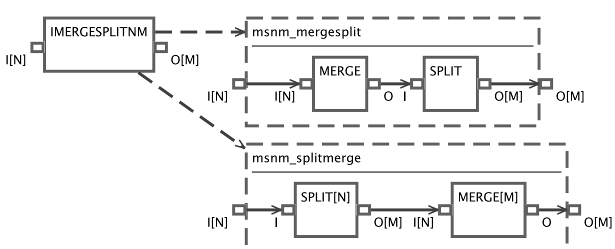


Fig. 11. MERGE − SPLIT cross product.

ReFlO has specific rules for replicating connectors (*ie* connectors linking replicated ports or ports of replicated boxes). Using the notation B.P to represent port P of box B, given a connector from output port O of box B to input port I of box C, the rules are:

- When O is replicated N times and B is not (which implies that either I or C is also replicated N times), connectors link $B.O_i$ to $C.I_i$ or $C_i.I$ (depending on which is replicated), for $i \in \{1 \ldots N\}$.
- When B is replicated N times and O is not (which implies that either I or C is also replicated N times), connectors link $B_i.O$ to $C.I_i$ or $C_i.I$ (depending on which is replicated), for $i \in \{1 \ldots N\}$.
- When B is replicated N times and O is replicated M times (which implies that both C and I are also replicated), connectors link $B_i.O_j$ to $C_j.I_i$, thereby implementing a *crossbar*, for $i \in \{1 \ldots N\}$ and $j \in \{1 \ldots M\}$ (this implies that C is replicated M times, and I is replicated N times).

Example: Figure 12 is the result of setting N and M to 2 in algorithm msnm_splitmerge from Figure 11. Note the crossbar resulting from connectors that link replicated ports of replicated boxes.

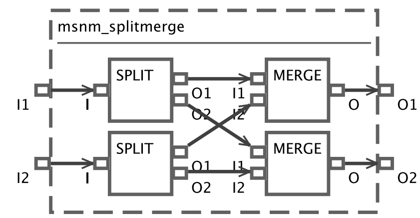


Fig. 12. msnm_splitmerge pattern without replication.

The mapping of a PIM to a PSM in ReFlO is discussed next.

4 Interactive Derivation of PSMs from a PIM

ReFlO is an interactive tool that allows designers to (1) define an RDM, (2) define a PIM, and (3) use the transformations of an RDM to progressively rewrite a PIM into a PSM. In the typical use, a domain expert starts by using ReFlO to reverse engineer legacy programs. During this process, he replays the development process, adding to the RDM the transformations that he, sometimes unconsciously, applied to code. The RDM may then be used by other developers to optimize their programs (directly in ReFlO, or exporting the RDM to an external tool [21]).

The actions domain experts and developers can invoke when transforming a PnF graph are:

- **Refine** replaces a user selected interface with one of its implementations. ReFlO examines each potential refinement and only displays those that satisfy the $\mathcal{I} \rightsquigarrow \mathcal{G}$ constraints of Section 2.2.[5] If only one option is available, ReFlO automatically selects it.[6]
- **Flatten** removes the modular boundaries of the selected graph that result from refining a PnF graph. If the graph to be flattened was replicated, this information is pushed down to its internal boxes.
- **Abstract** replaces the selected boxes with the interface they implement. ReFlO matches selected boxes with the

4. At design time, the variable only allow us to determine whether to elements are replicated the same number of times. These variables can be instantiated when generating code.

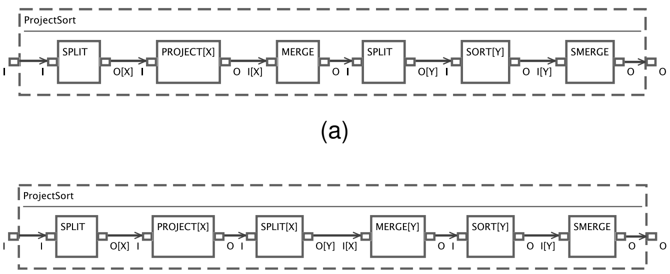
5. In Section 5.2 we provide additional details about how ReFlO verify these constraints.

6. Replication parameters of an interface are used to set the replication parameter(s) of an implementation. If an implementation has replication parameters that are not present in the interface, the user is asked to provide a value for the parameter.

patterns in the RDM. Unlike in refinements, no preconditions check is needed to decide whether a pattern can be replaced by the interface. However, to decide whether the selected boxes are an instance of the pattern $\mathcal{G}$ we need to put the modular boundaries of $\mathcal{G}$ around the boxes, and verify if $\mathcal{G}$ preconditions are met. That is, it is not enough to verify if the selected boxes have the shape of the pattern. If one match is found, the pattern is replaced by its interface. If multiple patterns match, the user is asked to choose one.[7]

- **Optimize** performs an abstraction, refinement, and flattening as a single step, replacing the selected set of boxes with an equivalent implementation.

  Example: `ReFlO` maps Figure 13(a) to 13(b) by applying the optimization of Figure 11. (Note the replication variables `X` and `Y` of the original graph are used to define the replication variables of the new graph.)



(a)

(b)

Fig. 13. Optimizing a parallel version of `ProjectSort`.

- **Find Optimization** locates all possible matches for the patterns in the RDM that exist inside the selected graph. The interfaces that comprise the matches are identified setting their attribute *label* to contain a tag identifying the match(es).

  Example: Applying find optimization to the `ProjectSort` graph of Figure 3b results in the graph of Figure 14, where we can see that two boxes are part of a match (of pattern `ms_mergesplit`).

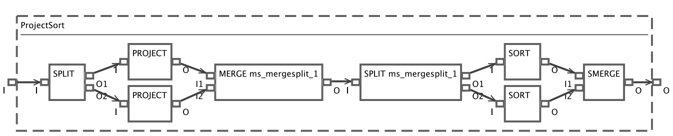


Fig. 14. The label shown after the name of boxes `MERGE` and `SPLIT` indicates that they are part of a match of pattern `ms_mergesplit`.

- **Expand** expands replicated boxes and ports of a graph. For each replicated box, a copy is created. For each replicated port, a copy is created (suffixes `1` and `2` are added to names of original port and its copy, respectively, as two ports cannot have the same name). Connectors are copied according to the rules previously defined.

  Example: Figure 15 is an expansion of Figure 13(b).

7. The values of replication parameters of the pattern are used to define the replication parameters of the interface. The same is done to define the values of the additional parameters of the new interface.

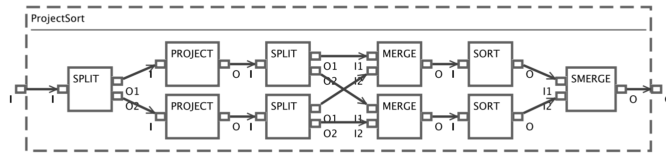


Fig. 15. Expanding the parallel, replicated `ProjectSort`.

5 Foundational Concepts: Part II

5.1 Interpretations

A PnF graph `P` may have many interpretations. The default is to interpret each box of `P` as the component it represents. That is, `SORT` means "sort the input stream". We call this the *standard interpretation* $\mathcal{S}$. The standard interpretation of box `B` is denoted $\mathcal{S}$(`B`) or simply `B`, *eg* $\mathcal{S}$(`SORT`) is "sort the input stream". The standard interpretation of graph `P` is $\mathcal{S}$(`P`) or simply `P`.

There are other interpretations of `P`. $\mathcal{ET}$ interprets each box `B` as a computation that *estimates the execution time* of $\mathcal{S}$(`B`), given statistics about $\mathcal{S}$(`B`)'s inputs. So $\mathcal{ET}$(`SORT`) is "return an estimate of the execution time to produce `SORT`'s output stream". Each box `B` $\in$ `P` has exactly the same number of inputs and outputs as $\mathcal{ET}$(`B`) $\in$ $\mathcal{ET}$(`P`), but the meaning of each box as well as the types of each of its I/O ports are different.

> **Example:** $\mathcal{ET}$(`ProjectSort`) estimates the execution time of `ProjectSort` for an input `I` whose statistics (tuple size, stream length, etc.) is $\mathcal{ET}$(`I`).

> **Example:** We said in Section 1 that an RDM can be used to forward-engineer (*eg* derive) all possible PSMs from an input PIM. The estimated run-time of a PSM `P` is determined by executing $\mathcal{ET}$(`P`). The most efficient PSM that implements the PIM is the one with the lowest estimated cost [22].

> **Example:** $\mathcal{M}2\mathcal{T}$(`ProjectSort`) is a model-to-text interpretation that maps `ProjectSort` to executable code.

> **Example:** Pre- and postconditions guarantee the correctness of `ReFlO` graphs. Each is encoded as a distinct interpretation, discussed further in Section 5.2.

In general, an interpretation $\mathcal{I}$ of graph `P` is an isomorphic graph $\mathcal{I}$(`P`), where each box `b` $\in$ `P` is mapped to a unique box $\mathcal{I}$(`b`) $\in$ $\mathcal{I}$(`P`) and each edge $b_1 \to b_2 \in$ `P` is mapped to a unique edge $\mathcal{I}(b_1) \to \mathcal{I}(b_2) \in \mathcal{I}$(`P`). In `ReFlO`, graph $\mathcal{I}$(`P`) is identical to `P`, except that the bindings of all boxes to computations are different.

5.1.1 Implementing Interpretations

It is reasonable to expect that each interpretation would be written in its own *domain-specific language (DSL)*. Creating such DSLs was not critical to our goal of developing and demonstrating `ReFlO`. Indeed, this would be an entire research project unto itself. Instead, we chose to write each interpretation in Java. For each interpretation and box, a Java class must be provided by a developer. Every interpretation is represented by a collection of classes, one per box, that is stored in a unique Java package whose name identifies the interpretation. Thus if there are `n` interpretations, there will be `n` Java packages provided by a domain designer.

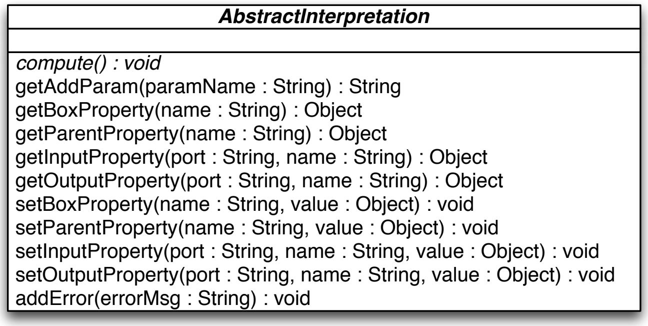


Fig. 16. The `AbstractInterpretation` class.

Each class has the name of its box and must extend abstract class `AbstractInterpretation` that is provided by `ReFlO` (see Figure 16). Interpretations grow in two directions: (i) new boxes can be added to the domain, which requires new classes to be added to each package, and (ii) new interpretations can be added, which requires new packages.

Each interpretation maintains its own data, which we call *properties*. The behavior of an interpretation is specified in method `compute`. It computes and stores properties that are associated with its box or ports. For each box/port, properties are stored in a map that associates a value with a property identifier.[8] `AbstractInterpretation` provides `get` and `set` methods for accessing and modifying properties.

A typical class structure for interpretations is shown in Figure 17(a), where all classes inherit directly from `AbstractInterpretation`. Nevertheless, more complex structures arise. For example, one interpretation may inherit from another (this is common when defining preconditions, as an algorithm has the same preconditions of the interface it implements), or there may be an intermediate class that implements part (or all) of the behavior of several classes (usually of the same interpretation), as depicted in Figure 17(b). Besides requiring classes to extend `AbstractInterpretation`, `ReFlO` allows developers to choose the most convenient class structure for the interpretation at hand.

Although `ReFlO` expects a Java class for each box, if none is provided, `ReFlO` automatically selects an appropriate default class with an empty `compute` method. That is, in cases where there are no properties to set, no class needs to be provided.

> **Example:** `ReFlO` generates complete executables in $\mathcal{M2T}$ interpretations; so interface boxes have no mappings to code.
>
> **Example:** Interpretations that set a ports' property usually do not need to provide a class for algorithms, as the properties of their ports are set when executing the `compute` methods of their internal boxes. This is the case of interpretations that compute postconditions, or interpretations that compute data sizes.
>
> However, there are cases where properties of an algorithm cannot be inferred from its internal boxes. A prime example is the `do_nothing` algorithm—it has preconditions, but its internals suggest nothing. (In such cases, a Java class is written for an algorithm to express its preconditions.)

`ReFlO` executes an interpretation in the following way: for each box in a graph, its `compute` method is executed, with the execution order being determined by the topological order of the boxes (in the case of hierarchical graphs, the interpretation of an algorithm box is executed before the interpretations of its internal boxes). After execution, a developer (or `ReFlO`) may select any box and examine its properties.

8. This map is similar to `java.util.Properties` except that values are of type `Object` instead of `String`.

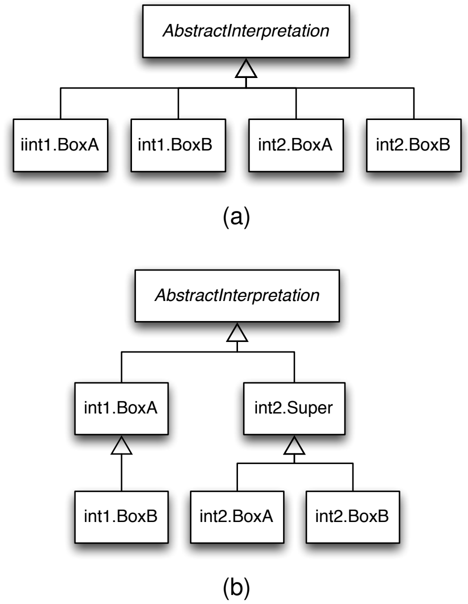


Fig. 17. Class diagrams for two interpretations `int1` and `int2`.

5.1.2 *Forward and Backward Interpretations*

Usually edges of an interpretation $\mathcal{I}$ have the same direction of the corresponding edge of interpretation $\mathcal{S}$. We have found cases where to compute some property about a graph it is convenient to invert the direction of the edges, so that information flows right-to-left. In this case, an edge $\mathtt{b}_1 \to \mathtt{b}_2 \in \mathtt{P}$ maps to a unique edge $\mathcal{I}(\mathtt{b}_1) \leftarrow \mathcal{I}(\mathtt{b}_2) \in \mathcal{I}(\mathtt{P})$. We call such interpretations *backward* and the others are *forward*.

5.1.3 *Composition of Interpretations*

To make all of the above work, interpretations must be composable. Each interpretation computes certain properties of a program `P`, and it may need properties that are computed by other interpretations, *eg* to estimate the execution cost of a box, we may need an estimate of the volume of data output by a box. The same property (volume of data) may be needed for other interpretations (*eg* preconditions). Therefore, it is useful to separate the computation of each property, in order to improve interpretation modularity and reusability.

`ReFlO` supports the composition of interpretations, where two or more interpretations are executed in sequence. An interpretation has access to the properties computed by previously executed interpretations. For example, an interpretation to compute data sizes ($\mathcal{DS}$) can be composed with one that uses data size estimates to form cost estimates ($\mathcal{ET}$). This is the compound interpretation $(\mathcal{ET} \circ \mathcal{DS})(\mathtt{P}) = \mathcal{ET}(\mathtt{P}) \circ \mathcal{DS}(\mathtt{P})$. This allows interpretation $\mathcal{DS}$ to be composed (reused) with other interpretations that also need data sizes.

5.2 Pre- and Postconditions

We use interpretations to compute box postconditions and then verify their preconditions, rather than providing a custom DSL for this purpose (*ie* pre- and postconditions are specified in the same language/framework used for other interpretations, currently Java).

Postconditions are evaluated by the $\mathcal{POST}$ interpretation. $\mathcal{POST}$ computes the properties that are output by a box given the properties that are input to that box. The postconditions of algorithms and patterns are inferred from the postconditions of their internal boxes.[9]

Preconditions are evaluated by the $\mathcal{PRE}$ interpretation. $\mathcal{PRE}$ reads the values of the properties about box inputs (computed by $\mathcal{POST}$), and checks if the preconditions of that box are satisfied. The method `addError` is used to send a message to `ReFlO` signaling a failure validating precondition. Thus `ReFlO` uses $\mathcal{PRE} \circ \mathcal{POST}$ for computing postconditions and validating preconditions.

When a user tries to apply a transformation, `ReFlO` builds the list of possible replacements for the selected box(es). The $\mathcal{POST}$ interpretation is then executed, to compute the postconditions for each box in the graph that is to be transformed. `ReFlO` then evaluates the $\mathcal{PRE}$ interpretation on each replacement graph. If no precondition error is reported, the replacement graph is legal, otherwise it is disallowed.

> **Example:** In Section 2.2 we mentioned the `do_nothing` implementation of `SORT`. To use such rewrite rule we are required to keep track of how streams are sorted. Thus, we associate a property to output ports, called `SortKey`. When a stream is sorted, `SortKey` is set to the sorting attribute. If unsorted, `SortKey` has an undefined value. The `SORT` box sets this property to its sort key, to specify its output is sorted. Other boxes may change the order of the stream without sorting it, in which case the `SortKey` property is set to undefined. Alternatively, a box may preserve stream order, in which case the sort key property of the input stream is copied to the sort key property of the output stream. The `do_nothing` algorithm reads the value of `SortKey` for its input stream, and compares it to the value of the desired order. If the sort keys are different, the `do_nothing` rewrite is invalid.

6 Case Study: Gamma Hash Join

This section serves a dual purpose: (1) to present a case study using DxT to re-engineer a legacy PnF application and (2) to illustrate how an RDM can be populated with rewrites. We have observed that there can be many ways in which a complex PnF graph can be derived; each derivation uses a slightly different or larger set of rewrites than other derivations. By exploring multiple derivations, the RDM is enriched and a better understanding of a design is achieved. *Each of the rewrites that we present in this section have been proven correct [23].*

Gamma was (and perhaps still is) the most sophisticated relational database machine built in academia [24]. It was created in the late 1980s and early 1990s without the aid of modern software architectural models. We focus on Gamma's join parallelization, which is typical of modern relational database machines, and use `ReFlO` screenshots to incrementally illustrate Gamma's derivations.[10] [11]

6.1 A Modicum of Domain Knowledge

Of course, to appreciate the rewrites that Gamma uses, one needs a modicum of domain knowledge about relational query processing. We assume this, providing references that elaborate such knowledge.

Look at Figure 18: it shows interface `HJOIN` (read "hash join") with three different implementations: a primitive, a map-reduce algorithm, and a bloom-filter algorithm.

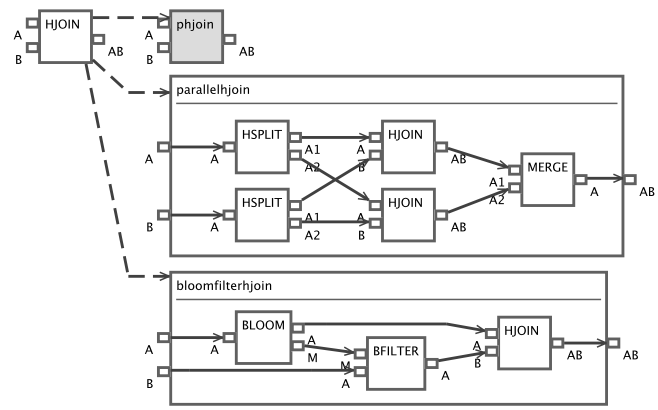


Fig. 18. `HJOIN` rewrite rules.

The primitive hash join implementation is simple: read all tuples of stream `A` into a main-memory hash table, where the join key of `A` tuples are hashed. Then read stream `B`, one tuple at a time. By hashing a `B` tuple's join key, one can quickly identify all `A` tuples that join with the `B` tuple. This algorithm has linear complexity in that each `A` and `B` tuple is read once.

The parallelization of `HJOIN` is textbook [25]: both input streams `A`, `B` are hash-split on their join keys using the same hash function. Each stream $\mathtt{A_i}$ is joined with stream $\mathtt{B_i}$ ($\mathtt{i} \in \{1, 2\}$), as we know that $\mathtt{A_i} \bowtie \mathtt{B_j} = \emptyset$ for all $\mathtt{i} \neq \mathtt{j}$ (equal keys must hash to the same value). By merging the joins of $\mathtt{A_i} \bowtie \mathtt{B_i}$ ($\mathtt{i} \in \{1, 2\}$), $\mathtt{A} \bowtie \mathtt{B}$ is produced as output.

A very different `HJOIN` algorithm makes use of Bloom filters to reduce the number of tuples to join [26]. It uses two new boxes: `BLOOM` (to create the filter) and `BFILTER` (to apply the filter). We call this algorithm `bloomfilterhjoin`. Here's how it works: the `BLOOM` box takes a stream of tuples `A` as input and outputs exactly the same stream `A` along with a bitmap `M`. The `BLOOM` box first clears `M`. Each tuple of `A` is read, its join key is hashed, the corresponding bit (indicated by the hash) is set in `M`, and the `A` tuple is output. After all `A` tuples are read, `M` is output. `M` is the *Bloom filter*.

The `BFILTER` box takes Bloom filter `M` and a stream of tuples `A` as input, and eliminates tuples of `A` that cannot join with tuples used to build the Bloom filter. The algorithm begins by reading `M`. Stream `A` is read one tuple at a time; the `A` tuple's join key is hashed, and the corresponding bit in `M` is checked.

9. `ReFlO` ignores the specification of explicit postconditions for algorithms or patterns. This prevents postconditions from being specified that are stronger than those computed from its internal boxes.

10. The RDM used in this derivation is available at `http://cs.utexas.edu/users/schwartz/DxT/case-studies/gamma/models/databases.html`.

11. For simplicity, the derivation presented does not use replication. A derivation using replication is available at `http://cs.utexas.edu/users/schwartz/DxT/case-studies/gamma/architectures/cascadejoin-rep/`.

If the bit is unset, the A tuple is discarded as there is no tuple to which it can be joined. Otherwise the A tuple is output. A new A stream is the result.

Finally, output stream A of BLOOM and output stream A of BFILTER are joined. Given the behaviors of the BLOOM, BFILTER, and HJOIN boxes, it is easy to prove that bloomfilterhjoin does indeed produce A ⋈ B [23].

We are now ready to present two derivations of Gamma: the first and simplest refines HJOIN by map-reduce first and then by bloom-filter. The second swaps the order by refining HJOIN with bloom-filter first, and then map-reduce. This seemingly minor difference yields a surprising wealth of rewrites.

6.2 Gamma – A Short Derivation

A *hash join* is an implementation of a relational equi-join; it takes two streams (A, B) of tuples as input and produces their equi-join A ⋈ B as output (AB). Figure 19 is Gamma's PIM. It just uses the HJOIN interface to specify the desired behavior.

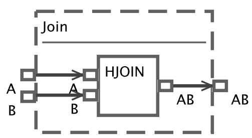


Fig. 19. The PIM: Join.

Our derivation starts by refining the HJOIN interface with its parallel map-reduce algorithm parallelhjoin (Figure 20).

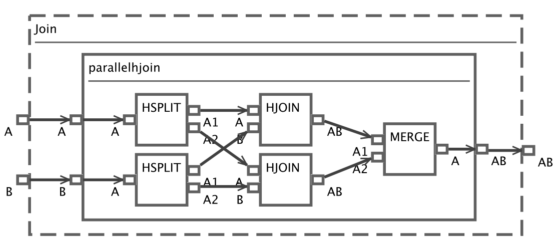


Fig. 20. Parallel Join graph.

Next, bloomfilterhjoin algorithm refines each of the HJOIN interfaces of Figure 20 to produce Figure 21. Flattening

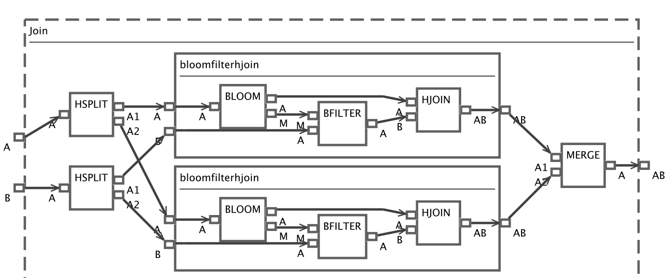


Fig. 21. Parallel Join graph, using Bloom filters.

Figure 21, and refining each interface with its lone primitive yields Gamma's PSM (Figure 22).

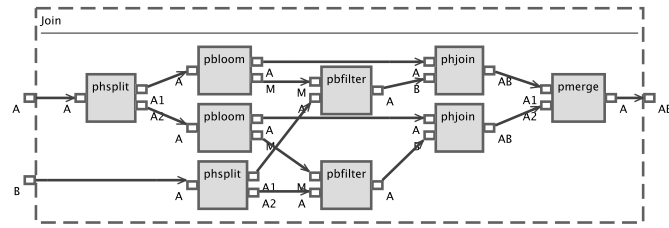


Fig. 22. Optimized parallel implementation of Gamma.

6.3 Gamma – An Alternative Derivation

A second, more involved derivation of Figure 22 exposes new rewrites. Historically, we discovered this derivation first, and only years later recognized the shorter derivation.

We start by applying the bloomfilterhjoin refinement. Doing so, we obtain the graph depicted in Figure 23.

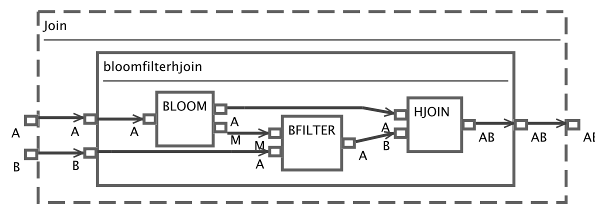


Fig. 23. Join graph using Bloom filters.

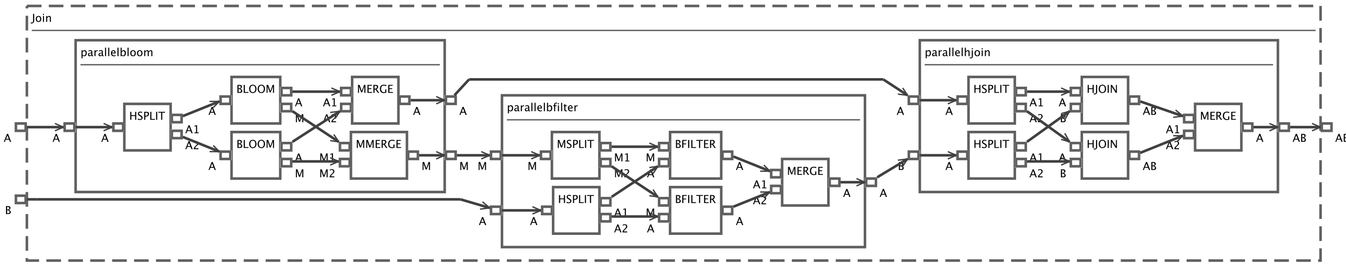
The next step is to parallelize the BLOOM, BFILTER, and HJOIN boxes by refining each with their map-reduce versions (Figure 24(a)).

A BLOOM box is parallelized by hash-splitting its input stream A into substreams A_1, A_2, creating a Bloom filter M_1, M_2 for each substream, coalescing A_1, A_2 back into A, and merging bit maps M_1, M_2 into a single map M. A BFILTER box is parallelized by hash-splitting its input stream A into substreams A_1, A_2. Map M is decomposed into submaps M_1, M_2 and substream A_i is filtered by M_i. The reduced substreams A_1, A_2 output by BFILTER boxes are coalesced into stream A. The same hash function must be used by all algorithms.

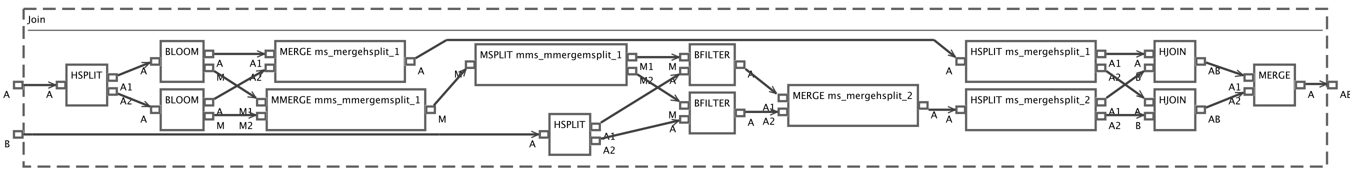
This alternative derivation already requires two additional refinements to map interfaces BLOOM and BFILTER to their map-reduce algorithms. Still, this graph is not yet the optimized Gamma PSM.

In this derivation, refinement is insufficient to produce Gamma's PSM. The graph of Figure 24(a) has three *serialization bottlenecks* which degrade performance. Consider the MERGE of substreams A_1, A_2 (produced by BLOOM) into A, followed by a HSPLIT to reconstruct A_1, A_2. *There is no need to materialize* A: the (MERGE, HSPLIT) pair can also be implemented by the identity map: $A_i \rightarrow A_i$. The same applies for the (MERGE, HSPLIT) pair for collapsing and reconstructing substreams produced by BFILTER. The removal of (MERGE, HSPLIT) pairs eliminates two serialization bottlenecks. This optimization is encoded in the graph presented in Figure 25(a).

The third bottleneck combines maps M_1, M_2 into M and then decomposes M back into M_1, M_2. The (MMERGE, MSPLIT) pair can also be implemented by an identity map: $M_i \rightarrow M_i$. This optimization removes the (MMERGE, MSPLIT) boxes and reroutes



(a)



(b)

Fig. 24. Parallelization of `Join` graph, and its bottlenecks.

the streams appropriately.[12] This optimization is encoded in the model presented in Figure 25(b).

Using the *Find Optimization* tool available in `ReFlO`, the bottlenecks are identified, as depicted in Figure 24(b). After applying the identity optimizations, we can refine the interfaces used with primitive implementations, to obtain the optimized Gamma graph, already presented in Figure 22.

6.4 An Interpretation Example – Costs Estimates

During the process of deriving a PSM, it is useful for the developers to be able to estimate values of quality attributes they are trying to improve. This is a typical application for interpretations.

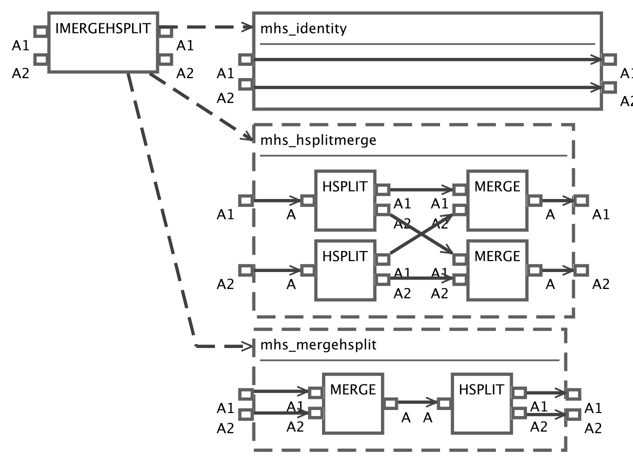
For databases, estimates for execution time are computed by adding the execution cost of each interface or primitive present in a graph. The cost of an interface[13] or primitive is computed based on the size of the data being processed. The $\mathcal{DS}$ interpretation takes estimates of input data sizes and computes estimates of output data sizes.

Size estimates are used to build a cost expression representing the cost of executing interfaces and primitives. We build a string containing a cost symbolic expression, as during design time we do not have concrete values for properties needed to compute costs. Thus, we associate a variable (string) to those properties, and we use those strings to build the symbolic expression representing the costs.

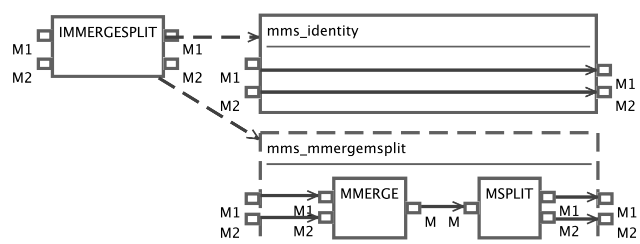
Figure 26 shows the code used to generate a cost estimate for `phjoin` primitive. `phjoin` is executed by reading each tuple of stream `A` and storing it in a hash table (`cHJoinAItem` is a constant that represents the cost of processing a tuple of stream `A`), and then each tuple of stream `B` is read and joined with

12. There are many ways in which `MMERGE` and `MSPLIT` can be realized. The simplest is this: `M` is a $2 \times k$ bitmap. The join key of an `A` tuple is hashed twice: once to determine the row of `M`, the second to determine the column within the selected row. Thus, all tuples of substream $\mathtt{A_i}$ hash to row `i` of `M`. `MMERGE` combines $\mathtt{M_1}, \mathtt{M_2}$ into `M` by boolean disjunction. For each `i`, `MSPLIT` extracts row `i` from `M` and zeros out the rest of $\mathtt{M_i}$.

13. An interface cost is set to that of its most general primitive implementation.



(a)



(b)

Fig. 25. Gamma optimizations.

tuples of `A` (`cHJoinBItem` is a constant that represents the cost of processing a tuple of stream `B`). Thus, the cost of `phjoin` is given by $\mathtt{size_a} * \mathtt{cHJoinAItem} + \mathtt{size_b} * \mathtt{cHJoinBItem}$. As `HJOIN` can always be implemented by `phjoin`, we can use the same cost expression for `HJOIN`. The $\mathcal{COSTS}$ interpretation is backward, as the costs of an algorithm are computed from the costs of its internal boxes (*ie* we need to compute costs of internal boxes first). So the costs are progressively sent to their parent boxes, until they reach the outermost box, where the costs of all boxes are aggregated, providing a cost estimate for the entire graph. Figure 27 shows the code used by interpretations of algorithm boxes, that simply add their costs to the aggregated costs stored on their parent boxes.

```
public class phjoin extends AbstractInterpretation {
  public void compute() {
    String sizeA=(String)getInputProperty("A","Size");
    String sizeB=(String)getInputProperty("B","Size");
    String cost="("+sizeA+") * cHJoinAItem + ("
        +sizeB+") * cHJoinBItem";
    setBoxProperty("Cost",cost);
    String parentCost=(String)getParentProperty("Cost");
    if(parentCost==null) parentCost=cost;
    else parentCost="("+parentCost+") + ("+cost+")";
    setParentProperty("Cost", parentCost);
  }
}
```

Fig. 26. Interpretation that estimates `phjoin` cost.

```
public class Algorithm extends AbstractInterpretation {
  public void compute() {
    String cost=(String) getBoxProperty("Cost");
    String parentCost=(String)getParentProperty("Cost");
    if(parentCost==null) parentCost=cost;
    else parentCost="("+parentCost+") + ("+cost+")";
    setParentProperty("Cost", parentCost);
  }
}
```

Fig. 27. Interpretation that processes costs for algorithm boxes.

7 Perspective

To round out our presentation, we sketch a general process on how to use `ReFlO` effectively and provide some insights on `ReFlO`'s limitations.

7.1 A Process on How to Use `ReFlO`

`ReFlO` can be used for different purposes, namely to reverse engineer existing PnF applications (*ie* to deduce a sequence of transformations that were used in a legacy application to map its PIM to its PSM) or to build new optimized programs, starting from a PIM. In either case, the process starts with a domain analysis [27], where an expert catalogs the fundamental operations of a domain with their implementations. The domain expert also knows that certain compositions of operations are inefficient, thus he needs to identify optimizations as well. It is also his job to provide evidence (*eg* a proof) that each transformation is correct and to specify the pre- and postconditions of each box.

This "minimal" model may be enhanced further. To explore different implementations of a program (*eg* efficiency or availability), additional interpretations are needed to estimate a program's quality attributes.

This knowledge can then be used by developers (or by the domain expert itself) to derive efficient programs. Typically, a developer starts with a PIM of a target application. `ReFlO` can be used incrementally to apply transformations and derive various PSMs, until a PSM is found that meets desired constraints on quality attributes. The developer may also export a domain model to an external tool to automatically search the space of a given PIM for a desirable PSM [21].

Domain analysis and derivations are often conducted in parallel. The domain model is usually built while reverse engineering existing programs, *ie* domain experts may be using `ReFlO` to derive programs and to build the domain model at the same time.

Finally, we note that `ReFlO` was developed specifically with pipe-and-filter software architectures in mind. We believe that `ReFlO` should be useable in other practical applications, such as dataflow and workflow applications, as well as functional-based application designs.

7.2 Limitations of `ReFlO`

We have used `ReFlO` to derive the designs of other applications—*crash fault tolerant (CFT)* servers [18] and dense linear algebra algorithms [22].

We chose a PnF notation to model programs and transformations that was influenced by the case-studies we explored. Although in certain domains a program's structure easily fits this architecture style (*eg* streaming applications [28], dataflow applications [29]), we are aware that some domains may require more effort to mine than others, and existing code may need to be adapted in order to provide code implementations for domain components. `ReFlO` seems best suited for mature and well-understood domains, although our use of `ReFlO` to explore designs of CFT servers is an example of a domain that hardly qualifies as mature. Further, `ReFlO` is not limited to domains with stateless computations either. The CFT servers that we studied were stateful [18].

The graphical notation (syntax) provided by `ReFlO` is not sufficient to encode domain knowledge. Pre- and postconditions are specified in Java; quality attribute definitions and computations are also specified in Java. Further, we found that many transformations are simple variations of each other; using templates substantially reduces the effort to encode rule variants. This combination of ideas and representations were sufficient to derive optimized programs in the different domains that we have studied.

It is possible that DSLs may simplify the task of writing different (and standardized) interpretations, rather than writing Java code. We leave this exploration to future work.

`ReFlO` promotes correct by construction derivations. Providing proofs of correctness for rewrite rules takes effort. Nevertheless, (i) proving individual transformations correct is usually simpler than proving the entire system correct and (ii) proofs for transformations are reusable whereas the proof for an entire system is usually not. Moreover, experts are far better able to provide proofs than developers who simply use the components and rewrite rules that experts have defined. *Although having proof of correctness is important,* `ReFlO` *does not require such proofs.*

Finally, we are hardly the first to notice that implementations of an interface can be specialized for particular inputs and particular conditions [16]. This forced Perry, and now us, to use PSP. It is worth observing that violations of the LSP are documented in the widely used JDK (*eg* `TreeMap` implementation of `Map` [30]).

8 Related Work

`ReFlO` is a tool to specify model transformations. Common tools/languages for model transformation, such as ATL [31] or Epsilon [32], specify transformations using executable code.

Our approach specifies transformations by providing examples, which has two advantages.

First, it makes it easier for domain experts (the ones with the knowledge about the valid domain transformations) to specify transformations [33], [34], [35], [36], [37]. Other approaches have been proposed to address this challenge. Baar and Whittle [34] explain how a metamodel (*eg* for PnF graphs) can be extended to also support the specification of transformations over models. In this way, a concrete syntax, similar to the syntax used to define models, is used to define model transformations, making those transformations easier to read and understand by humans. In ReFlO transformations are also specified using a concrete syntax.

Model transformation by example (MTBE) [33], [35] proposes to (semi-)automatically derive transformation rules based on set of key examples of mappings between source and target models. The approach was improved with the use of Inductive Logic Programming to derive the rules [38]. The rules may later be manually refined. Our rules provide examples in minimal context, and unlike in MTBE, we do not need to relate the objects of the source and target model (ports of interfaces are implicitly related with the ports of their implementations). Additionally, MTBE is more suited for exogenous transformations, whereas we use endogenous transformations [39], [40], [41].

More recently, a similar approach, *model transformation by demonstration* [36] was proposed, where users show how source models are edited in order to be mapped to the target models. A tool [42] captures the user actions and derives the transformations conditions and the operations needed to perform the transformations. When using ReFlO it is enough to provide the original element and its possible replacements.

Graph grammars [43] also provide a declarative way to define model/graph transformations using examples. In particular, our rules are specified in a similar way to productions in the *double-pushout approach* for hypergraphs [44]. AGG [45] is probably the most similar tool to ReFlO. It deals with graph rewrite rules, whereas our transformations are better captured by hypergraph rewrite rules, due to the role of ports in the transformations (that specify the gluing points in the transformation). Moreover, it is not clear whether these other approaches would be able to capture pre- and postconditions, which are essential for correct PnF graph derivation.

Another advantage is that ReFlO rewrites make domain knowledge more accessible to non-experts, as ReFlO encodes domain knowledge in a graphical and abstract way, relating alternative ways of implementing a particular behavior. Capturing algebraic identities is on the base of algebraic specifications and term rewriting systems. Relational query optimization [46], [47] is one of the most successful examples of application of these ideas, where, as in ReFlO, the goal is to optimize programs. Program verification tools, such as CafeOBJ [48] or Maude [49], are another common application. ReFlO was developed to support DxT approach, where transformations are specified as graph rewrites, instead of term rewriting.

More generally, ReFlO provides a framework for program transformation, that allows developers to interactively transform high-level program specifications into optimized implementations. SPIRAL [50] and AMPHION [51], are examples of projects with a similar goal, *ie* to synthesize efficient implementations for high-level specifications. Besides the differences in the way as they model the domain knowledge, and the strategies used to transform programs, the focus of these tools was on the automation of the synthesis process, whereas ReFlO is a tool for interactive development. Tools such as SPIRAL or AMPHION are useful when we have a complete model of a domain, whereas ReFlO is a tool to be used both by domain experts in the process of building those domain models, and later by other developers to optimize their programs. ReFlO is able to export its models to code that can be used with DxTer [9], [21] a tool that, like SPIRAL and AMPHION, automates the search for the optimized implementation.

Several tools for PnF modeling have been proposed, such as LabVIEW [3], Simulink [5], Weaves [52], Fractal [6], or StreamIt [53]. However, they focus on component specification and construction of systems composing those components. We realized that transformations (in particular optimizations) play an essential role when building efficient architectures using components. LabVIEW does support optimizations, but only when mapping a LabVIEW model to an executable. Users can *not* define refinements and optimizations, but LabVIEW compiler technicians can. More than a tool for the specification of PnF graphs, ReFlO provides the ability for users to capture domain specific graph transformations and to apply them to PnF designs.

The interpretation framework provided by ReFlO offers a way to perform model simulation/animation, which allows developers to predict properties of the system being modeled without having to actually build it. LabVIEW and Simulink are typical examples of tools to simulate PnF architectures. Ptolemy II [54] provides modeling and animation support for heterogeneous models.

Other tools exist for different types of models, such as UML [55], [56], or Colored Petri Nets [57]. Our work has some similarities with *Model-Driven Performance Engineering (MDPE)* [58]. However, we focus on endogenous transformations, and how those transformations improve architecture's quality attributes, not exogenous transformations, as it is common in MDPE. Our solution for cost estimation can be compared with the *coupled model transformations* proposed by Becker [59]. However, the cost estimates (as well as other interpretations) are transformed in parallel with the PnF graph, not during M2T transformations. Other solutions have proposed for component based systems [60]. KLAPER [61] provides a language to automate the creation of performance models from component models. Kounev [62] shows how *Queueing Petri Nets* can be used to model systems, allowing prediction of its performance characteristics. The *Palladio Component Model* [63] provides a powerful metamodel to support performance prediction, adapted to the different developer roles. We do not provide a specific framework for cost/performance estimates, as the expressiveness of ReFlO's interpretations framework allow us to support this capability.

ReFlO allows *properties* to be assigned to boxes. Properties are similar to attributes in an *attributed graph* [64]. Those properties are then used to specify pre- and postconditions. Allowing the implementations to have stronger preconditions, we may say that the rewrite rules may have *applicability predicates* [64] or *attribute conditions* [45], that specify a predicate over the attributes of a graph when a match/morphism is not enough to specify whether a transformation can be applied. Pre- and postconditions were used in other component systems, such as Inscape [16], with the goal of validating component compositions. In our case, the main purpose of pre- and postconditions is to decide when transformations can be applied. Nevertheless, they may also be used to validate component compositions.

Abstract interpretations [65], [66] define properties about a program's state, and specify how instructions affect those properties. The properties are correct, but often imprecise. Still, they provide useful information to allow compilers to perform certain transformations. In `ReFlO`, postconditions play a similar role. They compute properties about operation outputs based on properties of their inputs, and the properties may be used to decide whether a transformation can be applied or not. As for abstract interpretations, the properties computed by postconditions have to describe output values correctly. In contrast, properties used to compute costs, for example, are often just estimates, and therefore may not be correct, but in this case approximations are usually enough. The Broadway compiler [67] used the same idea of propagating properties about values, to allow the compiler to transform the program. The Broadway compiler separated the compiler infrastructure from domain expertise, and like in `ReFlO`, the goal was to allow users to specify domain specific optimizations. Specifying pre- and postconditions as properties that are propagated is also not new. This was the approach used in the Inscape environment [68], [69]. Interpretations provide alternative views of a PnF graph, that are synchronized as it is incrementally changed [70].

9 Conclusions

`ReFlO` was motivated by a lack of technology that would help us understand legacy pipe-and-filter (PnF) applications. Unless PnF graphs are very simple, they are spaghetti diagrams—difficult to understand, impossible to know if they are correct, and without tool support, difficult to analyze. Existing PnF tools, by in large, apply basic checks and convert a PnF graph into an executable, but not much else.

MDE places such tools in context of a much larger paradigm—the ability, indeed desire, to derive PnF applications using domain-specific rewrites that are implicitly used by experts, capturing and systematizing domain knowledge that would otherwise be lost or easily forgotten. Given a legacy PnF application, `ReFlO` makes it possible to derive its design with rewrite rules that are used by experts, and as we showed in this paper, rules that can be proven correct. To the best of our knowledge, this is the first derivation of Gamma that has been proven correct.

In this paper, we presented the core ideas behind `ReFlO`. We showed that the Perry Substitution Principle, rather than the Liskov Substitution Principle, is a foundation for `ReFlO` graph rewrites. We explained how a few basic operations (refine, flatten, abstract, optimize, find optimization, and expand) could be used by designers to derive PnF designs. Further, `ReFlO` is itself an extensible framework in which different interpretations of a PnF graph (which arise in checking pre- and postconditions, or cost evaluations) can be both added and composed as needed. Further, we illustrated a technique that we have used to populate `ReFlO` libraries with domain knowledge—*ie* different derivations of a design utilize different fundamental rewrites of a domain.

We believe `ReFlO` is a valuable step toward interactive design tools that aid domain-specific program development and knowledge collection.

Availability: `ReFlO` is available online at `http://www.cs.utexas.edu/users/schwartz/DxT/reflo/`. All PnF figures in this paper are screenshots from `ReFlO`. `ReFlO` is a Eclipse [71] plugin. The modeling languages were specified using Ecore [72], and the model editors were implemented using the GEF [73] and GMF [74]. The model transformations and model validation were implemented using the Epsilon [32] family of languages.

Acknowledgements: We gratefully acknowledge support for this work by NSF grants CCF 0724979 and OCI-1148125. Rui Gonçalves and João Sobral are funded by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-010152. Rui Gonçalves is additionally funded by FCT grant SFRH/BD/47800/2008.

References

[1] D. Garlan and M. Shaw, "An introduction to software architecture," Carnegie Mellon University, Tech. Rep., 1994.
[2] I. Crnkovic, *Building Reliable Component-Based Software Systems*, M. Larsson, Ed. Artech House, Inc., 2002.
[3] "The LabVIEW Environment," http://www.ni.com/labview/.
[4] M. M. Gorlick and R. R. Razouk, "Using weaves for software construction and analysis," in *ICSE '91: Proceedings of the 13th international conference on Software engineering*, 1991, pp. 23–34.
[5] "Simulink - Simulation and Model-Based Design," http://www.mathworks.com/products/simulink/.
[6] E. Bruneton, T. Coupaye, and J. Stefani, "The Fractal Component Model," http://fractal.ow2.org, 2004.
[7] D. S. Frankel, *Model Driven Architecture: Applying MDA to Enterprise Computing*. John Wiley & Sons, Inc., 2003.
[8] C. Green, D. Luckham, R. Balzer, T. Cheatham, and C. Rich, "Report on a knowledge-based software assistant," Kestrel Institute, Tech. Rep., 1983.
[9] B. Marker, J. Poulson, D. Batory, and R. van de Geijn, "Designing linear algebra algorithms by transformation: Mechanizing the expert developer," in *iWAPT '12: International Workshop on Automatic Performance Tuning*, 2012.
[10] D. Das, "Making database optimizers more extensible," Ph.D. dissertation, The University of Texas at Austin, 1995.
[11] N. Wirth, "Program development by stepwise refinement," *Communications of the ACM*, vol. 14, no. 4, pp. 221–227, 1971.
[12] B. H. Liskov and J. M. Wing, "A behavioral notion of subtyping," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 6, pp. 1811–1841, 1994.
[13] N. Medvidovic, D. S. Rosenblum, and R. N. Taylor, "A language and environment for architecture-based software development and evolution," in *ICSE '99: Proceedings of the 21st international conference on Software engineering*, 1999, pp. 44–53.
[14] Wikipedia, "Component-based software engineering," http://en.wikipedia.org/wiki/Component-based_software_engineering, 2013.
[15] D. Batory and S. W. O'Malley, "The design and implementation of hierarchical software systems with reusable components," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 1, no. 4, pp. 355–398, 1992.
[16] D. E. Perry, "Version control in the inscape environment," in *ICSE '87: Proceedings of the 9th international conference on Software Engineering*, 1987, pp. 142–149.
[17] R. F. Paige, D. S. Kolovos, and F. A. C. Polack, "Refinement via consistency checking in MDA," *Electronic Notes in Theoretical Computer Science*, vol. 137, no. 2, pp. 151–161, 2005.
[18] T. L. Riché, R. C. Gonçalves, B. Marker, and D. Batory, "Pushouts in software architecture design," in *GPCE '12: Proceedings of the 11th ACM international conference on Generative programming and component engineering*, 2012, pp. 84–92.
[19] R. C. Gonçalves, "Parallel Programming by Transformation," Ph.D. dissertation, Departamento de Informática, Universidade do Minho, (To appear).
[20] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin, "On the use of higher-order model transformations," in *ECMDA-FA '09: Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, 2009, pp. 18–33.

[21] B. Marker, D. Batory, and C. Shepherd, "DxTer: A program synthesizer for dense linear algebra," The University of Texas at Austin, Department of Computer Science, Tech. Rep., 2012.

[22] B. Marker, A. Terrel, J. Poulson, D. Batory, and R. van de Geijn, "Mechanizing the expert dense linear algebra developer," in *PPoPP '12: Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, 2012, pp. 289–290.

[23] D. Batory and B. Marker, "Correctness Proofs of the Gamma Database Machine Architecture," University of Texas Department of Computer Science TR-11-17, 2011.

[24] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen, "The gamma database machine project," *IEEE Transactions on Knowledge and Data Engineering*, vol. 2, no. 1, pp. 44–62, 1990.

[25] F. Baru, "DB2 Parallel Edition," *IBM Sys. Journal*, vol. 34, no. 2, 1995.

[26] B. H. Bloom, "Space/Time Trade-Offs in Hash Coding with Allowable Errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[27] J. M. Neighbors, "Software construction using components," Ph.D. dissertation, Department of Information and Computer Science, University of California, Irvine, 1980.

[28] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "StreamIt: A language for streaming applications," in *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, 2002, pp. 179–196.

[29] "Dataflow application areas," http://www.ni.com/labview/applications/, 2013.

[30] "TreeMap (Java Platform SE 7)," http://docs.oracle.com/javase/7/docs/api/java/util/TreeMap.html, 2013.

[31] "ATL - a model transformation technology," http://www.eclipse.org/atl/.

[32] "Epsilon," http://www.eclipse.org/gmt/epsilon/.

[33] D. Varró, "Model transformation by example," in *MODELS '06: Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, 2006, pp. 410–424.

[34] T. Baar and J. Whittle, "On the usage of concrete syntax in model transformation rules," in *PSI '06: Proceedings of the 6th international Andrei Ershov memorial conference on Perspectives of systems informatics*, 2006, pp. 84–97.

[35] M. Wimmer, M. Strommer, H. Kargl, and G. Kramler, "Towards model transformation generation by-example," in *HICSS '07: Proceedings of the 40th Annual Hawaii International Conference on System Sciences*, 2007.

[36] Y. Sun, J. White, and J. Gray, "Model transformation by demonstration," in *MODELS '09: Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, 2009, pp. 712–726.

[37] H. Saada, X. Dolquesa, M. Huchard, C. Nebut, and H. Sahraoui, "Generation of operational transformation rules from examples of model transformations," in *MODELS '12: Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems*, 2012, pp. 546–561.

[38] D. Varró and Z. Balogh, "Automating model transformation by example using inductive logic programming," in *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, 2007, pp. 978–984.

[39] A. Egyed, N. R. Mehta, and N. Medvidovic, "Software connectors and refinement in family architectures," in *IW-SAPF-3: Proceedings of the International Workshop on Software Architectures for Product Families*, 2000, pp. 96–106.

[40] R. Heckel and S. Thöne, "Behavior-preserving refinement relations between dynamic software architectures," in *WADT' 04: Proceedings of the 17th International Workshop on Algebraic Development Techniques*, 2004, pp. 1–27.

[41] T. Mens and P. Van Gorp, "A taxonomy of model transformation," *Electronic Notes in Theoretical Computer Science*, vol. 152, pp. 125–142, 2006.

[42] Y. Sun, J. Gray, and J. White, "MT-scribe: an end-user approach to automate software model evolution," in *ICSE '11: Proceedings of the 33rd International Conference on Software Engineering*, 20011, pp. 980–982.

[43] G. Rozenberg, *Handbook of Graph Grammars and Computing by Graph Transformation, Vol I: Foundations*. World Scientific, 1997.

[44] A. Habel, *Hyperedge Replacement: Grammars and Languages*. Springer-Verlag New York, Inc., 1992.

[45] G. Taentzer, "AGG: A graph transformation environment for modeling and validation of software," in *Applications of Graph Transformations with Industrial Relevance*. Springer Berlin / Heidelberg, 2004, vol. 3062, pp. 446–453.

[46] G. M. Lohman, "Grammar-like functional rules for representing query optimization alternatives," in *SIGMOD '88: Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, 1988, pp. 18–27.

[47] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access path selection in a relational database management system," in *SIGMOD '79: Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, 1979, pp. 23–34.

[48] R. Diaconescu, K. Futatsugi, and S. Iida, "Component-based algebraic specification and verification in CafeOBJ," in *FM '99: Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems-Volume II*, 1999, pp. 1644–1663.

[49] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada, "Maude: specification and programming in rewriting logic," *Theoretical Computer Science*, vol. 285, no. 2, pp. 187–243, 2002.

[50] M. Püschel, J. M. F. Moura, B. Singer, J. Xiong, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson, "Spiral: A generator for platform-adapted libraries of signal processing algorithms," *International Journal of High Performance Computing Applications*, vol. 18, no. 1, pp. 21–45, 2004.

[51] M. R. Lowry, A. Philpot, T. Pressburger, and I. Underwood, "Amphion: Automatic programming for scientific subroutine libraries," in *ISMIS '94: Proceedings of the 8th International Symposium on Methodologies for Intelligent Systems*, 1994, pp. 326–335.

[52] M. M. Gorlick and R. R. Razouk, "Using Weaves for Software Construction and Analysis," in *ICSE*, 1991.

[53] W. Thies, "Language and compiler support for stream programs," Ph.D. dissertation, MIT, 2008.

[54] J. Eker, J. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Sachs, Y. Xiong, and S. Neuendorffer, "Taming heterogeneity - the ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, 2003.

[55] B. Combemale, X. Crégut, J.-P. Giacometti, P. Michel, and M. Pantel, "Introducing simulation and model animation in the MDE topcased toolkit," in *ERTS '08: 4th European Congress Embedded Real Time Software*, 2008.

[56] D. Dotan and A. Kirshin, "Debugging and testing behavioral UML models," in *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, 2007, pp. 838–839.

[57] A. V. Ratzer, L. Wells, H. M. Lassen, M. Laursen, J. F. Qvortrup, M. S. Stissing, M. Westergaard, S. Christensen, and K. Jensen, "CPN tools for editing, simulating, and analysing coloured petri nets," in *ICATPN '03: Proceedings of the 24th international conference on Applications and theory of Petri nets*, 2003, pp. 450–462.

[58] M. Fritzsche and J. Johannes, "Putting performance engineering into model-driven engineering: Model-driven performance engineering," in *Models in Software Engineering*. Springer-Verlag, 2008, pp. 164–175.

[59] S. Becker, "Coupled model transformations," in *WOSP '08: Proceedings of the 7th international workshop on Software and performance*, 2008, pp. 103–114.

[60] H. Koziolek, "Performance evaluation of component-based software systems: A survey," *Performance Evaluation*, vol. 67, no. 8, pp. 634–658, 2010.

[61] V. Grassi, R. Mirandola, and A. Sabetta, "From design to analysis models: a kernel language for performance and reliability analysis of component-based systems," in *WOSP '05: Proceedings of the 5th international workshop on Software and performance*, 2005, pp. 25–36.

[62] S. Kounev, "Performance modeling and evaluation of distributed component-based systems using queueing petri nets," *IEEE Transactions on Software Engineering*, vol. 32, no. 7, pp. 486–502, 2006.

[63] S. Becker, H. Koziolek, and R. Reussner, "The palladio component model for model-driven performance prediction," *Journal of Systems and Software*, vol. 82, no. 1, pp. 3–22, 2009.

[64] H. Bunke, "Attributed programmed graph grammars and their application to schematic diagram interpretation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 4, no. 6, pp. 574–582, 1982.

[65] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or

approximation of fixpoints," in *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1977, pp. 238–252.
[66] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Springer-Verlag, 1999.
[67] S. Z. Guyer and C. Lin, "Broadway: A compiler for exploiting the domain-specific semantics of software libraries," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 342–357, 2005.
[68] D. E. Perry, "The inscape environment," in *ICSE '89: Proceedings of the 11th international conference on Software engineering*. ACM, 1989, pp. 2–11.
[69] ——, "The logic of propagation in the inscape environment," *ACM SIGSOFT Software Engineering Notes*, vol. 14, no. 8, pp. 114–121, 1989.
[70] I. Ráth, G. Varró, and D. Varró, "Change-driven model transformations," in *MODELS '09: Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, 2009, pp. 342–356.
[71] "Eclipse," http://eclipse.org.
[72] "Eclipse Modeling Framework," http://www.eclipse.org/modeling/emf/.
[73] "Graphical Editing Framework," http://www.eclipse.org/gef/.
[74] "Eclipse graphical modeling framework," http://www.eclipse.org/gmf/.

APPENDIX

Figure 22 is not the last word on Gamma's graph. Optimizations identical to those presented in Section 3.2.4 are used to optimize the processing of cascading joins, where the output of one join becomes the input of another (see Figure 28).

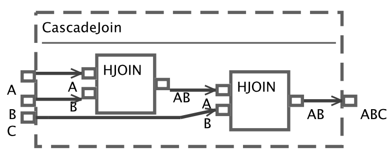
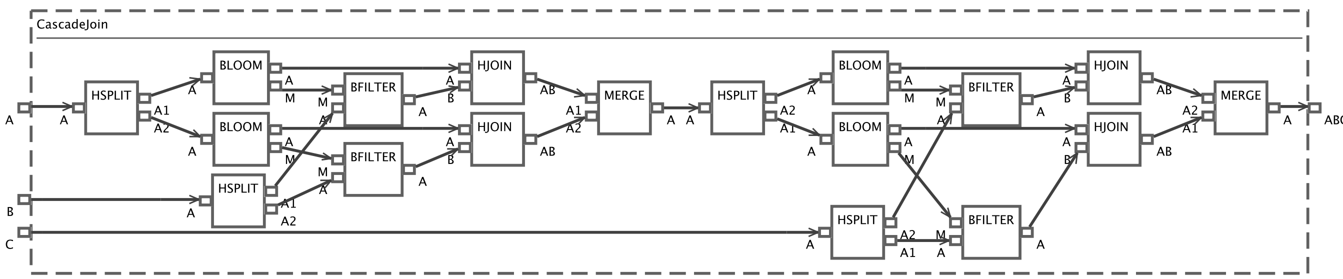


Fig. 28. `CascadeJoin` graph.

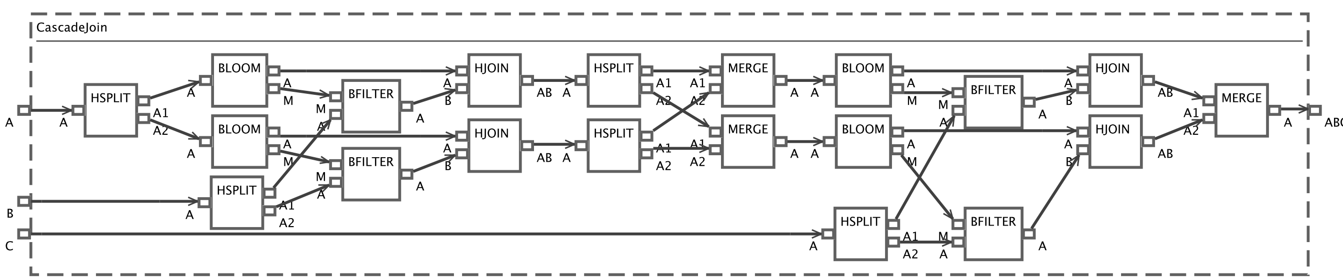
Applying the refinements `parallelhjoin` and `bloomfilterhjoin`, as described in Section 6.2, we get the graph depicted in Figure 29(a). This example further shows the importance of deriving the PnF graphs, instead of just using pre-built optimized implementations for the operations present in the initial PIM (in this case, `HJOIN` operations). The use of the optimized implementations for `HJOIN` would have resulted in an implementation equivalent to the one depicted in Figure 29(a). However, when we compose two (or more) instances of `HJOIN`, new opportunities for optimization arise. We have again a serialization bottleneck, formed by a composition of boxes `MERGE` (that merges the output streams of the first group of `HJOIN`s) and `HSPLIT` (that hash-splits the stream again).

Here again, refinement is insufficient to derive Gamma's graph; encapsulation boundaries must be broken to eliminate serialization bottlenecks. Unlike the bottlenecks in the previous section, cascading joins use different keys to hash the tuples, so the partitioning of the stream *before* its merge is different than the partitioning *after* the hash-split. Therefore, we cannot use algorithm `mhs_identity` to optimize this subgraph.[14] Instead, we use a rewrite that removes these bottlenecks by swapping (`MERGE`, `HSPLIT`) pairs (algorithm `mhs_hsplitmerge`). Each input stream is hash-split into two substreams, that are sent to the each `MERGE` box. The substreams with the same hash values are then merged.

14. We prevent algorithm `mhs_identity` from being chosen using preconditions.



(a)



(b)

Fig. 29. Rotation of `MERGE` and `HSPLIT`.