This is the author's version of the work. Published in SoSyM.
The final publication is available at Springer via http://dx.doi.org/10.1007/s10270-015-0495-8

From Software Extensions to Product Lines of Dataflow Programs

Rui C. Gonçalves · Don Batory · João L. Sobral · Taylor L. Riché

Abstract Dataflow programs are widely used. Each program is a directed graph where nodes are computations and edges indicate the flow of data. In prior work, we reverse-engineered legacy dataflow programs by deriving their optimized implementations from a simple specification graph using graph transformations called *refinements* and *optimizations*. In MDE-speak, our derivations were PIM-to-PSM mappings. In this paper, we show how *extensions* complement refinements, optimizations, and PIM-to-PSM derivations to make the process of reverse engineering complex legacy dataflow programs tractable. We explain how optional functionality in transformations can be encoded, thereby enabling us to encode product lines of transformations as well as product lines of dataflow programs. We describe the implementation of extensions in the `ReFlO` tool and present two non-trivial case studies as evidence of our work's generality.

Keywords MDE, PIM, PSM, Model Transformations, Software Extensions, Dataflow Programs, Software Product Lines

Rui C. Gonçalves
INESC TEC, 4710–057 Braga, Portugal
E-mail: `rgoncalves@di.uminho.pt`

Don Batory
The University of Texas at Austin, Austin, TX 78712, USA
E-mail: `batory@cs.utexas.edu`

João L. Sobral
Universidade do Minho, 4710–057 Braga, Portugal
E-mail: `jls@di.uminho.pt`

Taylor L. Riché
National Instruments, Austin, TX 78759, USA
E-mail: `taylor.riche@ni.com`

1 Introduction

Dataflow programs (DfP) abound in today's world. They are fault-tolerant servers [14,46], relational query execution plans [20,46], dense linear algebra kernels [39, 38], virtual instruments [53,49,26], stream processing applications [26,54], and large-scale cloud data processing applications [13,25]. A DfP is a directed graph: nodes called *boxes* are components or computations; edges indicate the flow of data. Edges that flow into a box are box inputs; edges leaving a box are box outputs. The graph of a DfP is referred to as its *architecture*.

In prior work [24] we explored an MDE approach to encode DfP design knowledge as graph transformations so that a simple easy-to-understand DfP could be transformed into a complex and optimized implementation, also expressed as a DfP. That is, we transformed an initial DfP (graph), always preserving its behavior, until we reached another DfP that had the desired implementation properties regarding, for example, efficiency or availability. This DfP could then be mapped to code using a domain-specific code generator. The transformations we used were *refinements* (replace a box with an implementing graph) or *optimizations* (replace a subgraph with a semantically equivalent graph).

Our derivations begin with a simple DfP (high-level specification or *platform independent model (PIM)*). In this paper, we show how a complex PIM can be constructed incrementally. That is, we start with an elementary PIM that defines only part of the desired behavior. New behavior is incrementally added to this PIM, until we arrive at a PIM with the desired behavior [47]. Adding behavior is *extension*; an increment in behavior (or functionality) that is added is called a *feature*.

In its most basic form, an extension maps a box `A` without a functionality `F` to a box `B` with the functional-

ity of A *and* F. *Refinements and optimizations preserve behavior; extensions (in contrast) extend behavior.*

Extensions are not new. They are basic to classical approaches to software development [51,1]. A simple specification A_0 is progressively extended to produce a desired specification, say Z_0 (Figure 1). The final specification, Z_0, is then used as the starting point for a derivation, using refinements, to produce the desired implementation Z_Ω, called a *platform specific model (PSM)* (Figure 1).

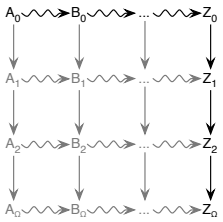

Fig. 1: Extension vs. derivation

Our initial interest in this topic arose when we tried to reverse engineer legacy D*f*Ps, to understand and encode the domain knowledge used by experts to build them. We used the model-driven approach described above, where expert knowledge was captured as model transformations in the process of mapping a PIM to a PSM. We discovered that when a PIM is complex, it is almost impossible to recognize, understand, and explain the transformations that map it to a PSM. In short, the classical approach of Figure 1 is not always practical. We devised an alternate way to derive such PSMs. Namely, we derive an implementation A_Ω from a simple specification A_0 and then *incrementally extend this derivation* to that which maps Z_0 to Z_Ω in Figure 1 [47]. Extensions are higher-order transformations [55] in our approach.

Reverse engineering legacy systems to extract and systematize the transformations (knowledge) used to build them is only the first step of a larger process. The knowledge gathered and encoded can later be used to mechanize/automate forward engineering [39,38], *i.e.*, to generate new programs. Extensions provided a mechanism to ease the process of extracting knowledge from legacy systems, and allowed us to improve the tools to assist experts in this task. Moreover, extensions can express optional features. When this happens, a *software product line (SPL)* of D*f*Ps arises [2]. In this paper, we show how program derivations are extended to product lines of derivations.

To summarize, our initial work [24] focused on multi-step derivations of implementations from specifications. This paper generalizes [24] and [47] to examine and reveal the rich relationships among extensions, refinements, optimizations, derivations, dataflow graphs, and SPLs. The contributions of this paper are to:

- define fundamental relationships among these concepts;
- illustrate a pragmatic way to extend dataflow graphs, graph transformations, and PIM-to-PSM derivations;
- distinguish transformations (refinements and optimizations) from higher-order transformations (extensions);
- explain how extensions can be encoded to support incremental development (or reverse engineering) of D*f*Ps and the specification of dataflow SPLs; and
- describe how extensions were implemented in the ReFlO framework [24].

We start with a review of background concepts that are central to our approach.

2 Background

Fundamental ideas in *Object-Oriented Programming (OOP)* are "implements" and "extends". Figure 2a is a UML declaration that class C implements interface I. Interface I specifies some abstract behavior for which class C provides an implementation. Figure 2b provides some additional declarations. Interface IX extends interface I, *i.e.*, it specifies some additional behavior when compared to I. Similarly, class CX extends class C, *i.e.*, it extends implementation C, so that CX also implements the additional behavior required by IX.

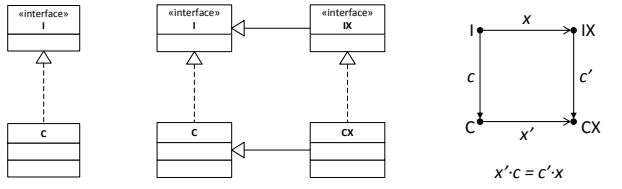

Fig. 2: Fundamental ideas in OOP and product lines.

Extensions define increments in behavior (either on a specification/interface or on an implementation/class). SPLs generalize the scale of extension. A *feature* embodies an increment in functionality (a.k.a. requirement). Instead of limiting extension to one class or one interface at a time, a feature extends an *entire* class diagram [7]. This is what we have in Figure 2b: diagram "C *implements* I" is extended to diagram "CX *implements* IX".

By reversing the arrows in Figure 2b, the mapping relationships of a *commuting diagram* in Figure 2c are exposed. The key property of a commuting diagram is that all paths between any two nodes yield the same result [43]. So in Figure 2c, we can develop class CX in two ways starting with I. One way is to extend I to IX and then IX is implemented by CX. Alternatively, we

could have implemented `I` by `C` and then extend `C` to `CX`.

In this paper, we apply these ideas to design SPLs of DfPs. DfPs have interfaces. An interface `I` can be implemented by a code component `C`, a relation we define as an *implementing rewrite rule* `I` $\rightarrow$ `C`, meaning `I` can be replaced by `C`. Or more generally, `I` could be implemented by a dataflow graph `G`, written `I` $\rightarrow$ `G`, where `G` implements the behavior of `I`.

We also need extensions: a dataflow interface `I` is extended to interface `IX`, written as an *extension mapping* `I` $\rightsquigarrow$ `IX`. Similarly, extension mappings can also be defined for a code component or a dataflow graph. We use the latter to express the changes a feature makes to a dataflow graph.

We now make a critical observation: *the rewrite rules that we expose and use in our work encapsulate basic steps or modules of domain-specific DfP construction.* These rewrites are identified with the help of domain experts: they know and implicitly use these "identities" in their designs. Our work provides a means for experts to (a) articulate them in a machine-processable form and (b) use them to design—or explain the design—of domain-specific DfPs.

Here is how we will proceed: we first explore how an abstract dataflow graph `AG` (PIM)—consisting only of interconnected interfaces—can be progressively elaborated using one or more implementation rewrites to an implementing graph `IG` (PSM). We denote such a mapping by `AG` $\rightarrow^*$ `IG` ($\rightarrow^*$ represents multiple behavior-preserving derivation steps). This is our earlier work [24].

We then show how `AG` maps to a more elaborate abstract dataflow graph `EAG` using extensions, denoted `AG` $\rightsquigarrow^*$ `EAG`, and then how the derivation of the implementing graph `IG` can be extended to the derivation of the graph that implements `EAG`, which is `EIG`. That is, we show (`AG` $\rightarrow^*$ `IG`) $\rightsquigarrow^*$ (`EAG` $\rightarrow^*$ `EIG`). How derivations are extended is a primary contribution of this paper.

To our readers: The meaning of "implements" and "extends" in OOP are well-understood. Countless successful programs have been built using these ideas without a hint of formal models behind them. Our work is the same vein: we rely on standard OOP concepts of "implements" and "extends" to express how we re-engineered complex legacy dataflow applications (*e.g.*, `EIG` above) by starting with a simple description `AG` of the application and developing a commuting diagram that allows us to reconstruct `EIG` using fundamental implementation and extension rewrites of the target domain.

The next section presents an illustrative example of our approach. Section 6 presents real-world case studies.

3 Motivating Examples and Methodology

3.1 Refinements and Optimizations

Figure 3 shows a DfP or PIM called `Server` that projects (eliminates) fields and sorts a stream of database tuples. The tuples are displayed by `WSERVER` and then are transmitted as the output of `Server`. Boxes `PROJECT`, `SORT` and `WSERVER` are *interfaces* as they do not imply any particular implementation.

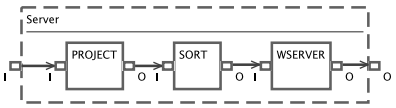

Fig. 3: Initial `Server` DfP.

These interfaces express operations that are well-known to domain experts. An expert also knows different ways of implementing these interfaces, which can be used to derive implementations of programs providing certain properties, such as efficiency or availability. `Server` can be parallelized (to improve efficiency) by replacing the `PROJECT` and `SORT` interfaces with their parallel implementations, which a domain expert specifies as rewrite rules `PROJECT` $\rightarrow$ `parallel_project` and `SORT` $\rightarrow$ `parallel_sort` [24].

Figure 4 shows the `SORT` $\rightarrow$ `parallel_sort` rewrite: an interface box (`SORT`) is linked—using an implementation connector (dashed arrow)—to a parallel implementation following a map-reduce strategy (`parallel_sort`) [37]. A *refinement* is the application of a rewrite rule that replaces an interface by an implementation. After using the aforementioned rewrite rules to refine `Server`, we obtain the DfP of Figure 5.

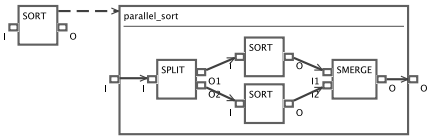

Fig. 4: `SORT` $\rightarrow$ `parallel_sort` rewrite rule.

The DfP of Figure 5, if directly mapped to code, would be inefficient. It has two identical `SPLIT` boxes. The substreams that are output by the `PROJECT` boxes are merged into a single stream, which is then split to reconstruct the substreams that were merged! This is clearly unnecessary work. We can apply the optimization of Figure 6 to eliminate this overhead: it replaces a

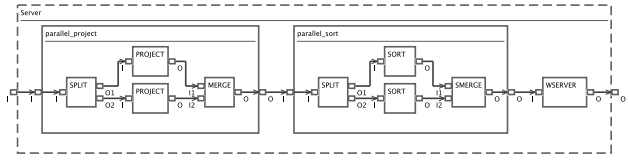

Fig. 5: `Server` DfP after map-reduce refinements.

sequential composition of boxes `MERGE` − `SPLIT` with direct connectors from inputs to outputs, producing the optimized DfP of Figure 7. Again, such inefficiencies are known to domain experts. They, in turn, write optimization rewrite rules to remove them.

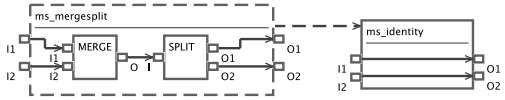

Fig. 6: The `MERGE` − `SPLIT` optimization.

An *optimization* is a transformation that replaces a subgraph `G′` with another graph `G` that preserves the semantics of `G′`, but implements `G′` in a different way. Optimization `G′` → `G` is really a pair of transformations: `G′` is abstracted to the interface `AI` that it implements, and then `AI` is replaced by an alternative implementation `G`.

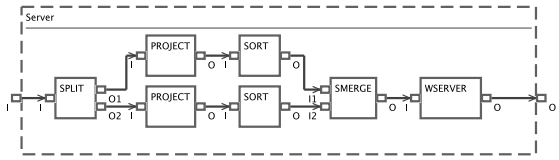

Fig. 7: Optimized `Server` DfP.

3.2 Extensions

Let `A` be a dataflow interface, component, or graph. We write `F.A` to denote the `F` extension of `A`.

Suppose we want to add new functionality to the `Server` PIM. We want `WSERVER` to change the sort key attribute at runtime. How would this change be made? Answer: by extending the `Server` with feature `K` (short for `Key`), `Server` ⇝ `K.Server`, resulting in the PIM of Figure 8.

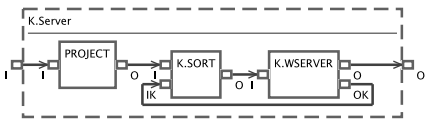

Fig. 8: The DfP `K.Server`.

Methodology. Extension of a DfP is accomplished by a two-step procedure. Think of `K` as a function `G` ⇝ `K.G` that maps a graph `G` to the `K`-extended graph `K.G`. In general, each element `e` ∈ `G`—where an element `e` is a box, port or connector—is either mapped to an element `K.e` ∈ `K.G` or removed from `K.G`. Element `K.e` is an extension of `e`: a connector may carry more data, a box has a new port or its ports may accept data conforming to an extended data type.[1] Occasionally `K` does nothing, *i.e.*, `K.e` = `e`. Whatever the outcome may be, an expert would know—it is not always evident to non-experts. For our `Server` example, the effects of extension `K` are not difficult to deduce.

The first step is to perform the `K` mapping. Figure 9 shows that the only elements changed by `K` are `SORT` and `WSERVER`. Box `K.SORT`, which `K`-extends `SORT`, has sprouted a new input (to specify the sort key parameter), and `K.WSERVER` has sprouted a new output (to specify a sort key parameter). The resulting DfP (Figure 9) is *provisional*—it is not yet complete.

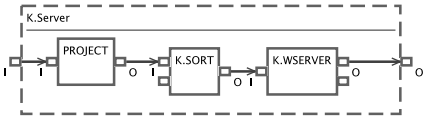

Fig. 9: Applying `K` to `Server`.

The second step completes the provisional DfP: the new input of `K.SORT` needs to be supplied. An expert would connect the new output of `K.WSERVER` to the new input of `K.SORT`. This yields Figure 8 and the `Server` ⇝ `K.Server` extension is complete.

Now suppose we want `K.WSERVER` to change the list of attributes that are projected at runtime. Another extension accomplishes this: `K.Server` ⇝ `L.K.Server`, where `L` denotes feature `List`. This extension produces the PIM of Figure 10.

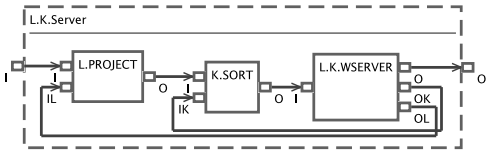

Fig. 10: The DfP `L.K.Server`.

Methodology. The procedure defined above is applied. `List` maps each element `e` ∈ `K.Server`

[1] In object-oriented parlance, `E` is an extension of `C` iff `E` is a subclass of `C`.

to `L.e` $\in$ `L.K.Server`. Namely, box `L.PROJECT` sprouts a new input port (to specify the list of attributes to project) and `L.K.WSERVER` sprouts a new output port (to provide that list of attributes). This produces the provisional DfP of Figure 11.

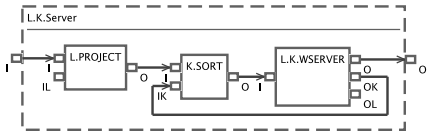

Fig. 11: Applying `L` to `K.Server`.

Figure 11 is completed by connecting the new input of `L.Project` to the new output of `L.K.WSERVER`. This yields Figure 10, the `K.Server` $\rightsquigarrow$ `L.K.Server` extension.

With the `Key` and `List` features, we defined three PIMs: `Server`, `K.Server`, and `L.K.Server`. There is a fourth: extend `Server` with just the `List` feature. Figure 12 depicts the different PIMs that can be built and the extension relationships among them. Starting from `Server`, we can either extend it with feature `Key` (obtaining `K.Server`) or with feature `List` (obtaining `L.Server`). Taking either of these PIMs, we can add the remaining feature to obtain `L.K.Server`. By doing so, we have created a tiny product line of `Server`s where `Server` is the base product and `Key` and `List` are optional features.

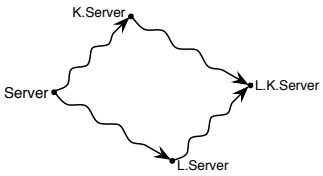

Fig. 12: A `Server` product line.

Henceforth, we assume the order in which features are composed is irrelevant: `L.K.Server` = `K.L.Server` as both mean `Server` is extended with features `List` and `Key`. This assumption is standard in the SPL literature where a product is identified by its *set* of features. Of course, dependencies among features can exist, where one feature requires (or disallows) another [2,6]. This is not the case for `Server`; nevertheless, our work does not preclude such constraints, as the user can provide a *feature model* [17] specifying those constraints.

3.3 Rewrite Rules, Derivations and Their Extensions

We now step back from the previous sections to expose our use of a rule base $\mathcal{R}$ of refinements and optimizations. Given a PIM `S`, we use one or more rules in $\mathcal{R}$ to derive a PSM `G`, written `S` $\xrightarrow{\mathcal{R}}{}^{*}$ `G`. Further, observe that PIM `S` is not unique: we could use the rules of $\mathcal{R}$ to derive a PSM for any one of a large collection of PIMs. A rule base $\mathcal{R}$ therefore encodes reusable steps in many PIM $\rightarrow^{*}$ PSM mappings.

Extensions fit into this universe in an interesting way. Just as extensions elaborate DfPs, extensions also elaborate rule bases and derivations:

- the `E` extension of rule base $\mathcal{R}$ is another rule base `E.`$\mathcal{R}$;
- the `E` extension of derivation (`S` $\xrightarrow{\mathcal{R}}{}^{*}$ `G`) is (`E.S` $\xrightarrow{\texttt{E.}\mathcal{R}}{}^{*}$ `E.G`).

Recall Section 3.1. We started with the `Server` PIM and derived its PSM. We used three rewrite rules: `SORT` $\rightarrow$ `parallel_sort` (we denote it by $\texttt{r}_1$), `PROJECT` $\rightarrow$ `parallel_project` (denoted by $\texttt{r}_2$), and the `ms_mergesplit` $\rightarrow$ `ms_identity` optimization (denoted by $\texttt{r}_3$). We presented the derivation `Server` $\xrightarrow{\texttt{r}_1}$ $\texttt{Server}_1$ $\xrightarrow{\texttt{r}_2}$ $\texttt{Server}_2$ $\xrightarrow{\texttt{r}_3}$ $\texttt{Server}_\Omega$ where $\texttt{Server}_\Omega$ is the PSM of Figure 7. The sequence of rewrites applied is $\texttt{r}_3 \cdot \texttt{r}_2 \cdot \texttt{r}_1$,[2] so we can write the derivation as `Server` $\xrightarrow{\texttt{r}_3\cdot\texttt{r}_2\cdot\texttt{r}_1}$ $\texttt{Server}_\Omega$.

Recall Section 3.2. We extended the `Server` PIM with features `Key` and `List`. Consider feature `Key`. We want to derive the PSM for `K.Server`. We can approximate this derivation by extending each rewrite rule $\texttt{r}_\texttt{i}$ to $\texttt{K.r}_\texttt{i}$, and applying them in order to yield the derivation `K.Server` $\xrightarrow{(\texttt{K.r}_3)\cdot(\texttt{K.r}_2)\cdot(\texttt{K.r}_1)}$ $\texttt{K.Server}_\Omega$, where $\texttt{K.Server}_\Omega$ is the PSM of `K.Server`.

Rule extension is a consequence of the concepts we discussed earlier. Figure 13 illustrates (`SORT` $\rightarrow$ `parallel_sort`) $\rightsquigarrow$ (`K.SORT` $\rightarrow$ `K.parallel_sort`), *i.e.*, how the $\texttt{r}_1$ rewrite rule is extended by the `Key` feature.

Methodology. Extending rewrite rules is no different than extending DfPs. To spell it out, a rewrite rule `L` $\rightarrow$ `R` specifies that `L` can be replaced with `R`. When feature/extension `K` is applied, `L` is mapped to a provisional `K.L` and `R` is mapped to a provisional `K.R`. These provisional DfPs are then completed by an expert to yield the non-provisional `K.L` and `K.R`. Rule extension follows: (`L` $\rightarrow$ `R`) $\rightsquigarrow$ (`K.L` $\rightarrow$ `K.R`).[3] The same holds for optimization rewrites.

[2] In standard function composition notation, the order in which the transformations are listed is the reverse of the order in which they are applied.

[3] Rule extensions need not be unique. Our experience to-date is that they are, largely because the increments in DfP functionality are sufficiently small for unique extensions to present themselves.

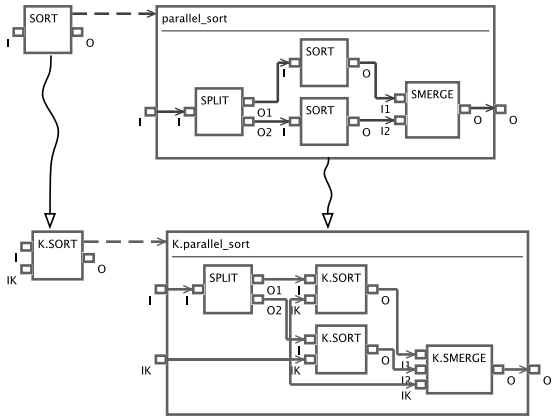

Fig. 13: Extending the `SORT` → `parallel_sort` rewrite rule.

3.4 Bringing It All Together

Figure 14 summarizes this section:

- we began with PIM `Server`;
- by rewriting `Server` using rules from a rule set $\mathcal{R}$, we derived an implementing DfP $\texttt{Server}_\Omega$;
- we extended `Server` with features `Key` and then `List` to create a small product line of PIMs, namely {`Server` ... `L.K.Server`};
- we extended $\mathcal{R}$ to rule sets $\texttt{K.}\mathcal{R}$, $\texttt{L.K.}\mathcal{R}$, and $\texttt{L.}\mathcal{R}$ by applying features `Key` and `List`;
- we extended the $\texttt{Server} \xrightarrow{\mathcal{R}} {}^*\texttt{Server}_\Omega$ derivation to corresponding derivations of implementations of `K.Server`, `L.K.Server`, and `L.Server`:

$$\texttt{K.Server} \xrightarrow{\texttt{K.}\mathcal{R}} {}^*\texttt{K.Server}_\Omega \tag{3.1}$$

$$\texttt{L.K.Server} \xrightarrow{\texttt{L.K.}\mathcal{R}} {}^*\texttt{L.K.Server}_\Omega \tag{3.2}$$

$$\texttt{L.Server} \xrightarrow{\texttt{L.}\mathcal{R}} {}^*\texttt{L.Server}_\Omega \tag{3.3}$$

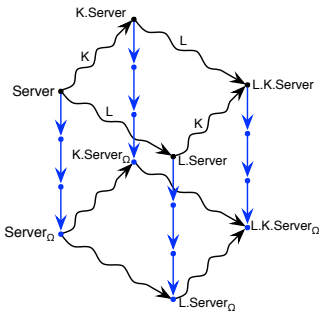

Fig. 14: Extending derivations and PSMs.

`Server` is a simple example. In more complex DfPs, obtaining extended derivations may require additional rewrite rules (not just the extended counterparts of previously used rewrites), or for previously used rewrites to be dropped. Such changes we cannot automate—they would have to be specified by a domain-expert. Nevertheless, a considerable amount of tool support can be provided to users and domain-experts in program derivation, precisely because the basic pattern of extension that we use is straightforward.

3.5 Commuting Diagrams of DfP Designs

Our approach to the design and derivation of DfPs is visual. Figure 15 shows the commuting diagram whose upper-left node is the `Server` PIM of Figure 3 and whose lower-right node is $\texttt{L.K.Server}_\Omega$ of Figure 10. This figure is digitally enlargable so that readers can "zoom" in on a particular design $\texttt{n}_{\texttt{ij}}$, extension $\texttt{n}_{\texttt{ij}} \rightsquigarrow \texttt{n}_{\texttt{i(j+1)}}$ or refinement/optimization $\texttt{n}_{\texttt{ij}} \rightarrow \texttt{n}_{\texttt{(i+1)j}}$ step.

The commuting diagrams that `ReFlO` produces are large, simply because systems that are being modeled are complex, they may have many steps in their PIM-to-PSM mappings, and they have many features.

Finally, we hinted at the possibility that feature-extended derivations may use additional refinements or optimizations that did not exist in the non-extended derivation (or vice versa). This means we may have identity transformations in the commuting diagram, so that the before-and-after extension derivations are of the same length, and that each vertical arrow $\texttt{X} \rightarrow \texttt{Y}$ in the original derivation corresponds to the `E`-extended arrow $\texttt{E.X} \rightarrow \texttt{E.Y}$ in the extended derivation (and vice versa). Identity-transformation-padded derivations are visible in our case studies.

In the next section, we explain how these ideas were implemented in our `ReFlO` tool, and then the basic workflow for using the approach we propose. After that, we present some steps of our derivations of two real applications using `ReFlO`.

4 Encoding Extensions (Higher-Order Rewrites)

There are many ways to encode (*i.e.*, express in a machine-understandable notation) extensions. At the core of `ReFlO` is its ability to store rewrite rules. Given an initial rule base $\mathcal{R}$, for each rule $\texttt{r} \in \mathcal{R}$ we maintain a (small) product line of rules: an initial rule `r` and each of its extensions. For a reasonable number of features (say < 20), a simple way to encode the variations of a rewrite rule is to form the union of it variants, and then annotate each element of the result to specify for which combinations of features it is to appear (and hence for which combinations of features it should be discarded).

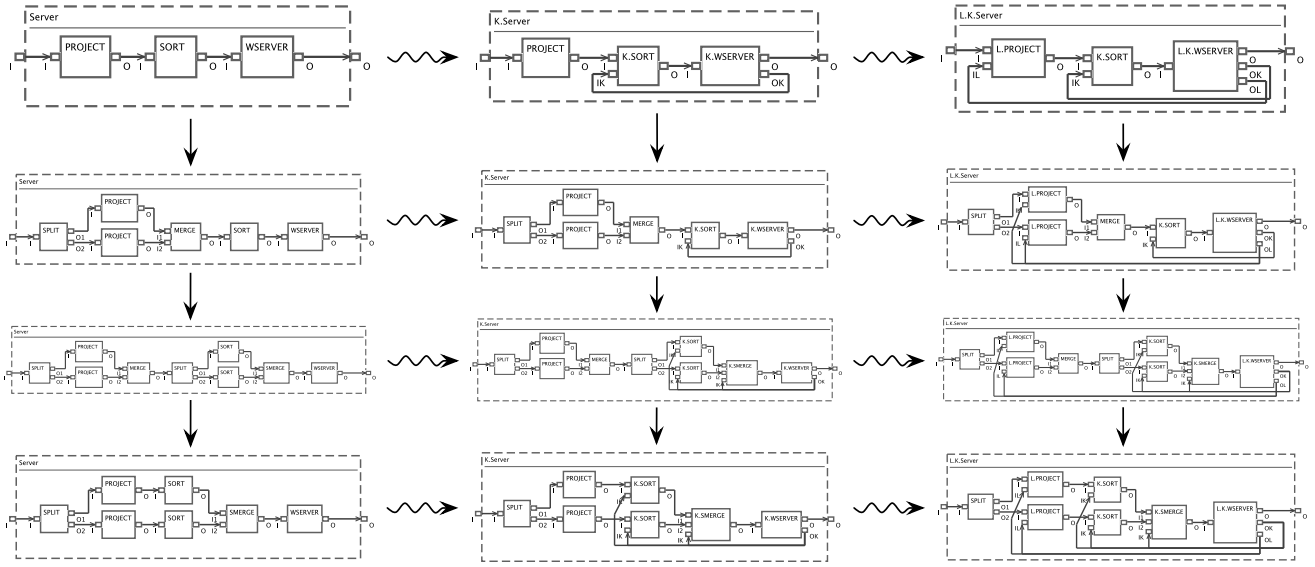

Fig. 15: The $\texttt{Server}_{\searrow \texttt{L.K.Server}_{\Omega}}$ commuting diagram.

This is an *annotative approach* to product line implementation [16].

Annotations specify how we can "project" a variant of a rewrite rule, for a certain combination of features, from the union of all variants of that rule. They may even specify that a rewrite rule should disappear for a certain combination of features. We do this for all rewrite rules of a rule base.

Rewrite rules (and its elements) are annotated with two attributes: a feature predicate and a feature tags set. The *feature predicate* determines when a box, port, or connector is part of a rewrite rule. The *feature tags set* determines how boxes are tagged/labeled, *i.e.*, `K` is a tag for feature `Key`. In this section we explain how these annotations specify a product line of rule bases, and how they enable the projection of the rewrite rules variants.

4.1 eXtended `ReFlO` Domain Models

In `ReFlO`, a rule base is encoded in a `ReFlO` *Domain Model (RDM)*. We defined its UML class diagram metamodel in [24]. With annotations, we enhanced this metamodel (see Figure 16). Now an initial rule base and its extensions are superimposed into a single artifact called an *eXtended* `ReFlO` *Domain Model (XRDM)*, which encodes a product line of RDMs. A projection of an XRDM produces an RDM supporting a given set of features. That is, whereas an RDM defines rewrite rules supporting a *fixed set of features*, an XRDM is the result of superimposing multiple RDMs, effectively defining rewrite rules supporting *multiple sets of features*.

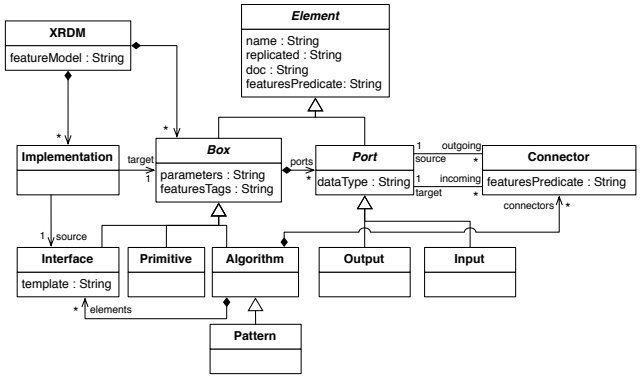

Fig. 16: UML class diagram metamodel.

Boxes, ports, and connectors now have a `featuresPredicate` attribute. Given a subset of features $\mathcal{S} \subseteq \mathcal{F}$ and a model element with predicate $\texttt{P} : \mathcal{P}(\mathcal{F}) \to \{\texttt{true}, \texttt{false}\}$ (where $\mathcal{P}$ denotes the power set), $\texttt{P}(\mathcal{S})$ is true if and only if the element is part of the RDM when $\mathcal{S}$ are the enabled features. We use a propositional formula to specify `P`, where its atoms represent the features of the domain. $\texttt{P}(\mathcal{S})$ is computed by evaluating the propositional formula associating `true` to the atoms corresponding to features in $\mathcal{S}$ and associating `false` to the remaining.

Boxes now have another attribute, `featuresTags`. It is a set of abbreviated feature names that determines box tagging. A *tag* is a prefix that is added to a box's name to identify the variant of the box being used (*e.g.*, `L` and `K` are tags of box `L.K.WSERVER`, specifying that this

box is a variant of the WSERVER with features L(ist) and K(ey)).

Example. Recall our web server example. We can define rewrite rule WSERVER → pwserver to specify a primitive implementation (direct code implementation [24]) for WSERVER (see Figure 17a).
When feature Key (abbreviated as K) is applied to this rule, a new port (OK) is added to the WSERVER and pwserver boxes. As these ports are present only when feature Key is enabled, they are annotated with predicate key. Further, the boxes now provide extra behavior, therefore we need to add the K tag to each. The result is depicted in Figure 17b (red boxes show tags sets, and blue boxes show predicates).
When feature List (abbreviated as L) is applied, another port (OL) is added to both boxes. Again, these ports are annotated with a predicate (in this case, list specifies the ports are only part of the model when feature List is enabled). The set of tags of each box also receives an additional tag L. The final model is depicted in Figure 17c.

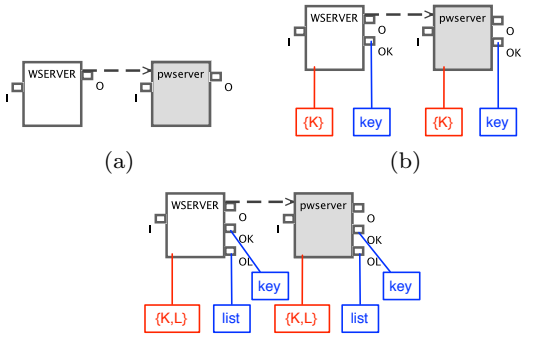

Fig. 17: Incrementally specifying a rewrite rule.

This information encodes the extensions of the rule base and allows us to project an RDM for a specific set of features from the XRDM.

Features may have inter-dependencies, *i.e.*, a certain feature may require or exclude another, which in SPLs are typically specified by a feature model. Therefore, the XRDM has an additional attribute, featureModel, allowing users to specify a feature model that expresses the valid combinations of features, capturing their dependencies and incompatibilities. ReFlO uses the grammar notation of [6] to specify the feature model.

4.2 Projection of an RDM from the XRDM

A new transformation is needed to map an XRDM to an RDM with the desired features enabled. This transformation takes an XRDM, and a given set of active features, and *projects* the RDM for that set of features. The projection is done by examining all model elements and hiding (or making inactive) those elements whose predicate is evaluated to false for the given list of features. To simplify predicate specifications, we use implicit rules that determine when an element must be hidden regardless of the result of evaluating its predicate. The idea is that when a certain element is hidden, its dependent elements must also be hidden. For example, when a box is hidden, all of its ports must also be hidden. A similar reasoning may be applied in other cases. The rules are:

- if the lhs of a rewrite rule is hidden, the rhs is hidden;
- if a box is hidden, all of its ports are hidden;
- if a graph is hidden, so too are its internal boxes and connectors;
- if a port is hidden, the connectors linked to that port are hidden.

These rules greatly reduce the effort needed to specify an XRDM, as repetition of formulas is avoided. Consequently, the projection algorithm we use is straightforward.

Part of a projection is to determine which tags are attached to each box. Given the set $\mathcal{S}$ of selected features, and given box B with tag set T, the tags of B after the projection are $\mathtt{T} \cap \mathcal{S}$. That is, T specifies the features that change the behavior of B, but we are only interested in the enabled features specified by $\mathcal{S}$.

Example. Consider the rewrite rule from Figure 17c and assume $\mathcal{S} = \{\mathtt{K}\}$. Projection yields Figure 18. Ports OK, that depend on feature Key, are present. However, ports OL, that depend on feature List, are hidden. Additionally, both boxes are tagged with K (as $\{\mathtt{K},\mathtt{L}\} \cap \{\mathtt{K}\} = \{\mathtt{K}\}$).

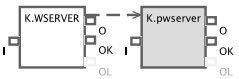

Fig. 18: Projection of feature K from rewrite rule WSERVER → pwserver (note the greyed out OL ports).

The projection is only allowed if the selected combination of features is valid (according to the user specified feature model).

5 Approach Workflow

Our goal is to build a knowledge base by reverse engineering existing systems, a process conducted by domain experts (or by a developer with the support of a

domain expert). As the knowledge base becomes larger, it will eventually be usable for forward engineering, allowing developers to automatically explore the space of implementations for a domain of programs.

An expert starts with a simplified specification (PIM) of a system he wants to reverse engineer—it is simplified as some features of the target system have been removed. He creates interfaces for required domain operations along with their possible implementations and optimizations, and codifies them as `ReFlO` rewrite rules. Using these rules, he derives a PSM that implements the PIM.

Next, the expert adds a feature `F` to the PIM to elaborate it, and then he reviews existing rewrite rules to `F`-extend them. That is, he examines each rewrite rule, determines how it is affected by `F`, and adds the new model elements and annotations needed to support `F`. In the end, the expert can apply a projection to the XRDM, to obtain the RDM with the rewrite rules supporting the new feature, and repeat the derivation to produce an `F`-extended PSM for the `F`-extended PIM. During this process, the domain expert may realize that completely new rewrite rules are also needed, which he adds to the XRDM. This process is repeated by progressively adding features, until a PIM is reached that matches the target system. The PIM-to-PSM mappings produced along the way are extended too, yielding a PSM that matches the target system. At this point, assuming the refinements, optimizations, and extensions used are correct, the domain expert has a correct-by-construction design of the target system.

The expert may choose different orders in which to add features. Different orders will expose feature interactions in different orders—in the end, all features and interactions will be accounted for. But he should also take into account the feature dependencies imposed by the feature model. That is, if feature `G` depends on feature `F`, he must add feature `F` before `G`. The result of evaluating a feature predicate is the same for a given set of features, regardless of the order in which they were added. The set of features chosen must comply with the feature model, though the projection operation verifies this constraint.

Although `ReFlO` does not guarantee the behavioral correctness of the rewrite rules of an XRDM, it provides a *safe composition* [52] mechanism, which allows experts to check whether all projections that can obtained from an XRDM are syntactically correct and whether types of interfaces and implementations are compatible. Moreover, the interpretations mechanism provided by `ReFlO` can be explored to implement more complex validation rules. See [24] for details.

6 Case Studies

We highlight the most sophisticated dataflow applications that we reverse-engineered in this section: a *crash fault-tolerant (CFT)* server called UpRight [14] and a parallel *molecular dynamics (MD)* [22] simulator called MolDyn [12,48].

As said earlier, `ReFlO` rewrites and extensions encode deep domain knowledge—knowledge that typically is appreciated (only) by domain experts. Consequently, we expect few readers of this paper to be experts in either CFT or MD. Admittedly for us, only the fourth author (Riché) was an expert in CFT (he was a co-author of UpRight) and the first author (Gonçalves) was familiar with an MD application. Without expertise, our case studies read like a semantics-free structured evolution of graphs.

Of course, this is not the case. For UpRight, Riché built a lightweight, concurrent actor framework in Python [45], and coded each of its different derivations and extensions by hand. (He finished this work prior to the completion of `ReFlO`.) At each step, he ran a regression suite to ensure that all tests passed. After each extension, more tests were included to check that the extra functionality was correct. As for MolDyn, existing C++ code components and test suite were used to support the translation from models to code, and to verify code (and models) correctness [41]. *That is, from a software engineering viewpoint, our approach allowed us to write tests to confirm that each of our design modifications were correct.*

The next sections give an overview of our CFT and MD designs and extensions.

6.1 UpRight

Figure 19a shows an SPL of four variant PIMs of UpRight and their derivations: Synchronous CFT (SCFT), Asynchronous CFT (ACFT), Authenticated Synchronous CFT (ASCFT), and Authenticated Asynchronous CFT (AACFT). The shaded region in Figure 19a denotes a commuting diagram that we highlight below; the remaining commuting diagrams (of which there are many) illustrate the same ideas of prior sections, except they are more complicated.

The full commuting diagram with `SCFT` PIM in the upper-left corner and $\texttt{AACFT}_{\Omega}$ in the lower-right is given in Appendix A. For more details, see [23].

6.1.1 Highlighted Commuting Diagram

Figure 20 is an enlarged version of the highlighted commuting diagram. The ACFT PIM (upper left) has a set

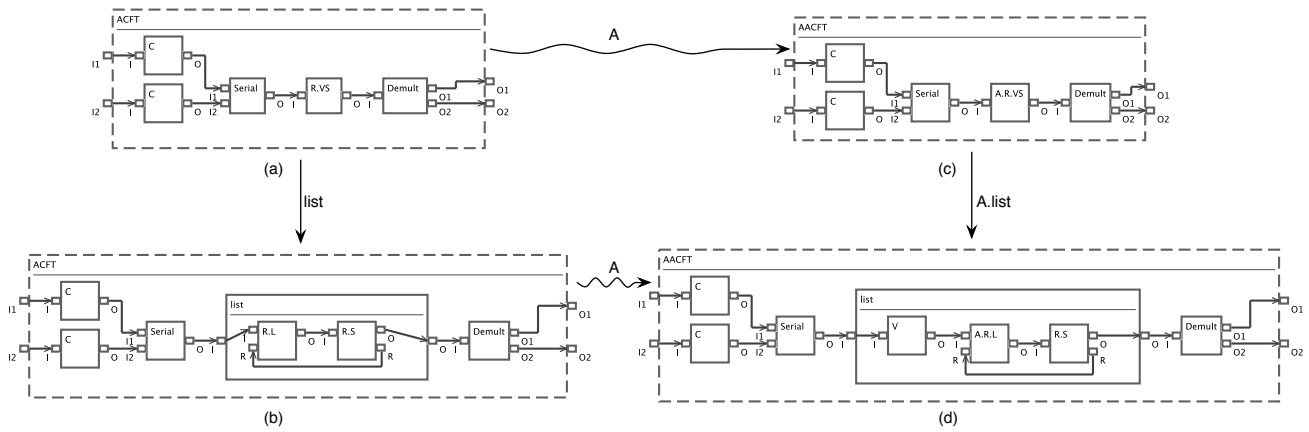

Fig. 20: An UpRight commuting diagram.

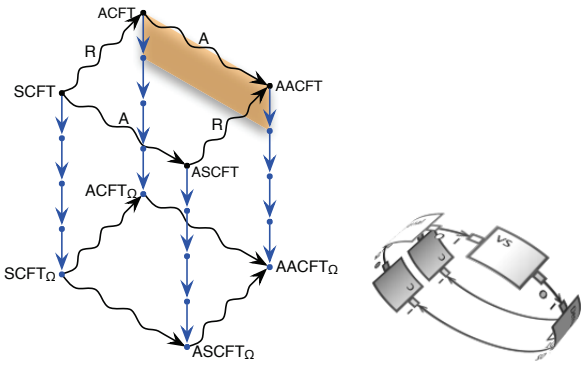

Fig. 19: UpRight's (a) SPL derivations and (b) PIM cylinder.

of client `C` boxes (only two are shown, but an arbitrary number is supported) that submit messages to a `Serial` box that multiplexes messages to a recoverable virtual server `R.VS`. Each message is processed by the server, the server updates its state, and then sends its result to a `Demult` box that routes the response message back to its originating client. ACFT PIM of Figure 20a is formed by unrolling the cylinder in Figure 19b.

The first refinement of ACFT is `list`; it replaces server `R.VS` with a recoverable list `R.L`, which queues messages and remembers its state (for purposes of recovery), followed by a recoverable server `R.S`. Box `R.L` sends messages to `R.S` for processing (this is the `R.L` → `R.S` connector in Figure 20b). When `R.S` recovers from a failure, it requests/needs state information from `R.L` (this is the `R.S` → `R.L` connector in Figure 20b).

Adding `Authentication` to ACFT extends it to the AACFT PIM (Figure 20a ⇝ Figure 20c). The only box affected by `Authentication` is the recoverable virtual server, *i.e.*, `R.VS` ⇝ `A.R.VS`.

`A.list`, an `A`-extended `list` rewrite, is the first refinement of AACFT. Messages are first validated (authenticated) by box `V`, which discards invalid messages. Valid messages are then sent to the list box `A.R.L` which queues and forwards messages to a recoverable server `R.S`. As before, when `R.S` is recovering from a failure, it requests state information from the list box `A.R.L`.

6.1.2 Encoding Extensions

For each of the four UpRight designs in Figure 19a, there is a unique definition of the list rewrite, namely `list`, `R.list`, `A.list`, and `A.R.list`. Figure 21 shows the annotations of this rewrite rule with the features `A` and `R` that allow `ReFlO` to project the correct version of the rewrite given a set `F` of features, where $\texttt{F} \subseteq \{\texttt{R}, \texttt{A}\}$.

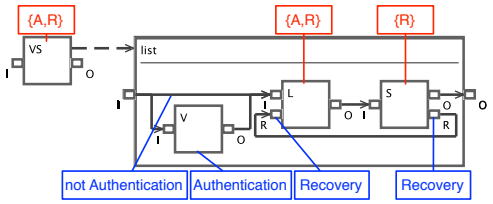

Fig. 21: Annotated `VS` → `list` rewrite rule.

6.2 MolDyn

Figure 22 shows an SPL with six variant PIMs of MolDyn and their derivations: the base MDCore, which can be enhanced with features Neighbors, Blocks, and Cells (which requires the Blocks feature). The shaded region represents the commuting diagram (Figure 22) that we highlight in this section. The full commuting diagram with `MDCore` PIM in the upper-left corner and $\texttt{CBNMDCore}_{\Omega}$ in the lower-right is given in Appendix B, noting the initial derivation `MDCore` → $\texttt{MDCore}_{\Omega}$ is visually simpler than the final derivation `CBNMDCore` → $\texttt{CBNMDCore}_{\Omega}$. For more details, see [23].

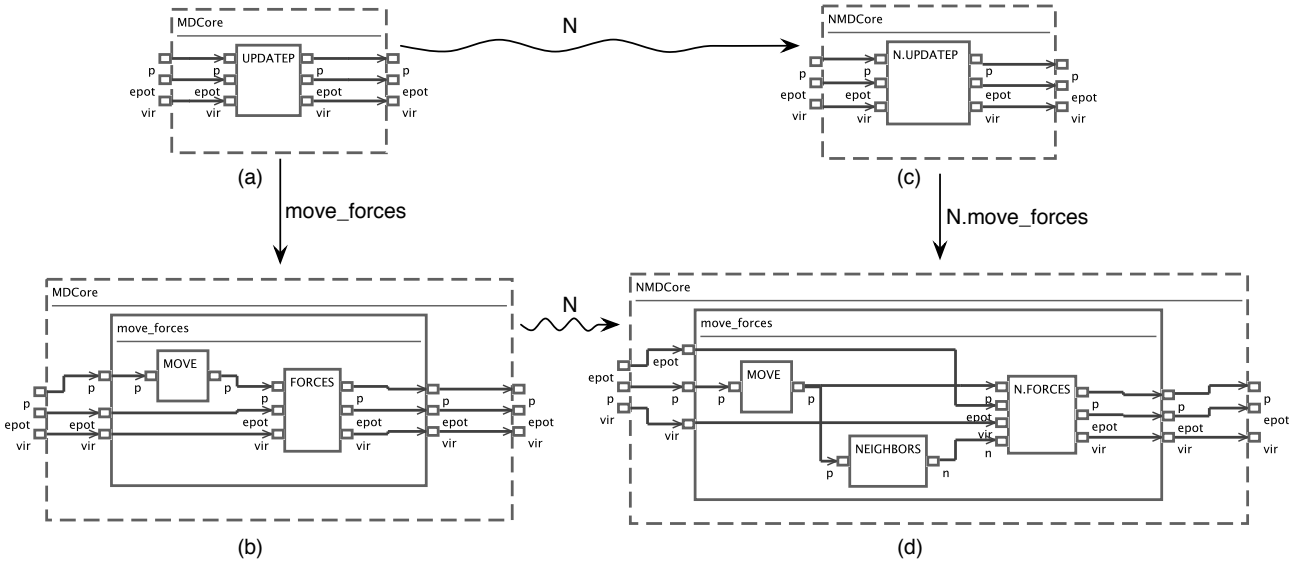

Fig. 23: A MolDyn commuting diagram.

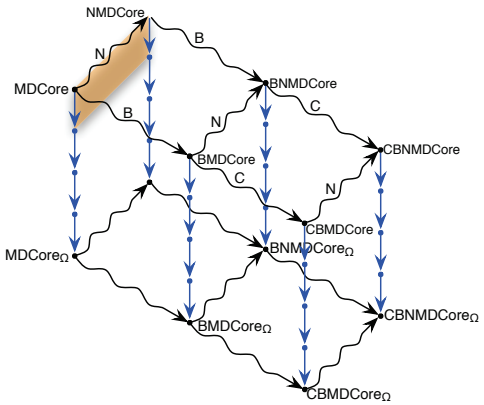

Fig. 22: MolDyn derivations.

6.2.1 Highlighted Commuting Diagram

MD simulations use computational resources to predict properties of materials [22]. Materials are modeled by a set of particles (*e.g.*, atoms or molecules) with certain properties (*e.g.*, position, velocity, and force). The set of particles is initialized with properties such as density and initial temperature. The simulation starts by computing interactions between particles, iteratively updating properties, until the system stabilizes, at which point the properties of the material can be studied/measured.

The expensive part of MD simulation is computing particle interactions (forces among particles), where a naive implementation has a complexity of $\texttt{O}(\texttt{N}^2)$ where `N` is the number of particles. Figure 23a shows the base PIM we use. It contains `UPDATEP` box, which express the core operation of an MD simulation, the update of particles. The final goal of this derivation is to obtain an optimized parallel implementation of MolDyn (with support for shared memory, distributed memory, or both). The selected commuting diagram of Figure 23 shows only the first step of this derivation, where we expose the two operations needed to update the particles: first particles positions are updated (`Move` box), then forces among particles are recomputed (`FORCES` box).

A common technique used in MD simulations to reduce the base $\texttt{O}(\texttt{N}^2)$ complexity is pre-computing and caching the list of particles that interact with another particle [57]; doing so improves performance. Called the `Neighbors` feature, `N` may or may not change the behavior of the simulation.[4] Implementing `N` requires an extension of the internal boxes used by the program, which results in the PIM `NMDCore` shown Figure 23c, which is identical (sans `N.UPDATEP`) to Figure 23a.

To obtain the implementation for this PIM we need to extend the `move_forces` algorithm. The extended algorithm exposes the two steps of updating particles as before, through boxes `MOVE` and `FORCES`. However, box `FORCES` is extended and now has a new input, which receives a list of neighbors of each particle (*i.e.*, the particles that interact with a certain particle). Moreover, a new box (`NEIGHBORS`) was added to compute this list of neighbors, using as input the set of particles output by box `MOVE`. Box `MOVE` itself is not affected by this extension.

[4] We can "relax" the correctness criteria of the simulation (and therefore change the behavior of the program) [57] to improve performance.

6.2.2 Encoding Extensions

Figure 24 shows the annotated rewrite rule `UPDATEP` → `move_forces` after specifying the `Neighbors` extension, which allow `ReFlO` to project both rewrite rules used in the commuting diagram of Figure 23.

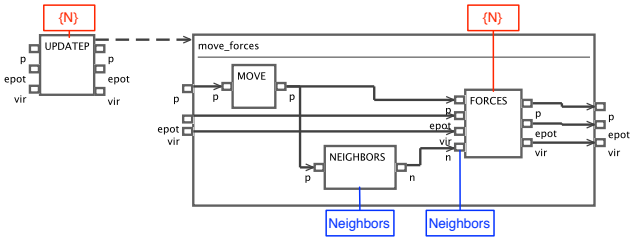

Fig. 24: Annotated `UPDATEP` → `move_forces` rewrite rule.

6.3 Discussion

We re-engineered the dataflow designs of UpRight and MolDyn by starting with a simple PIM-to-PSM derivation $\Delta_{\mathtt{A}}$ starting with a simple PIM $\mathtt{A}_0$ and deriving its PSM $\mathtt{A}_\Omega$:

$$\Delta_{\mathtt{A}} \;=\; \mathtt{A}_0 \rightarrow^* \mathtt{A}_\Omega$$

An expert validated the PIM design, the PSM design, and the steps (rewrites) used in derivation. We validated this work by an implementation.

We then extended the derivation by adding a feature `B` to the PIM, to the rewrite rules, and including additional rules that were needed to produce the feature-extended PSM and its derivation:

$$\Delta_{\mathtt{B}} \;=\; \mathtt{B}.(\mathtt{A}_0 \rightarrow^* \mathtt{A}_\Omega) \;=\; (\mathtt{B}.\mathtt{A}_0) \rightarrow^* (\mathtt{B}.\mathtt{A}_\Omega) \;=\; \mathtt{B}_0 \rightarrow^* \mathtt{B}_\Omega$$

Again, we confirmed the correctness of the extension via implementation. By repeating the process of adding more features, we extended the original derivation $\Delta_{\mathtt{A}}$ to a derivation $\Delta_{\mathtt{Z}}$ whose PSM is the design of the target system we wanted to re-engineer:

$$\Delta_{\mathtt{Z}} \;=\; \mathtt{Z}_0 \rightarrow^* \mathtt{Z}_\Omega$$

This is the process that we followed and that we recommend others to follow to re-engineer legacy dataflow applications.

Our extensions approach is integrated with `ReFlO` framework, which does not impose a particular model of computation to our DfP, *i.e.*, different domains may specify different rules for how a DfP is to be translated to code and executed. The dataflow computing model [31] is an obvious candidate and it is the one used by UpRight. In contrast, MolDyn (and in some other domains analyzed) the translation to code simply treats each box as a function that must be executed with some order constraints (and parallelism is obtained executing multiple instances of the DfP, following an *single program multiple data* model [19]). When specifying the rewrite rules, extensions or PIMs, the user should make sure they are correct with regard to the model used to generate code and execute the DfP.

7 Related Work

There is a strong connection between classical work on formal program development and the approach we propose. Z [51], Event-B [1], and *Abstract State Machines (ASMs)* [10] have notions of refinement and extension. (Event-B uses different terms—horizontal refinement and vertical refinement—for the same ideas.) Event-B and ASMs focus on state transition representations of programs, whereas we deal with DfPs. Our emphasis is not on developing proofs of program correctness, but instead what it takes to encode the knowledge used/need as rewrite rules to build complex and efficient DfPs and product lines of DfPs in an MDE context.

We use extensions to explain the effects of optional features in dataflow graphs, allowing us to encode a SPL of dataflow graphs. There are several techniques in which features of SPLs can be implemented. Some are compositional, including AHEAD [5], FeatureHouse [3], *Delta-Oriented Programming (DOP)* of Bettini et al. [9], and AOP [33], all of which work mainly at code level. Other solutions have been proposed to handle SPLs of higher-level models [40,44].

We use an annotative approach where a set of artifacts containing all features/variants are superimposed. Artifacts (*e.g.*, code, model elements) are annotated with feature predicates to determine when these artifacts are visible in a particular combination of features. Preprocessors are a primitive example [35] of a similar technique. Code with preprocessor directives can be made more understandable by tools that color code [21] or that extract views from it [50]. More sophisticated solutions exist, such as XVCL [29], Spoon [42], Spotlight [15], or CIDE [32]. However, our solution works at a model level, not code.

Other annotative approaches also work at the model level. Ziadi et al. [60] proposed a UML profile to specify model variability in UML class diagrams and sequence diagrams. Czarnecki and Antkiewicz [16] proposed a template approach, where model elements are annotated with presence conditions (similar to our feature predicates) and meta-expressions. FeatureMapper [28] allows the association of model elements (*e.g.*, classes and associations in a UML class diagram) to features.

Instead of annotating program architectures directly (usually too complex), we annotate model transformations (simpler) that are used to derive program implementations. This reduces the complexity of the annotated models, and it also makes extensions available when deriving other implementations, thereby making extensions more reusable.

An appealing alternative to annotations, and closer in spirit to compositional approaches, is work by Haber, et al. on Delta Simulink [27]. Simulink models are component (multi-)graphs, syntactically similar to dataflow graphs [49]. Deltas (from DOP) are a sequence of block add, remove, replace, and modify operations on Simulink graphs. Deltas are used to express features (extensions) of Simulink graphs, and is an alternative to Czarnecki's annotative approach. To us, graph rewrites provide a higher-level and a more natural modeling approach when it comes to correct-by-construction program derivation. Whether annotations or deltas are better for encoding extensions remains an open problem for SPL construction, in general, and not just graphs.

Our work can be used to extract an SPL from legacy applications. RE-PLACE [8] is an alternative to reengineer existing systems into SPLs. Other approaches have been proposed with similar intent, employing refactoring techniques [34,36,56].

Extracting variants from a XRDM is similar to program slicing [59]. Slicing has been applied to models [30, 4] to reduce model complexity and make easier for developers to analyze models. These approaches are focused on the understandability of the artifacts, whereas in our work focuses on rule variability. Nevertheless, `ReFlO` projections remove elements from rewrite rules that are not needed for a certain combination of features, which we believe also contribute to improved rewrite rule understandability. In [58] Wasowski proposes a slice-based solution where SPLs are specified using restrictions, that remove features from a model, so that a variant can be obtained.

We use a dataflow notation in our work. Similar graphical notation has been used by several other tools such as LabVIEW [53], Simulink [49], Weaves [26], Fractal [11], or StreamIt [54]. However, they focus on component specification and construction of systems composing those components. We are unaware of any support for extensions in these tools.

`ReFlO` supports analyses to verify whether all projections of a XRDM that can be obtained meet the metamodel constraints. The analysis method used is based on solutions previously proposed by Czarnecki and Pietroszek [18] and Thaker *et al.* [52].

8 Conclusions

Designing program architectures always has an element of "magic". The term "spaghetti diagram" was coined to express architectural diagrams that are indecipherable, except to their authors. **How** these architectures work and **why** they work remains a mystery but to a few people.

Classical formal approaches to software development recognized this problem by elaborating designs in an incremental manner. One starts with a simple specification. This specification is progressively elaborated by understandable increments in functionality. When the full specification is produced, a derivation of its implementation is undertaken, using refinements and optimizations.

Our work explored this classical approach from the applied perspective of deriving complex dataflow programs (DfPs) from an MDE perspective. Our involvement stemmed from the need to reverse-engineer complex legacy DfPs [24,47]. In prior work [24], we showed how refinement and optimization design knowledge of DfPs could be captured as graph transformations, so that a complex and optimized DfP (or PSM) could be derived from a high-level DfP (or PIM). We observed that starting PIMs could be complicated, and the transformations that refined and optimized such PIMs were themselves complex: hard to recognize, hard to define, and (frankly) hard to believe [46].

This paper blends ideas from classical formal work on program design with our prior work [24,47]. We begin with an elementary PIM and derive its PSM using simple, domain-specific refinements and optimizations. This set of graph rewrites defines a rule base $\mathcal{R}$. We showed how increments in program functionality, called features, could be integrated into this universe. A feature `F` extends a graph `G` to graph `F.G` through the addition, modification, and removal of boxes and connectors. Each graph rewrite rule `L`→`R` is extended by feature `F` to an extended rule `F.(L`→`R)` = `F.L`→`F.G`. `F`-extending a rule base $\mathcal{R}$, which adds new rules, delete rules, and modifies existing rules, yields an `F`-extended rule base `F`.$\mathcal{R}$.

Further, we showed that PIM$\xrightarrow{\mathcal{R}}{}^*$PSM derivations (the notation meaning apply one or more rules $\mathtt{r} \in \mathcal{R}$ to map a PIM graph to a PSM graph), were also subject to extensions. That is, the `F`-extension of derivation PIM$\xrightarrow{\mathcal{R}}{}^*$PSM was `F`.PIM$\xrightarrow{\mathtt{F}.\mathcal{R}}{}^*$`F`.PSM.

We explained how extensions were implemented in the `ReFlO` tool; we used an annotative approach to superimpose rules that were related by extension. By a simple projection operation, we could recreate the rule that was specific to a set of features. By doing this for

all rules, we could recreate the rule base that was specific to a set of features.

Our approach relies on domain experts to define the rewrite rules and extensions (that comprise the domain knowledge). To make the approach easier for those domain experts, we use a declarative notation that relates graphs providing equivalent behavior, where the expert does not have to care about the steps that are required to actually perform the transformations on a DfP.

The approach we propose, supported by the `ReFlO` tool/framework, provides an important contribution to encode and systematize domain knowledge that is used by experts, and to show **how** and **why** complex architectures work in a more understandable way. `ReFlO` includes model-to-text capabilities, which enable the generation of runnable code from models.

Availability. The `ReFlO` framework with extensions support can be downloaded at `http://cs.utexas.edu/users/schwartz/DxT/reflo/x/`.

Acknowledgements We gratefully acknowledge helpful feedback from B. Marker (U. Texas), and from the anonymous reviewers. 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 projects FCOMP-01-0124-FEDER-010152, FCOMP-01-0124-FEDER-011413 and UID/CEC/00319/2013. Rui Gonçalves is additionally funded by FCT grant SFRH/BD/47800/2008. We also gratefully acknowledge support for this work by NSF grants CCF-0724979, CCF-1421211, and OCI-1148125.

References

1. J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 1st edition, 2010.
2. S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines*. Springer Berlin Heidelberg, 2013.
3. S. Apel, C. Kästner, and C. Lengauer. Featurehouse: Language-independent, automated software composition. In *ICSE '09: Proceeding of the 31st International Conference on Software Engineering*, pages 221–231, 2009.
4. J. H. Bae, K. Lee, and H. S. Chae. Modularization of the UML metamodel using model slicing. In *ITNG '08: Proceedings of the 5th International Conference on Information Technology: New Generations*, pages 1253–1254, 2008.
5. D. Batory. Feature-oriented programming and the AHEAD tool suite. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 702–703, 2004.
6. D. Batory. Feature models, grammars, and propositional formulas. In *SPLC '05: Proceedings of the 9th international conference on Software Product Lines*, pages 7–20, 2005.
7. D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004.
8. J. Bayer, J.-F. Girard, M. Würthner, J.-M. DeBaud, and M. Apel. Transitioning legacy assets to a product line architecture. *ACM SIGSOFT Software Engineering Notes*, 24(6):446–463, 1999.
9. L. Bettini, F. Damiani, and I. Schaefer. Compositional type checking of delta-oriented software product lines. *Acta Informatica*, 50(2):77–122, 2013.
10. E. Borger and R. F. Stark. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.
11. E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Software—Practice & Experience*, 36(11-12):1257–1284, 2006.
12. J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A benchmark suite for high performance java. *Concurrency: Practice and Experience*, 12(6):81–88, 1999.
13. C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. Flumejava: Easy, efficient data-parallel pipelines. In *PLDI '10: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 363–375, 2010.
14. A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. L. Riché. UpRight cluster services. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 277–290, 2009.
15. D. Coppit, R. R. Painter, and M. Revelle. Spotlight: A prototype tool for software plans. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 754–757, 2007.
16. K. Czarnecki and M. Antkiewicz. Mapping features to models: a template approach based on superimposed variants. In *GPCE '05: Proceedings of the 4th international conference on Generative Programming and Component Engineering*, pages 422–437, 2005.
17. K. Czarnecki and U. W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., 2000.
18. K. Czarnecki and K. Pietroszek. Verifying feature-based model templates against well-formedness OCL constraints. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 211–220, 2006.
19. F. Darema. The SPMD model: Past, present and future. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 2131. Springer Berlin Heidelberg, 2001.
20. 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*, 2(1):44–62, 1990.
21. J. Feigenspan, M. Papendieck, C. Kästner, M. Frisch, and R. Dachselt. Featurecommander: Colorful #ifdef world. In *SPLC '11: Proceedings of the 15th International Software Product Line Conference*, pages 48:1–48:2, 2011.
22. D. Frenkel and B. Smit. *Understanding molecular simulation: from algorithms to applications*. Academic press, 2001.
23. R. C. Gonçalves. *Parallel Programming by Transformation*. PhD thesis, Universidades do Minho, Aveiro e Porto, Braga, 2015.
24. R. C. Gonçalves, D. Batory, and J. L. Sobral. ReFlO: An interactive tool for pipe-and-filter domain specification

and program generation. *Software and Systems Modeling*, 2014.
25. Google Cloud Dataflow. `https://cloud.google.com/dataflow/`.
26. 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*, pages 23–34, 1991.
27. A. Haber, C. Kolassa, P. Manhart, P. M. S. Nazari, B. Rumpe, and I. Schaefer. First-class variability modeling in matlab/simulink. In *VaMoS '13: Proceedings of the 7th International Workshop on Variability Modelling of Software-intensive Systems*, pages 4:1–4:8, 2013.
28. F. Heidenreich, J. Kopcsek, and C. Wende. FeatureMapper: mapping features to models. In *ICSE Companion '08: Companion of the 30th international conference on Software engineering*, pages 943–944, 2008.
29. S. Jarzabek, P. Bassett, H. Zhang, and W. Zhang. XVCL: XML-based variant configuration language. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 810–811, 2003.
30. H. Kagdi, J. I. Maletic, and A. Sutton. Context-free slicing of UML class models. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 635–638, 2005.
31. G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing '74: Proceedings of the IFIP Congress*, pages 471–475, 1974.
32. C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 311–320, 2008.
33. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP '97: Proceedings of the 11th European Conference on Object-Oriented Programming*, pages 220–242, 1997.
34. R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi. A case study in refactoring a legacy component for reuse in a product line. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 369–378, 2005.
35. J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 105–114, 2010.
36. J. Liu, D. Batory, and C. Lengauer. Feature oriented refactoring of legacy applications. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 112–121, 2006.
37. Mapreduce. `http://en.wikipedia.org/wiki/MapReduce`.
38. B. Marker, D. Batory, and R. A. van de Geijn. Understanding performance stairs: Elucidating heuristics. In *ASE '14: Automated Software Engineering*, 2014.
39. B. Marker, J. Poulson, D. Batory, and R. van de Geijn. Designing linear algebra algorithms by transformation: Mechanizing the expert developer. In *High Performance Computing for Computational Science—VECPAR 2012*, volume 7851 of *Lecture Notes in Computer Science*, pages 362–378. Springer Berlin Heidelberg, 2013.
40. A. McNeile and N. Simons. State machines as mixins. *Journal of Object Technology*, 2(6):85–101, 2003.
41. Md product line. `http://alba.di.uminho.pt/research/md-product-line/`.
42. R. Pawlak. Spoon: Compile-time annotation processing for middleware. *IEEE Distributed Systems Online*, 7(11), 2006.
43. B. Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, 1991.
44. C. Prehofer. Plug-and-play composition of features and feature interactions with statechart diagrams. *Software and Systems Modeling*, 3(3):221–234, 2004.
45. Python code generator. `http://code.google.com/p/stepwise-ft/`.
46. T. L. Riché, D. Batory, R. C. Gonçalves, and B. Marker. Architecture design by transformation. Technical Report TR-10-39, The University of Texas at Austin, Department of Computer Sciences, 2010.
47. 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*, pages 84–92, 2012.
48. R. A. Silva and J. L. Sobral. Optimizing molecular dynamics simulations with product lines. In *VaMoS '11: Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, pages 151–157, 2011.
49. Simulink - Simulation and Model-Based Design. `http://www.mathworks.com/products/simulink/`.
50. N. Singh, C. Gibbs, and Y. Coady. C-CLR: a tool for navigating highly configurable system software. In *ACP4IS '07: Proceedings of the 6th workshop on Aspects, components, and patterns for infrastructure software*, 2007.
51. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 1989.
52. S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In *GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering*, pages 95–104, 2007.
53. The LabVIEW Environment. `http://www.ni.com/labview/`.
54. W. Thies. *Language and Compiler Support for Stream Programs*. PhD thesis, MIT, 2008.
55. 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*, pages 18–33, 2009.
56. S. Trujillo, D. Batory, and O. Diaz. Feature refactoring a multi-representation program into a product line. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 191–200, 2006.
57. L. Verlet. Computer "experiments" on classical fluids. i. thermodynamical properties of lennard-jones molecules. *Physical Review*, 159(1):98–103, 1967.
58. A. Wasowski. Automatic generation of program families by model restrictions. In *Software Product Lines*, volume 3154 of *Lecture Notes in Computer Science*, pages 73–89. Springer Berlin Heidelberg, 2004.
59. M. Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, 1981.
60. T. Ziadi, L. Hélouët, and J.-M. Jézéquel. Towards a UML profile for software product lines. In *Software Product-Family Engineering*, volume 3014 of *Lecture Notes in Computer Science*, pages 129–139. Springer Berlin Heidelberg, 2004.

A UpRight Commuting Diagrams

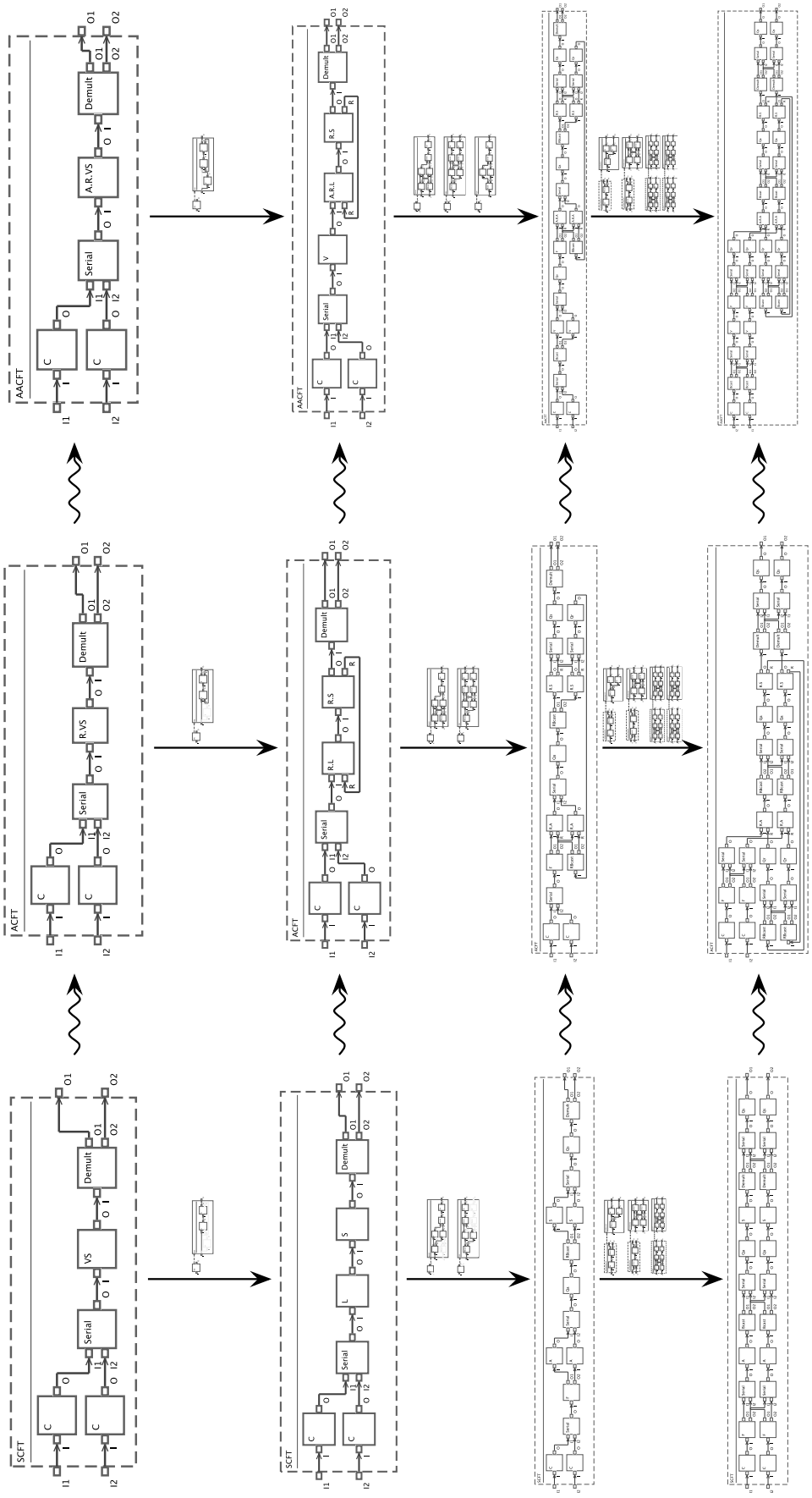

Fig. 25: The $\mathtt{SCFT}_{\searrow \mathtt{AACFT}_\Omega}$ commuting diagram focusing the DfPs of the derivations.

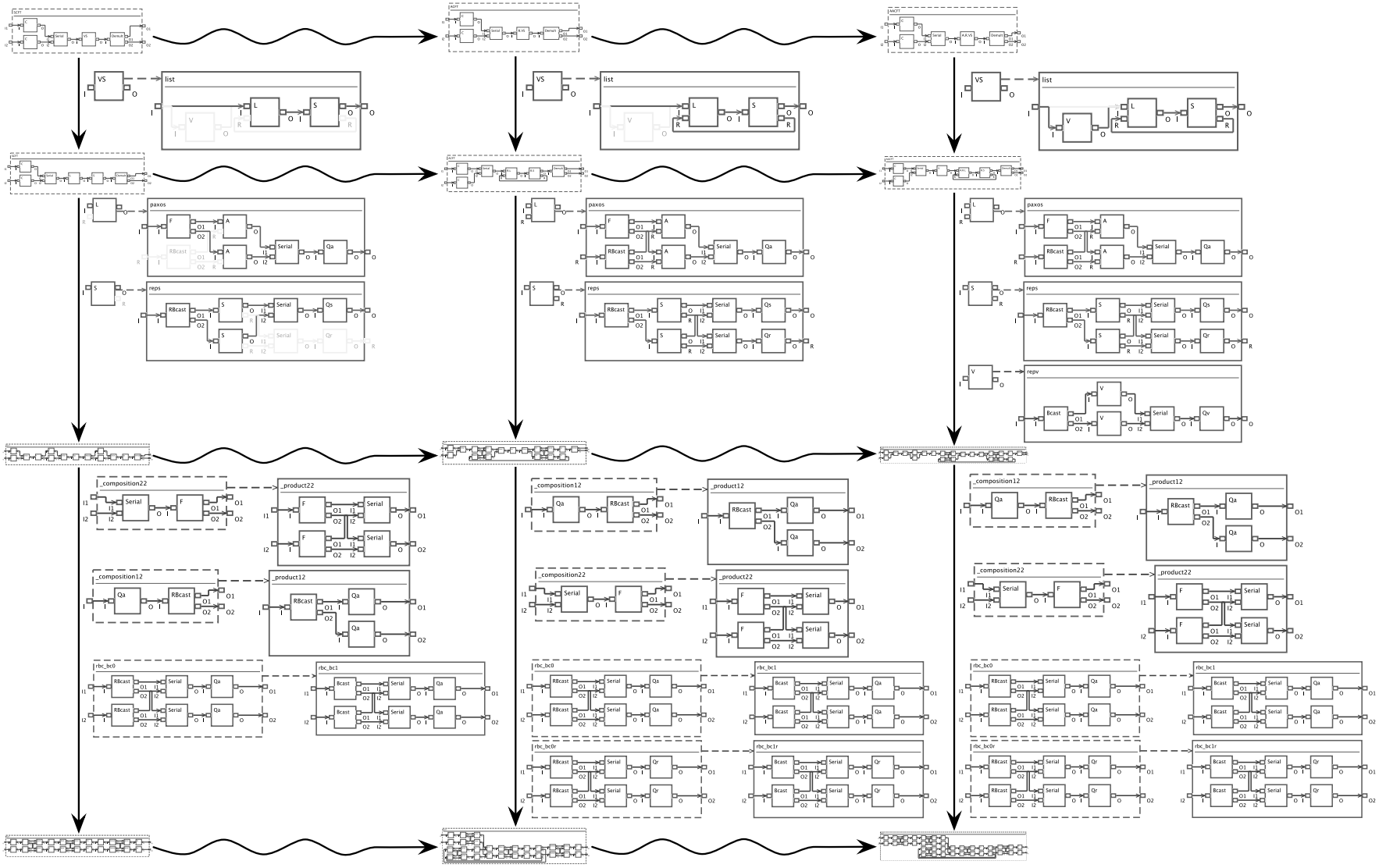

Fig. 26: The $^{\texttt{SCFT}}\searrow_{\texttt{AACFT}_\Omega}$ commuting diagram focusing the rewrite rules used.

B MolDyn Commuting Diagrams

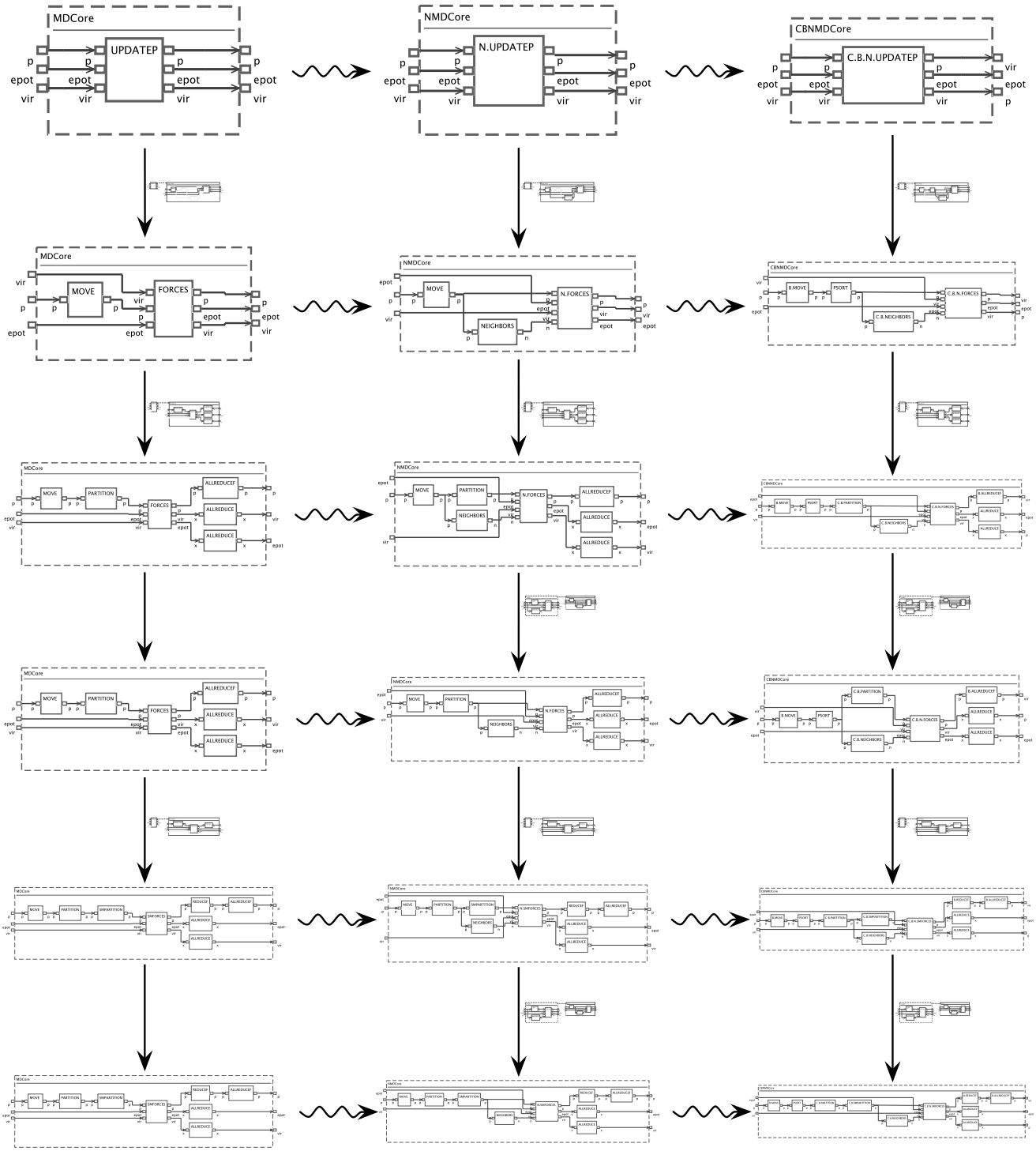

Fig. 27: The $^{\texttt{MDCore}}\!\searrow_{\texttt{CBNMDCore}_\Omega}$ commuting diagram focusing the DfPs of the derivations.

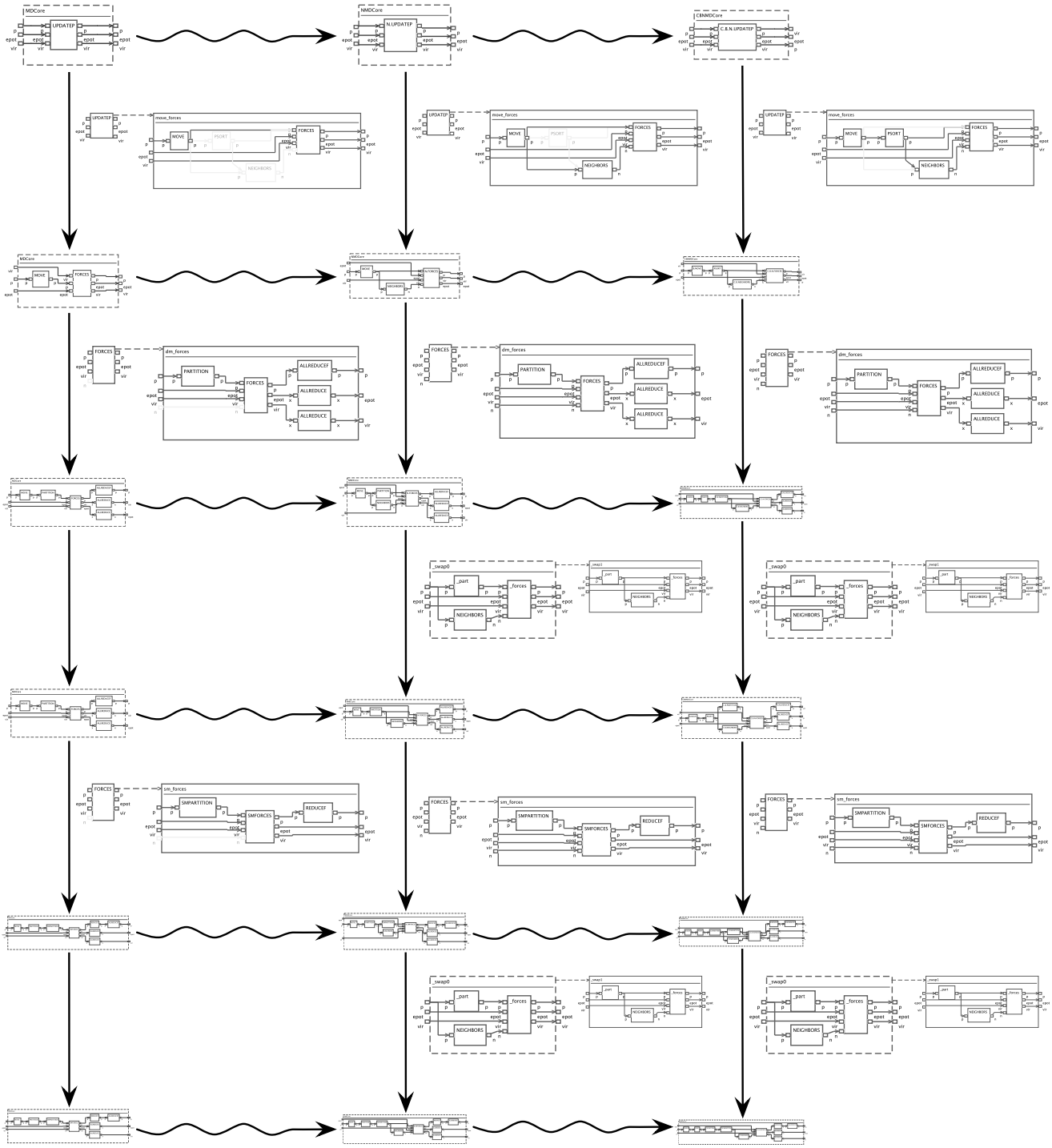

Fig. 28: The $\texttt{MDCore} \searrow_{\texttt{CBNMDCore}_\Omega}$ commuting diagram focusing the rewrite rules used.