Product lines of dataflows

Michael Lienhardt[a,*], Maurice H. ter Beek[b], Ferruccio Damiani[c]

[a] *ONERA, Palaiseau, France*
[b] *ISTI–CNR, Pisa, Italy*
[c] *University of Turin, Turin, Italy*

Abstract

Data-centric parallel programming models such as *dataflows* are well established to implement complex concurrent software. However, in a context of a configurable software, the dataflow used in its computation might vary with respect to the selected options: this happens in particular in fields such as Computational Fluid Dynamics (CFD), where the shape of the domain in which the fluid flows and the equations used to simulate the flow are all options configuring the dataflow to execute.

In this paper, we present an approach to implement product lines of dataflows, based on Delta-Oriented Programming (DOP) and term rewriting. This approach includes several analyses to check that all dataflows of a product line can be generated. Moreover, we discuss a prototype implementation of the approach and demonstrate its feasibility in practice.

1. Introduction

Over the past decades, with the end of Moore's law and the multiplication of parallel architectures such as multi-core CPUs and GPUs, many data-centric programming paradigms were developed in order to continue having always more efficient programs with such new hardware. This trend is clearly visible in HPC where many data-centric languages and libraries have been developed, such as Chapel [1], StarPU [2], HPX [3], Charm++ [4], Legion [5] and DAPP [6, 7]. The core model of data-centric computation is the *dataflow* [8, 9] which can be expressed as an acyclic directed graph stating how data is generated and used by side-effect-free tasks.

While dataflows can efficiently be deployed on parallel and heterogeneous architectures, their structure is very static with no conditional nor loops. Libraries like HPX [3] or Legion [5] alleviate this restriction by extending the

*Corresponding author

Email addresses: `michael.lienhardt@onera.fr` (Michael Lienhardt), `maurice.terbeek@isti.cnr.it` (Maurice H. ter Beek), `ferruccio.damiani@unito.it` (Ferruccio Damiani)

model with conditional and runtime tasks creations, at the cost of a less efficient computation model. Moreover such extensions are not well-suited to large configuration spaces that occur in industrial tools like *elsA* [10] and *Fun3D* [11].

elsA is a tool that implements *Computational Fluid Dynamics* (CFD), i.e., it simulates the flow of fluids in a given input mesh and outputs information of interest to the user (e.g., the pressure that a material pushed by the fluid must be able to sustain, or some modification of its shape that would make the fluid flow more efficiently). The principle of *elsA*'s computation is a fixpoint loop: it executes the same code in loop until the computed flow is close enough to what would happen in reality. And since the loop's code could be executed millions of time, expressing it as a dataflow would greatly improve the efficiency of *elsA*.

However, *elsA* has an infinite configuration space – structured into three parts – that has a huge impact on the shape of the executed dataflow. The first part of the configuration space consists of about 2000 options that configure which fluid flow computation to perform. Indeed, fluid flow is given by the Navier–Stokes equations that do not have an analytic solutions, and so hundreds of approximations of these equations have been defined, with various precision and stability characteristics: it is up to the user to decide which approximation s/he wants to use. The second part of the configuration space is the output information provided to the user: virtually any data could interest her/him since it depends on which phenomena s/he's studying. So s/he must provide the list of these data to *elsA* which in turn must compute them by extending its dataflow. The last part of the configuration space is the shape of the input mesh itself. Meshes are usually structured in various *zones* (modelling the domain in which the fluids flow) and *boundaries* (modelling walls of different materials, fluid injection or extraction, or even infinite domains): fluid flow simulation must be performed on every zone of the mesh, and specific computation must be performed on each boundary depending on its type (e.g., the effect of a wall on a flow is different from the effect of a fluid injection).

In this paper, we propose an approach to automatically generate dataflows given a configuration space close to *elsA*'s: instead of considering an arbitrary input mesh, we consider that its variability space could be expressed with features. Our approach is structured in two main parts: first, it uses Software Product Line (SPL) techniques [12, Sect. 6.6.1] to express the variability of tasks w.r.t. the configuration space, and configures them given the options selected by the user; then, it uses term rewriting to assemble these configured tasks into a dataflow that computes the data requested by the user. Figure 1 details the structure of our approach:

- first, we use a Domain Specific Language (DSL) duly extended with concepts from Delta-Oriented Programming (DOP) [13] to model the variability of the dataflow's tasks. This DSL allows us to specify which tasks, with which inputs and which outputs, are available to construct a dataflow. Then,

- given an input *Product* specifying the values of the different options, the

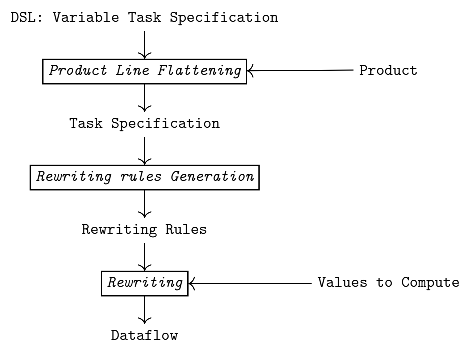

Figure 1: Dataflow generation pipeline

Product Line Flattening process automatically generates the *Task Specification* corresponding to that specific product; then

- the *Rewriting rules Generation* process automatically translates the specification into term rewriting rules; and
- given a list of *Values to Compute*, we simply apply the generated rewriting rules on this data to obtain a correct dataflow computing these values by using the tasks available in the specification.

We presented a preliminary version of this approach in [14]. In this paper, we: *i*) replace the ad-hoc dependency solver with rewriting; *ii*) give a precise algorithm for each step of our dataflow generation process; and *iii*) add a static analysis to guarantee that, for each product, a well-formed dataflow is generated.

Outline. Section 2 illustrates variability on dataflows with an example from Computational Fluid Dynamics (CFD), and shows how such a dataflow can be encoded with terms in order to motivate our approach of using term rewriting. Section 3 introduces our DSL, its DOP extension, and how to use them to generate a dataflow. Section 4 describes the different analyses guaranteeing that a DSL can generate a correct dataflow for all its products. Section 5 introduces our prototype implementation and presents some benchmarks illustrating the feasibility of the approach in practice. Finally, Section 6 discusses related work, while Section 7 concludes the paper.

2. Running Example

In this section, we present the running example that is used throughout the paper to illustrate our approach. This example is inspired from the variability and computation that occurs in *elsA* and is structured in three parts: first we introduce a simple feature model; second we present two simple dataflows that corresponds to some products of the feature model; and finally, we show how

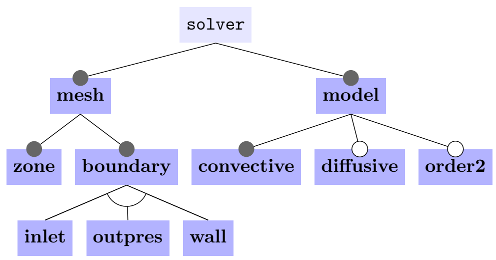

Figure 2: Simple CFD feature model

these dataflows could be encoded as terms. In this example and in the rest of the paper, we will use the Maude term rewriting language [15] to write down terms and rewriting rules.

2.1. Feature Model

Figure 2 shows the feature model of our running example, which is structured in two main parts. The first part encodes the variability of the mesh and is identified with the **mesh** feature. A mesh always has a unique zone, and between one and three boundaries of different types: **inlet** models the injection of fluid; **outpres** models a possible output of the fluid flow; and **wall** models a wall. The second part encodes which approximation of the fluid dynamics is considered in the computation and is identified with the **model** feature. The mandatory **convective** feature only considers convective dynamics (which are triggered by pressure). The optional **diffusive** feature extends the flow dynamics by also considering its viscosity. Finally, the optional **order2** feature changes the behaviour of the selected flow dynamics and asks them to have a more precise computation (i.e., instead of approximating the equations with a polynomial of degree one, they are now approximated with a polynomial of degree 2).

2.2. Dataflows

Figure 3 shows the dataflow computing the value `Rhs` only (i.e., the update of the fluid flow fixpoint computation) while selecting the boundaries **inlet** and **outpres**, and no optional model feature. The computation starts with the data `Conservative` which models the currently computed flow on the unique zone of the mesh. Then on one side, it uses one function per boundary (resp. *inlet* and *outpres*) to compute data (resp. `BC(Inlet)` and `BC(Outpres)`) encoding the effect of these boundaries on the flow. On the other side, it uses the *primitive* function to normalize the `Conservative` data into `Primitive`. Then the functions *convectiveFluxBC* and *convectiveFlux* compute the update of the convective fluid flow respectively on the boundaries (`FxcBC`) and on the zone (`Fxc`) of the mesh. Finally, these two flows are merged with the *fluxBalance* function and normalized with the *explicitIncrement* to generate the data `Rhs`.

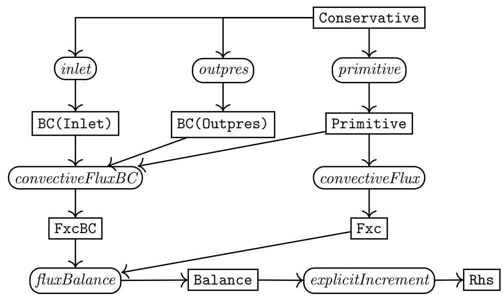

Figure 3: Dataflow computing `Rhs` with **inlet**, **outpres** selected

Figure 4 shows a more complex dataflow that computes the value `Rhs` and the gradient (i.e., the derivative) of the pressure (which is used to identify shocks) while selecting the boundary **wall** and all optional model features. Compared to the dataflow presented in Figure 3, this dataflow has one modified part and three additional parts (depicted in gray). The part that is modified corresponds to the management of the boundaries: since the boundary of the mesh only contains a wall (and not an injection and exit flow as in Figure 3), the effect of this boundary on the flow is now modelled by the data `BC(Wall)`, computed by the *wall* function. The first additional part occurs on the right of `Primitive`, where some new tasks appeared to compute the gradient of the the pressure `grad(Pressure)` requested by the user: the function *pressure* computes the `Pressure` from `Primitive`, and then the *gradient* function is used to compute `grad(Pressure)`. The second additional part is triggered by the selection of the **order2** feature: the functions *convectiveFlux* and *convectiveFluxBC* now take the additional parameter `grad(Primitive)`, computed by the *gradient* function applied on `Primitive`. The last additional part consists of the management of the diffusive flow: in addition to the *convectiveFlux* and *convectiveFluxBC* functions, the similar functions *diffusiveFlux* and *diffusiveFluxBC* were added, that take an extra input `grad(grad(Primitive))` (computed by the *gradient* function applied on `grad(Primitive)`).

Finally, similarly to the dataflow in Figure 3, the function *fluxBalance* collects all the computed flows into the `Balance` data, which is then normalized with the *explicitIncrement* to generate the data `Rhs`.

2.3. Dataflows as Terms

Listing 1 presents the abstract syntax that encodes a dataflow in the form of a Maude module. In this encoding, a dataflow is a DAG that contains two main kinds of nodes: data and tasks. We model these nodes with the term constructors `data` of sort `Data` and `task` of sort `Task`. A data node can either be a root of the dataflow, or be computed by a task. In both cases, data has a value (of sort `Value`). A task is identified by the ID of its function (of sort `FunctionID`) and the list of its data parameters.

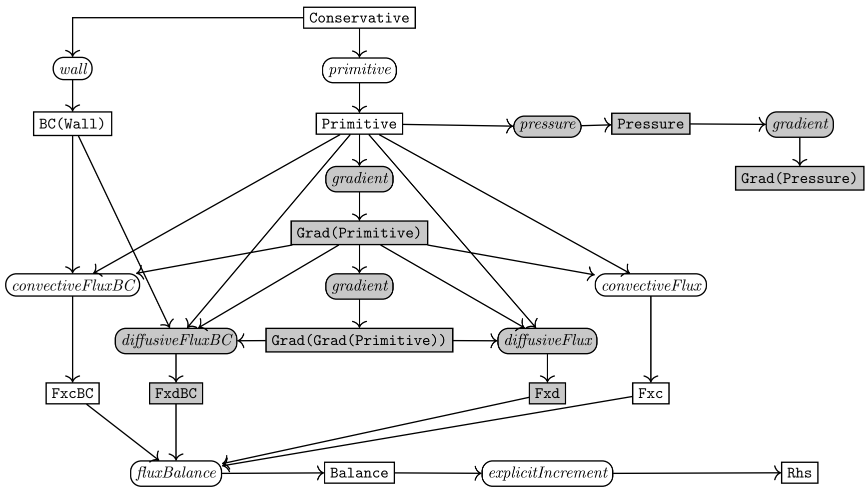

Figure 4: Dataflow computing `Rhs` and `Grad(Pressure)`
with **wall**, **diffusive** and **order2** selected

```
mod DATAFLOW is
 sorts Data Task Value FunctionID DataList .

 subsort Data < DataList .
 op _ _ : Data DataList -> DataList [ctor] .

 op data : Value -> Data [ctor] .
 op data : Value Task -> Data [ctor] .

 op task : FunctionID DataList -> Task [ctor] .
endm
```

Listing 1: The `DATAFLOW` Maude module: abstract syntax of a dataflow

In the rest of this section, we use the `DATAFLOW` Maude module as a basis to encode the two dataflows in Figures 3 and 4 into terms.

First, the Maude module `EXAMPLE-CORE` in Listing 2 provides the sorts and term constructors used to encode our dataflow examples. We have three sorts in this module: `Value` and `FunctionID` are the same as in the `DATAFLOW` module, and `BC` is a new sort for values held on a mesh boundary. The rest of the module is structured in three parts, each one declaring the constructor for a specific sort. First, the module declares the base values of our dataflow: `Conservative`, `Primitive`, etc. These constructors do not have any parameters, except for `BC` that takes a boundary value in parameter (this models the fact that these values are somewhat special), and `Grad` that takes a value in parameter (as illustrated in Figure 4, it is indeed possible to compute the gradient of any kind of value). Second, the module declares the three BC of our example: `Inlet`, `Outpres` and

```
mod EXAMPLE-CORE is
 sort Value BC FunctionID .

 *** values
 op Conservative : -> Value [ctor] .
 op Primitive : -> Value [ctor] .
 op Fxc : -> Value [ctor] .
 op Fxd : -> Value [ctor] .
 op FxcBC : -> Value [ctor] .
 op FxdBC : -> Value [ctor] .
 op Balance : -> Value [ctor] .
 op Rhs : -> Value [ctor] .
 op Pressure: -> Value [ctor] .
 op BC : BC -> Value [ctor] .
 op Grad : Value -> Value [ctor] .

*** BCs
 op Inlet : -> BC [ctor] .
 op Outpres : -> BC [ctor] .
 op Wall : -> BC [ctor] .

 *** function ids
 op inlet : -> FunctionID [ctor] .
 op outpres : -> FunctionID [ctor] .
 op wallslip : -> FunctionID [ctor] .
 op primitive : -> FunctionID [ctor] .
 op pressure : -> FunctionID [ctor] .
 op gradient : -> FunctionID [ctor] .
 op convectiveFlux : -> FunctionID [ctor] .
 op convectiveFluxBC : -> FunctionID [ctor] .
 op diffusiveFlux : -> FunctionID [ctor] .
 op diffusiveFluxBC : -> FunctionID [ctor] .
 op fluxBalance : -> FunctionID [ctor] .
 op explicitIncrement : -> FunctionID [ctor] .
endm
```

Listing 2: The `EXAMPLE-CORE` Maude module: all declarations for our running example

`Wall`. And finally, the module declares all the `FunctionID` corresponding to functions in our dataflow.

The Maude module `EXAMPLE-1` in Listing 3 encodes the dataflow of Figure 3. This module is structured in five parts:

1. we first include and merge the two modules performing the core declarations, i.e., `DATAFLOW` and `EXAMPLE-CORE`;

2. we then declare the term `dconservative`, which corresponds to the root node of our dataflow containing the value `Conservative`;

3. we then construct the part of the dataflow computing `Fxc`: `Primitive` (in node `dprimitive`) is computed by applying the function `primitive` on `Conservative`; and `Fxc` (in the node `dFxc`) is computed by applying the function `convectiveFlux` on `Primitive`;

4. we then construct the part of the dataflow computing `FxcBC`: `BC(Inlet)` (in the node `dinlet`) is computed by applying the function `inlet` on `Conservative`; `BC(Outpres)` (in the node `doutpres`) is computed by applying the function `outpres` on `Conservative`; and `FxcBC` (in the node `dFxcBC`) is computed by applying the function `convectiveFluxBC` on `Primitive`, `BC(Inlet)`, and `BC(Outpres)`;

5. and finally, we conclude the dataflow with the computation of the `Rhs` data: `Balance` (in the node `dBalance`) is computed by applying the function `fluxBalance` on `Fxc` and `FxcBC`; and `Rhs` (in the node `dRhs`) is computed by applying the function `explicitIncrement` on `Balance`.

The Maude module `EXAMPLE-2` in Listing 4 encodes the dataflow of Figure 4. Since this dataflow is more complex than the one of Figure 3, it is structured in seven parts (rather than in five parts as in Listing 3):

```
mod EXAMPLE-1 is
 *** includes dataflow abstract syntax, value and function declaration
 protecting DATAFLOW + EXAMPLE-CORE .

 *** root of the dataflow
 op dconservative : -> Data .
 eq dconservative = data(Conservative) .

 *** Fxc computation
 op dprimitive : -> Data .
 eq dprimitive = data(Primitive, task(primitive, dconservative)) .
 op dFxc : -> Data .
 eq dFxc = data(Fxc, task(convectiveFlux, dprimitive)) .

 *** FxcBC computation
 op dinlet : -> Data .
 eq dinlet = data(BC(Inlet), task(inlet, dconservative)) .
 op doutpres : -> Data .
 eq doutpres = data(BC(Outpres), task(outpres, dconservative)) .
 op dFxcBC : -> Data .
 eq dFxcBC = data(FxcBC, task(convectiveFluxBC, dprimitive dinlet doutpres)) .

 *** Rhs computation
 op dBalance : -> Data .
 eq dBalance = data(Balance, task(fluxBalance, dFxc dFxcBC)) .
 op dRhs : -> Data .
 eq dRhs = data(Rhs, task(explicitIncrement, dBalance)) .
endm
```

Listing 3: The `EXAMPLE-1` Maude module: encodes the dataflow of Figure 3

1. and 2. these parts are identical to the ones in Listing 3: they first include the modules `DATAFLOW` and `EXAMPLE-CORE`, and then declare the root node of our dataflow `dconservative` that contains the value `Conservative`;

3. we then construct the chain of the `Primitive` data and its gradients: `Primitive` (in node `dprimitive`) is computed by applying the function `primitive` on `Conservative`; `Grad(Primitive)` (in node `dgprimitive`) is computed by applying the function `gradient` on `Primitive`; and `Grad(Grad(Primitive))` (in node `dggprimitive`) is computed by applying the function `gradient` a second time, on `Grad(Primitive)`;

4. We then construct the computation of gradient of the pressure: `Pressure` (in node `dpressure`) is computed by applying the function `pressure` on `Primitive`; and `Grad(Pressure)` (in node `dgpressure`) is computed by applying the function `gradient` on `Pressure`;

5. we then construct the computation of the convective and diffusive flows on the zone: `Fxc` (in the node `dFxc`) is computed by applying the function `convectiveFlux` on `Primitive` and `Grad(Primitive)`; `Fxd` (in the node `dFxd`) is computed by applying the function `diffusiveFlux` on `Primitive`, `Grad(Primitive)` and `Grad(Grad(Primitive))`;

6. similarly, we construct the computation of the convective and diffusive flows on the boundary: `BC(Wall)` (in the node `dwall`) is computed by applying the function `wall` on `Conservative`; `FxcBC` (in the node `dFxcBC`)

```
mod EXAMPLE-2 is
 *** includes dataflow abstract syntax, value and function declaration
 protecting DATAFLOW + EXAMPLE-CORE .

 *** root of the dataflow
 op dconservative : -> Data .
 eq dconservative = data(Conservative) .

 *** Primitive with gradient computation
 op dprimitive : -> Data .
 eq dprimitive = data(Primitive, task(primitive, dconservative)) .
 op dgprimitive : -> Data .
 eq dgprimitive = data(Grad(Primitive), task(gradient, dgprimitive)) .
 op dggprimitive : -> Data .
 eq dggprimitive = data(Grad(Grad(Primitive)), task(gradient, dgprimitive)) .

 *** Grad(Pressure) computation
 op dpressure : -> Data .
 eq dpressure = data(Pressure, task(pressure, dprimitive)) .
 op dgpressure : -> Data .
 eq dgpressure = data(Grad(Pressure), task(gradient, dpressure)) .

 *** Fxc and Fxd computation
 op dFxc : -> Data .
 eq dFxc = data(Fxc, task(convectiveFlux, dprimitive dgprimitive)) .
 op dFxd : -> Data .
 eq dFxd = data(Fxd, task(diffusiveFlux, dprimitive dgprimitive dgprimitive)) .

 *** FxcBC and FxdBC computation
 op dwall : -> Data .
 eq dwall = data(BC(Wall), task(wall, dconservative)) .
 op dFxcBC : -> Data .
 eq dFxcBC = data(FxcBC, task(convectiveFluxBC, dwall dprimitive dgprimitive)) .
 op dFxdBC : -> Data .
 eq dFxdBC = data(FxdBC, task(diffusiveFluxBC, dwall dprimitive dgprimitive dggprimitive)) .

 *** Rhs computation
 op dBalance : -> Data .
 eq dBalance = data(Balance, task(fluxBalance, dFxc dFxd dFxcBC dFxdBC)) .
 op dRhs : -> Data .
 eq dRhs = data(Rhs, task(explicitIncrement, dBalance)) .
endm
```

Listing 4: The `EXAMPLE-2` Maude module: encodes the dataflow of Figure 4

is computed by applying the function `convectiveFluxBC` on `BC(Wall)`, `Primitive` and `Grad(Primitive)`; `FxdBC` (in the node `dFxdBC`) is computed by applying the function `diffusiveFluxBC` on `BC(Wall)`, `Primitive`, `Grad(Primitive)` and `Grad(Grad(Primitive))`;

7. and finally, we conclude the dataflow with the computation of the `Rhs` data: `Balance` (in the node `dBalance`) is computed by applying the function `fluxBalance` on `Fxc`, `Fxd`, `FxcBC` and `FxdBC`; and `Rhs` (in the node `dRhs`) is computed by applying the function `explicitIncrement` on `Balance`.

3. Model

This section presents the main elements of our approach. First, we recall the concepts of signatures and terms used in rewriting. We then present our DSL without its DOP extension and describe the algorithm that transforms any DSL program into rewriting rules. Finally, we describe the full version of our DSL, and introduce the *flattening* algorithm that generates the variant of an input DSL program L for an input product p of the associated SPL.

3.1. Dataflow Term Signature

As illustrated in Section 2, a signature that encodes our dataflow is structured into two parts: one part that creates the dataflow structure (with, e.g., constructors `data` and `task`) and is common to all dataflows, and one *user* part that declares which values and functions are available in the dataflow construction, which is specific to each dataflow.

Here, we first recall the definitions for order-sorted signatures and terms. We then define the two parts of a dataflow signature.

3.1.1. Preliminary Definitions

We provide some notation and a basic definition over sets and ordered sets.

Definition 1 ($\mathcal{S}$-sorted set)**.** *Given a set S and an S-indexed family $V = \{V_s\}_{s\in S}$, we write $v : s \in V$ for $v \in V_s$. Moreover, we denote by S^∞ the set $\bigcup_{i=1}^{\infty} S^i$.*

Given a partially ordered set (poset) $\mathcal{S} = (S, <)$, an $\mathcal{S}$-sorted set *is an S-indexed family $V = \{V_s\}_{s\in S}$ such that $s < s'$ implies $V_s \subseteq V_{s'}$. Moreover, we add a partial order to S^∞ with*

$$(s_1, \ldots, s_n) < (s'_1, \ldots, s'_n) \textit{ iff } (\exists 1 \leq i \leq n,\ s_i < s'_i) \wedge (\forall 1 \leq j \leq n,\ s_i \leq s'_i)$$

The following definition specifies the arity of term constructors for a given ordered set of sorts $\mathcal{S}$.

Definition 2 ($\mathcal{S}$-arity)**.** *Given a poset $\mathcal{S} = (S, <)$, an* $\mathcal{S}$-arity *A is a subset of S^∞ such that for all $(s_1, \ldots, s_n), (s'_1, \ldots, s'_n) \in A$ with $s_i \leq s'_i$ for all $1 \leq i \leq n-1$, then $s_n \leq s'_n$. For all $(s_1, \ldots, s_n) \in A$ with $n > 1$, we write $(s_1, \ldots, s_{n-1}) \to s_n$ as syntactic sugar for $(s_1, \ldots, s_n)$.*

The following definition introduces an order-sorted signature for a given ordered set of sorts $\mathcal{S}$.

Definition 3 ($\mathcal{S}$-sorted signature)**.** *Given a poset $\mathcal{S} = (S, <)$, an* $\mathcal{S}$-sorted signature *is an $\mathcal{A}$-sorted set F with $\mathcal{A}$ being an $\mathcal{S}$-arity.*

Finally, we can define the set of terms given an ordered set of sorts $\mathcal{S}$ and a signature F.

Definition 4 ($\mathcal{S}$-sorted set of terms)**.** *Given a poset $\mathcal{S} = (S, <)$, an $\mathcal{S}$-sorted signature F, and an $\mathcal{S}$-sorted set of variables V, the $\mathcal{S}$-sorted set $\mathcal{T}(F, V)$ of* terms *is inductively defined as follows:*

- $v : s \in \mathcal{T}(F, V)$ *if* $v : s \in V$
- $f(t_1, \ldots, t_n) : s \in \mathcal{T}(F, V)$ *if* $t_1 : s_1, \ldots, t_n : s_n \in \mathcal{T}(F, V)$, $f : s_1, \ldots, s_n \to s' \in F$ *and* $s' \leq s$.

3.1.2. Dataflow

The next definition states when a signature is *dataflow-safe*, i.e., it is a valid user part of a dataflow signature.

Definition 5 (Dataflow-safe poset and $\mathcal{S}$-sorted signature)**.** *A poset $\mathcal{S} = (S, <)$ is* dataflow-safe *iff both of the following hold:*

1. *$\mathcal{S}$ contains* `Value` *and* `FunctionID`
2. *$\mathcal{S}$ contains neither* `Data`*, nor* `Task`

An $\mathcal{S}$-sorted signature F is dataflow-safe *iff for all $s \in S$,* `data` : `Value` $\to s \notin F$.

Example 1 (A dataflow-safe signature)**.** *The Maude module* `EXAMPLE-CORE` *in Listing 2 is a dataflow-safe signature.*

For the remainder of this paper, we assume a given dataflow-safe ordered set $\mathcal{S} = (S, <)$ and an $\mathcal{S}$-sorted signature F.

3.2. Function Specification

We now present the syntax of our function specification DSL, shown in Figure 5. We use the following name categories: f is a function name; v is a term variable; and s is a sort. Moreover, t is a term.

A specification starts with the declaration of a list of term variables, which is then followed by the list of function specifications. A function has a name f, a list of inputs, and a list of outputs (modelled by terms of sort `Value`).

The declaration order of variables and functions does not matter, so we consider this syntax up to declaration reordering (this will be used later to simplify specification transformation).

Example 2 (Dataflow specification)**.** *Listing 5 shows the specification for the dataflow in Figure 3. This specification starts with the functions used in the* `Fxc computation` *part of Listing 3:* `primitive` *takes the value* `Conservative`

$S ::= \texttt{vars}\ (v : s)^*\ (F)^*$	Specification
$F ::= \texttt{fun}\ f :\ \texttt{inputs}\ (t)^*\ \texttt{outputs}\ (t)^*$	Function Specification

Figure 5: Static syntax of function specification

```
// Fxc computation
fun primitive: inputs Conservative outputs Primitive
fun convectiveFlux: inputs Primitive outputs Fxc

// FxcBC computation
fun inlet: inputs Conservative outputs BC(Inlet)
fun outpres: inputs Conservative outputs BC(Outpres)
fun convectiveFluxBC: inputs Primitive, BC(Inlet), BC(Outpres) outputs FxcBC

// Rhs computation
fun fluxBalance: inputs Fxc, FxcBC outputs Balance
fun explicitIncrement: inputs Balance outputs Rhs
```

Listing 5: Function specification for the dataflow in Listing 3

$$\texttt{fun}\ f:\ \texttt{inputs}\ i_1,\ldots,i_n\ \texttt{outputs}\ o_1,\ldots,o_m$$

$$\rhd \begin{cases} \texttt{rl data(}o_1\texttt{) => data(}o_1\texttt{, task(}f\texttt{, data(}i_1\texttt{)}\ldots\texttt{data(}i_n\texttt{))).} \\ \ldots \\ \texttt{rl data(}o_m\texttt{) => data(}o_m\texttt{, task(}f\texttt{, data(}i_1\texttt{)}\ldots\texttt{data(}i_n\texttt{))).} \end{cases}$$

$$\frac{F_1 \rhd r_1 \quad \ldots \quad F_m \rhd r_m}{\texttt{vars}\ v_1 : s_1,\ldots,v_n,s_n\ F_1\ \ldots, F_m \rhd \texttt{var}\ v_1\ \texttt{:}\ s_1\ \texttt{.}\ \ldots\ \texttt{var}\ v_n\ \texttt{:}\ s_n\ \texttt{.}\ r_1\ \ldots\ r_m}$$

Figure 6: Translation rules from function specifications into Maude

in parameter and returns the value `Primitive`*; and* `convectiveFlux` *takes* `Primitive` *in parameter and returns* `Fxc`*.*

The second part of the specification describes the functions used in the `FxcBC computation` *part of Listing 3:* `inlet` *takes* `Conservative` *in parameter and returns* `BC(Inlet)`*; then* `outpres` *takes* `Conservative` *in parameter and returns* `BC(Outpres)`*; and finally* `convectiveFluxBC` *takes* `Primitive`*,* `BC(Inlet)`*, and* `BC(Outpres)` *in parameter and returns* `FxcBC`*.*

Finally, the last part of the specification describes the functions used in the `Rhs computation` *part of Listing 3:* `fluxBalance` *takes* `Fxc` *and* `FxcBC` *in parameter and returns* `Balance`*, while* `explicitIncrement` *takes* `Balance` *in parameter and returns* `Rhs`*.*

3.3. From Function Specifications to Maude Rewriting Rules

Figure 6 shows the rules to translate function specifications into Maude rewriting rules. The first rule takes the specification of a function f and for each of its outputs o_i, it generates a rewriting rule that adds to any node containing o_i not being computed (modelled by $\texttt{data(}o_i\texttt{)}$) the task that uses f to compute o_i (modelled by $\texttt{task(}f\texttt{, data(}i_1\texttt{)}\ldots\texttt{data(}i_n\texttt{))}$ with $i_1,\ldots,i_n$ the inputs of f).

The second rule takes a complete specification, and translates it into Maude by: replacing the variables declarations by equivalent Maude declarations; and replacing all function specifications by rewriting rules.

```
1
2  *** Fxc computation
3  rl data(Primitive) => data(Primitive, task(primitive, data(Conservative) )) .
4  rl data(Fxc) => data(Fxc, task(convectiveFlux, data(Primitive) )) .
5
6  *** FxcBC computation
7  rl data(BC(Inlet)) => data(BC(Inlet), task(inlet, data(Conservative) )) .
8  rl data(BC(Outpres)) => data(BC(Outpres), task(outpres, data(Conservative) )) .
9  rl data(FxcBC) =>
10  data(FxcBC, task(convectiveFluxBC, data(Primitive) data(BC(Inlet)) data(BC(Outpres)) )) .
11
12 *** Rhs computation
13 rl data(Balance) => data(Balance, task(fluxBalance, data(Fxc) data(FxcBC) )) .
14 rl data(Rhs) => data(Rhs, task(explicitIncrement, data(Balance) )) .
```

Listing 6: Rules generated from the specification in Figure 5

Example 3 (Rules generated from a dataflow specification)**.** *Listing 6 shows the rules generated from the specification in Figure 5. For clarity, we added sections in this generated set of rewriting rules to illustrate its relation to Listing 5. This generated file starts with the rewriting rules corresponding to the functions declared in the* `Fxc computation` *part of Listing 5: adding to the node* `data(Primitive)` *the task* `task(primitive, data(Conservative))`*; and adding to node* `data(Fxc)` *the task* `task(convectiveFlux, data(Primitive))`*.*

The second part of the set describes the rewriting rules corresponding to the functions declared in the `FxcBC computation` *part of Listing 5: it adds to the node* `BC(Inlet)` *the task* `task(inlet, data(Conservative))`*; it adds to the node* `BC(Outpres)` *the task* `task(outpres, data(Conservative))`*; and it adds to the node* `data(FxcBC)` *the following task*

```
task(convectiveFluxBC,
  data(Primitive) data(BC(Inlet))  data(BC(Outpres)))
```

Finally, the last part of the specification describes rewriting rules corresponding to the functions declared in the `Rhs computation` *part of Listing 5: adding to node* `data(Balance)` *the task* `task(fluxBalance, data(Fxc)  data(FxcBC))`*; and to node* `data(Rhs)` *the task* `task(explicitIncrement, data(Balance))`*.*

Rewriting the term `data(Rhs)` *with these rules will give the same term* `dRhs` *as in* `EXAMPLE-1` *in Listing 3. Indeed, rewriting consists of applying the rewriting rules wherever on the input term, until none can be applied anymore. First,* `data(Rhs)` *matches the pattern of the rule in Line 14: the rule is applied, resulting in the term* `data(Rhs,task(explicitIncrement, data(Balance)))`*. Second, within that term,* `data(Balance)` *matches the pattern of the rule in Line 13: the rule is applied, resulting in the term*

```
data(Rhs,task(explicitIncrement,
  data(Balance, task(fluxBalance, data(Fxc) data(FxcBC) ))))
```

Following this principle, the subterms `data(Fxc)` *and* `data(FxcBC)` *will be then rewritten with the rules in lines 4 and 9 which expands further our encoding of the dataflow. Ultimately, all data and all tasks of the dataflow in* `EXAMPLE-1` *will be added by different applications of the rules in Figure 5.*

$$\frac{t \in \mathrm{dom}(\Gamma)}{\Gamma \vdash t : \Gamma(t)} \qquad \frac{\Gamma \vdash t_i : s_i \quad f : s_1, \ldots, s_n \to s \in F}{\Gamma \vdash f(t_1, \ldots, t_n) : s}$$

$$\frac{\Gamma \vdash t_1 : s_1 \quad s_1 \leq \texttt{Value} \quad \ldots \quad \Gamma \vdash t_m : s_m \quad s_m \leq \texttt{Value}}{\Gamma \vdash \texttt{fun}\ f :\ \texttt{inputs}\ t_1, \ldots, t_n\ \texttt{outputs}\ t_{n+1}, \ldots, t_m}$$

$$\frac{v_1 : s_1, \ldots, v_n : s_n \vdash F_1 \quad \ldots \quad v_1 : s_1, \ldots, v_n : s_n \vdash F_m}{\vdash \texttt{vars}\ v_1 : s_1, \ldots, v_n : s_n\ F_1\ \ldots, F_m}$$

Figure 7: Checking the correctness of Input / Output definition

3.4. Specification Correction

Specifications in our DSL must validate some sanity conditions to ensure that the generated rewriting rules are well formed. Next we list these conditions.

3.4.1. Naming

A first standard condition is the absence of name clashes: all variables and all functions must have different names. Since this condition is standard, in the remainder we assume that it is always satisfied. This condition can be straightforwardly checked by a standard static analysis.

3.4.2. Term Construction

The second set of sanity conditions ensures that the left-hand side and the right-hand side of every generated rewriting rules are well sorted, with sort `data`. First, all function names must be declared in the signature F, with sort `FunctionID`. Moreover, the inputs and outputs of the functions need to have sort `Value` in F. The rules to check the sort of the input and output terms of every function specification in a DSL program are shown in Figure 7.

The first rule states that the sort of a variable is given by its declaration (stored in Γ). The second rule states that if the term constructor f has arity $s_1, \ldots, s_n \to s$ and has parameters t_i of sorts s_i, then $f(t_1, \ldots, t_n)$ has sort s. The third rule states that all inputs and outputs of a function declaration must have sort (or subsort of) `Value`. Finally, the fourth rule creates the store Γ from the variable declaration (as previously stated) and ensures that all function specifications are correct.

Any variable in a function's input must be present in all of the function's output. In any term rewriting system, rewriting rules should not introduce fresh variables, i.e., the variables in the right-hand side of a rule must all be present in the left-hand side. This constraint translates into our DSL by the fact that for every function specification, all variables in the inputs of the function must be declared in every output of the function. This constraint is formalized by the following equation, where $fv(t)$ denotes the set of variables in t for any term t:

$$\forall k \in I,\ fv(t_k) \subseteq \bigcap_{o \in O} fv(t_o) \tag{1}$$

L	::=	`features` $(o)^*$ `with` ϕ; $S\ (Dd)^*\ CK$	Software Product Line
Dd	::=	`delta` $d; (Do)^*$	Delta
Do	::=	`add` F | `remove` f | `modify` $f\ (Dm)^*$	Delta Operation
		| `add var` $v : s$ | `remove var` v	
Dm	::=	$Op\ Cat\ t$	Function Modification
Op	::=	`add` | `remove`	Modification Operation
Cat	::=	`input` | `output`	Modification Category
CK	::=	$DAC\ DAO$	Configuration Knowledge
DAC	::=	`delta` d `when` ϕ	Activation Condition
DAO	::=	$(d)^+(< (d)^+)^+$	Delta Ordering

Figure 8: Syntax of deltas

Optional: data-driven functions. This last condition is not mandatory to ensure the correct construction of the rewriting rules, but ensures that every function specification corresponds to a data-driven function, i.e., a function's outputs only depend on its inputs. This dependency translates into our DSL by the fact that for every function specification, the variables in the function's outputs are declared in its inputs. This constraint is formalized by the following equation:

$$\bigcup_{k \in I} fv(t_k) = \bigcup_{o \in O} fv(t_o) \tag{2}$$

3.5. Variable Function Specification

Finally, the syntax of an SPL over function specifications is given in Figure 8.

An SPL starts with the definition of a feature model, with features $(o)^*$ and a propositional formula ϕ over features. This feature model is then followed by the *core* of the SPL, i.e., the initial set of variables and function declarations S, using the syntax presented in Figure 5. The rest of the SPL declares a set of deltas $(Dd)^*$ that manipulate S, and a configuration knowledge CK.

Each delta specifies a number of changes to S. A delta comprises the keyword `delta` followed by the delta's name, a semicolon, and a sequence of delta operations $(Do)^*$. A *delta operation* Do can add/remove a function specification definition, or modify it by adding/removing inputs and outputs (via modifying operations Dm). Moreover, a *delta operation* can declare or remove variables.

Configuration knowledge CK provides a mapping from products to variants by describing the connection between deltas and features. First, the DAC entries specifies an activation condition ϕ (a propositional formula over features) for each delta in the SPL. Second, the DAO entries specify an application order between deltas: each of these entries specifies a partial order over the set of deltas in terms of a total order on disjoint subsets of delta names.

The overall delta application order $\prec$ is the transitive closure of the union of these partial orders. In this paper, we assume that $\prec$ is *consistent* (i.e., $\prec$ is a partial order) and *unambiguous* (i.e., all the total delta application orders that respect $\prec$ generate the same variant for each product). Techniques that allow one to check that $\prec$ is unambiguous are described in the literature [16, 17]. Without loss of generality, we assume that the total order in which delta definitions are listed is compatible with $\prec$.

Flattening rules. Figure 9 shows the flattening rules that, given an input SPL L and a product p of that SPL, apply the activated deltas in L in order on the core part of L. The first two rules describe the flattening process at the SPL level. Rule (SPL-1) first ensures that p is a product of the SPL (with $p \vdash \phi$), takes the first delta Dd of the SPL, and applies it to the core S. This application is written $[\![Dd]\!][\phi'](S)$ where ϕ' is the activation condition of the delta Dd. Rule (SPL-2) is used when all delta have been applied (i.e., the list of deltas that are left is empty): it still ensures that p is a product of the SPL, and then simply returns the core S of the SPL.

The next two rules deal with the application of deltas. Rule (D-1) is used when the delta is activated (checked with $p \vdash \phi$) and contains at least one operation Do: in that case, we apply that operation on the core S (denoted by $[\![Do]\!][S]$), and use this as parameter for the rest of the operations in the deltas. Rule (D-2) is used when the delta is not activated (checked with $p \nvdash \phi$) or does not contain any operations: in that case we simply return the core S unchanged.

The next three rules describe the flattening of delta operations on functions. Rule (F-ADD) states that to add a function F to a core S, that function must not be declared already in S (this is checked by $\mathit{name}(F) \notin \{\mathit{name}(F_i) \mid 1 \leq i \leq n\}$). If this condition is validated, we return S extended with the new function. Rule (F-REM) states that to remove a function named f from a core S, that function must be present in S (this is checked with extracting from S the function F with $\mathit{name}(F) = f$). If this condition is validated, we return S without its function f. Rule (F-MOD) states that to modify a function named f in a core S, that function must be present in S (this is checked as in rule (F-REM)). If this condition is validated, we return S with its function f modified by the set of operations $(Dm)^*$.

The next two rules deal with the delta operations on variables, and are very similar to rules (F-ADD) and (F-REM): to add a variable, that variable must not be present already in the core; and, dually, to remove a variable, that variable must be declared in the core.

Finally, the last four rules describe the flattening of modification operations Dm. Rule (I-ADD) states that to add an input to a function specification, the added term must not already be an input of that function. Rule (O-ADD) states that to add an output to a function specification, the added term must not already be an output of that function. Rule (I-REM) states that to remove an input from a function specification, the removed term must be an input of that function. Rule (O-REM) states that to remove an output from a function specification, the removed term must be an output of that function.

Example 4 (A delta-oriented SPL of dataflows)**.** *Listing 7 presents the SPL that contains the complete specification for our running example. Lines 1–4 present the feature model of our running example, with the root feature* `solver`*,*

[SPL-1]
$$\frac{p \vdash \phi \quad \mathit{name}(Dd) = d}{\begin{array}{c}(\texttt{features}\ (o)^*\ \texttt{when}\ \phi; S\ Dd\ (Dd)^*\ \texttt{delta}\ d\ \texttt{when}\ \phi';\ CK) \\ \rhd_p (\texttt{features}\ (o)^*\ \texttt{when}\ \phi; [Dd][\phi'](S)\ (Dd)^*\ CK)\end{array}}$$

[SPL-2]
$$\frac{p \vdash \phi}{(\texttt{features}\ (o)^*\ \texttt{when}\ \phi; S\ CK) \rhd_p S}$$

[D-1]
$$\frac{p \vdash \phi}{\begin{array}{c}[\texttt{delta}\ d; Do\ (Do)^*][\phi](S) \\ \rhd_p [\texttt{delta}\ d; (Do)^*][\phi]([Do](S))\end{array}}$$

[D-2]
$$\frac{p \nvdash \phi \vee (Do)^* = \varepsilon}{[\texttt{delta}\ d; (Do)^*][\phi](S) \rhd_p S}$$

[F-ADD]
$$\frac{\mathit{name}(F) \notin \{\mathit{name}(F_i) \mid 1 \leq i \leq n\}}{[\texttt{add}\ F](\texttt{vars}\ (v:s)^*\ F_1\ \ldots\ F_n) \rhd_p \texttt{vars}\ (v:s)^*\ F_1\ \ldots\ F_n\ F}$$

[F-REM]
$$\frac{\mathit{name}(F) = f}{[\texttt{remove}\ f](\texttt{vars}\ (v:s)^*\ F\ (F)^*) \rhd_p \texttt{vars}\ (v:s)^*\ (F)^*}$$

[F-MOD]
$$\frac{\mathit{name}(F) = f}{[\texttt{modify}\ f\ (Dm)^*](\texttt{vars}\ (v:s)^*\ F\ (F)^*) \rhd_p \texttt{vars}\ (v:s)^*\ [(Dm)^*](F)\ (F)^*}$$

[V-ADD]
$$\frac{v \notin \{v_i \mid 1 \leq i \leq n\}}{[\texttt{add var}\ v:s](\texttt{vars}\ v_1 : s_1, \ldots, v_n : s_n\ (F)^*) \rhd_p \texttt{vars}\ v_1 : s_1, \ldots, v_n : s_n, v : s\ (F)^*}$$

[V-REM]
$$[\texttt{remove var}\ v](\texttt{vars}\ v:s,\ (v:s)^*\ (F)^*) \rhd_p \texttt{vars}\ (v:s)^*\ (F)^*$$

[I-ADD]
$$\frac{t \notin \{t_i \mid 1 \leq i \leq n\}}{\begin{array}{c}[\texttt{add input}\ t](\texttt{fun}\ f:\ \texttt{inputs}\ t_1\ \ldots\ t_n\ \texttt{outputs}\ (t')^*) \\ \rhd_p \texttt{fun}\ f:\ \texttt{inputs}\ t_1\ \ldots\ t_n\ t\ \texttt{outputs}\ (t')^*\end{array}}$$

[O-ADD]
$$\frac{t \notin \{t_i \mid 1 \leq i \leq n\}}{\begin{array}{c}[\texttt{add output}\ t](\texttt{fun}\ f:\ \texttt{inputs}\ (t')^*\ \texttt{outputs}\ t_1\ \ldots\ t_n) \\ \rhd_p \texttt{fun}\ f:\ \texttt{inputs}\ (t')^*\ \texttt{outputs}\ t_1\ \ldots\ t_n\ t\end{array}}$$

[I-REM]
$$\begin{array}{c}[\texttt{remove input}\ t](\texttt{fun}\ f:\ \texttt{inputs}\ t\ (t')^*\ \texttt{outputs}\ (t'')^*) \\ \rhd_p \texttt{fun}\ f:\ \texttt{inputs}\ (t')^*\ \texttt{outputs}\ (t'')^*\end{array}$$

[O-REM]
$$\begin{array}{c}[\texttt{remove output}\ t](\texttt{fun}\ f:\ \texttt{inputs}\ (t')^*\ \texttt{outputs}\ t\ (t'')^*) \\ \rhd_p \texttt{fun}\ f:\ \texttt{inputs}\ (t')^*\ \texttt{outputs}\ (t'')^*\end{array}$$

Figure 9: Flattening Rules

```
1  features solver mesh zone boundary inlet outpres wall
2   model convective diffusive order2 with
3    solver /\ mesh /\ boundary /\ zone /\ model /\
         convective
4      /\ (inlet \/ outpres \/ wall);
5
6  vars valueV: Value
7
8  //// Base Artifact
9  // functions on zone
10 fun primitive: inputs Conservative
11  outputs Primitive
12 fun gradient: inputs valueV
13  outputs Grad(valueV)
14 fun pressure: inputs Primitive
15  outputs Pressure
16 fun convectiveFlux: inputs Primitive
17  outputs Fxc
18 fun diffusiveFlux: inputs Primitive
19  Grad(Primitive) outputs Fxd
20
21 // functions on boundaries
22 fun inlet: inputs Conservative
23  outputs BC(Inlet)
24 fun outpres: inputs Conservative
25  outputs BC(Outpres)
26 fun wall: inputs Conservative
27  outputs BC(Wall)
28 fun convectiveFluxBC: inputs Primitive
29  outputs FxcBC
30 fun diffusiveFluxBC: inputs Primitive
31  Grad(Primitive) outputs FxdBC
32
33 // Rhs
34 fun fluxBalance: inputs Fxc, FxcBC
35  outputs Balance
36 fun explicitIncrement: inputs Balance
37  outputs Rhs
38 //// DELTA
39 // boundaries
40 delta d_inlet;
41  modify convectiveFluxBC
42   add input BC(Inlet)
43  modify diffusiveFluxBC
44   add input BC(Inlet)
45 delta d_outpres;
46  modify convectiveFluxBC
47   add input BC(Outpres)
48  modify diffusiveFluxBC
49   add input BC(Outpres)
50 delta d_wall;
51  modify convectiveFluxBC
52   add input BC(Wall)
53  modify diffusiveFluxBC
54   add input BC(Wall)
55
56 // computation
57 delta d_diffusive;
58  modify fluxBalance
59   add input Fxd add input FxdBC
60 delta d_order2;
61  modify convectiveFlux
62   add input Grad(Primitive)
63  modify convectiveFluxBC
64   add input Grad(Primitive)
65  modify diffusiveFlux
66   add input Grad(Grad(Primitive))
67 modify diffusiveFluxBC
68   add input Grad(Grad(Primitive))
69
70 delta d_inlet when inlet;
71 delta d_outpres when outpres;
72 delta d_wall when wall;
73 delta d_diffusive when diffusive;
74 delta d_order2 when order2;
```

Listing 7: Complete Software Product Line of our running example

the features `mesh`*,* `zone`*,* `boundary`*,* `inlet`*,* `outpres` *and* `wall` *for the structure of the mesh, and the features* `model`*,* `convective`*,* `diffusive` *and* `order2` *for the computation.*

Line 6 declares the variable `valueV` *of sort* `Value` *that is used for the declaration of the* `gradient` *function.*

Lines 10–19 define the base specification of all the functions working on the mesh's zone that are used in our dataflow construction: `primitive` *takes the value* `Conservative` *in parameter and returns the value* `Primitive`*;* `gradient` *can take any value in parameter (modelled with the variable* `valueV`*), and returns the gradient of that value (modelled with the term* `Grad(valueV)`*);* `pressure` *takes the value* `Primitive` *in parameter and returns the value* `Pressure`*; the function* `convectiveFlux` *takes* `Primitive` *in parameter and returns* `Fxc`*; and, finally,* `diffusiveFlux` *takes* `Primitive` *and* `Grad(Primitive)` *in parameter and returns* `Fxd`*.*

Lines 22–31 define the base specification of all the functions working on the mesh's boundaries that are used in our dataflow construction: `inlet` *takes*

`Conservative` *in parameter and returns* `BC(Inlet)`*;* `outpres` *takes* `Conservative` *in parameter and returns* `BC(Outpres)`*;* `wall` *takes* `Conservative` *in parameter and returns* `BC(Wall)`*;* `convectiveFluxBC` *takes* `Primitive`*, in parameter and returns* `FxcBC`*; and* `diffusiveFluxBC` *takes* `Primitive`*, in parameter and returns* `FxdBC`*.*

Finally, lines 34–37 define the base specification of the remaining functions: `fluxBalance` *takes* `Fxc` *and* `FxcBC` *in parameter and returns* `Balance`*; while* `explicitIncrement` *takes* `Balance` *in parameter and returns* `Rhs`*.*

The rest of the SPL declares the deltas that modify the base specifications with respect to the selected features.

Lines 40–44 describe the delta `d_inlet` *that adds* `BC(Inlet)` *as input of the functions* `convectiveFluxBC` *and* `diffusiveFluxBC` *in case the feature* `inlet` *is selected. Lines 45 to 49 describe the delta* `d_outpres` *that adds* `BC(Outpres)` *as input of the functions* `convectiveFluxBC` *and* `diffusiveFluxBC` *in case the feature* `outpres` *is selected. Lines 50–54 describe the delta* `d_wall`*, which adds* `BC(Wall)` *as input of the functions* `convectiveFluxBC` *and* `diffusiveFluxBC` *in case the feature* `wallslip` *is selected. Lines 57–59 state that if the user want to also compute the diffusive part of the flux (i.e., if the feature* `diffusive` *is selected), the function* `fluxBalance` *now takes two more arguments:* `Fxd` *and* `FxdBC`*. Lines 60–68 state that if the user wants to compute the flux with an order 2 precision, i.e., when the feature* `order2` *is selected, the functions* `convectiveFlux` *and* `convectiveFluxBC` *now take also* `Grad(Primitive)` *in arguments, and the functions* `diffusiveFlux` *and* `diffusiveFluxBC` *now take also* `Grad(Grad(Primitive))` *in arguments.*

Finally, lines 70–74 define the previously described activation conditions of the deltas. No order between deltas is specified: there are no restrictions on the order in which they can be applied.

4. Static Analysis

This section describes the different analyses ensuring the correct definition of an SPL L. These analyses are structured in three categories: the first analysis ensures that all products generate a variant by checking that the application conditions of all delta in L are validated; the second set of analyses ensures that a generated variant is well constructed, i.e., all the input and output terms are well sorted, with sort `Value`, and the constraints modelled by Equations 1 and 2 are validated; and, finally, the last analysis ensures that a dataflow can be generated from a variant by ensuring that the corresponding rewriting rules always terminate.

Most of these analyses are inspired by [18] and follow the same principle of generating a SAT constraint that is valid if and only if the analysed property holds. The exception is the analysis of termination, which generates a universally quantified SAT constraint. Moreover, like in [18], our analysis is based on the *type uniformity* guideline, which is stated as follows in our context.

Type Uniformity Guideline Ensure that every time a variable v is declared or added, it always has the same sort.

This guideline ensures that in all variants, given that every used variable is declared, the analysis checking that the functions' inputs and outputs are well sorted, with sort `Value`, is always the same and can be performed on the SPL directly with the rules already given in Figure 7. That way, we reduce checking the term well-sortedness in variants into checking that all used variables are declared.

Example 5 (Type uniformity)**.** *Our complete running example presented in Listing 7 contains only one variable* `valueV`*, declared in the base artifact with sort* `Value`*. Hence this variable always has the same sort every time it is declared, and so our running example is type uniform.*

To simplify the presentation of our analysis, we consider in the rest of this section that the base artifact of an SPL is modelled by a delta named base which is always activated and always applied before the other deltas. Before we present the different analyses, we introduce a set of getters on top of which these different analyses are constructed.

4.1. Getters on SPLs

All our analyses are based on only two sets of getters. First, we have getters that introspect the variability of an SPL.

Notation 1. *Given an SPL L, we denote by $fm(L)$ the constraint over feature names corresponding to the feature model of the SPL. Moreover, we denote by $act(L)$ the constraint stating that every delta name in L is equivalent to its activation condition.*

Second, we need to relate the SPL variability to the variable names, function names, inputs, and outputs, which are manipulated during the application of the SPL's deltas. So, we first define the notion of *path* to have a common notation for all these manipulated elements, and then introduce the three getters we use.

Definition 6 (Paths in an SPL)**.** *A* path *is either a variable name v, a function name f, or a word of the form $f.field\_t.T$, where f is a function name; $field\_t$ is an element of $\{\mathtt{input}, \mathtt{output}\}$; and T is a term of sort* `Value`*.*

For a specification SPL L, we denote by $\mathtt{P}(L)$ the set of paths occurring in L.

Definition 7 (Getters on paths)**.** *Given a specification SPL L and a path $\rho \in \mathtt{P}(L)$, we denote by:*

- $\mathtt{add}(L, \rho)$ *the set of delta names d that add the path ρ in L;*
- $\mathtt{rem}(L, \rho)$ *the set of delta names d that remove the path ρ in L;*
- $\mathtt{mod}(L, \rho)$ *the set of delta names d that modify the path ρ in L.*

Moreover, given a path ρ, we denote by $prefix(\rho)$ the set of prefixes of ρ.

Example 6 (Paths and getters)**.** *We illustrate the previous two definitions by giving the value of the path getters for the running example L presented in Listing 7. Since this list is long, we split it into 13 different parts.*

1. *The variable* `valueV` *corresponds to the path* `valueV`*, and since that variable is declared in the base artifact without ever being manipulated, we have* $\mathtt{add}(L, \mathtt{valueV}) = \{\mathtt{base}\}$ *and* $\mathtt{rem}(L, \mathtt{valueV}) = \mathtt{mod}(L, \mathtt{valueV}) = \emptyset$.

2. *The function* `primitive` *corresponds to the three paths*

   `primitive`, `primitive.input.Conservative`, `primitive.output.Primitive`

   and since that function is declared in the base artifact without ever being manipulated, we have $\mathtt{add}(L, \rho) = \{\mathtt{base}\}$ *and* $\mathtt{rem}(L, \rho) = \mathtt{mod}(L, \rho) = \emptyset$ *for* ρ *being any of these paths.*

3. *The function* `gradient` *corresponds to the three paths*

   `gradient`, `gradient.input.valueV`, `gradient.output.Grad(valueV)`

   and since that function is declared in the base artifact without ever being manipulated, we have $\mathtt{add}(L, \rho) = \{\mathtt{base}\}$ *and* $\mathtt{rem}(L, \rho) = \mathtt{mod}(L, \rho) = \emptyset$ *for* ρ *being any of these paths.*

4. *The function* `pressure` *corresponds to the three paths*

   `pressure`, `pressure.input.Primitive`, `pressure.output.Pressure`

   and since that function is declared in the base artifact without ever being manipulated, we have $\mathtt{add}(L, \rho) = \{\mathtt{base}\}$ *and* $\mathtt{rem}(L, \rho) = \mathtt{mod}(L, \rho) = \emptyset$ *for* ρ *being any of these paths.*

5. *The function* `convectiveFlux` *corresponds to the four paths*

   `convectiveFlux`, `convectiveFlux.input.Primitive`,
   `convectiveFlux.input.Grad(Primitive)`, `convectiveFlux.output.Fxc`

   This function is first declared in the base artifact, and then modified by the `d_order2` *delta, which gives:*

   - $\mathtt{add}(L, \rho) = \{\mathtt{base}\}$ *for*

     $$\rho \in \left\{ \begin{array}{c} \mathtt{convectiveFlux}, \mathtt{convectiveFlux.input.Primitive}, \\ \mathtt{convectiveFlux.output.Fxc} \end{array} \right\}$$

   - $\mathtt{add}(L, \mathtt{convectiveFlux.input.Grad(Primitive)}) = \{\mathtt{d\_order2}\}$
   - $\mathtt{rem}(L, \rho) = \emptyset$ *for* ρ *being any path related to* `convectiveFlux`
   - $\mathtt{mod}(L, \mathtt{convectiveFlux}) = \{\mathtt{d\_order2}\}$

6. *The function* `diffusiveFlux` *corresponds to the five paths*

$$\texttt{diffusiveFlux}, \texttt{diffusiveFlux.input.Primitive}, \\ \texttt{diffusiveFlux.input.Grad(Primitive)}, \\ \texttt{diffusiveFlux.input.Grad(Grad(Primitive))}, \texttt{convectiveFlux.output.Fxd}$$

This function is first declared in the base artifact, and then modified by the `d_order2` *delta, which gives:*

- $\texttt{add}(L, \rho) = \{\texttt{base}\}$ *for*

$$\rho \in \left\{ \begin{array}{c} \texttt{diffusiveFlux}, \texttt{diffusiveFlux.input.Primitive}, \\ \texttt{diffusiveFlux.input.Grad(Primitive)}, \texttt{diffusiveFlux.output.Fxd} \end{array} \right\}$$

- $\texttt{add}(L, \texttt{diffusiveFlux.input.Grad(Grad(Primitive))}) = \{\texttt{d\_order2}\}$
- $\texttt{rem}(L, \rho) = \emptyset$ *for* ρ *being any path related to* `diffusiveFlux`
- $\texttt{mod}(L, \texttt{diffusiveFlux}) = \{\texttt{d\_order2}\}$

7. *The function* `inlet` *corresponds to the three paths*

   $$\texttt{inlet}, \texttt{inlet.input.Primitive}, \texttt{inlet.output.BC(Inlet)}$$

   and since that function is declared in the base artifact without ever being manipulated, we have $\texttt{add}(L, \rho) = \{\texttt{base}\}$ *and* $\texttt{rem}(L, \rho) = \texttt{mod}(L, \rho) = \emptyset$ *for* ρ *being any of these paths.*

8. *The function* `outpres` *corresponds to the three paths*

   $$\texttt{outpres}, \texttt{outpres.input.Primitive}, \texttt{outpres.output.BC(Outpres)}$$

   and since that function is declared in the base artifact without ever being manipulated, we have $\texttt{add}(L, \rho) = \{\texttt{base}\}$ *and* $\texttt{rem}(L, \rho) = \texttt{mod}(L, \rho) = \emptyset$ *for* ρ *being any of these paths.*

9. *The function* `wall` *corresponds to the three paths*

   $$\texttt{wall}, \texttt{wall.input.Primitive}, \texttt{wall.output.BC(Wall)}$$

   and since that function is declared in the base artifact without ever being manipulated, we have $\texttt{add}(L, \rho) = \{\texttt{base}\}$ *and* $\texttt{rem}(L, \rho) = \texttt{mod}(L, \rho) = \emptyset$ *for* ρ *being any of these paths.*

10. *The function* `convectiveFluxBC` *corresponds to the seven paths*

    $$\texttt{convectiveFluxBC}, \texttt{convectiveFluxBC.input.Primitive}, \\ \texttt{convectiveFluxBC.input.Grad(Primitive)}, \\ \texttt{convectiveFluxBC.input.BC(Inlet)}, \\ \texttt{convectiveFluxBC.input.BC(Outpres)}, \texttt{convectiveFluxBC.input.BC(Wall)}, \\ \texttt{convectiveFluxBC.output.FxcBC}$$

    This function is first declared in the base artifact, and then modified by the `d_inlet`, `d_outpres`, `d_wall` *and* `d_order2`, *deltas, which gives:*

- $\texttt{add}(L,\rho) = \{\texttt{base}\}$ *for*

$$\rho \in \left\{ \begin{array}{c} \texttt{convectiveFluxBC}, \texttt{convectiveFluxBC.input.Primitive}, \\ \texttt{convectiveFluxBC.output.FxcBC} \end{array} \right\}$$

- $\texttt{add}(L, \texttt{convectiveFluxBC.input.BC(Inlet)}) = \{\texttt{d\_inlet}\}$
- $\texttt{add}(L, \texttt{convectiveFluxBC.input.BC(Outpres)}) = \{\texttt{d\_outpres}\}$
- $\texttt{add}(L, \texttt{convectiveFluxBC.input.BC(Wall)}) = \{\texttt{d\_wall}\}$
- $\texttt{add}(L, \texttt{convectiveFluxBC.input.Grad(Primitive)}) = \{\texttt{d\_order2}\}$
- $\texttt{rem}(L,\rho) = \emptyset$ *for* ρ *being any path related to* `convectiveFluxBC`
- $\texttt{mod}(L, \texttt{convectiveFluxBC}) = \{\texttt{d\_inlet}, \texttt{d\_outpres}, \texttt{d\_wall}, \texttt{d\_order2}\}$

11. *The function* `diffusiveFluxBC` *corresponds to the eight paths*

`diffusiveFluxBC`, `diffusiveFluxBC.input.Primitive`,
`diffusiveFluxBC.input.Grad(Primitive)`,
`diffusiveFluxBC.input.Grad(Grad(Primitive))`,
`diffusiveFluxBC.input.BC(Inlet)`, `diffusiveFluxBC.input.BC(Outpres)`,
`diffusiveFluxBC.input.BC(Wall)`, `diffusiveFluxBC.output.FxcBC`

This function is first declared in the base artifact, and then modified by the `d_inlet`, `d_outpres`, `d_wall` *and* `d_order2`, *deltas, which gives:*

- $\texttt{add}(L,\rho) = \{\texttt{base}\}$ *for*

$$\rho \in \left\{ \begin{array}{c} \texttt{diffusiveFluxBC}, \texttt{diffusiveFluxBC.input.Primitive}, \\ \texttt{diffusiveFluxBC.input.Grad(Primitive)}, \\ \texttt{diffusiveFluxBC.output.FxcBC} \end{array} \right\}$$

- $\texttt{add}(L, \texttt{diffusiveFluxBC.input.BC(Inlet)}) = \{\texttt{d\_inlet}\}$
- $\texttt{add}(L, \texttt{diffusiveFluxBC.input.BC(Outpres)}) = \{\texttt{d\_outpres}\}$
- $\texttt{add}(L, \texttt{diffusiveFluxBC.input.BC(Wall)}) = \{\texttt{d\_wall}\}$
- $\texttt{add}(L, \texttt{diffusiveFluxBC.input.Grad(Grad(Primitive))}) = \{\texttt{d\_order2}\}$
- $\texttt{rem}(L,\rho) = \emptyset$ *for* ρ *being any path related to* `convectiveFluxBC`
- $\texttt{mod}(L, \texttt{diffusiveFluxBC}) = \{\texttt{d\_inlet}, \texttt{d\_outpres}, \texttt{d\_wall}, \texttt{d\_order2}\}$

12. *The function* `fluxBalance` *corresponds to the six paths*

`fluxBalance`, `fluxBalance.input.Fxc`, `fluxBalance.input.Fxd`,
`fluxBalance.input.FxcBC`, `fluxBalance.input.FxdBC`,
`fluxBalance.output.Balance`

This function is first declared in the base artifact, and then modified by the `d_diffusive` *delta, which gives:*

- $\texttt{add}(L, \rho) = \{\texttt{base}\}$ *for*

$$\rho \in \left\{ \begin{array}{c} \texttt{fluxBalance}, \texttt{fluxBalance.input.Fxc}, \\ \texttt{fluxBalance.input.FxcBC}, \texttt{convectiveFlux.output.Balance} \end{array} \right\}$$

- $\texttt{add}(L, \rho) = \{\texttt{d\_diffusive}\}$ *for*

$$\rho \in \{\texttt{fluxBalance.input.Fxd}, \texttt{fluxBalance.input.FxdBC}\}$$

- $\texttt{rem}(L, \rho) = \emptyset$ *for* ρ *being any path related to* `fluxBalance`
- $\texttt{mod}(L, \texttt{fluxBalance}) = \{\texttt{d\_diffusive}\}$

13. *Finally, the function* `explicitIncrement` *corresponds to the three paths*

    `explicitIncrement`, `explicitIncrement.input.Balance`,
    `explicitIncrement.output.Rhs`

    and since that function is declared in the base artifact without ever being manipulated, we have $\texttt{add}(L, \rho) = \{\texttt{base}\}$ *and* $\texttt{rem}(L, \rho) = \texttt{mod}(L, \rho) = \emptyset$ *for* ρ *being any of these paths.*

4.2. Applicability Constraints

Applicability corresponds to the fact that delta operations do not fail (i.e., they all can be applied). As previously stated, this analysis corresponds to the generation of a constraint, which comprises three validation parts: checking if delta operations adding a path are valid; checking if delta operations removing a path are valid; and checking if delta operations modifying a path are valid.

4.2.1. Addition Operation

Given a specification SPL L, the constraint for checking that no addition operation of a path $\rho \in \texttt{P}(L)$ fails is as follows:

$$\mathit{predADD}(L, \rho) = \bigwedge_{d \neq d'} \left(d \wedge d' \Rightarrow \bigvee_{d''} d'' \right)$$

$$\text{with } \left\{ \begin{array}{l} d, d' \in \texttt{add}(L, \rho), d'' \in \bigcup_{\rho' \in \mathit{prefix}(\rho)} \texttt{rem}(L, \rho') \\ \text{and } d' \prec d'' \prec d \end{array} \right.$$

This constraint states that if two deltas add the same path, then there must be a third one in between that removes it.

Example 7 ($\mathit{predADD}(L, \rho)$ constraint)**.** *Consider the running example L presented in Listing 7: since each path* ρ *in this product line is introduced only once,* $\texttt{add}(L, \rho)$ *is a singleton. Hence,* $\mathit{predADD}(L, \rho)$ *is true for every path in* L.

4.2.2. Removal Operation

Given a specification SPL L, the constraint for checking that no removal operation of a path $\rho \in \mathtt{P}(L)$ fails is as follows:

$$predREM(L,\rho) = \bigwedge_{d} \left(d \Rightarrow \Big(\bigvee_{d''} \big(d'' \wedge \bigwedge_{d'} \neg d' \big) \Big) \right)$$
$$\text{with } \begin{cases} d \in \mathtt{rem}(L,\rho), d' \in \bigcup_{\rho' \in prefix(\rho)} \mathtt{rem}(L,\rho'), d'' \in \mathtt{add}(L,\rho) \\ \text{and } d'' \prec d' \prec d \end{cases}$$

This constraint states that for a removal operation to succeed (in delta d), there must be a previous delta d'' that added the path to remove, with no other delta d' in between removing it first.

Example 8 ($predREM(L,\rho)$ constraint)**.** *Consider the running example L presented in Listing 7: since this example does not contain any removal operation (i.e.,* $\mathtt{rem}(L,\rho) = \emptyset$ *for all path* ρ *in L),* $predREM(L,\rho)$ *is true for every path* ρ *in L.*

4.2.3. Modification Operation

Given a specification SPL L, the constraint for checking that no modification operation of a path $\rho \in \mathtt{P}(L)$ fails is as follows:

$$predMOD(L,\rho) = \bigwedge_{d} \left(d \Rightarrow \Big(\bigvee_{d''} \big(d'' \wedge \bigwedge_{d'} \neg d' \big) \Big) \right)$$
$$\text{with } \begin{cases} d \in \mathtt{mod}(L,\rho), d' \in \bigcup_{\rho' \in prefix(\rho)} \mathtt{rem}(L,\rho'), d'' \in \mathtt{add}(L,\rho) \\ \text{and } d'' \prec d' \prec d \end{cases}$$

This constraint has the same structure as $predREM(L,\rho)$ before: for a modification operation to succeed (in delta d), there must be a previous delta d'' that added the path to remove, with no other delta d' in between removing it first.

Example 9 ($predMOD(L,\rho)$ constraint)**.** *Consider the running example L presented in Listing 7. In Example 6, we have seen that several paths corresponding to functions are modified. And since all functions are declared in the base artifact, which is always executed before any delta, we thus have the following equalities:*

$$predMOD(L, \mathtt{convectiveFlux}) = (\mathtt{d\_order2} \Rightarrow \mathtt{base})$$
$$predMOD(L, \mathtt{diffusiveFlux}) = (\mathtt{d\_order2} \Rightarrow \mathtt{base})$$
$$predMOD(L, \mathtt{convectiveFluxBC}) = \begin{pmatrix} (\mathtt{d\_inlet} \Rightarrow \mathtt{base}) \wedge (\mathtt{d\_outpres} \Rightarrow \mathtt{base}) \\ \wedge (\mathtt{d\_wall} \Rightarrow \mathtt{base}) \wedge (\mathtt{d\_order2} \Rightarrow \mathtt{base}) \end{pmatrix}$$
$$predMOD(L, \mathtt{diffusiveFluxBC}) = \begin{pmatrix} (\mathtt{d\_inlet} \Rightarrow \mathtt{base}) \wedge (\mathtt{d\_outpres} \Rightarrow \mathtt{base}) \\ \wedge (\mathtt{d\_wall} \Rightarrow \mathtt{base}) \wedge (\mathtt{d\_order2} \Rightarrow \mathtt{base}) \end{pmatrix}$$
$$predMOD(L, \mathtt{fluxBalance}) = (\mathtt{d\_diffusive} \Rightarrow \mathtt{base})$$

4.2.4. Full Applicability Constraint

We can now combine all the previous constraints to ensure that all delta operations are valid:

$$\mathit{predAPP}(L) = \bigwedge_{\rho \in \mathrm{P}(L)} \Big(\mathit{predADD}(L, \rho) \wedge \mathit{predREM}(L, \rho) \wedge \mathit{predMOD}(L, \rho) \Big)$$

Finally, we can bind this constraint to the variability model of the SPL to obtain the formula

$$(\mathit{fm}(L) \wedge \mathit{act}(L)) \Rightarrow \mathit{predAPP}(L)$$

This formula states that if we take a product p (i.e., a model of $\mathit{fm}(L)$), and extend it to the set of delta's activated by p (i.e., a model of $\mathit{act}(L)$), then if the resulting model validates the constraints, then all delta operations triggered by the product p will succeed, i.e., the corresponding variant can be generated. This property is formalized in the following theorem.

Theorem 1. (Applicability consistency). *Consider an SPL L and the following two properties on L:*

1. *The constraint $(\mathit{fm}(L) \wedge \mathit{act}(L)) \Rightarrow \mathit{predAPP}(L)$ is valid.*
2. *All variants of L can be generated.*

Then Property 1 is equivalent to Property 2.

Proof. See Appendix A.3. □

Example 10 (Applicability consistency)**.** *We illustrate Theorem 1 by using the running example L presented in Listing 7. In Examples 7 and 8 we have seen that $\mathit{predADD}(L, \rho)$ and $\mathit{predREM}(L, \rho)$ are valid for all paths ρ. With Example 9, we thus have that*

$$\mathit{predAPP}(L) = \begin{pmatrix} \mathit{predMOD}(L, \texttt{convectiveFlux}) \wedge \mathit{predMOD}(L, \texttt{diffusiveFlux}) \\ \wedge \mathit{predMOD}(L, \texttt{convectiveFluxBC}) \\ \wedge \mathit{predMOD}(L, \texttt{diffusiveFluxBC}) \wedge \mathit{predMOD}(L, \texttt{fluxBalance}) \end{pmatrix}$$

By removing duplicate implications, we thus have

$$\begin{aligned} \mathit{predAPP}(L) = {} & (\texttt{d\_inlet} \Rightarrow \texttt{base}) \wedge (\texttt{d\_outpres} \Rightarrow \texttt{base}) \\ & \wedge (\texttt{d\_wall} \Rightarrow \texttt{base}) \wedge (\texttt{d\_order2} \Rightarrow \texttt{base}) \wedge (\texttt{d\_diffusive} \Rightarrow \texttt{base}) \end{aligned}$$

On the other hand, we have by definition:

$$\begin{aligned} \mathit{fm}(L) &= \texttt{solver} \wedge \texttt{mesh} \wedge \texttt{boundary} \wedge \texttt{zone} \wedge \texttt{model} \wedge \texttt{convective} \\ & \quad \wedge (\texttt{inlet} \vee \texttt{outpres} \vee \texttt{wall}) \\ \mathit{act}(L) &= \texttt{base} \wedge (\texttt{d\_inlet} \Leftrightarrow \texttt{inlet}) \wedge (\texttt{d\_outpres} \Leftrightarrow \texttt{outpres}) \wedge (\texttt{d\_wall} \Leftrightarrow \texttt{wall}) \\ & \quad \wedge (\texttt{d\_order2} \Leftrightarrow \texttt{order2}) \wedge (\texttt{d\_diffusive} \Leftrightarrow \texttt{diffusive}) \end{aligned}$$

Hence, looking at the definition of the constraint in Theorem 1, since $\mathit{act}(L)$ (in the left hand side of the implication) selects the Boolean variable `base`*, all* *implications in the right hand side are satisfied. This constraint is thus valid, and indeed, we can see that every product of the SPL can be generated.*

Now consider an erroneous definition of the SPL: suppose that the `diffusiveFlux` *function is declared in the* `d_diffusive` *delta instead of in the base artifact. This changes the path getters into* $\texttt{add}(L,\rho) = \{\texttt{d\_diffusive}\}$ *for*

$$\rho \in \left\{ \begin{array}{c} \texttt{diffusiveFlux}, \texttt{diffusiveFlux.input.Primitive}, \\ \texttt{diffusiveFlux.input.Grad(Primitive)}, \texttt{diffusiveFlux.output.Fxd} \end{array} \right\}$$

But more importantly in our case, $\mathit{predMOD}(L, \texttt{diffusiveFlux})$ *is modified into*

$$\texttt{d\_order2} \Rightarrow \texttt{False}$$

since the delta that adds `diffusiveFlux` *(i.e.,* `d_diffusive`*) may not be applied before* `d_order2`*. Consequently, the contraint in Theorem 1 is not valid in this case, because the Boolean variable* `d_order2` *may be selected in the left hand side of the implication which leads the right hand side to be false (and indeed, the variant generation will fail for any product with the feature* `d_order2` *selected).*

4.3. Specification Validation

The analyses presented in this section check the correct definition of the generated variants, i.e., if the input and output terms are well sorted, and if the function specifications validate Equation 1 and optionally Equation 2. Since these analyses manipulate the paths and the variables that are present in a variant, we first define several constraints that state the presence status of these different elements in a variant.

4.3.1. Path Presence

Given a specification SPL L, the fact that a path $\rho \in \texttt{P}(L)$ is present in a variant is given by the following constraint:

$$\mathit{Pre}(L,\rho) = \bigvee_{d} \Big(d \wedge \big(\bigwedge_{d'} \neg d \big) \Big) \text{ with } \left\{ \begin{array}{l} d \in \texttt{add}(L,\rho), d' \in \bigcup_{\rho' \in \mathit{prefix}(\rho)} \texttt{rem}(L,\rho') \\ \text{and } d \prec d' \end{array} \right.$$

This constraint states that for ρ to exist, a delta must add it with no delta removing it later. Furthermore, we denote by $\mathit{input}(L,f)$ (by $\mathit{output}(L,f)$, respectively) the set $\{T \mid f.\texttt{input}.T \in \texttt{P}(L)\}$ (the set $\{T \mid f.\texttt{output}.T \in \texttt{P}(L)\}$, respectively).

Example 11 ($\mathit{Pre}(L,\rho)$ constraint)**.** *Since our running example L in Listing 7 has no remove operation, and since* $\texttt{add}(L,\rho)$ *is a singleton for any path ρ, we have that $\mathit{Pre}(L,\rho)$ is equal to the name of the delta adding ρ for any path ρ. For instance,* $\mathit{Pre}(L, \texttt{convectiveFlux.input.Grad(Primitive)}) = \texttt{d\_order2}$ *and* $\mathit{Pre}(L, \texttt{valueV}) = \texttt{base}$*.*

4.3.2. Variable Presence

Given a specification SPL L, for all function names $f \in \mathtt{P}(L)$, we define the set of term variables as follows:

$$\mathtt{fv}(L, f) = \bigcup_{ft \in \{\mathtt{input}, \mathtt{output}\}} \{fv(T) \mid f.ft.T \in \mathtt{P}(L)\}$$

For all $v \in \mathtt{fv}(L, f)$, we define:

$$PrI(L, f, v) = \bigvee_{T \in input(L,f) \wedge v \in fv(T)} Pre(L, f.\mathtt{input}.T)$$

$$PrO(L, f, v) = \bigvee_{T \in output(L,f) \wedge v \in fv(T)} Pre(L, f.\mathtt{output}.T)$$

$$AbsO(L, f, v) = \bigvee_{T \in output(L,f) \wedge v \notin fv(T)} Pre(L, f.\mathtt{output}.T)$$

Here, $PrI(L, f, v)$ states when the variable v is present in an input of f; $PrO(L, f, v)$ states when the variable v is present in an output of f; and, finally, $PrO(L, f, v)$ states whether there are outputs of f that do not contain the variable v.

Example 12 (Variable presence constraints)**.** *Looking at Example 6, with L being the running example in Listing 7, we have that*

$$\mathtt{fv}(L, \rho) = \begin{cases} \{\mathtt{valueV}\} & \text{if } \rho = \mathtt{gradient} \\ \emptyset & \text{else} \end{cases}$$

Consequently, the getters PrI, PrO, and AbsO are only defined on the pair `(gradient, valueV)` *and we have:*

$$PrI(L, \mathtt{gradient}, \mathtt{valueV}) = PrO(L, \mathtt{gradient}, \mathtt{valueV}) = \mathtt{base}$$
$$AbsO(L, \mathtt{gradient}, \mathtt{valueV}) = \mathtt{False}$$

4.3.3. Validating Function Specifications

Our first analysis in this section ensures that all declarations in a variant are well sorted. First, we need to ensure that all declared functions are in the signature F, sorted with `FunctionID`. This check does not depend on the variability, and can be done by simply parsing the SPL and checking that every function name in the SPL is declared in F with the correct sort.

Second, we need to check that the inputs and outputs of every function specification are well sorted, with sort `Value`. As discussed in the beginning of this section, we use the *type uniformity* guideline to reduce this check to two simpler tests: first, we use the rules in Figure 7 to check the well sortedness of the inputs and outputs on the SPL directly; and second, we check that the variables used in any term present in a variant are declared in that variant. This second test is modelled by the following constraint:

$$
\begin{aligned}
\mathit{decl}(L) = \bigwedge_{f \in \mathtt{P}(L)} \Bigg(&\Big(\bigwedge_{T \in \mathit{input}(L,f)} \mathit{Pre}(L, f.\mathtt{input}.T) \Rightarrow \bigwedge_{v \in \mathit{fv}(T)} \mathit{Pre}(L, v) \Big) \\
&\wedge \Big(\bigwedge_{T \in \mathit{output}(L,f)} \mathit{Pre}(L, f.\mathtt{output}.T) \Rightarrow \bigwedge_{v \in \mathit{fv}(T)} \mathit{Pre}(L, v) \Big) \Bigg)
\end{aligned}
$$

The following theorem states the property expressed by the *decl* predicate.

Theorem 2. (Variable presence). *Consider an SPL L such that all variants are generable. Moreover, consider the following two properties on L:*

1. *The constraint $(\mathit{fm}(L) \wedge \mathit{act}(L)) \Rightarrow \mathit{decl}(L)$ is valid.*
2. *All variants of L are such that all their variables are declared.*

Then Property 1 is equivalent to Property 2.

Proof. See Appendix A.5. □

Example 13 (Variable presence)**.** *We illustrate Theorem 2 using the running example L presented in Listing 7. From our discussion in Example 12, we can see that*

$$
\begin{aligned}
\mathit{decl}(L) = &(\mathit{Pre}(L, \mathtt{gradient.input.valueV}) \Rightarrow \mathit{Pre}(L, \mathtt{valueV})) \\
&\wedge (\mathit{Pre}(L, \mathtt{gradient.output.Grad(valueV)}) \Rightarrow \mathit{Pre}(L, \mathtt{valueV}))
\end{aligned}
$$

From Example 11, we can apply the definition of Pre to get

$$\mathit{decl}(L) = (\mathtt{base} \Rightarrow \mathtt{base}) \wedge (\mathtt{base} \Rightarrow \mathtt{base})$$

which is valid, and so the constraint in Theorem 2 is also valid.

If instead the variable `valueV` *were declared in the* `d_order2` *delta, $\mathit{decl}(L)$ would have been equal to* `d_order2` $\Rightarrow$ `base`*. And since $\mathit{act}(L)$ states that* `base` *is always selected while* `d_order2` *is not, the constraint in Theorem 2 would not be valid (and indeed, any variant generated from a product without* `d_order2` *selected would be erroneous).*

4.3.4. Validating Equation 1

The second analysis of this section ensures that Equation 1 is validated by every function in every variant of the SPL. As before, we define this analysis with the constrution of a SAT constraint, namely, given a specification SPL L, we define the following formula:

$$\mathit{nFree}(L) \quad = \quad \bigwedge_{f \in \mathtt{P}(L)} \bigwedge_{v \in \mathtt{fv}(L,f)} \Big(\mathit{PrI}(L, f, v) \quad \Rightarrow \quad \big(\mathit{PrO}(L, f, v) \wedge \neg \mathit{AbsO}(L, f, v) \big) \Big)$$

This constraint states that if a function f has an input or attribute T, then the variables of T must be present in one output of f. The property expressed by this constraint is stated in the following theorem.

Theorem 3. (Input variable relevance). *Consider an SPL L such that all variants are generable. Moreover, consider the following two properties on L:*

1. *The constraint $(\mathit{fm}(L) \wedge \mathit{act}(L)) \Rightarrow \mathit{nFree}(L)$ is valid.*
2. *All variants of L validate Equation 1 from Section 3.4.2.*

Then Property 1 is equivalent to Property 2.

Proof. See Appendix A.5. □

Example 14 (Input variable relevance)**.** *We illustrate Theorem 3 using the running example L presented in Listing 7. From our discussion in Example 12, we can see that*

$$\begin{aligned}
\mathit{nFree}(L) &= (\mathit{PrI}(L, \texttt{gradient}, \texttt{valueV}) \Rightarrow \\
&\qquad (\mathit{PrO}(L, \texttt{gradient}, \texttt{valueV}) \wedge \neg \mathit{AbsO}(L, \texttt{gradient}, \texttt{valueV}))) \\
&= \texttt{base} \Rightarrow (\texttt{base} \wedge \neg \texttt{False}) \\
&\equiv \texttt{True}
\end{aligned}$$

Consequently, the constraint in Theorem 3 is valid.

Suppose now that another variable `valueVV` *of sort* `Value` *is declared in the delta* `d_order2`*, and that the* `gradient` *function has a second output* `valueVV` *added in the delta* `d_order2`*. Since* `valueVV` *never appears in the input of* `gradient`*, we have that*

$$\begin{aligned}
&\mathit{PrI}(L, \texttt{gradient}, \texttt{valueVV}) = \texttt{False} \\
&\mathit{PrO}(L, \texttt{gradient}, \texttt{valueVV}) = \texttt{d_order2} \\
&\mathit{AbsO}(L, \texttt{gradient}, \texttt{valueVV}) = \texttt{base}
\end{aligned}$$

In this case, we thus have

$$\begin{aligned}
\mathit{nFree}(L) &= (\mathit{PrI}(L, \texttt{gradient}, \texttt{valueV}) \Rightarrow \\
&\qquad (\mathit{PrO}(L, \texttt{gradient}, \texttt{valueV}) \wedge \neg \mathit{AbsO}(L, \texttt{gradient}, \texttt{valueV}))) \\
&\quad \wedge (\mathit{PrI}(L, \texttt{gradient}, \texttt{valueVV}) \Rightarrow \\
&\qquad (\mathit{PrO}(L, \texttt{gradient}, \texttt{valueVV}) \wedge \neg \mathit{AbsO}(L, \texttt{gradient}, \texttt{valueVV}))) \\
&= (\texttt{base} \Rightarrow (\texttt{base} \wedge \neg \texttt{False})) \wedge (\texttt{False} \Rightarrow (\texttt{d_order2} \wedge \neg \texttt{base})) \\
&\equiv \texttt{True}
\end{aligned}$$

Consequently, the constraint in Theorem 3 is valid also in this case: indeed, the considered modification added a new output, which is transparent for Equation 1.

Suppose finally that the new variable `valueVV` *is now used as an input of the function* `gradient` *when the delta* `d_order2` *is activated. Since* `valueVV` *never* *appears in the output of* `gradient`*, we have that*

$$\begin{aligned}
&PrI(L, \texttt{gradient}, \texttt{valueVV}) = \texttt{d_order2}\\
&PrO(L, \texttt{gradient}, \texttt{valueVV}) = \texttt{False}\\
&AbsO(L, \texttt{gradient}, \texttt{valueVV}) = \texttt{base}
\end{aligned}$$

In this case, we thus have

$$\begin{aligned}
nFree(L) = {} & (PrI(L, \texttt{gradient}, \texttt{valueV}) \Rightarrow \\
& \qquad (PrO(L, \texttt{gradient}, \texttt{valueV}) \wedge \neg AbsO(L, \texttt{gradient}, \texttt{valueV}))) \\
& \quad \wedge (PrI(L, \texttt{gradient}, \texttt{valueVV}) \Rightarrow \\
& \qquad (PrO(L, \texttt{gradient}, \texttt{valueVV}) \wedge \neg AbsO(L, \texttt{gradient}, \texttt{valueVV}))) \\
= {} & (\texttt{base} \Rightarrow (\texttt{base} \wedge \neg\texttt{False})) \wedge (\texttt{d_order2} \Rightarrow (\texttt{False} \wedge \neg\texttt{base})) \\
\equiv {} & \texttt{d_order2} \Rightarrow \texttt{False}
\end{aligned}$$

Since `d_order2` *may be selected in the left hand side of the constraint in Theorem 3 (i.e.,* $fm(L) \wedge act(L)$*), this constraint is not valid in this case. And indeed, in this case, if* `d_order2` *is selected, the* `gradient` *function has an input* `valueVV` *that contains a variable that is not in its outputs, invalidating Equation 1.*

4.3.5. Validating Equation 2

This third analysis of this section ensures that Equation 2 is validated by every function in every variant of the SPL. As before, we define this analysis by the construction of a SAT constraint, namely, given a specification SPL L, we define the following formula:

$$nAmbiguous(L) = \bigwedge_{f \in \texttt{P}(L)} \bigwedge_{v \in \texttt{fv}(L,f)} \Big(PrI(L, f, v) \Leftrightarrow PrO(L, f, v) \Big)$$

This constraint states that for all functions f in a variant of the SPL, if a variable v is in an input or attribute, then it must also be in an output, and reciprocally. The property expressed by this constraint is stated in the following theorem.

Theorem 4. (Output variable dependency). *Consider an SPL L such that all variants are generable. Moreover, consider the following two properties on L:*

1. *The constraint* $(fm(L) \wedge act(L)) \Rightarrow nAmbiguous(L)$ *is valid.*
2. *All variants of L validate Equation 2 from Section 3.4.2.*

Then Property 1 is equivalent to Property 2.

Proof. See Appendix A.5. □

Example 15 (Output variable dependency)**.** *We illustrate Theorem 4 using the running example L presented in Listing 7. From our discussion in Example 12, we can see that*

$$\begin{aligned}
nAmbiguous(L) &= (PrI(L, \texttt{gradient}, \texttt{valueV}) \Leftrightarrow PrO(L, \texttt{gradient}, \texttt{valueV})) \\
&= \texttt{base} \Leftrightarrow \texttt{base} \\
&\equiv \texttt{True}
\end{aligned}$$

Consequently, the constraint in Theorem 3 is valid.

Suppose now that another variable `valueVV` *of sort* `Value` *is declared in the delta* `d_order2`*, and that the* `gradient` *function has a second output* `valueVV` *added in the delta* `d_order2`*. Since* `valueVV` *never appears in the input of* `gradient`*, we have that*

$$
\begin{aligned}
&PrI(L, \texttt{gradient}, \texttt{valueVV}) = \texttt{False}\\
&PrO(L, \texttt{gradient}, \texttt{valueVV}) = \texttt{d\_order2}\\
&AbsO(L, \texttt{gradient}, \texttt{valueVV}) = \texttt{base}
\end{aligned}
$$

In this case, we thus have

$$
\begin{aligned}
nAmbiguous(L) = &(PrI(L, \texttt{gradient}, \texttt{valueV}) \Leftrightarrow PrO(L, \texttt{gradient}, \texttt{valueV}))\\
&\wedge(PrI(L, \texttt{gradient}, \texttt{valueVV}) \Leftrightarrow PrO(L, \texttt{gradient}, \texttt{valueVV}))\\
= &(\texttt{base} \Leftrightarrow \texttt{base}) \wedge (\texttt{False} \Leftrightarrow \texttt{d\_order2})\\
\equiv &\texttt{False} \Leftrightarrow \texttt{d\_order2}
\end{aligned}
$$

Since `d_order2` *may be selected in the left hand side of the constraint in Theorem 3 (i.e., $fm(L) \wedge act(L)$), this constraint is not valid in this case. And indeed, in this case, if* `d_order2` *is selected, the* `gradient` *function has an output* `valueVV` *that contains a variable that is not in its inputs, invalidating Equation 2.*

Suppose finally that the new variable `valueVV`*is now used as an input of the function* `gradient` *when the delta* `d_order2` *is activated. Since* `valueVV` *never appears in the output of* `gradient`*, we have that*

$$
\begin{aligned}
&PrI(L, \texttt{gradient}, \texttt{valueVV}) = \texttt{d\_order2}\\
&PrO(L, \texttt{gradient}, \texttt{valueVV}) = \texttt{False}\\
&AbsO(L, \texttt{gradient}, \texttt{valueVV}) = \texttt{base}
\end{aligned}
$$

In this case, we thus have

$$
\begin{aligned}
nAmbiguous(L) = &(PrI(L, \texttt{gradient}, \texttt{valueV}) \Leftrightarrow PrO(L, \texttt{gradient}, \texttt{valueV}))\\
&\wedge(PrI(L, \texttt{gradient}, \texttt{valueVV}) \Leftrightarrow (PrO(L, \texttt{gradient}, \texttt{valueVV}))\\
= &(\texttt{base} \Leftrightarrow \texttt{base}) \wedge (\texttt{d\_order2} \Leftrightarrow \texttt{False})\\
\equiv &\texttt{d\_order2} \Leftrightarrow \texttt{False}
\end{aligned}
$$

Since `d_order2` *may be selected in the left hand side of the constraint in Theorem 3 (i.e., $fm(L) \wedge act(L)$), this constraint is not valid in this case. And indeed, in this case, if* `d_order2` *is selected, the* `gradient` *function has an input* `valueVV` *that contains a variable that is not in its outputs, invalidating Equation 2.*

4.4. Terminating Specification

Our last analysis ensures that the set of rewriting rules derived from any variant of an SPL terminates. This property implies that for any variant of an SPL and any data to compute, a corresponding dataflow model can be generated.

Our analysis is based on [19], where it is proved that the termination of a set of rewriting rules is equivalent to the existence of a *well-founded*, *weakly monotonic*, and *substitution-closed* partial order between some terms extracted from the rewriting rules. Such a partial order is defined as follows.

Definition 8 (A *well-founded*, *weakly monotonic*, and *substitution-closed* partial order)**.** *A partial order over terms $<$ is* well-founded *iff there exists no infinite sequence $(a_i)_{i\in\mathbb{N}}$ with $a_{i+1} < a_i$. Moreover, $<$ is* weakly monotonic *iff $t < t'$ implies $f(t_1,\dots,t_i,t,t_{i+1},\dots t_n) \leq f(t_1,\dots,t_i,t',t_{i+1},\dots t_n)$ for all terms $t_1,\dots,t_n \in T$. Finally, $<$ is* closed under substitution *iff for all $(l,r) \in \leq$ and all substitutions σ, it holds that $(\sigma(l),\sigma(r)) \in \leq$.*

In the following, if $<$ is well-founded, weakly monotonic*, and* substitution-closed*, then we denote this by $\mathit{WF}(<)$.*

Moreover, due to the structure of our generated rewriting rules, in our case the terms that must be ordered are the input and output of the different functions. Hence, we can define this analysis by the following constraint:

$$\begin{aligned}&\mathit{terminating}(L) = \exists <\ \ \mathit{WF}(<).\\ &\bigwedge_{f\in\mathtt{P}(L)}\ \bigwedge_{T\in \mathit{output}(L,f)}\ \bigwedge_{T'\in \mathit{input}(L,f)} \Big(\big(\mathit{Pre}(f.\mathtt{output}.T) \wedge \mathit{Pre}(f.\mathtt{input}.T')\big) \Rightarrow T' < T\Big)\end{aligned} \tag{3}$$

This constraint states that for any variant of the SPL L, there must exist a *well-founded*, *weakly monotonic*, and *substitution-closed* partial order $<$ such that if T and T' are the input and the output of a function, respectively, then $T' < T$ must hold. The property expressed by this constraint is stated in the following theorem.

Theorem 5. (Terminating specification). *Consider an SPL L such that all variants are generable and with all the variables declared. Moreover, consider the following two properties on L:*

1. *The constraint $(\mathit{fm}(L) \wedge \mathit{act}(L)) \Rightarrow \mathit{terminating}(L)$ is valid.*
2. *Each variant of L results in a terminating TRS.*

Then Property 1 is equivalent to Property 2.

Proof. See Appendix A.6. □

It is important to underline that since term rewriting is Turing complete, Equation 3 is not decidable. However, there are many sound but incomplete techniques, such as [19, 20, 21], that translate the problem of finding the partial order $<$ into SAT or into linear constraints, and these techniques typically have good results in practice. Therefore, it is possible to use these techniques to transform our constraint into an existentially quantified SAT or linear constraint problem that can be managed by existing SAT or SMT solvers.

Example 16 (Terminating specification)**.** *We illustrate Theorem 5 using the running example L presented in Listing 7. Define as follows the function* `rank` *that takes in parameter terms of sort* `Value` *and returns an integer:*

$$\texttt{rank(Conservative)} = 1 \quad \texttt{rank(Primitive)} = 2 \quad \texttt{rank(BC(Inlet))} = 2$$
$$\texttt{rank(BC(Outpres))} = 2 \quad \texttt{rank(BC(Wall))} = 2 \quad \texttt{rank(Pressure)} = 3$$
$$\texttt{rank(Grad(}T\texttt{))} = \texttt{rank(}T\texttt{)} + 1 \quad \texttt{rank(Fxc)} = 5 \quad \texttt{rank(Fxd)} = 5$$
$$\texttt{rank(FxcBC)} = 5 \quad \texttt{rank(FxdBC)} = 5 \quad \texttt{rank(Balance)} = 6$$
$$\texttt{rank(Rhs)} = 7$$

Now state that given two terms T_1 *and* T_2 *of sort* `Value`*, we have* $T_1 < T_2$ *iff both:* $fv(T_1) = fv(T_2)$ $\texttt{rank}(\sigma(T_2)) < \texttt{rank}(\sigma(T_2))$ *with* σ *mapping any variable in* $fv(T_1)$ *to* `Conservative`*.*

We can first see that $WF(<)$ *holds:*

- $<$ *is a partial order: it is indeed clearly irreflexive, asymmetric and transitive;*
- *any sequence of decreasing terms* $(T_i)_i$ *corresponds to a sequence of the same length of decreasing natural numbers* $(\texttt{rank}(T_i))_i$*, and since the order on natural numbers is well-founded, so is* $<$*;*
- *since* $\texttt{rank(Grad(}T\texttt{))} = \texttt{rank(}T\texttt{)} + 1$*, we have* $\texttt{Grad(}T_1\texttt{)} < \texttt{Grad(}T_2\texttt{)}$ *for all* $T_1 < T_2$*; moreover, since* `Grad` *is the only term constructor that has parameters,* $<$ *is weakly monotonic;*
- *finally, we can see that* $<$ *is substitution-closed by induction on the structure of the terms.*

We can also see that for each function declared in our running example, we have $T_1 < T_2$ *for any of its inputs* T_1 *and any of its outputs* T_2*:*

- *for the* `primitive` *function: we have* `Conservative` $<$ `Primitive`
- *for the* `gradient` *function: we have* `valueV` $<$ `Grad(valueV)`
- *for the* `pressure` *function: we have* `Primitive` $<$ `Pressure`
- *for the* `convectiveFlux` *function: we have*

$$\texttt{Primitive} < \texttt{Fxc}, \texttt{Grad(Primitive)} < \texttt{Fxc}$$

- *for the* `diffusiveFlux` *function: we have*

$$\texttt{Primitive} < \texttt{Fxd}, \texttt{Grad(Primitive)} < \texttt{Fxd}, \texttt{Grad(Grad(Primitive))} < \texttt{Fxd}$$

- *for the* `inlet` *function: we have* `Conservative` $<$ `BC(Inlet)`
- *for the* `outpres` *function: we have* `Conservative` $<$ `BC(Outpres)`
- *for the* `wall` *function: we have* `Conservative` $<$ `BC(Wall)`
- *for the* `convectiveFluxBC` *function: we have*

$$\texttt{BC(Inlet)} < \texttt{FxcBC}, \texttt{BC(Outpres)} < \texttt{FxcBC}, \texttt{BC(Wall)} < \texttt{FxcBC},$$
$$\texttt{Primitive} < \texttt{FxcBC}, \texttt{Grad(Primitive)} < \texttt{FxcBC}$$

- *for the* `diffusiveFluxBC` *function: we have*

$$\texttt{BC(Inlet)} < \texttt{FxdBC}, \texttt{BC(Outpres)} < \texttt{FxdBC}, \texttt{BC(Wall)} < \texttt{FxdBC},$$
$$\texttt{Primitive} < \texttt{FxdBC}, \texttt{Grad(Primitive)} < \texttt{FxdBC}, \texttt{Grad(Grad(Primitive))} < \texttt{FxdBC}$$

- *for the* `fluxBalance` *function: we have*

$$\texttt{Fxc} < \texttt{Balance}, \texttt{Fxd} < \texttt{Balance}, \texttt{FxcBC} < \texttt{Balance}, \texttt{FxdBC} < \texttt{Balance}$$

- *for the* `explicitIncrement` *function: we have* $\texttt{Balance} < \texttt{Rhs}$

Hence, terminating(L) *is valid, which implies that the constraint in Theorem 5 is valid as well. Following Theorem 5, we thus have that every dataflow generation request submitted to our running example would terminate.*

5. Empirical Evaluation

In this section, we evaluate the approach described in this paper on a prototype. Our evaluation focuses on the feasibility of dataflow generation: we evaluate the time used for the *product line flattening* and *rewriting* steps presented in Figure 1 and check if our approach is quick enough to consider it for an industrial application.

We first give some insights into our prototype, present our testing protocol and the corresponding results. We conclude by discussing the threats to the validity of our experiments.

5.1. Prototype Implementation

Our prototype was designed together with the *elsA* development team in order to evaluate if the approach proposed in this article could serve as a basis for a new CFD tool. We constructed our implementation around three design choices.

1. We first embedded in python3 the DSL described in Section 3. This choice was motivated by the fact that python3: (*i*) was already well-known by the *elsA* development team; (*ii*) is a flexible language that easily embeds DSLs; and (*iii*) can orchestrate complex and efficient libraries implemented in other languages.

2. We then used the `pydop` python library [22] to handle the variability aspect of our DSL. Indeed, this library can construct Delta-Oriented Product Lines over any python object, and was thus particularly suited for our approach, where our DSL manipulates abstract function specification and terms.

3. For the term and signature part of our DSL, we implemented an ad-hoc rewriting tool in C++. The main reason we implemented an ad-hoc tool instead of using an existing rewriting engine is because of the DAG structure of the dataflows. Indeed, existing rewriting engines like Maude

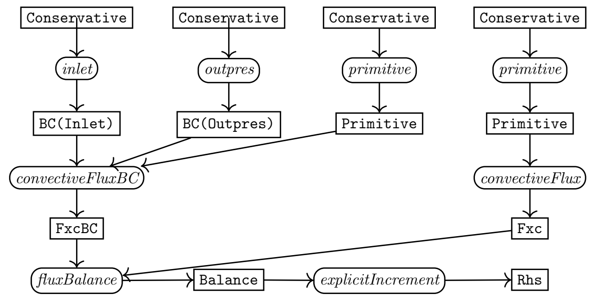

Figure 10: Tree version of the dataflow presented in Figure 3

create a tree instead of a DAG when applying the rewriting rules. For instance, Figure 10 would be the dataflow generated by Maude in place of the dataflow in Figure 3: all shared subtrees are duplicated. While this difference is not relevant semantically (two objects representing the same term are logically the same and we can easily identify the identical subtrees to construct the DAG dataflow), on large dataflows the equivalent tree version generated by existing rewriting engines would have a size that is orders of magnitude larger than the expected dataflow, and would take significantly more time to generate. Our ad-hoc tool implements a naive algorithm for rewriting rule application, but ensures that each shared term is created only once.

5.2. Testing Protocol

Since there are no standard benchmarks for dataflow generation, we constructed 597 dataflow generation problems to evaluate. We first implemented a test product line with the *elsA* development team, which contains a subpart of the configuration space available in *elsA*. This SPL extends our running example and contains 97 features, 173 functions and 1493 deltas.

Then, following our dataflow generation pipeline presented in Figure 1, every run of our prototype needs two inputs: a product and a value to compute. For the product, we use the *uniform random configuration generator* `unigen` [23, 24] to randomly pick 597 products of the test product line[1]. For the data to compute, we simply chose the value `Rhs` in all our runs.

Finally, each of the 597 runs of our prototype were executed 10 times on a single 2.5GHz Intel Xeon CPU with 32GB of memory that was hosting a CentOS 8 operating system.

[1]The version of `unigen` that was available for our tests had a bug that made the tool fail whenever we specified the number of products to generate. So we used the default behaviour of the tool, which gave us the arbitrary number of 597 generated products. We also tried to use the `smarch` tool [25], but never succeeded to compile it.

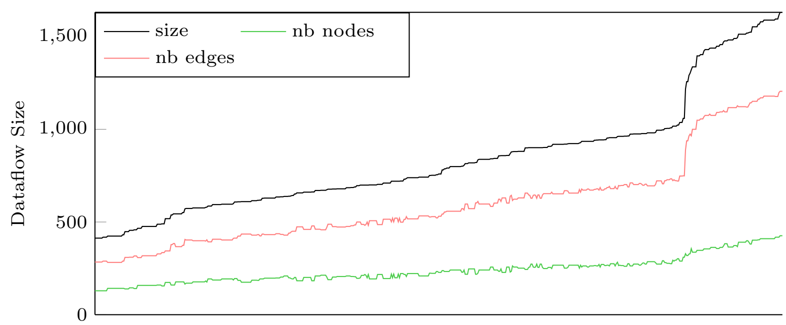

Figure 11: Size of generated Dataflows – the x-axis lists the 597 generated dataflows ordered by size (i.e., number of nodes + number of edges)

5.3. Results

To facilitate the discussion of the experiments, the figures presenting our results use a fixed ordering of the 597 dataflow generation problems we considered along the x-axis; this ordering is determined by the size (the sum of the number of nodes and the number of edges) of the generated dataflow for a given problem.

Figure 11 illustrates the size of the generated dataflows. The smallest generated dataflow has 129 nodes and 284 edges, while the largest has 426 nodes and 1205 edges. Moreover, in all dataflows, the number of edges is between two and three times the number of nodes. This confirms our concern discussed in Section 5.1 that many subtrees of the dataflows are shared, and so existing rewriting engines would not perform efficiently on these dataflow generation problems. Indeed, we computed the size of the trees these engines would have generated: they would contain between 26043 and 150983896 nodes with an average of 10 million nodes.

Figure 12 presents the average computation time for the *product line flattening* and *rewriting* steps of our prototype. The product line flattening step takes between 51ms (executing 342 deltas) and 168ms (executing 902 deltas); and the rewriting step (performing the dataflow generation itself) takes between 2ms and 8ms. The difference of execution time between these two steps can be explained by the fact that the SPL part of our prototype is implemented in python while the rewriting part is implemented in C++.

Moreover while the time taken by the *product line flattening* step is bounded for a given SPL by the time needed to execute all its deltas, the *rewriting* step can take an arbitrary amount of time, since the *Value to Compute* is arbitrary. Figure 13 shows that in our test, the execution time for this step evolves linearly w.r.t. the number of nodes in the dataflow. Hence, we believe that our approach can scale to larger dataflows.

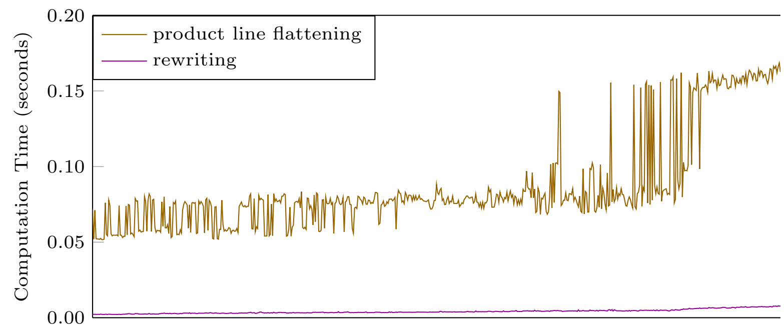

Figure 12: Computation Time w.r.t. Dataflow Size – the x-axis lists the 597 generated Dataflows ordered by size (i.e., number of nodes + number of edges)

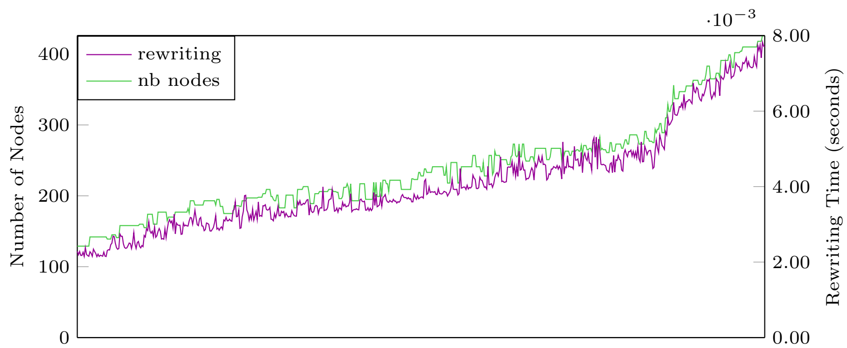

Figure 13: Rewriting Time w.r.t. Number of Nodes – the x-axis lists the 597 generated Dataflows ordered by size (i.e., number of nodes + number of edges)

5.4. Threats to Validity

We now conclude this section by discussing the external and internal threats to the validity of our experiments.

5.4.1. External Validity

The results of the evaluation strongly depend on the dataflow generation problems considered in our test protocol. Due to the lack of standard benchmarks, we only performed our tests on one SPL, on which we considered 597 randomly selected dataflow generation problems. We plan to investigate other dataflow generation problems: in particular, in addition to problems coming from CFD applications, we would like to study other application domains to get more insights. For instance, it would be interesting to investigate how the shape of the dataflows varies w.r.t. the application domain.

5.4.2. Internal Validity

Our prototype is constructed on top of two separate libraries: pydop and our ad-hoc rewriting engine. Using other existing tools, like FeatureIDE [26] (for the SPL part of our approach) and any of the existing rewriting engines [27] may affect the execution time of our approach. We plan to repeat the experiments using other tools for comparison.

6. Related Work

Dataflows have a structure that is similar to statecharts and transition systems, on top of which variability has already been defined, e.g., by using DOP on statecharts, resulting in delta-statecharts [28]; and the annotative approach on transition systems, resulting in Featured Transition Systems (FTSs) [29, 30]. While delta-statecharts and FTSs could in principle be used for modelling the dataflow model of our running example, the variation on the value of one option could have consequences all over the dataflow since a variable function could appear in many place in a dataflow. For instance, the *grad* function is variable, with different inputs and outputs, and can be used in many different tasks. Consequently, the use of delta-statecharts or FTSs would imply that the variability of *grad* must be duplicated in every task in which it is used, which is clearly not satisfactory.

Different approaches to implement SPL on specifications and code have been proposed in the literature. In our DSL we used the delta-oriented approach. We refer to a couple of surveys [31, 32] for a discussion of the different approaches.

Our approach for dataflow generation was largely inspired by work on type inhabitation [33, 34, 35], in particular [35] uses rewriting to generate terms of a given type. Indeed, if we consider that the *Value to Compute* in Figure 1 is a type, then constructing a dataflow computing this value corresponds to finding a term (i.e., a composition of functions) that has this type. Finally, in [36], the authors use type inhabitation to help programmers to use complex libraries: their tool suggests expressions of the expected type constructed from the libraries' functions.

7. Conclusion

We presented an approach to automatically generate dataflow models in an SPL setting, based on DOP and term rewriting. We provided an analysis that allows to check that for any variant of the SPL and any data to compute, a corresponding dataflow model can be generated. Moreover, we also implemented a prototype for our approach and evaluated its execution time.

In future work, we would like to address several limitations of our current approach. First, our running example considered a mesh with at most three boundaries of different types: in practice, there can be an arbitrary number of boundaries with arbitrary types. Note that this flexibility makes it so that dataflows do not have an upper bound on their size, since there is at least one task per boundary. Consequently annotative approaches on graphs like FTS, while not being satisfactory in this work, can no longer be used.

Moreover, we would like to investigate extending our DSL with the possibility to include delta operations on the $\mathcal{S}$-sorted signature. That way, we could express more easily the fact that the signature is constructed together with the rest of the variant (e.g., function declaration corresponds to adding a new term constructor of sort `FunctionID`) instead of having a signature that is the same for all variants of an SPL. Finally, we intend to conclude the evaluation of our approach by implementing and testing the analyses described in this paper.

Acknowledgements

The authors would like to thank the anonymous reviewers and Chiara Oberti for their useful and constructive comments. They also would like to thank the *elsA* development team for its work on the test product line, and in particular Bertrand Michel and Bruno Maugars. The second author acknowledges funding from the MUR PRIN 2020TL3X8X project T-LADIES (Typeful Language Adaptation for Dynamic, Interacting and Evolving Systems).

References

[1] B. L. Chamberlain, D. Callahan, H. P. Zima, Parallel programmability and the Chapel language, Int. J. High Perform. Comput. Appl. 21 (3) (2007) 291–312. doi:10.1177/1094342007078442.

[2] C. Augonnet, S. Thibault, R. Namyst, P.-A. Wacrenier, StarPU: a unified platform for task scheduling on heterogeneous multicore architectures, Concurrency Computat.: Pract. Exper. 23 (2) (2011) 187–198. doi:10.1002/cpe.1631.

[3] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, D. Fey, HPX – A Task Based Programming Model in a Global Address Space, in: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models (PGAS'14), ACM, 2014, pp. 6:1–6:11. doi:10.1145/2676870.2676883.

[4] L. V. Kale, S. Krishnan, CHARM++: A Portable Concurrent Object Oriented System Based On C++, in: Proceedings of the 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93), ACM, 1993, pp. 91–108. doi:10.1145/165854.165874.

[5] M. E. Bauer, Legion: Programming distributed heterogeneous architectures with logical regions, Ph.D. thesis, Stanford University (2014).

[6] T. Ben-Nun, J. de Fine Licht, A. N. Ziogas, T. Schneider, T. Hoefler, Stateful Dataflow Multigraphs: A Data-Centric Model for Performance Portability on Heterogeneous Architectures, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'19), ACM, 2019, pp. 81:1–81:14. doi:10.1145/3295500.3356173.

[7] O. Rausch, T. Ben-Nun, N. Dryden, A. Ivanov, S. Li, T. Hoefler, A Data-Centric Optimization Framework for Machine Learning, in: Proceedings of the International Conference on Supercomputing (ICS'22), ACM, 2022, pp. 36:1–36:13. doi:10.1145/3524059.3532364.

[8] K. M. Kavi, A. K. Deshpande, Specification of concurrent processes using a dataflow model of computation and partially ordered events, J. Syst. Softw. 16 (2) (1991) 107–120. doi:10.1016/0164-1212(91)90004-P.

[9] G. Roumage, S. Azaiez, S. Louise, A survey of main dataflow MoCCs for CPS design and verification, in: Proceedings of the 15th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC'22), IEEE, 2022, pp. 1–9. doi:10.1109/MCSoC57363.2022.00010.

[10] L. Cambier, S. Heib, S. Plot, The Onera *elsA* CFD software: input from research and feedback from industry, Mech. Ind. 14 (3) (2013) 159–174. doi:10.1051/meca/2013056.

[11] W. K. Anderson, R. T. Biedron, J.-R. Carlson, C. T. D. J. Joseph M. Derlaga, P. A. Gnoffo, D. P. Hammond, W. T. J. Kevin E. Jacobson, B. Kleb, E. M. Lee-Rausch, G. C. Nastac, M. A. P. Eric J. Nielsen, C. L. Rumsey, J. L. Thomas, A. C. W. Kyle B. Thompson, L. Wang, S. L. Wood, W. A. Wood, B. Diskin, Y. Liu, X. Zhang, FUN3D Manual: 14.0. 1, Tech. Rep. NASA/TM–20230004211, NASA (2023).
URL https://fun3d.larc.nasa.gov/papers/FUN3D_INTG_Manual-14.0.1.pdf

[12] S. Apel, D. S. Batory, C. Kästner, G. Saake, Feature-Oriented Software Product Lines: Concepts and Implementation, Springer, 2013. doi:10.1007/978-3-642-37521-7.

[13] I. Schaefer, L. Bettini, V. Bono, F. Damiani, N. Tanzarella, Delta-Oriented Programming of Software Product Lines, in: J. Bosch, J. Lee (Eds.), Proceedings of the 14th International Conference on Software Product Lines: Going Beyond (SPLC'10), Vol. 6287 of LNCS, Springer, 2010, pp. 77–91. doi:10.1007/978-3-642-15579-6_6.

[14] F. Damiani, M. Lienhardt, B. Maugars, B. Michel, Towards a Modular and Variability-Aware Aerodynamic Simulator, in: W. Ahrendt, B. Beckert, R. Bubel, E. B. Johnsen (Eds.), The Logic of Software: A Tasting Menu of Formal Methods, Vol. 13360 of LNCS, Springer, 2022, pp. 147–172. doi:10.1007/978-3-031-08166-8_8.

[15] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott (Eds.), All About Maude — A High-Performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic, Vol. 4350 of LNCS, Springer, 2007. doi:10.1007/978-3-540-71999-1.

[16] L. Bettini, F. Damiani, I. Schaefer, Compositional type checking of delta-oriented software product lines, Acta Inform. 50 (2) (2013) 77–122. doi:10.1007/s00236-012-0173-z.

[17] M. Lienhardt, D. Clarke, Conflict Detection in Delta-Oriented Programming, in: T. Margaria, B. Steffen (Eds.), Proceedings of the 5th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation: Technologies for Mastering Change (ISoLA'12), Vol. 7609 of LNCS, Springer, 2012, pp. 178–192. doi:10.1007/978-3-642-34026-0_14.

[18] F. Damiani, M. Lienhardt, On Type Checking Delta-Oriented Product Lines, in: E. Ábrahám, M. Huisman (Eds.), Proceedings of the 12th International Conference on Integrated Formal Methods (IFM'16), Vol. 9681 of LNCS, Springer, 2016, pp. 47–62. doi:10.1007/978-3-319-33693-0_4.

[19] T. Arts, J. Giesl, Termination of term rewriting using dependency pairs, Theor. Comput. Sci. 236 (1) (2000) 133–178. doi:10.1016/S0304-3975(99)00207-8.

[20] N. Hirokawa, A. Middeldorp, Polynomial Interpretations with Negative Coefficients, in: B. Buchberger, J. A. Campbell (Eds.), Proceedings of the 7th International Conference on Artificial Intelligence and Symbolic Computation (AISC'04), Vol. 3249 of LNCS, Springer, 2004, pp. 185–198. doi:10.1007/978-3-540-30210-0_16.

[21] A. Yamada, Tuple Interpretations for Termination of Term Rewriting, J. Autom. Reason. 66 (4) (2022) 667–688. doi:10.1007/s10817-022-09640-4.

[22] M. Lienhardt, PYDOP: A Generic Python Library for Delta-Oriented Programming, in: Proceedings of the 27th ACM International Systems and Software Product Line Conference (SPLC'23), Vol. B, ACM, 2023, pp. 30–33. doi:10.1145/3579028.3609011.

[23] S. Chakraborty, D. J. Fremont, K. S. Meel, S. A. Seshia, M. Y. Vardi, On Parallel Scalable Uniform SAT Witness Generation, in: C. Baier, C. Tinelli (Eds.), Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'15), Vol. 9035 of LNCS, Springer, 2015, pp. 304–319. doi:10.1007/978-3-662-46681-0_25.

[24] M. Soos, S. Gocht, K. S. Meel, Tinted, Detached, and Lazy CNF-XOR Solving and Its Applications to Counting and Sampling, in: S. K. Lahiri,

C. Wang (Eds.), Proceedings of the 32nd International Conference on Computer Aided Verification (CAV'20), Vol. 12224 of LNCS, Springer, 2020, pp. 463–484. doi:10.1007/978-3-030-53288-8_22.

[25] J. Oh, P. Gazzillo, D. Batory, M. Heule, M. Myers, Scalable Uniform Sampling for Real-World Software Product Lines (2020).
URL https://api.semanticscholar.org/CorpusID:219699491

[26] J. Meinicke, T. Thüm, R. Schröter, F. Benduhn, T. Leich, G. Saake, Mastering Software Variability with FeatureIDE, Springer, 2017. doi:10.1007/978-3-319-61443-4.

[27] H. Garavel, M. Tabikh, I. Arrada, Benchmarking Implementations of Term Rewriting and Pattern Matching in Algebraic, Functional, and Object-Oriented Languages: The 4th Rewrite Engines Competition, in: V. Rusu (Ed.), Proceedings of the 12th International Workshop on Rewriting Logic and Its Applications (WRLA'18), Vol. 11152 of LNCS, Springer, 2018, pp. 1–25. doi:10.1007/978-3-319-99840-4_1.

[28] M. Lienhardt, F. Damiani, L. Testa, G. Turin, On checking delta-oriented product lines of statecharts, Sci. Comput. Program. 166 (2018) 3–34. doi:https://doi.org/10.1016/j.scico.2018.05.007.

[29] A. Classen, M. Cordy, P. Schobbens, P. Heymans, A. Legay, J. Raskin, Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking, IEEE Trans. Softw. Eng. 39 (8) (2013) 1069–1089. doi:10.1109/TSE.2012.86.

[30] M. H. ter Beek, F. Damiani, M. Lienhardt, F. Mazzanti, L. Paolini, Efficient static analysis and verification of featured transition systems, Empir. Softw. Eng. 22 (1) (2022) 10:1–10:43. doi:10.1007/s10664-020-09930-8.

[31] I. Schaefer, R. Rabiser, D. Clarke, L. Bettini, D. Benavides, G. Botterweck, A. Pathak, S. Trujillo, K. Villela, Software diversity: state of the art and perspectives, Int. J. Softw. Tools Technol. Transf. 14 (5) (2012) 477–495. doi:10.1007/s10009-012-0253-y.

[32] T. Thüm, S. Apel, C. Kästner, I. Schaefer, G. Saake, A Classification and Survey of Analysis Strategies for Software Product Lines, ACM Comput. Surv. 47 (1) (2014) 6:1–6:45. doi:10.1145/2580950.

[33] P. Urzyczyn, Inhabitation in Typed Lambda-Calculi (A Syntactic Approach), in: P. de Groote (Ed.), Proceedings of the 3rd International Conference on Typed Lambda Calculi and Applications (TLCA'97), Vol. 1210 of LNCS, Springer, 1997, pp. 373–389. doi:10.1007/3-540-62688-3_47.

[34] A. Dudenhefner, J. Rehof, The Complexity of Principal Inhabitation, in: D. Miller (Ed.), Proceedings of the 2nd International Conference on Formal Structures for Computation and Deduction (FSCD'17), Vol. 84 of LIPIcs,

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017, pp. 15:1–15:14. doi:10.4230/LIPIcs.FSCD.2017.15.

[35] S. Alves, S. Broda, A Unifying Framework for Type Inhabitation, in: H. Kirchner (Ed.), Proceedings of the 3rd International Conference on Formal Structures for Computation and Deduction (FSCD'18), Vol. 108 of LIPIcs, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018, pp. 5:1–5:16. doi:10.4230/LIPIcs.FSCD.2018.5.

[36] T. Gvero, V. Kuncak, I. Kuraj, R. Piskac, Complete Completion using Types and Weights, in: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13), ACM, 2013, pp. 27–38. doi:10.1145/2491956.2462192.

[37] P. Thoman, K. Dichev, T. Heller, R. Iakymchuk, X. Aguilar, K. Hasanov, P. Gschwandtner, P. Lemarinier, S. Markidis, H. Jordan, T. Fahringer, K. Katrinis, E. Laure, D. S. Nikolopoulos, A taxonomy of task-based parallel programming technologies for high-performance computing, J. Supercomput. 74 (4) (2018) 1422–1434. doi:10.1007/s11227-018-2238-4.

[38] M. H. ter Beek, F. Damiani, M. Lienhardt, F. Mazzanti, L. Paolini, Static Analysis of Featured Transition Systems, in: Proceedings of the 23rd International Systems and Software Product Line Conference (SPLC'19), ACM, 2019, pp. 39–51. doi:10.1145/3336294.3336295.

[39] T. Gvero, V. Kuncak, R. Piskac, Code completion using quantitative type inhabitation, Tech. rep., EPFL (2011).
URL http://infoscience.epfl.ch/record/170040

[40] T. Gvero, V. Kuncak, I. Kuraj, R. Piskac, On fast code completion using type inhabitation, Tech. rep., EPFL (2012).
URL http://infoscience.epfl.ch/record/180086

A. Proofs

A.1. Preliminary Notations

Given an SPL L, we write:

- $\texttt{spec}(L)$ for the constraint $\mathit{fm}(L) \wedge \mathit{act}(L)$
- $\mathit{predAPP}^{\bullet}(L)$ for the constraint $\texttt{spec}(L) \Rightarrow \mathit{predAPP}(L)$
- $\mathit{decl}^{\bullet}(L)$ for the constraint $\texttt{spec}(L) \Rightarrow \mathit{decl}(L)$
- $\mathit{nFree}^{\bullet}(L)$ for the constraint $\texttt{spec}(L) \Rightarrow \mathit{nFree}(L)$
- $\mathit{nAmbiguous}^{\bullet}(L)$ for the constraint $\texttt{spec}(L) \Rightarrow \mathit{nAmbiguous}(L)$
- $\mathit{terminating}^{\bullet}(L)$ for the constraint $\texttt{spec}(L) \Rightarrow \mathit{terminating}(L)$

A.2. Correspondence Product Model

Lemma 1. *Given a specification SPL L with $\mathcal{F}$ its set of features, and a product p of L, then there exists exactly one model I of $\texttt{spec}(L)$ such that $p = \{o \mid o \in \mathcal{F} \wedge I(o)\}$. More precisely, I is such that: i) the domain of I is the set of features $\mathcal{F}$ plus the set of delta names in L; ii) all the variables corresponding to features selected for that product are set to* **true***; iii) all the variables corresponding to modules activated for the construction of this product's variant are set to* **true***; and iv) all the other variables in* $\mathrm{dom}(I)$ *are set to* **false**.

Reciprocally, if $\texttt{spec}(L)$ has a model I, then the set $\{o \mid o \in \mathcal{F} \wedge I(o)\}$ is a product of L.

Proof. This follows direct from the way the formula $\texttt{spec}(L)$ is constructed. □

A.3. Proof of Theorem 1 (Applicability Consistency)

Theorem 1. (Applicability consistency). *Consider an SPL L and the following two properties on L:*

1. *The constraint $(\mathit{fm}(L) \wedge \mathit{act}(L)) \Rightarrow \mathit{predAPP}(L)$ is valid.*
2. *All variants of L can be generated.*

Then Property 1 is equivalent to Property 2.

Proof. Let first consider that L has no product: by Lemma 1, $\texttt{spec}(L)$ has no model, and so the constraint $\mathit{predAPP}^{\bullet}(L)$ is valid. Moreover, since L has no product, it also has no variant, and so all of them can be generated. Hence, both Property 1 and Property 2 are valid statements.

Let us now consider that the product line has at least one product: we prove the equivalence by proving each implication independently.

Case $1 \Rightarrow 2$. Suppose chosen a specific product p of L: by Lemma 1, there exists exactly one model I of $\texttt{spec}(L)$ such that $p = \{o \mid o \in \mathcal{F} \wedge I(o)\}$. Because $\mathit{predAPP}^{\bullet}(L)$ is valid and $\mathrm{dom}(I) = \mathit{fv}(\mathit{predAPP}^{\bullet}(L))$, I is also a model of $\mathit{predAPP}(L)$. Let us now consider the sequence $d_1, \ldots, d_n$ of delta that are applied to generate the variant corresponding to p (d_1 being the core of L): we prove that for all $i \in [1..n]$ no errors occurs in the deltas $d_1, \ldots, d_i$ by induction on i. With $i = 1$, as d_1 is the core of L, it only adds variables and function specifications to an empty specification. It thus trivially succeeds. Let now consider $i = j + 1$ with $d_1, \ldots, d_j$ containing no errors. We have eight cases.

1. If d_i contains no operations, then d_i succeed for any input specification, and so $d_1, \ldots, d_i$ contains no errors.

2. If d_i adds a function F with $\mathit{name}(F) = f$: by construction $d_i \in \texttt{add}(L, f)$. Since I is a model of $\mathit{predADD}(L, f)$ and $I(d_i)$ is **true**, we have that $I \vdash \bigwedge_{d'} d' \Rightarrow \bigvee_{d''} d''$ with $d' \prec d'' \prec d_i$, $d' \in \texttt{add}(L, f)$ and $d'' \in \texttt{rem}(L, f)$. Hence, if there exists $1 \leq k \leq j$ with $d_k \in \texttt{add}(L, f)$, there must exist $k < l \leq j$ with $d_l \in \texttt{rem}(L, f)$. Consequently, the specification in input of d_i does not contain f, and so the operation succeeds.

3. If d_i removes a function F with $\mathit{name}(F) = f$: by construction $d_i \in \texttt{rem}(L, f)$. Since I is a model of $\mathit{predREM}(L, f)$ and $I(d_i)$ is **true**, we have that $I \vdash \bigvee_{d''}(d'' \wedge \bigwedge_{d'} \neg d')$ with $d'' \prec d' \prec d_i$, $d' \in \texttt{rem}(L, f)$ and $d'' \in \texttt{add}(L, f)$. Hence, there must exist $1 \leq k \leq j$ with $d_k \in \texttt{add}(L, f)$, such that no $d_l \in \texttt{rem}(L, f)$ with $k < l \leq j$. Consequently, the specification in input of d_i does contain f, and so the operation succeeds.

4. If d_i modifies a function F with $\mathit{name}(F) = f$: by construction $d_i \in \texttt{mod}(L, f)$. Since I is a model of $\mathit{predMOD}(L, f)$ and $I(d_i)$ is **true**, we have that $I \vdash \bigvee_{d''}(d'' \wedge \bigwedge_{d'} \neg d')$ with $d'' \prec d' \prec d_i$, $d' \in \texttt{rem}(L, f)$ and $d'' \in \texttt{add}(L, f)$. Hence, there must exist $1 \leq k \leq j$ with $d_k \in \texttt{add}(L, f)$, such that no $d_l \in \texttt{rem}(L, f)$ with $k < l \leq j$. Consequently, the specification in input of d_i does contain f, and so the operation succeeds.

5. If d_i adds an input T in the function f: by construction

   $$d_i \in \texttt{add}(L, f.\texttt{input}.T) \cap \texttt{mod}(L, f)$$

   Since $d_i \in \texttt{mod}(L, f)$ with Case 4 we can deduce that f is in the input specification of d_i, and with a reasoning similar to Case 2, we can show that T is not an input of f in that specification. Hence the operation succeeds.

6. If d_i removes an input T from the function f: by construction

   $$d_i \in \texttt{rem}(L, f.\texttt{input}.T) \cap \texttt{mod}(L, f)$$

   Since $d_i \in \texttt{mod}(L, f)$ with Case 4 we can deduce that f is in the input specification of d_i, and with a reasoning similar to Case 3, we can show that T is an input of f in that specification. Hence the operation succeeds.

7. If d_i adds an output T in the function f: by construction

$$d_i \in \texttt{add}(L, f.\texttt{output}.T) \cap \texttt{mod}(L, f)$$

Since $d_i \in \texttt{mod}(L, f)$ with Case 4 we can deduce that f is in the input specification of d_i, and with a reasoning similar to Case 2, we can show that T is not an output of f in that specification. Hence the operation succeeds.

8. If d_i removes an output T from the function f: by construction

$$d_i \in \texttt{rem}(L, f.\texttt{output}.T) \cap \texttt{mod}(L, f)$$

Since $d_i \in \texttt{mod}(L, f)$ with Case 4 we can deduce that f is in the input specification of d_i, and with a reasoning similar to Case 3, we can show that T is an output of f in that specification. Hence the operation succeeds.

Consequently, all possible operations in d_i succeed, and so $d_1, \ldots, d_i$ contains no errors.

Case $1 \Leftarrow 2$. We prove this result by contraposition: we assume $\mathit{predAPP}^{\bullet}(L)$ is not valid and prove that there is one variant that cannot be generated.

Let us consider I with $\mathrm{dom}(I) = \mathit{fv}(\mathit{predAPP}^{\bullet}(L))$ such that I is not a model of $\mathit{predAPP}^{\bullet}(L)$. Consequently, I is a model of $\texttt{spec}(L)$ and by Lemma 1, $p = \{o \mid o \in \mathcal{F} \wedge I(o)\}$ is a product of L. Let now consider the sequence $d_1, \ldots, d_n$ of delta that are applied to generate the variant corresponding to p (d_1 being the core of L). For all $\rho \in \texttt{P}(L)$ we define the following sets:

$$\begin{array}{l}
S(\texttt{add}, \rho) = \{d_i \mid \exists 1 \leq j < i.d_i, d_j \in \texttt{add}(L, \rho) \wedge \forall j < k < i.d_k \notin \texttt{rem}(L, \rho)\} \\
S(\texttt{modify}, \rho) = \{d_i \mid d_i \in \texttt{rem}(L, \rho) \\
\qquad\qquad \wedge \forall 1 \leq j < i.\exists j < k < i.d_j \in \texttt{add}(L, \rho) \wedge d_k \in \bigcup_{\rho' \in \mathit{prefix}(\rho)} \texttt{rem}(L, \rho')\} \\
S(\texttt{remove}, \rho) = \{d_i \mid d_i \in \texttt{mod}(L, \rho) \\
\qquad\qquad \wedge \forall 1 \leq j < i.\exists j < k < i.d_j \in \texttt{add}(L, \rho) \wedge d_k \in \bigcup_{\rho' \in \mathit{prefix}(\rho)} \texttt{rem}(L, \rho')\}
\end{array}$$

From the definition of these sets: if there exist $\rho \in \texttt{P}(L)$ such that I does not model $\mathit{predADD}(L, \rho)$, then $S(\texttt{add}, \rho)$ is not empty; if there exist $\rho \in \texttt{P}(L)$ such that I does not model $\mathit{predREM}(L, \rho)$, then $S(\texttt{remove}, \rho)$ is not empty; and if there exist $\rho \in \texttt{P}(L)$ such that I does not model $\mathit{predMOD}(L, \rho)$, then $S(\texttt{modify}, \rho)$ is not empty; Since I does not model $\mathit{predAPP}^{\bullet}(L)$, it does not model $\mathit{predAPP}(L)$, which implies that the following set is not empty:

$$S = \bigcup_{\rho \in \texttt{P}(L)} \bigcup_{op \in \{\texttt{add}, \texttt{modify}, \texttt{remove}\}} S(op, \rho)$$

Let us consider i minimal with $d_i \in S$: we have that $d_1, \ldots, d_{i-1}$ succeeds. Let us moreover consider the first operation in d_i, identified by a pair (op, ρ) with

$op \in \{\texttt{add}, \texttt{modify}, \texttt{remove}\}$ and $\rho \in \mathtt{P}(L)$ such that $d_i \in S(op, \rho)$. We have three cases:

1. $op = \texttt{add}$. By definition of $S(\texttt{add}, \rho)$, we have that $d_i \in \mathtt{add}(L, \rho)$ and there exists d_j such that $j < i$ and $d_j \in \mathtt{add}(L, \rho)$ and for all $j < k < i$, $d_k \notin \mathtt{rem}(L, \rho)$. This means that the function specification on which d_i is applied does contain ρ. Hence, by the Rules 1, 4, 6, and 7 of Figure 9, d_i fails.

2. $op = \texttt{remove}$. By definition of $S(\texttt{remove}, \rho)$, we have that $d_i \in \mathtt{rem}(L, \rho)$ and for all d_j such that $j < i$ and $d_j \in \mathtt{add}(L, \rho)$, there exist d_k with $j < k < i$ and $d_k \in \bigcup_{\rho' \in \mathit{prefix}(\rho)}$. This means that the function specification on which d_i is applied does not contain ρ. Hence, by the Rules 2, 5, 8, and 9 of Figure 9, d_i fails.

3. $op = \texttt{modify}$. By definition of $S(\texttt{modify}, \rho)$, we have that $d_i \in \mathtt{mod}(L, \rho)$ and for all d_j such that $j < i$ and $d_j \in \mathtt{add}(L, \rho)$, there exist d_k with $j < k < i$ and $d_k \in \bigcup_{\rho' \in \mathit{prefix}(\rho)}$. This means that the function specification on which d_i is applied does not contain ρ. Hence, by the third rule of Figure 9, d_i fails.

□

A.4. Presence Constraints

Lemma 2. *Given a specification SPL L with $\mathcal{F}$ its set of features and a product p of L such that the corresponding variant can be generated, consider the model I of $\mathtt{spec}(L)$ corresponding to p with Lemma 1. Then for all $\rho \in \mathtt{P}(L)$, the two following statements are equivalent:*

1. *I is a model of $\mathit{Pre}(L, \rho)$.*

2. *The variant corresponding to p contains ρ.*

Proof. We prove the equivalence by proving each implication independently.

Case $1 \Rightarrow 2$. Since I is a model of $\mathit{Pre}(L, \rho)$, there exists $d \in \mathtt{add}(L, \rho)$ with $I(d)$ and for each delta name d' such that $d \prec d'$and $d' \in \mathtt{rem}(L, \rho)$, we have $\neg I(d')$. By construction of the sets $\mathtt{add}(L, \rho)$ and $\mathtt{rem}(L, \rho)$, this means that during the variant generation, ρ is added by d and is never removed afterward. Hence ρ is present in the variant.

Case $1 \Leftarrow 2$. We prove this result by contraposition: we suppose that I is not a model of $\mathit{Pre}(L, \rho)$ and prove that the variant cannot contain ρ. Since I is not a model of $\mathit{Pre}(L, \rho)$, we have that for all $d \in \mathtt{add}(L, \rho)$ with $I(d)$, there exists $d' \in \mathtt{rem}(L, \rho)$ with $d \prec d'$ and $I(d')$. By construction of the sets $\mathtt{add}(L, \rho)$ and $\mathtt{rem}(L, \rho)$, this means that during the variant generation, every time a delta d adds ρ, the path ρ is removed afterward. Hence ρ is not present in the variant. □

Lemma 3. *Given a specification SPL L with $\mathcal{F}$ its set of features and a product p of L such that the corresponding variant can be generated, consider moreover the model I of* $\mathtt{spec}(L)$ *corresponding to p with Lemma 1. Then for all $f \in \mathtt{P}(L)$ and $v \in \mathtt{fv}(L, f)$, the two following statements are equivalent:*

1. *I is a model of $PrI(L, f, v)$.*
2. *The variant corresponding to p contains the function f and one of the inputs of f contains the variable v.*

Proof. By construction of $PrI(L, \rho)$, I validates it iff there exists $T \in \mathit{input}(L, f)$ with $v \in \mathit{fv}(T)$ and $I \vdash \mathit{Pre}(f.\mathtt{input}.T)$. By Lemma 2, this is equivalent to $f.\mathtt{input}.T$ being present in the generated variant. By construction of paths, this is equivalent to f being included in the generated variant, and v begin a variable of an input of f. □

Lemma 4. *Given a specification SPL L with $\mathcal{F}$ its set of features and a product p of L such that the corresponding variant can be generated, consider moreover the model I of* $\mathtt{spec}(L)$ *corresponding to p with Lemma 1. Then for all $f \in \mathtt{P}(L)$ and $v \in \mathtt{fv}(L, f)$, the two following statements are equivalent:*

1. *I is a model of $PrO(L, f, v)$*
2. *the variant corresponding to p contains the function f and one of the outputs of f contains the variable v*

Proof. This proof is similar to that of Lemma 3, replacing *input* by *ouptut*. □

Lemma 5. *Given a specification SPL L with $\mathcal{F}$ its set of features and a product p of L such that the corresponding variant can be generated, consider moreover the model I of* $\mathtt{spec}(L)$ *corresponding to p with Lemma 1. Then for all $f \in \mathtt{P}(L)$ and $v \in \mathtt{fv}(L, f)$, the two following statements are equivalent:*

1. *I is a model of $AbsO(L, f, v)$.*
2. *The variant corresponding to p contains the function f and one of the outputs of f does not contain the variable v.*

Proof. This proof is similar to that of Lemma 3, replacing *input containing v* by *ouptut not containing v*. □

A.5. Specification Validation: Proofs of Theorems 2 (Variable Presence), 3 (Input Variable Relevance), and 4 (Output Variable Dependency)

Theorem 2. (Variable presence). *Consider an SPL L such that all variants are generable. Moreover, consider the following two properties on L:*

1. *The constraint $(\mathit{fm}(L) \wedge \mathit{act}(L)) \Rightarrow \mathit{decl}(L)$ is valid.*
2. *All variants of L are such that all their variables are declared.*

Then Property 1 is equivalent to Property 2.

Proof. Let us first consider that L has no product: by Lemma 1, $\texttt{spec}(L)$ has no model, and so the constraint $\mathit{decl}^\bullet(L)$ is valid. Moreover, since L has no product, it also has no variant, and so all of them have their variables declared. Hence, both Property 1 and Property 2 are valid statements.

Let us now consider that the product line has at least one product: we prove the equivalence by proving each implication independently.

Case $1 \Rightarrow 2$. Suppose chosen a specific product p of L: by Lemma 1, there exists exactly one model I of $\texttt{spec}(L)$ such that $p = \{o \mid o \in \mathcal{F} \wedge I(o)\}$. Because $\mathit{decl}^\bullet(L)$ is valid and $\mathrm{dom}(I) = \mathit{fv}(\mathit{decl}^\bullet(L))$, I is also a model of $\mathit{decl}(L)$. Let us now consider a variable v used in the variant corresponding to p: by definition, there exist in the variant a term T in input or output of a function f that contains v. Without lost of generality, let consider that T is an input of f: by Lemma 2, we have that $I \vdash \mathit{Pre}(L, f.\texttt{input}.T)$. Hence, since I is a model of $\mathit{decl}(L)$, we must have$I \vdash \mathit{Pre}(L, v')$ for all $v' \in \mathit{fv}(T)$, including v. Consequently, by Lemma 2, we have that v is declared in the variant.

Case $1 \Leftarrow 2$. We prove this result by contraposition: we suppose that $\mathit{decl}^\bullet(L)$ is not valid and prove that there is one variant that does not declare a used variable.

Let us consider I with $\mathrm{dom}(I) = \mathit{fv}(\mathit{decl}^\bullet(L))$ such that I is not a model of $\mathit{decl}^\bullet(L)$. Consequently, I is a model of $\texttt{spec}(L)$, not a model of $\mathit{decl}(L)$, and by Lemma 1, $p = \{o \mid o \in \mathcal{F} \wedge I(o)\}$ is a product of L. Since I is not a model of $\mathit{decl}(L)$ there exists $f \in \texttt{P}(L)$ and either:

- an input T of f and $v \in \mathit{fv}(T)$ with $I \vdash \mathit{Pre}(f.\texttt{input}.T)$ and $I \nvdash v$; or
- an output T of f and $v \in \mathit{fv}(T)$ with $I \vdash \mathit{Pre}(f.\texttt{output}.T)$ and $I \nvdash v$.

In both cases, by Lemma 1, the variant corresponding to p contains a term T with a variable v that is not declared. □

Theorem 3. (Input variable relevance). *Consider an SPL L such that all variants are generable. Moreover, consider the following two properties on L:*

1. *The constraint $(\mathit{fm}(L) \wedge \mathit{act}(L)) \Rightarrow \mathit{nFree}(L)$ is valid.*
2. *All variants of L validate Equation 1 from Section 3.4.2.*

Then Property 1 is equivalent to Property 2.

Proof. Let us first consider that L has no product: by Lemma 1, $\texttt{spec}(L)$ has no model, and so the constraint $\mathit{nFree}^\bullet(L)$ is valid. Moreover, since L has no product, it also has no variant, and so all of them validate Equation 1. Hence, both Property 1 and Property 2 are valid statements.

Let us now consider that the product line has at least one product: we prove the equivalence by proving each implication independently.

Case 1 $\Rightarrow$ 2. Suppose chosen a specific product p of L: by Lemma 1, there exists exactly one model I of $\texttt{spec}(L)$ such that $p = \{o \mid o \in \mathcal{F} \wedge I(o)\}$. Because $\mathit{nFree}^{\bullet}(L)$ is valid and $\mathrm{dom}(I) = \mathit{fv}(\mathit{nFree}^{\bullet}(L))$, I is also a model of $\mathit{nFree}(L)$. Let now consider f declared in the variant corresponding to p, and an input term t of f containing a variable v. By Lemma 3, I is a model of $\mathit{PrI}(L, f, v)$, which implies that it is also a model of $\mathit{PrO}(L, f, v) \wedge \neg \mathit{AbsO}(L, f, v)$. By Lemma 4 and 5, this means that there exists an output of f that contains v, and there are no output of f that does not contain that variable.

Case 1 $\Leftarrow$ 2. We prove this result by contraposition: we suppose that $\mathit{nFree}^{\bullet}(L)$ is not valid and prove that there is one variant that does not validate Equation 1.

Let us consider I with $\mathrm{dom}(I) = \mathit{fv}(\mathit{nFree}^{\bullet}(L))$ such that I is not a model of $\mathit{nFree}^{\bullet}(L)$. Consequently, I is a model of $\texttt{spec}(L)$, not a model of $\mathit{nFree}(L)$, and by Lemma 1, $p = \{o \mid o \in \mathcal{F} \wedge I(o)\}$ is a product of L. Since I is not a model of $\mathit{nFree}(L)$ there exists $f \in \texttt{P}(L)$ and $v \in \texttt{fv}(L, f)$ such that I is a model of $\mathit{PrI}(L, f, v)$ and at least one of the following statement holds:

- I does not model $\mathit{PrO}(L, f, v)$ which implies by Lemma 4 that no output of f contains v; or
- I models $\mathit{AbsO}(L, f, v)$ which implies by Lemma 5 that there exists an output of f that does not contain v. □

Theorem 4. (Output variable dependency). *Consider an SPL L such that all variants are generable. Moreover, consider the following two properties on L:*

1. *The constraint $(\mathit{fm}(L) \wedge \mathit{act}(L)) \Rightarrow \mathit{nAmbiguous}(L)$ is valid.*
2. *All variants of L validate Equation 2 from Section 3.4.2.*

Then Property 1 is equivalent to Property 2.

Proof. Let us first consider that L has no product: by Lemma 1, $\texttt{spec}(L)$ has no model, and so the constraint $\mathit{nAmbiguous}^{\bullet}(L)$ is valid. Moreover, since L has no product, it also has no variant, and so all of them validate Equation 2. Hence, both Property 1 and Property 2 are valid statements.

Let us now consider that the product line has at least one product: we prove the equivalence by proving each implication independently.

Case 1 $\Rightarrow$ 2. Suppose chosen a specific product p of L: by Lemma 1, there exists exactly one model I of $\texttt{spec}(L)$ such that $p = \{o \mid o \in \mathcal{F} \wedge I(o)\}$. Because $\mathit{nAmbiguous}^{\bullet}(L)$ is valid and $\mathrm{dom}(I) = \mathit{fv}(\mathit{nAmbiguous}^{\bullet}(L))$, I is also a model of $\mathit{nAmbiguous}(L)$. Let now consider f declared in the variant corresponding to p, and an input term t of f containing a variable v. By Lemma 3, I is a model of $\mathit{PrI}(L, f, v)$, which implies that it is also a model of $\mathit{PrO}(L, f, v)$. By Lemma 4, that there exists an output of f that contains v. With a similar approach, we easily prove that all variables in the outputs of f are also in its inputs.

Case $1 \Leftarrow 2$. We prove this result by contraposition: we assume $nAmbiguous^{\bullet}(L)$ is not valid and prove that there is one variant that does not validate Equation 2.

Let us consider I with $\mathrm{dom}(I) = fv(nAmbiguous^{\bullet}(L))$ such that I is not a model of $nAmbiguous^{\bullet}(L)$. Consequently, I is a model of $\mathtt{spec}(L)$, not a model of $nAmbiguous(L)$, and by Lemma 1, $p = \{o \mid o \in \mathcal{F} \wedge I(o)\}$ is a product of L. Since I is not a model of $nAmbiguous(L)$ there exists $f \in \mathtt{P}(L)$ and $v \in \mathtt{fv}(L, f)$ such that I models either:

- $PrI(L, f, v) \wedge \neg PrO(L, f, v)$: by Lemma 3 and 4, this means that v is in an input of f but not in any of its outputs; or
- $PrO(L, f, v) \wedge \neg PrI(L, f, v)$: by Lemma 3 and 4, this means that v is in an output of f but not in any of its inputs. □

A.6. Proof of Theorem 5 (Terminating Specification)

The proof presented in this section is based on one of the main theorems of [19]. Hence, before presenting our proof, we first present an extended version of Definition 8 that introduces a new characterization of order relations. We then recall the concept of *dependency pairs*, which is the core contribution of [19]. Finally, we recall the theorem on which our proof is based.

A.6.1. Order relations

Definition 9 (Preorder, partial order, and well-founded partial order)**.** *Given a set A, and a binary relation $R \subseteq A \times A$. R is a* preorder *(a.k.a.* quasi-order*) iff it is reflexive and transitive. R is a* partial order *iff it is antisymmetric and transitive. Morever, R is a* well-founded *order iff it is a partial order and for all infinite sequence $(a_i)_{i \in \mathbb{N}}$ there exists $i \in \mathbb{N}$ with $(a_i, a_{i+1}) \notin R$.*

Finally, given a preorder $\leq$, we write $\leq_{\downarrow}$ the partial order $\{(x, y) \mid (x, y) \in \leq \wedge (y, x) \notin \leq\}$.

Definition 10 (Substitution-closed preorder over a set of terms)**.** *Given a set of terms $T = \mathcal{T}(F, V)$, a preorder relation $\leq$ over T is* weakly monotonic *iff $s \leq t$ implies $f(s_1, \ldots, s_i, s, s_{i+1}, \ldots s_n) \leq f(s_1, \ldots, s_i, t, s_{i+1}, \ldots s_n)$ for all $s_1, \ldots, s_n \in T$. Moreover, $\leq$ is* closed under substitution *iff for all $(l, r) \in \leq$ and all substitution σ, $(\sigma(l), \sigma(r)) \in \leq$.*

A.6.2. Dependency pairs

Definition 11 (T-TRS and terminanting T-TRS)**.** *Given a set of terms $T = \mathcal{T}(F, V)$, a T-*term rewriting system *(T-TRS) is a set $R \subseteq T \times T$ such that for all $(l : s_l, r : s_r) \in R$, $l \notin V$, $fv(r) \subseteq fv(l)$ and $\{(s_l, s_r), (s_r, s_l)\} \cap < \neq \emptyset$.*

A T-TRS R is terminating *there is no infinite sequence of terms $\{t_i)\}_{i \in \mathbb{N}}$ such that for all $i \in \mathbb{N}$, there exists a subterm t'_i of t_i, a substitution σ and a rewriting rule $(l, r) \in R$ such that $t'_i = \sigma(l)$ and $t_{i+1} = \sigma(r)$.*

Definition 12 (Dependence pair of a T-TRS)**.** *Given a set of terms $T = \mathcal{T}(F, V)$ and a term $t = f(t_1, \ldots, t_n) \in T \setminus V$, the* root symbol *of t, written*

$\mathtt{root}(t)$ *is* f*. Given a* T*-TRS* R *and writing* $F = \{F_a\}_{a\in A}$*, the set of* defined symbols *in* R *is the indexed family* $F[R] = \{F[R]_a\}_{a\in A}$ *with*

$$F[R]_a = \{f \mid f \in F_a \wedge \exists(l, r) \in R,\ f = \mathtt{root}(l)\}$$

For all $a \in A$*, we assume a set of fresh symbols* $F[R]^{\#}_a = \{f^{\#} \mid f \in F[R]_a\}$*, and for all terms* $t = f(t_1, \ldots, t_n)$ *with* $f \in F[R]$*, we write* $t^{\#}$ *for the term* $f^{\#}(t_1, \ldots, t_n)$*.*

A dependency pair *of* R *is a pair* $(l^{\#}, r^{\#})$ *such that* $(l, r') \in R$ *and* r *is a subterm of* r'*. We write* $\mathtt{DP}(R)$ *the set of dependency pairs of* R*.*

Theorem 6 ([19, Theorem 7])**.** *Given a set of terms* $T = \mathcal{T}(F, V)$ *and a* T*-TRS* R*,* R *is terminating iff there exists a weakly monotonic preordering* $\preceq$ *such that both* $\preceq$ *and* $\preceq_{\downarrow}$ *are closed under substitution,* $\preceq_{\downarrow}$ *is well-founded, and both*

- $r \preceq l$ *for all rules* $(l, r) \in R$*; and*
- $r \preceq_{\downarrow} l$ *for all dependency pairs* $(l, r) \in \mathtt{DP}(R)$

A.6.3. Proof of Theorem 5 (terminating specification)

Lemma 6. *Given a specification SPL* L *with* $\mathcal{F}$ *its set of features, and a product* p *of* L *such that the corresponding variant can be generated with all its variables declared. Then the term rewriting system generated from that variant is*

$$\begin{aligned} R = \{\, &(\mathtt{data}(o), \mathtt{data}(o, \mathtt{task}(f, \mathtt{data}(i_1), \ldots, \mathtt{data}(i_n)))) \mid f \in P(L) \\ &\wedge I \vdash \mathit{Pre}(L, f.\mathtt{output}.o) \wedge \{i_1, \ldots, i_n\} = \{r \mid I \vdash \mathit{Pre}(L, f.\mathtt{input}.r)\}\,\} \end{aligned}$$

with I *defined by Lemma 1. Moreover, we have*

$$\begin{aligned} \mathtt{DP}(R) = \{\, &(\mathtt{data}^{\#}(o), \mathtt{data}^{\#}(i)) \mid \\ &\exists f \in P(L),\ I \vdash \mathit{Pre}(L, f.\mathtt{output}.o) \wedge I \vdash \mathit{Pre}(L, f.\mathtt{input}.i)\,\} \end{aligned}$$

Proof. The first statement is a corollary of Lemma 2. The second one is a direct application of the definition of the dependency pairs. □

Theorem 5. (Terminating specification). *Consider an SPL* L *such that all variants are generable and with all the variables declared. Moreover, consider the following two properties on* L*:*

1. *The constraint* $(\mathit{fm}(L) \wedge \mathit{act}(L)) \Rightarrow \mathit{terminating}(L)$ *is valid.*
2. *Each variant of* L *results in a terminating TRS.*

Then Property 1 is equivalent to Property 2.

Proof. Let us first consider that L has no product: by Lemma 1, $\mathtt{spec}(L)$ has no model, and so the constraint $\mathit{terminating}^{\bullet}(L)$ is valid. Moreover, since L has no product, it also has no variant, and so all of them result in a terminating TRS. Hence, both Property 1 and Property 2 are valid statements.

Let us now consider that the product line has at least one product: we prove the equivalence by proving each implication independently.

Case $1 \Rightarrow 2$. Suppose chosen a specific product p of L: by Lemma 1, there exists exactly one model I of $\mathtt{spec}(L)$ such that $p = \{o \mid o \in \mathcal{F} \wedge I(o)\}$. Because $\mathit{terminating}^{\bullet}(L)$ is valid and $\mathrm{dom}(I) = \mathit{fv}(\mathit{terminating}^{\bullet}(L))$, I is also a model of $\mathit{terminating}(L)$. Hence, there exist a partial order $<$ on terms that is well-founded, weakly monotonic, closed under substitution and such that for all $f \in \mathtt{P}(L)$, all $t \in \mathit{output}(L, f)$ with $I \vdash \mathit{Pre}(L, f.\mathtt{output}.t)$, and all $t' \in \mathit{input}(L, f)$ with $I \vdash \mathit{Pre}(L, f.\mathtt{input}.t')$, we have $t' < t$. Since the $\mathcal{S}$-sorted signature F is dataflow-safe, $<$ contains no terms with $\mathtt{data}$ or $\mathtt{task}$ symbols. Let us define $\equiv$ being the transitive, reflexive closure of R that is closed under substitution. We moreover define the partial order $\prec$ as follows:

$$\prec \;=\; < \cup \{\, (\mathtt{data}(l), \mathtt{data}(r)) \mid (l, r) \in < \,\} \cup \{\, (\mathtt{data}^{\#}(l), \mathtt{data}^{\#}(r)) \mid (l, r) \in < \,\}$$

Let $\preceq = \prec \cup \equiv$. By construction of R, it is a preorder and $t \preceq_{\downarrow} = \prec$.

With Lemma 6, is thus clear that $\preceq$ validates Theorem 6.

Case $1 \Leftarrow 2$. Suppose chosen a specific product p of L: by Lemma 1, there exists exactly one model I of $\mathtt{spec}(L)$ such that $p = \{o \mid o \in \mathcal{F} \wedge I(o)\}$. Since the TRS R resulting of the variant corresponding to p terminates, we can consider the preorder $\preceq$ given by Theorem 6. Let us recall that $\preceq_{\downarrow}$ is a partial order on $\mathtt{DP}(R)$ that is well founded, weakly monotonic, and closed under substitution. Hence the following relation $<$ is also founded, weakly monotonic, and closed under substitution:

$$< = \preceq_{\downarrow} \cup \{\, (l, r) \mid (\mathtt{data}^{\#}(l), \mathtt{data}^{\#}(r)) \in \preceq_{\downarrow} \,\}$$

By Lemma 6, we thus have that $<$ is such that for all $f \in \mathtt{P}(L)$, all $t \in \mathit{output}(L, f)$ with $I \vdash \mathit{Pre}(L, f.\mathtt{output}.t)$, and all $t' \in \mathit{input}(L, f)$ with $I \vdash \mathit{Pre}(L, f.\mathtt{input}.t')$, we have $t' < t$. $\square$