Reengineering Product Lines of Dataflow Programs

Rui C. Gonçalves, João L. Sobral
Departamento de Informática
Universidade do Minho
Braga, Portugal
{rgoncalves,jls}@di.uminho.pt

Don Batory
Department of Computer Science
The University of Texas at Austin
Austin, TX, USA
batory@cs.utexas.edu

Taylor L. Riché
National Instruments
Austin, TX, USA
taylor.riche@ni.com

***Abstract*—*Dataflow programs (DfPs)* are widely used in computing. They are complex graphs where nodes are computations and edges indicate the flow of data. We reverse engineered a legacy D*f*P by deriving its graph from an elementary graph using domain-specific transformations. (In MDE-speak, our derivations are PIM to PSM mappings). In this paper, we explain how our tool ReFlO (a) implements transformations, (b) expresses a PIM to PSM mapping as a sequence of transformations, (c) encodes product lines RDMs using extensions, and (d) generalizes derivations of a D*f*P to a derivation of feature-extended D*f*P.**

I. Introduction

Dataflow programs (DfPs) abound in today's world. They are crash-fault tolerant file servers [1], [2], parallel relational query processors [3], [2], dense linear algebra kernels [4], [2], virtual instruments [5], [6], [7], and programs of stream processing languages [8]. A D*f*P is a directed graph; nodes are components or computations. Edges that flow into a node are node inputs; edges leaving a node are node outputs.

We confronted a common reverse engineering problem: given a legacy D*f*P, how can we explain both its dataflow graph and functionality? We found that a derivational approach to D*f*P development offers a realistic solution. We start with a simple D*f*P that is an abstract specification of a dataflow application or *Platform Independent Model (PIM)* [9]. We then apply a series of graph transformations (a.k.a. refinements and optimizations) to incrementally *derive* its concrete D*f*P or *Platform Specific Model (PSM)* which is executable. Each transformation encodes expert-known and expert-proven identities of that domain [10]. Thus, our approach to D*f*P development is correct-by-construction.

A *feature* is an increment in program functionality [11]. A *software product line (SPL)* is a family of related programs, where each family member is identified with a unique set of features [12]. Classical program development starts with a simple specification (A_1) that is progressively extended with desired features/functionality to produce more complex—but still abstract—designs ($A_1 \rightarrow A_2 \rightarrow A_3 \rightarrow A_4$ or more compactly $A_1 \mapsto A_4$). Then, the elaborated spec is incrementally mapped to an implementation ($A_4 \mapsto D_4$) using transformations as indicated by the bold arrows of Figure 1.

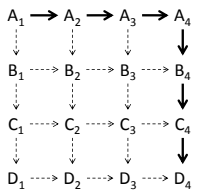

Fig. 1: Derivation paths.

When a derivational approach to D*f*P construction is coupled with product lines, a commuting diagram is formed (Figure 1) [13]. The nodes of this diagram represent the D*f*P at different stages or levels of abstraction. The further right in the diagram, the more features the program has. The further down, the more implementation details have been exposed. The upper-left corner (A_1) is the most abstract design of the D*f*P; the lower-right (D_4) is the most concrete and detailed design. *We obtain these designs and rewrite rules through a careful reverse engineering of the application and its domain.*

We showed in [13] that classical program development is impractical for reverse engineering legacy D*f*P applications. The fully-featured specification (A_4) is extraordinarily complex, making it almost impossible to extract the transformations needed to map it to its implementation ($A_4 \mapsto D_4$). We proposed a solution based on the commuting diagram of Figure 1. Given a mapping of a simple spec to its implementation ($A_1 \mapsto D_1$), we explained how each design and transformation in $A_1 \mapsto D_1$ is extended by the first feature to ($A_2 \mapsto D_2$), and then the next feature to ($A_3 \mapsto D_3$), and so on until the fully-featured mapping ($A_4 \mapsto D_4$) is derived. Doing so, the complex transformations that are needed to map the fully-featured spec to its implementation are constructed incrementally, simplifying the task of reverse engineering D*f*Ps.

In this paper, we explain how the above task is supported by tools. ReFlO is a framework for encoding domain knowledge as transformations (refinements and optimizations) and deriving the PSM of a D*f*P from a PIM using those transformations [14], [15]. *Moreover, we explain how the encoded knowledge used in the derivations of DfPs and extended DfPs is used to extract a product line of PSMs.* The contributions of our paper is the ReFlO tool and how it:

- expresses domain-specific design knowledge as rewrite rules. The collection of all such rules is a ReFlO *domain model (RDM)*,
- illustrates PIM↦PSM mappings using rules of an RDM,
- supports the process of incrementally reverse engineering D*f*Ps by enhancing and annotating an RDM,
- encodes a product line of RDMs using extensions, and
- generalizes a PIM↦PSM mapping using one RDM to a PIM↦PSM mapping of a feature-extended RDM.

We illustrate all of these ideas by a non-trivial case study,

which serves as our evaluation of ReFlO.

II. Basic Concepts

We use a standard pipe-and-filter notation to draw D*f*Ps [16]. Nodes or *boxes* are components with input and output ports. Input ports are drawn as nubs on the left-side of boxes; output ports are nubs on the right. A connector links an input port to an output port. Figure 2 shows a D*f*P drawn by our ReFlO tool [15] that models a program, called Server, that projects (eliminates) attributes of the tuples of its input stream and then sorts them, before sending them to a webserver. The webserver outputs results to clients (which are exterior to the Server interface/abstraction).

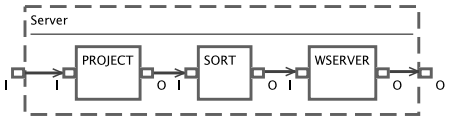

Fig. 2: The Server D*f*P.

Figure 2 is a PIM as it makes no reference to—or demands on—its concrete implementation. It is a high-level specification that can be mapped to a particular platform or for particular inputs. We call boxes PROJECT, SORT, and WSERVER *interfaces* as they specify only abstract behavior (their inputs and outputs, and, informally, their semantics).

We use transformations to map interfaces to *primitive* boxes, that represent concrete code implementations, or to *algorithms*, that express how an interface can be implemented as a D*f*P that references other interfaces. Figure 3 is an algorithm. It shows a D*f*P called parallel_sort of a map-reduce implementation of the SORT box in Figure 2. That is, the input stream is split into 2 (or more generally any number of) substreams, the substreams are sorted and then merged in sorted order.

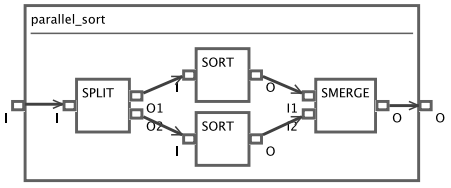

Fig. 3: parallel_sort implements SORT by map-reduce.

A. Refinements

A *refinement* [17] is the replacement of an interface with one of its implementations (primitive or algorithm). Refining Figure 2 by replacing SORT with parallel_sort and PROJECT with a map-reduce algorithm, we obtain the D*f*P of Figure 4.

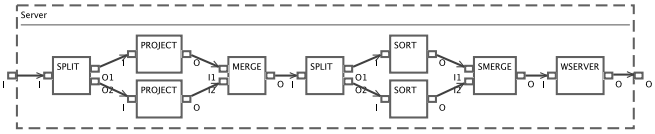

Fig. 4: Parallel version of Server.

B. Optimizations

Refinements alone are insufficient to derive efficient D*f*Ps. In Figure 4 we have a MERGE followed by a SPLIT, *i.e.* two streams are merged and the resulting stream is immediately split again. Let interface IMERGESPLIT be the box that receives two input streams, and produces two other streams containing their input tuples (see Figure 5). ms_mergesplit is one of its implementations. However, the ms_identity algorithm is a more efficient alternative that requires no computations at all. The reason is that SPLIT is the (right) inverse of MERGE.

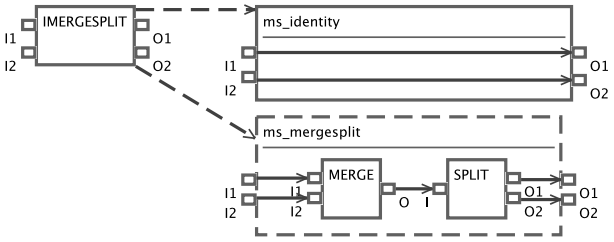

Fig. 5: Alternative implementations for IMERGESPLIT.

Figure 5 tells us how Figure 4 can be optimized. We *abstract* the ms_mergesplit composition as the IMERGESPLIT interface, obtaining Figure 6a. Then, we refine IMERGESPLIT with its ms_identity algorithm, to obtain the optimized Server (Figure 6b). An *optimization* is the action of abstracting an (inefficient) composition of boxes to an interface and then refining that interface to an alternative implementation.

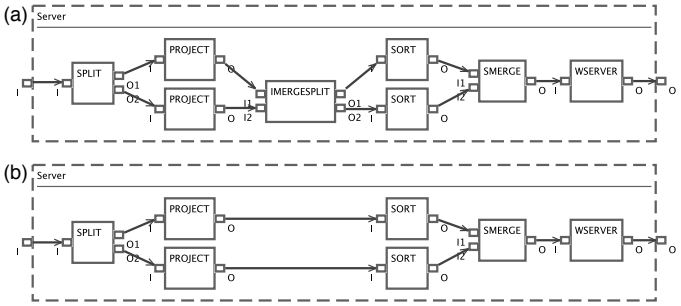

Fig. 6: Server optimization.

C. Domain Models

All of our graph rewrite rules pair an interface with an implementation. The optimization of Figure 5 is encoded as two rules defined by dashed arrows: IMERGESPLIT → ms_identity and IMERGESPLIT → ms_mergesplit. We call the set of all such rules a ReFlO *Domain Model (RDM)*.

> **Note:** There are preconditions for replacing an interface with an implementation; there are also preconditions for replacing an implementation with its interface [14]. We elide these details in our discussions, but indeed such preconditions are present.

In general, there can be many PIMs in a domain. *An RDM is the set of rules that can be used to map PIMs of a domain to their PSMs*. In the following sections, we illustrate two PIM↦PSM mappings using two different, but related PIMs.

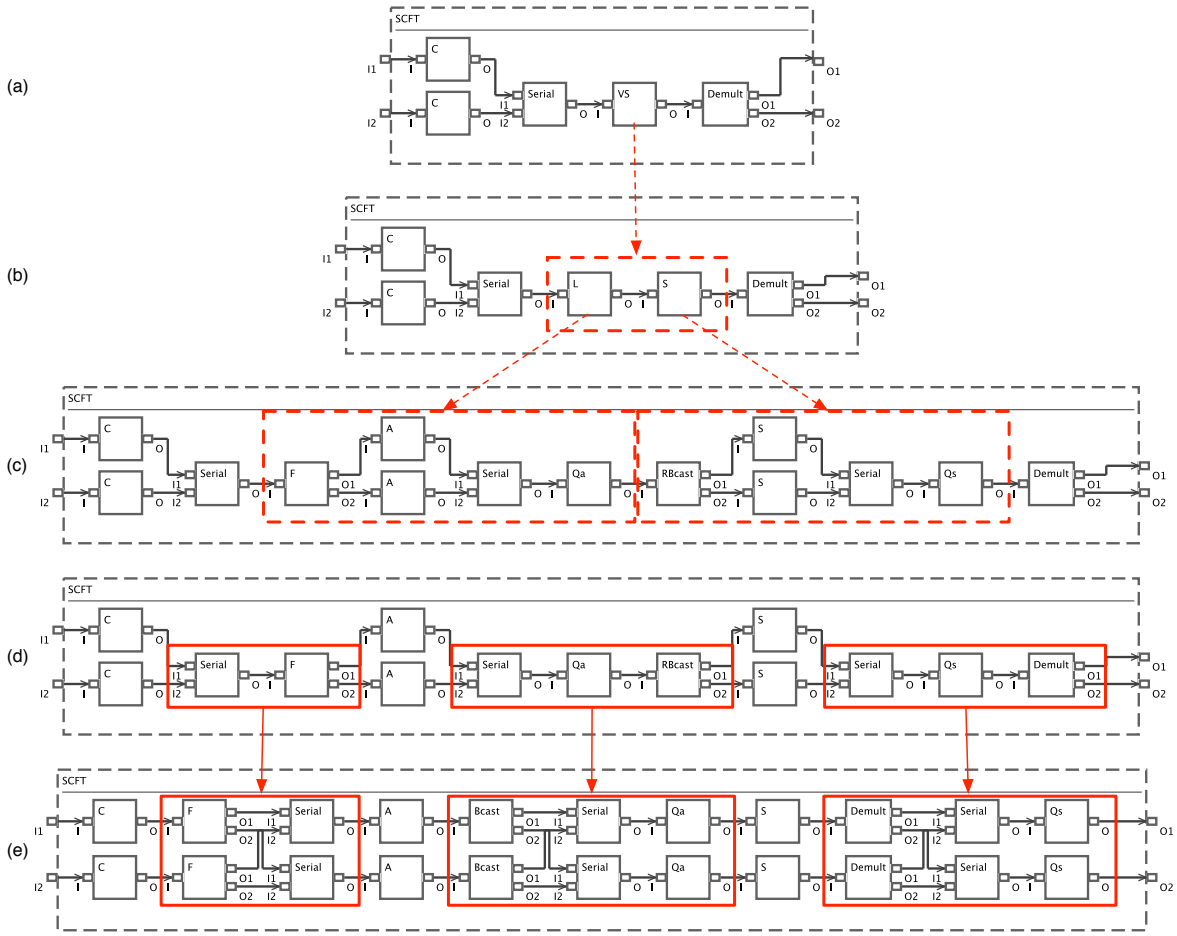

Fig. 7: SCFT PIM↦PSM derivation.

III. Illustrations

Upright is the most sophisticated *Byzantine Crash-Fault Tolerant (BCFT)* server that has been built to date [1]. We describe Upright as a sequence of progressively more complex server designs `SCFT` to `ACFT` to `AACFT` ... `BCFT`, ultimately terminating with its `BCFT` design.

In this section, we present two different derivations of Upright: one defining its *Synchronous Crash-Fault Tolerant (SCFT)* design and the second its *Asynchronous Crash-Fault Tolerant (ACFT)* design.

A. Example 1: An SCFT Derivation

The simplest design of Upright encodes a stateful SCFT server that processes requests from multiple clients. Figure 7 shows a digitally-enlargeable derivation of Upright's PIM↦PSM SCFT mapping. (Note: Upright is highly sophisticated, as indicated by its graphs. Such graphs must be magnified beyond a standard page size to be appreciated). One of us (Riché) was on an Upright implementation team, and validated our derivation by implementing it, to reproduce Upright's SCFT server.

The PIM is formed by two clients (the `C` boxes), a serializer (`Serial`) that serializes the requests from the different clients, a server (`VS`), and a demultiplexer (`Demult`) that redirects responses to the appropriate client. (Note: for simplicity we draw only two clients; there can be any number of clients).

The goal is to map the SCFT PIM to a PSM that has no *single points of failure (SPoF)*, *i.e.* a box that if it stopped processing messages would make the entire server abstraction to stop/fail. The boxes `Serial`, `VS`, and `Demult` in Figure 7a are SPoFs.

For exposition reasons, we do not explain the semantics of these rewrites or their justification—interested readers should consult [13]. We deliberately limit our discussions to syntactic rewrites (a) not to overwhelm readers with domain-specific semantics that few people on earth understand, and (b) to demonstrate the key contributions of this paper, namely how to derive D*f*Ps from domain-specific rewrite rules and how to encode product lines of D*f*Ps in a general way.

Briefly, the derivation starts with the transition from Figure 7a to Figure 7b, that refines `VS`, exposing a network queue (`L`) in front of the server (`S`). Next, the transition from Figure 7b to Figure 7c map-reduces both `L` and `S` [18]. Figure 7d is a copy of Figure 7c, and shows the subgraphs of the D*f*P that are to be optimized to eliminate SPoFs. The optimizations are performed in the transition from Figure 7d

to Figure 7e, which has no SPoFs. The D*f*P of Figure 7e is Upright's SCFT PSM for the PIM of Figure 7a.

B. Example 2: An ACFT Derivation

Upright's SCFT design tolerates a modicum of (permanent) box failures, eventually resulting in the failure of the entire system. Resilience to failures can be improved by adding recovery capabilities—the ability of a box to recover, much like a database system recovers after a crash—so that the system can recover from occasional network asynchrony. We now review the PIM↦PSM derivation of Upright's *Asynchronous Crash-Fault Tolerant (ACFT)* server [1], an enhanced variant of Upright's SCFT server PIM [13]. The goal is again to map its PIM to a PSM that has no SPoFs.

The PIM of Upright's ACFT is shown in the digitally-enlargable Figure 8a. Box `R.VS` denotes a recoverable server. Briefly the PIM↦PSM mapping is as follows: The first transition (Figure 8a to Figure 8b) exposes a recoverable network queue (`R.L`) in front of a recoverable server (`R.S`). A feedback loop connects `R.S` to `R.L` to provide recovery capabilities. The transition from Figure 8b to Figure 8c performs a map-reduce of both `R.L` and `R.S` (the refinements used are similar to the ones from Section III-A, but here additional boxes, ports, and connectors are needed to handle recovery data).

We again reach a point in the system's design where optimizations are needed to remove SPoFs. Replaying the optimizations already shown in Section III-A removes the SPoF enclosed in the red boxes in Figure 8d. These optimizations are not enough. To complete the ACFT PSM, we need to apply additional optimizations to the blue box. Doing so, the remaining SPoFs are eliminated, completing the mapping of Figure 8d to Figure 8e. The D*f*P of Figure 8e is a PSM for the PIM of Figure 8a.

C. Recap

We have just shown two different PIM↦PSM mappings: one for Upright's SCFT and another for Upright's ACFT. Both have a lot in common: their derivations are similar but not identical; their rewrites are similar but not identical. The same holds for more complete designs of Upright (see Section V-A).

With these examples under our belt, our next task to explain how `ReFlO` expresses relationships among these different but related designs and derivations, and how it avoids duplication when specifying the transformations used in a derivation.

IV. Product Line Concepts

A *feature* is an increment in program functionality; it extends a design that does not have a given functionality to a design that does. Features and extensions are well-known. In Z [19], simple specifications are progressively extended to more complex specifications. Here we follow a similar approach: we want to show how the SCFT PIM is extended to the ACFT PIM, *and in addition* show how all of the rewrite rules used in the SCFT PIM↦PSM derivation are extended to their counterparts in the ACFT PIM↦PSM derivation. To do so, we first define an extension to a D*f*P.

An *extension* of a D*f*P adds new ports and functionality to existing boxes and adds new boxes and connectors. Whereas refinements and optimizations preserve the semantics of boxes, extensions enhance semantics (*e.g.* adding new ports to existing boxes). Extensions allow developers to add new features to boxes by making explicit the relation between the extended and the base box. Features—increments in functionality—are fundamental to *software product lines (SPLs)*. We will return to SPLs shortly.

To illustrate, suppose we want our `Server` from Section II to have an additional feature `Key` that allows it to change the sort key attribute at run-time. Figure 9 shows `K.Server`, the `Key`-extended PIM of `Server`, where an extended webserver (`K.WSERVER`) may now change the attribute that is used to sort the stream by the (extended) sort operation (`K.SORT`). Interface `WSERVER` is extended with a new output port and has now the ability to ask for stream sorting according to a specific attribute. Interface `SORT` is extended with a new input port and has now the ability to change the attribute it uses to sort the input stream. A new connector links the `K.WSERVER` back to `K.SORT` to provide this extra information.

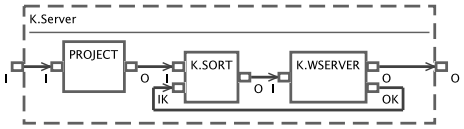

Fig. 9: The D*f*P `K.Server`.

If an interface is extended, so too must all of its implementations be extended. Figure 10 shows our design for a `Key`-extension of `parallel_sort` that adds new input ports and connectors.

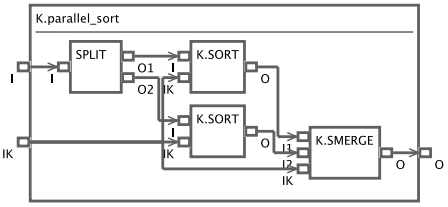

Fig. 10: Extended `K.parallel_sort` algorithm.

By replaying the derivation using the extended transformations, we obtain the extended implementation of `K.Server` (Figure 11).

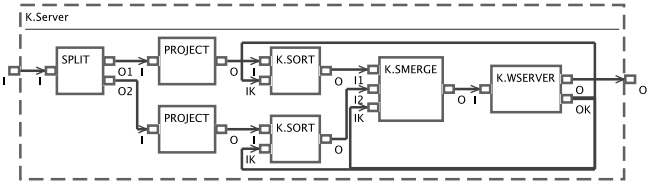

Fig. 11: The optimized D*f*P `K.Server`.

Here is where product lines arise: there can be many `Server` extensions. We could add feature `List` that enables the server

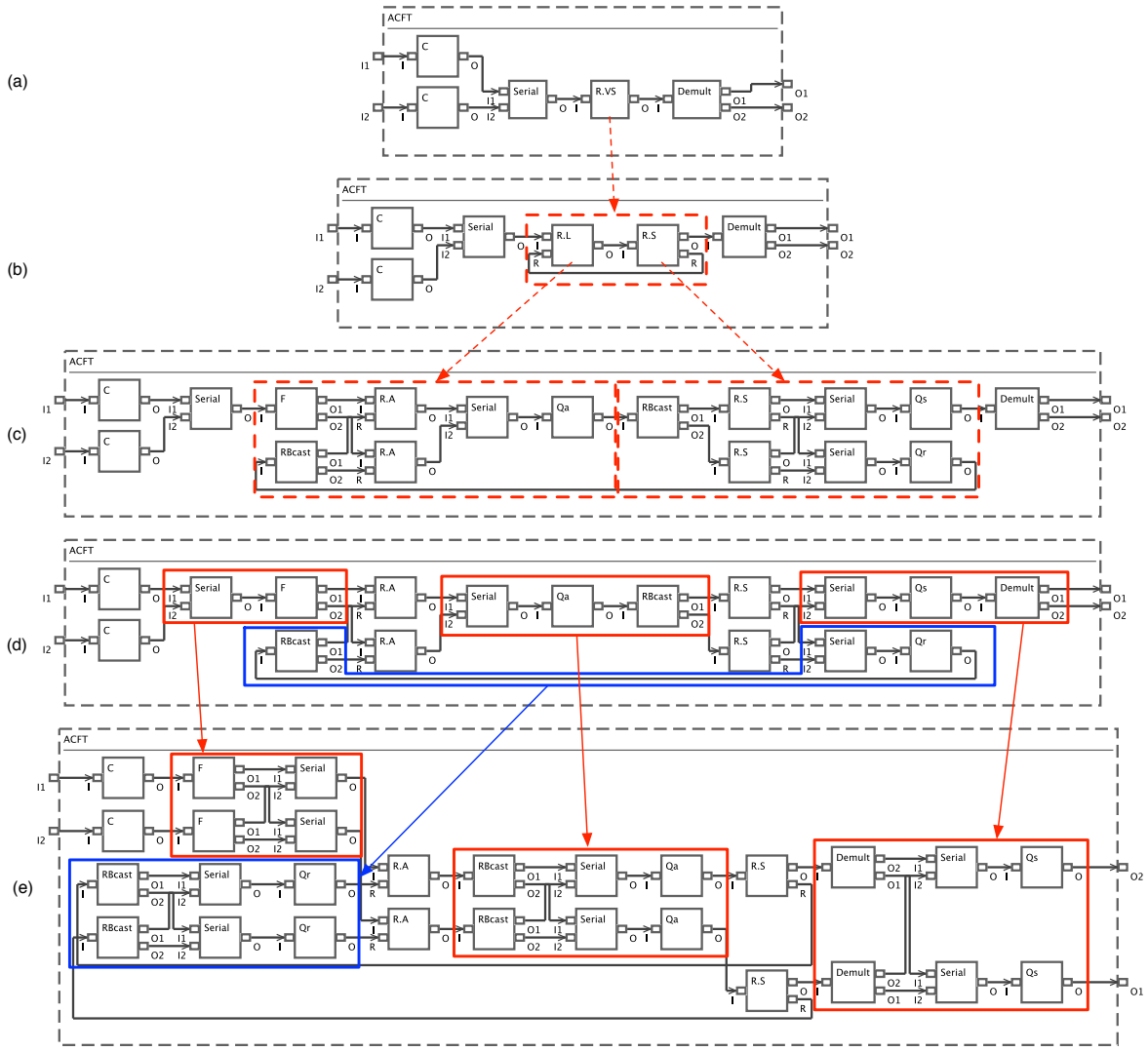

Fig. 8: ACFT PIM↦PSM derivation.

to change the list of attributes of the stream that are projected. Figure 12 shows the D*f*P `L.K.Server` that extends `K.Server` with `List`. The `L.K.WSERVER` is extended again with a new output port. The same happens to the `L.PROJECT` box, that now also receives a list of attributes to project.

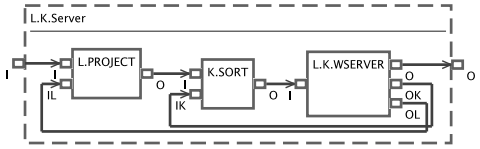

Fig. 12: The D*f*P `L.K.Server`.

After extending `K.parallel_sort` by `List` and the parallel implementation of `L.PROJECT`, we can again replay the derivation to obtain an implementation for the new extended specification.

Readers may have observed the following:

- `Server`, `K.Server`, and `L.K.Server` are members of a small product line of servers. This product line has two optional features `Key` and `List`, which means that another member of this product line is `L.Server`—a server with the ability to change the attributes that are projected.
- The PIMs of SCFT and ACFT are related by extension—a `R` (recovery) feature. That is, `ACFT` = `R.SCFT`.
- Not only can interfaces and algorithms be extended, so too can RDM rules. We saw examples above, such as `K.(SORT` → `parallel_sort)` = `K.SORT` → `K.parallel_sort`. That is, extensions of interfaces and algorithms induce extensions of their rewrite rules.

Henceforth we write `e` ⇝ `K.e` to mean element `e` is extended to element `K.e`, where an element can be an interface, primitive, algorithm, rewrite rule, or RDM.

A. *Example: Extending SCFT Rules with Recovery*

Illustrating SCFT rewrite rule extensions to their ACFT counterparts is the topic of this section. We illustrate how we obtain the rewrite rules used to derive ACFT PSM from the

rewrite rules previously used to derive SCFT PSM, and note other changes that need to be made to map the RDM for SCFT to the RDM for ACFT.

The SCFT PIM $\rightsquigarrow$ ACFT PIM is the transition from Figure 7a to Figure 8a. The only change is `VS` $\rightsquigarrow$ `R.VS`, that is, a server is extended to a recoverable server.

> **Note:** Applying an extension `E` to a D*f*P α involves applying `E` to each element `e` $\in \alpha$ (connectors, ports, and boxes) to produce an extended element `E.e`. In many cases, `E` has no effect, meaning there is a fixed point `e` = `E.e`. In the SCFT PIM $\rightsquigarrow$ ACFT PIM extension, only the server `VS` was effected by feature `R`. All other ports, connectors, and boxes of Figure 7a are unchanged. Readers may recognize additional fixed point mappings in the rule extensions discussed below.

Recall the rule `VS` $\rightarrow$ `list`. This rule is extended to `R.VS` $\rightarrow$ `R.list` by the indicated interface and algorithm extensions in Figure 13. That is, new ports were added to `R.L` and `R.S` and a new connector is added that connects these ports.

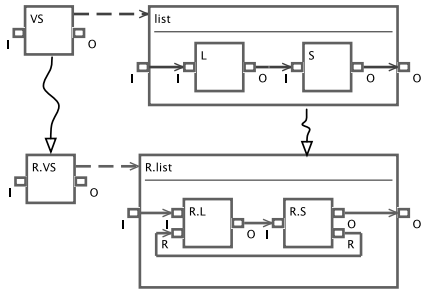

Fig. 13: Extending the `VS` $\rightarrow$ `list` rewrite rule.

As another example, the server replicating rewrite rule `S` $\rightarrow$ `reps` is extended to `R.S` $\rightarrow$ `R.reps` in Figure 14. Each server `S` becomes a recoverable server `R.S`. Further, because a server now must export a new port, new computations are needed to produce output for that port. This is manifested by the introduction of new `Serial` boxes, a new `Qr` box/interface, and additional connectors. Eventually, new refinements must be defined to map a `Qr` interface to its implementation(s). These additional rules are present in the ACFT RDM, but not in the SCFT RDM.

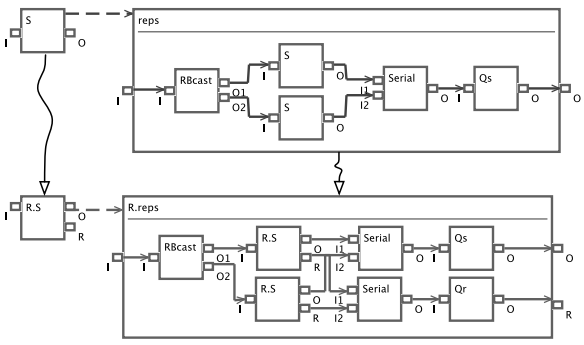

Fig. 14: Extending the `S` $\rightarrow$ `reps` rewrite rule.

SCFT derivation uses optimizations to remove existing SPoFs. The boxes involved in these optimizations are not modified by recovery, so optimization rewrites are unchanged and can be directly reapplied. There is also the need to add a new optimization, as part of the transformation from Figure 8d to Figure 8e, to eliminate SPoFs involving the boxes `Qr`, `RBcast`, `Serial` (as box `Qr` was not part of the RDM previously).

Thus, in general, experts know (and thus we know) how features extend individual boxes, how features extend individual rewrite rules, and how features require new rules to be added. This now brings us to describe the general process of how features map RDMs to extended RDMs.

B. Encoding Product Lines of RDMs

We encode a product line of RDMs using the template approach of Czarnecki [20]. All RDMs of a product line are superimposed into a single composite model, called an *eXtended Reflo Domain Model (XRDM)*, and a specific RDM (*i.e.* a specific member of the RDM product line) is obtained by projection.

To encode the product line of RDMs, rule elements are annotated with two attributes: a feature predicate and a feature tag set. A *feature predicate* determines when a box, port, or connector is part of an RDM. Given a subset of features $\mathcal{F}$, and a model element `e` with predicate `P`, `P`$(\mathcal{F})$ is true if and only if `e` is part of the RDM when $\mathcal{F}$ are the enabled features.

The *feature tag set* determines how a box is tagged. 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 `List` and `Key`).

These same ideas are used to generalize rewrite rules. Consider the `VS` $\rightarrow$ `list` rewrite rule extension shown in Figure 13. The annotated rewrite rule that encodes this extension is depicted in Figure 15. Port `R` of boxes `L` and `S` are annotated with predicate `Recovery`, *i.e.* they exist only when recovery is enabled. Boxes `VS`, `list`, `L`, and `S` have tag set {`R`} (*i.e.* its semantics depends on the presence of recovery features). When projecting this rewrite rule with only the base feature enabled, `ReFlO` produces the rewrite rule depicted in Figure 16, where the model elements associated with feature `Recovery` are hidden.

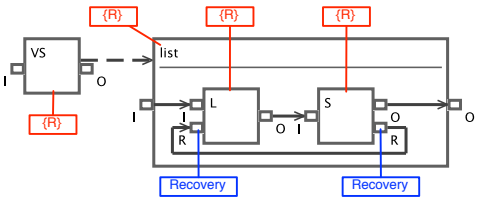

Fig. 15: `VS` $\rightarrow$ `list` rewrite rule.

C. Safe Composition

D*f*Ps must satisfy constraints in order to be valid (*e.g.* each active input port must have one and only one active incoming connector). `ReFlO` validates an RDM by testing the constraints

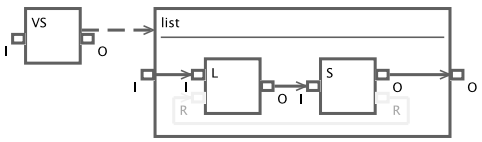

Fig. 16: Projected `list` algorithm.

of ReFlO's RDM meta-model. When extensions were added, ReFlO's validation was modified to check only whether the active elements in an XRDM (those permitted by the current set of features) formed a valid RDM.

This alone is insufficient. A more important question is whether *all* possible RDMs in the product line of RDMs that is encoded by an XRDM are valid. This is particularly important, as manually annotating an XRDM can be error prone.

ReFlO uses *safe composition* to verify that all members of a product line are valid [21], [22]. The constraints checked by ReFlO are:

- An interface must be defined if it is referenced by an algorithm.
- An interface that is referenced by an algorithm must have the same ports as the definition of that interface.
- An algorithm or primitive must have the same ports as the interface it implements.
- The input ports of interfaces that are referenced by an algorithm must have precisely one active incoming connector.
- The output ports of an algorithm must have precisely one active incoming connector.

ReFlO warns users if there is a combination of features that produces an invalid RDM, providing information about the type of error detected.

In addition, ReFlO also detects bad smells, *i.e.* situations that do not invalidate an RDM, but are uncommon and likely to be incorrect. The two cases we detect are:

- The input of an algorithm is not used.
- The output of an interface reference in an algorithm is unused.

Whenever ReFlO detects a bad smell, the developer is warned, so that (s)he can further check if the XRDM is correct.

D. Replaying Derivations

ReFlO keeps track of the transformations used in a PIM↦PSM derivation. In this way, after building a derivation for the base D*f*P, developers can specify a new D*f*P (with additional features) and ask ReFlO to replay the derivation.

ReFlO tries to reapply the same sequence of transformation (or rather their extended counterparts), to the extended PIM. As we have already seen, new transformations may be needed. In this case, the developer has to manually finish the derivation. We now illustrate these ideas with another example from Upright.

V. Illustration

A. Adding Authentication to ACFT

We saw earlier how the recovery feature mapped Upright's SCFT design, its derivation and rewrites to Upright's ACFT design, its derivation and rewrites. We abbreviate this complex mapping as SCFT ⇝ ACFT. Upright has other features in its complete design, e.g. SCFT ⇝ ACFT ⇝ AACFT ⇝ ... ⇝ BCFT. In this section we show how the ACFT server can be extended with another feature, *authentication*, which is a next stage in Upright's design.

Briefly, Figure 17a is the PIM of AACFT. Box A.R.VS represents a recoverable server with authentication. The transition from Figure 17a to Figure 17b exposes an authenticated recoverable network queue (A.R.L) before the recoverable server (R.S), which processes authenticated messages. Authentication adds a new box (V) before the network queue, that filters invalid requests (according to authentication requirements). As in the ACFT derivation, the process follows with the transition from Figure 17b to Figure 17c, that performs a map-reduce of A.R.L, R.S, and V.

SPoFs now need to be removed. In the transition from Figure 17d to Figure 17e, we replay the optimizations used in the ACFT derivation. Note that the sequential composition of boxes Serial and F is no longer present (left red group of boxes from Figure 8d). Instead, new optimizations are required to remove the SPoF enclosed by the green boxes in Figure 17d, and produce Figure 17e. The D*f*P of Figure 17e is Upright's PSM for the PIM of Figure 17a.

B. The XRDM

The PIM↦PSM base derivation of AACFT is obtained using three refinements, specified by the rewrite rules VS → list, L → paxos and S → reps, that are feature-extended.

The annotated VS → list rewrite rule is presented in Figure 18a. It contains the annotations already presented in Figure 15. Additionally, box V was added and it is annotated with predicate Authentication, *i.e.* it is only present when authentication is enabled. The connector linking input I of list and input I of L, when authentication is enabled, is replaced by box V (and its connectors), therefore it is annotated with predicate not Authentication (*i.e.* it is *not* present when authentication is enabled). Boxes VS, list, and L have the additional A (*i.e.* its semantics also depends on the presence of authentication features). Figure 18b and Figure 18c show how rewrite rules L → paxos and S → reps are annotated to define recovery and authentication features.

The new rewrite rules, added to specify implementations for new boxes that are introduced and to handle new opportunities for optimization that arise, are annotated with a predicate that is either Recovery or Authentication, depending on whether they are required for recovery or authentication.

C. Adding Authentication to SCFT

The Upright XRDM was incrementally created by starting with the RDM for SCFT and then adding support for recovery and then authentication. Given this set of features, there is

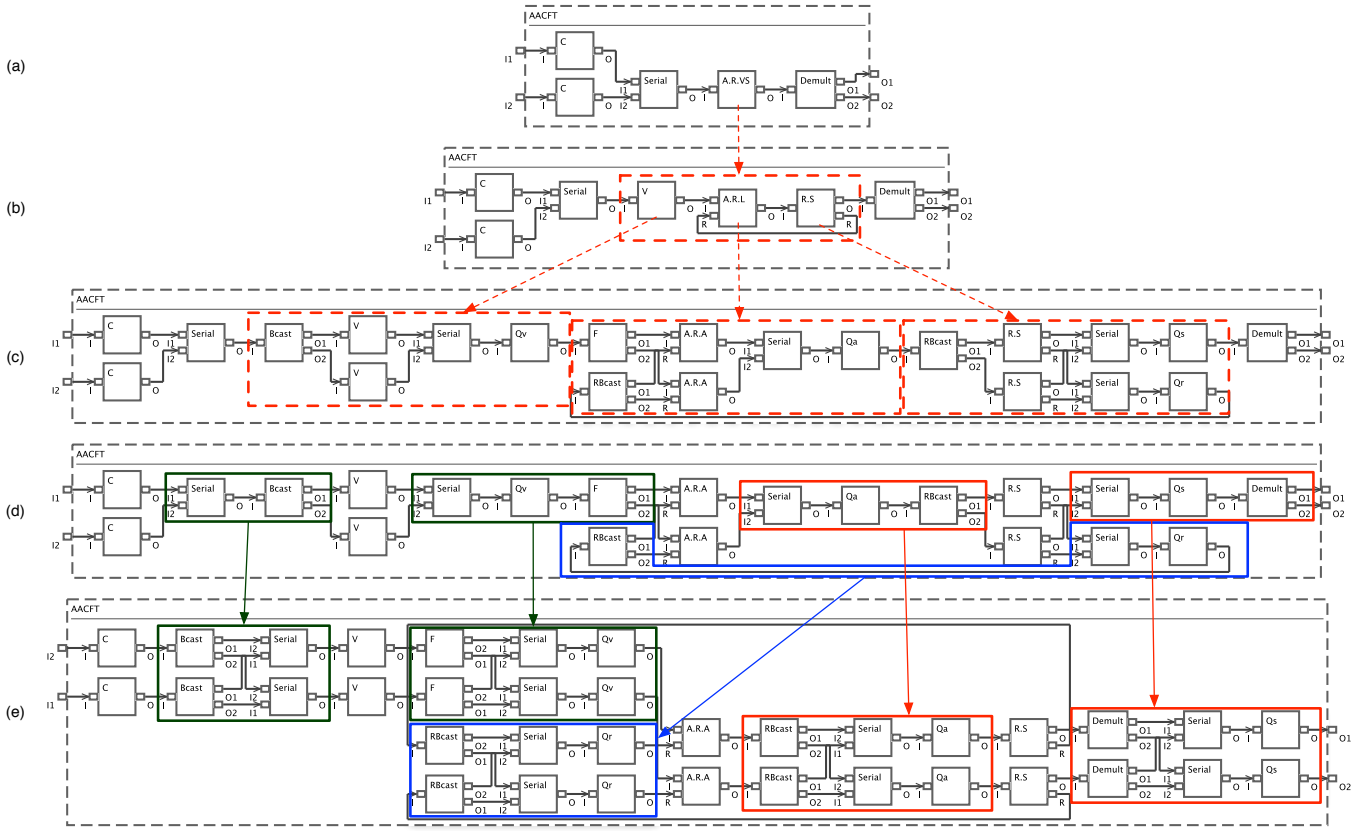

Fig. 17: AACFT PIM↦PSM derivation.

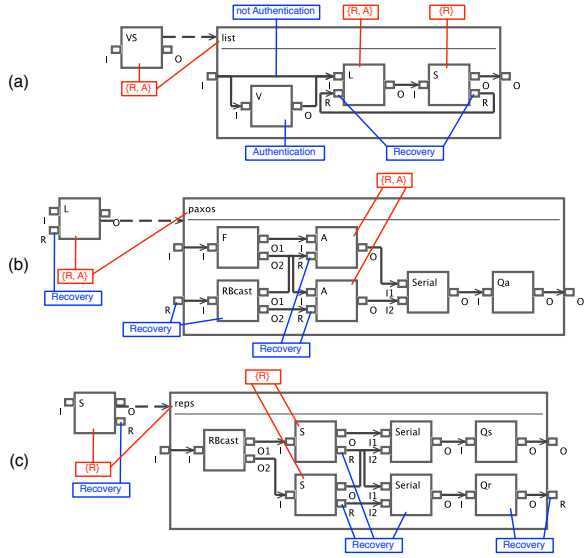

Fig. 18: Rewrite rules annotated.

another PIM↦PSM derivation we may be interested in: SCFT with authentication (ASCFT).

This derivation is obtained replaying the refinements of the derivation presented in Section V-A. In the absence of recovery, there are some boxes that are no longer present in the D*f*P. This affects the optimization step. We can again replay the optimizations used in the AACFT, but now one optimization was no longer applicable (the optimization of boxes enclosed in the blue box in Figure 17d). In this particular case, ReFlOs replay of a PIM↦PSM mapping is complete and requires no developer intervention.

VI. Recap and Perspective

Figure 19 overviews of the derivations presented in this paper. We started with a simple PIM of Upright (p_0) and showed how it was progressively refined and optimized to its synchronous crash fault PSM ($p_0 \mapsto$ SCFT). We then extended the SCFT PIM with recovery, and repeated the mapping (R.$p_0 \mapsto$ R.SCFT) to obtain Upright's asynchronous crash fault PSM (ACFT = R.SCFT). Next, we added the authentication feature and repeated the mapping (A.R.$p_0 \mapsto$ A.R.SCFT) to obtain Upright's authenticated ACFT PSM (AACFT = A.R.SCFT). We also explained a third variant of Upright, authenticated SCFT, in which authentication is added to Upright's SCFT design (A.$p_0 \mapsto$ A.SCFT).

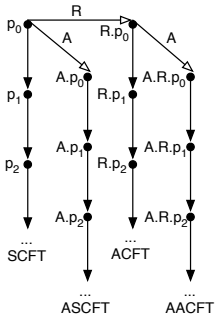

Fig. 19: Derivations presented.

This process was driven by creating the Upright XRDM (described in Section V-B). The XRDM is a library of feature-annotated rewrite rules. By specifying a particular set of

features, the XRDM can be projected (simplified) to a feature-specific version of RDM (itself a set of rewrites) from which a particular derivation of Upright can be reproduced.

> **Note:** We could have defined *higher-order (HO)* rewrites to map the rules of one RDM to another. We quickly realized that HO rewrites would ultimately be unreadable: every feature combination would have to include feature interactions, which would require separate HO rewrites by themselves.[1] When `n` features are composed, there could be up to 2^n feature interactions [23]. Although this is highly unlikely, we concluded that writing separate RDM↦RDM HO transformations for each interaction is impractical. As rewrites with different features are so similar, it is elementary to annotate superimposed rewrites with features, as described in [20], and just project out the rules that we need. If we discover an interaction is missing, it is easy to add missing pieces. In any case, we conceptually view all mappings (HO and otherwise) as transformations; we found that HO transformations are easier to implement by annotations than by explicit rewrite rules.

Encoded in the XRDM is a product line of RDMs—this includes features and feature interactions. Given a PIM↦PSM derivation using an RDM, `ReFlO` can project a derivation to reveal the derivation of a simpler design (one with fewer features) or `ReFlO` can show a partial derivation of a PIM↦PSM where additional features are added. In the latter case, a designer will need to add the missing transformations (refinements or optimizations) to complete the PIM↦PSM mapping.

VII. Related Work

Our work is an example of *Design by Transformation (DxT)* [2] uses refinements, optimizations, and extensions to incrementally map abstract D*f*P specifications to implementations. DxT is used for reverse-engineering as well as forward-engineering [4], [13].

To the best of our knowledge, this is the first time SPLs of D*f*Ps has been documented. We used an unusual combination of techniques to accomplish this. There are several ways in which features of SPLs can be implemented. Some are compositional, including AHEAD [24], FeatureHouse [25], and AOP [26], all of which work on code. Similar solutions have been proposed to handle SPLs of models [27], [28].

`ReFlO` uses an annotative approach, where a single set of artifacts containing all features/variants are superimposed, and the artifacts (*e.g.* code, model elements) are annotated with feature predicates to determine when these artifacts are visible in a SPL member. Preprocessors are a primitive example [29]. Code with preprocessor directives can be made more understandable by tools that color code [30] or that extract views [31]. More sophisticated solutions exist, such as XVCL [32], Spoon [33], Spotlight [34], or CIDE [35]. `ReFlO` differs from those solutions in that it works at a model level.

Other annotative approaches also work at the model level. In [36] an UML profile is proposed to specify model variability in UML class diagrams and sequence diagrams. Czarnecki and Antkiewicz [20] proposed a template approach, where model elements are annotated with presence conditions (similar to our feature predicates) and meta-expressions. FeatureMapper [37] allows the association of model elements (*e.g.*, classes and associations in a UML class diagram) to features. Instead of annotating D*f*P directly (usually too complex), we annotate model transformations (simpler) that are used to derive D*f*P implementations. This reduces the complexity of the annotated models, and it also makes the extensions available when deriving other implementations, making extensions more reusable.

We provide an approach to extract an SPL from legacy applications. RE-PLACE [38] is an alternative to reengineer existing systems into SPLs. Other approaches have been proposed with similar intent, employing refactoring techniques [39], [40], [41].

Extracting variants from a XRDM is similar to program slicing [42]. Slicing has been generalized to be used with models [43], [44], in order to reduce its complexity and make easier for developers to analyse models. In [45] 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.

`ReFlO` supports analyses to verify whether all variants that can be produced meet the metamodel constrains. The analysis is based on the solutions proposed by Czarnecki and Pietroszek [21] and Thaker *et al.* [22].

`ReFlO` uses a dataflow notation to model programs and transformations. Other tools specify D*f*P, such as LabVIEW [5], Simulink [6], Weaves [7], Fractal [46], or StreamIt [8]. More than a tool to specify D*f*Ps, `ReFlO` allows the specification of transformations that can be used to map a high-level D*f*P specification to an implementation with desired properties regarding efficiency or availability, for example. Moreover, with extensions `ReFlO` allows the specification of product lines of such transformations (that expresses a product line of D*f*Ps).

Graph grammars [47] provide a framework for specifying graph transformations. With extensions, `ReFlO` specifies not only graph transformations, but also how to map a set of transformations to another set of transformations that have been feature-extended.

VIII. Conclusions and Outlook

Reverse engineering a legacy D*f*P application is difficult. A key to its success is choosing a representation that can incrementally reveal a design's complexity; doing so is a step toward making D*f*P design more rigorous, analyzable, and automated. The rigor that we gain using our approach comes from expressing incremental design decisions as domain-specific rewrites rules (a.k.a. design patterns or identities) that

[1] If two features `A` and `B` *interact*, a third feature (`A#B`) indicates changes to be made to `A` or `B` or both so that they work correctly together. `A#B` is a 2-way interaction; `n`-way interactions among `n` different features are possible.

are well-known to experts. Domain-experts have proofs or demonstrations of correctness for these rewrites otherwise they would not use them [1], [4], [10]. Our approach is analyzable. We explained how ReFlO generalizes or simplifies derivations of D*f*P designs through the addition or removal of features. ReFlO can also guarantee the rewrite rules of a domain model are type correct (meaning that they conform to the constraints of rule metamodels). Our work provides a framework for automated software development in an MDE style, where model-to-text mappings can be used to generate the code for a ReFlO-specified-and-derived D*f*P.

We validated our approach with a real-world case study: Upright, a state-of-the-art crash fault tolerant server, and presented ReFlO derivations of its SCFT, ACFT, AACFT, and ASCFT designs. Other case studies are presented in [48]. We believe that ReFlO is the first of many interactive environments in which domain experts can use to reverse and forward engineer domain-specific dataflow applications.

Availability. ReFlO can be downloaded at http://cs.utexas.edu/users/schwartz/DxT/reflo/x/.

Acknowledgements. Gonçalves and Sobral are funded by project FCOMP-01-0124-FEDER-010152. Gonçalves is additionally funded by FCT grant SFRH/BD/47800/2008. We also gratefully acknowledge support for this work by NSF grants CCF 0724979 and OCI-1148125.

References

[1] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche, "Upright cluster services," in *SOSP*, 2009.

[2] T. L. Riché, D. Batory, R. C. Gonçalves, and B. Marker, "Architecture design by transformation," University of Texas at Austin, Dept. of Computer Sciences, Tech. Rep. TR-10-39, 2010.

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

[4] B. Marker, J. Poulson, D. Batory, and R. van de Geijn, "Designing linear algebra algorithms by transformation: Mechanizing the expert developer," in *iWAPT*, 2012.

[5] "The LabVIEW Environment," http://www.ni.com/labview/.

[6] "Simulink - Simulation and Model-Based Design," http://www.mathworks.com/products/simulink/.

[7] M. M. Gorlick and R. R. Razouk, "Using weaves for software construction and analysis," in *ICSE*, 1991.

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

[9] D. S. Frankel, *Model Driven Architecture: Applying MDA to Enterprise Computing*. John Wiley & Sons, Inc., 2003.

[10] D. Batory and B. Marker, "Correctness proofs of the gamma database machine architecture," University of Texas at Austin, Dept. of Computer Science, Tech. Rep. TR-11-17, 2011.

[11] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," Software Engineering Institute, Carnegie Mellon University, Tech. Rep. CMU/SEI-90-TR-021, 1990.

[12] P. C. Clements and L. M. Northrop, *Software product lines: practices and patterns*. Addison-Wesley Longman Publishing, 2001.

[13] T. L. Riché, R. C. Gonçalves, B. Marker, and D. Batory, "Pushouts in software architecture design," in *GPCE*, 2012.

[14] R. C. Gonçalves, D. Batory, and J. L. Sobral, "ReFlO: An interactive tool for pipe-and-filter domain specification and program generation," University of Texas at Austin, Dept. of Computer Science, Tech. Rep. TR-13-02, 2013.

[15] "ReFlO Framework," http://cs.utexas.edu/users/schwartz/DxT/reflo/.

[16] D. Garlan and M. Shaw, "An introduction to software architecture," Carnegie Mellon University, Tech. Rep., 1994.

[17] N. Wirth, "Program development by stepwise refinement," *Communications of the ACM*, vol. 14, no. 4, pp. 221–227, 1971.

[18] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, 1998.

[19] J. Spivey, *The Z Notation: A Reference Manual*. Prentice Hall, 1989.

[20] K. Czarnecki and M. Antkiewicz, "Mapping features to models: a template approach based on superimposed variants," in *GPCE*, 2005.

[21] K. Czarnecki and K. Pietroszek, "Verifying feature-based model templates against well-formedness OCL constraints," in *GPCE*, 2006.

[22] S. Thaker, D. Batory, D. Kitchin, and W. Cook, "Safe composition of product lines," in *GPCE*, 2007.

[23] D. Batory, P. Höfner, and J. Kim, "Feature interactions, products, and composition," in *GPCE*, 2011.

[24] D. Batory, "Feature-oriented programming and the AHEAD tool suite," in *ICSE*, 2004.

[25] S. Apel, C. Kastner, and C. Lengauer, "FEATUREHOUSE: Language-independent, automated software composition," in *ICSE*, 2009.

[26] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *ECOOP*, 1997.

[27] A. McNeile and N. Simons, "State machines as mixins," *Journal of Object Technology*, vol. 2, no. 6, pp. 85–101, 2003.

[28] C. Prehofer, "Plug-and-play composition of features and feature interactions with statechart diagrams," *Software and Systems Modeling*, vol. 3, no. 3, pp. 221–234, 2004.

[29] 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*, vol. 1, 2010.

[30] J. Feigenspan, M. Papendieck, C. Kästner, M. Frisch, and R. Dachselt, "Featurecommander: Colorful #ifdef world," in *SPLC*, 2011.

[31] N. Singh, C. Gibbs, and Y. Coady, "C-CLR: a tool for navigating highly configurable system software," in *ACP4IS*, 2007.

[32] S. Jarzabek, P. Bassett, H. Zhang, and W. Zhang, "XVCL: XML-based variant configuration language," in *ICSE*, 2003.

[33] R. Pawlak, "Spoon: Compile-time annotation processing for middleware," *IEEE Distributed Systems Online*, vol. 7, no. 11, 2006.

[34] D. Coppit, R. R. Painter, and M. Revelle, "Spotlight: A prototype tool for software plans," in *ICSE*, 2007.

[35] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in software product lines," in *ICSE*, 2008.

[36] T. Ziadi, L. Hlout, and J. Jzquel, "Towards a UML profile for software product lines," in *In PFE*. Springer, 2003.

[37] F. Heidenreich, J. Kopcsek, and C. Wende, "Featuremapper: mapping features to models," in *ICSE Companion*, 2008.

[38] J. Bayer, J.-F. Girard, M. Würthner, J.-M. DeBaud, and M. Apel, "Transitioning legacy assets to a product line architecture," *SIGSOFT Softw. Eng. Notes*, vol. 24, no. 6, pp. 446–463, 1999.

[39] 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*, 2005.

[40] J. Liu, D. Batory, and C. Lengauer, "Feature oriented refactoring of legacy applications," in *ICSE*, 2006.

[41] S. Trujillo, D. Batory, and O. Diaz, "Feature refactoring a multi-representation program into a product line," in *GPCE*, 2006.

[42] M. Weiser, "Program slicing," in *ICSE*, 1981.

[43] H. Kagdi, J. I. Maletic, and A. Sutton, "Context-free slicing of UML class models," in *ICSM*, 2005.

[44] J. H. Bae, K. Lee, and H. S. Chae, "Modularization of the UML metamodel using model slicing," in *ITNG*, 2008.

[45] A. Wasowski, "Automatic generation of program families by model restrictions," in *SPLC*, 2004.

[46] E. Bruneton, T. Coupaye, and J. Stefani, "The Fractal Component Model," http://fractal.ow2.org, 2004.

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

[48] R. C. Gonçalves, "Parallel Programming by Transformation," Ph.D. dissertation, Departamento de Informática, Universidade do Minho, (To appear).