RESEDA: Declaring Live Event-driven Computations as REactive SEmi-structured DAta

Full version [DRAFT]

João Costa Seco
Søren Debois
Thomas Hildebrandt
Tijs Slaats

Abstract

Enterprise computing applications generally consists of several inter-related business processes linked together via shared data objects and events. We address the open challenge of providing formal modelling and implementation techniques for such enterprise computing applications, introducing the declarative, data-centric and event-driven process language RESEDA for REactive SEmi-structured DAta. The language is inspired by the computational model of spreadsheets and recent advances in declarative business process modelling notations. The key idea is to associate either input events or reactive computation events to the individual elements of semi-structured data and declare reactive behaviour as explicit reaction rules and constraints between these events. Moreover, RESEDA comes with a formal operational semantics given as rewrite rules supporting both formal analysis and persistent execution of the application as sequences of rewrites of the data. The data, along with the set of constraints, thereby at the same time constitutes the specification of the data, its behaviour and the run-time execution component. This key contribution of the paper is to introduce the RESEDA language, its formal execution semantics and give a sufficient condition for liveness of programs. We also establish Turing-equivalence of the language independently of the choice of underlying data expressions and exemplify the use of RESEDA by a running example of an online store. A prototype implementation of RESEDA and the examples of the paper are available on-line at `http://dcr.tools/reseda`.

1 Introduction

Realistic enterprise computing applications contain several inter-related business processes defining the behaviour of the application and which are often linked together via shared data, such as customers, orders and shipments in a shop. The data and the activities of the processes are two sides of the same coin: Data classes have corresponding activities in the processes for the creation and change of the data objects. Moreover, cardinality relations between data elements in the data model are mirrored by behavioural orderings of activities in the processes. For instance, the data model may specify that a shipment is related to one or more orders belonging to the same customer. This is reflected by a behavioural constraint, that the activity for creating a shipment can only be executed after the activities of creating a customer and the first order of the customer.

The management of customers, orders and shipments of orders are likely to be independently defined and developed business processes, which are not necessarily hierarchically embedded in each other. However, as pointed out in [15,17], standard business process notations such as BPMN are not ideal for modelling nor implementing enterprise business applications with dependencies and shared data between activities belonging to multiple instances of such independent processes.

We propose in the present paper a novel, declarative data-centric process language RESEDA based on REactive SEmi-structured DAta as a unified specification of data, behaviour and run-time execution state. RESEDA is inspired by and generalises both the declarative Dynamic Condition Response (DCR) graph process notation [4,5,10] and the widely used reactive model of data and behaviour used in spreadsheets [14].

In a spreadsheet, data is organised in a matrix. The fields of the matrix are either empty and expect a data element to be *input* by the user or contain an expression, possibly referring to data of other fields, and expected to be *computed* by the spreadsheet processer. The computation of an expression happens *initially*, when the expression is defined and *reactively*, whenever a data element referenced by the expression changes its value.

In RESEDA we categorise data elements as either input or computation data elements similarly to spreadsheets, but instead of the matrix structure used in spreadsheets, RESEDA employs a semi-structured model for the data elements. Semi-structured data is widely used in practice, notably via XML and JSON, and allows more easily than matrices the modelling of inter-related, unbounded and dynamically growing and shrinking collections of data elements using the containment relation, attributes with unique ids (UIDs) and locational referencing of data elements via so-called path expressions.

Input data elements in RESEDA are thus data elements, for which the assignment of a value is triggered by an input event from the outside context of the process. In addition to an empty field of a spreadsheet, other analogies for such input data elements can be found in input fields of web forms, input parameters of web-services, or even IoT sensors e.g. measuring the location of a business object using a geo-location capability. Similarly to input fields in a web form being either required or optional, RESEDA allows for data elements to be declared as either required or optional. A required input data element must eventually be provided a value for the program to reach an accepting state. Similarly, a required computation data element must eventually be evaluated by the run-time system, for the program to reach an accepting state. Also, we allow for the declaration of additional rules, e.g. for dynamically creating new data and determining additional ordering constraints, e.g. that a new input or re-computation of a value for a data element is required as a response to another input or computation. A computation of a process is then only considered complete if every required input or computation of a data value is eventually executed.

The execution semantics of RESEDA makes the language directly applicable for execution as a sequence of rewrites of the data. The semantic foundation for describing both the reactive computation of expressions and the creation of new data obtained from the declarative DCR* process notation introduced in [4,5] and generalising DCR graphs [10] to allow dynamic creation of sub processes. The creation of new data (with its own behavioural constraints and reaction rules) is represented directly by the DCR* rule for dynamic spawning of new sub processes. RESEDA can thereby also be seen as an extension of DCR* to allow for associating events with semi-structured data and expressions, and using path expressions to indirectly refer to other events by their location instead of only their identity. Our approach in this way harmoniously integrates the dynamic construction of semi-structured data and concurrent declarative control flow.

The concurrent event-driven reactive computation raises issues of possible deadlocks and livelocks, as an event may in response lead to an infinite sequence of re-computations and creations of new data elements. As a main technical result, we provide sufficient conditions for deadlock and livelock freedom. In addition, we prove that Reseda generalises the DCR graphs process formalism, [4,5] and derive Turing completeness of the language as a corollary.

2 RESEDA

We illustrate and motivate the constructs of our language, RESEDA, by gradually designing the shopping process introduced in [18].

The data model shown in Figure 1 describes (using UML notation) classes of customers, orders, orderlines, products and deliveries. The behavioural model shown in Figure 2 describes behavioural constraints between key activities of the business process using a variant of DECLARE [13] notation used in the so-called Object-Centric Behavioural Constraint (OCBC) notation of [17,18]. Briefly, the process describes a shop in which registered customers can create orders (by the activity `create order` (1)), to which they must add one or more order lines (by the activity `pick item` (2)), each referring to a unique product. Each product mentioned in a order line must be wrapped (by the activity `wrap item` (3)) and eventually delivered to the customer (by the activity `deliver items` (4)). A delivery may contain items from different orders but restricted to only one customer.

The relations (5) and (6) from `create order` to `pick item` in Figure 2 define behavioural constraints denoting respectively that any `create order` event is followed by one or more `pick item` events, and that every `pick item` event is preceded by a unique `create order` event. The relation (7) between `pick item` and `wrap item` denote a one-to-one correspondence between events related to the two activities. Finally, the relations (8) and (9) define

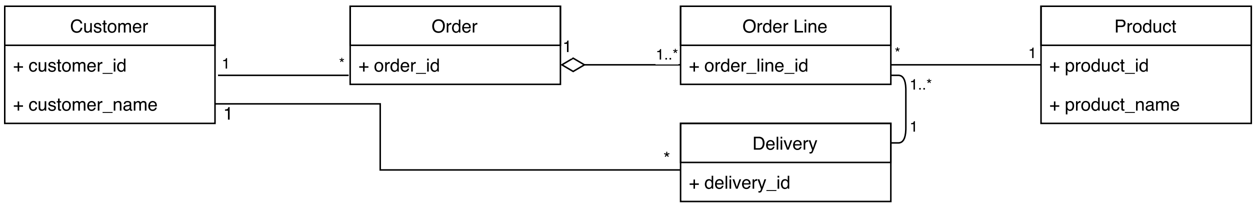

Figure 1: Data model of the shopping example from [18]

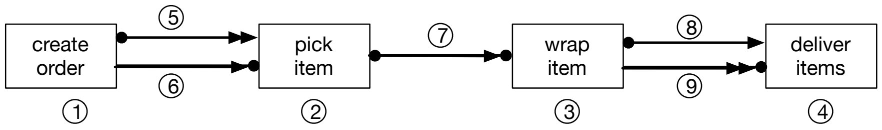

Figure 2: Behavioural constraints of the shopping example from [18]

respectively that every `wrap item` must be followed by exactly one `deliver items` event, and each `deliver items` event must be preceded by one or more `wrap item` events.

The OCBC notation of [17, 18]) makes it possible to combine the data model of Figure 1 and the declarative behavioural model of Figure 2. But OCBC does not provide notation for specifying the run-time data objects and process instances. In the following we describe a RESEDA process doing exactly that. We represent data items as named leaf elements of the tree-structured semi-structured data, and model the structure of the data model either by regular nesting or by linking using unique identifiers like in document bases databases.

In this section we use the concrete RESEDA syntax, used in our implementation. However, in the formal developments ahead we define a core calculus, using a slightly different, more abstract notation, for the sake of readability of the results. We invite the reader to experiment with the syntax and semantics of RESEDA in the on-line tool accompanying this paper, available at `http://dcr.itu.dk/reseda/`. For convenience, all major examples of the present section are pre-loaded in that tool for experimentation.

2.1 Static data

We begin by showing how to model static data with no behaviour in Example 1. Concretely, we model two data objects of class Customer of Figure 1.

Example 1 (Modelling structured data using constant values).

```
customer[] {
  customer_id(0)[],
  customer_name('John')[]
},
customer[]{
  customer_id(1)[],
  customer_name('Mary')[]
}
```

In Reseda, the basic building block are instances of data elements, with the general form,

```
M name(V)[E]{P}
```

where `name` defines a label for the data element, `V` defines an initial value for the element, `E` is the expression that defines the value for the element, `M` defines the behavioural state of the data element (to be explained later), and `P` defines the inner process of the data element that may contain more data elements and control elements (behavioural constraints and reaction rules, described ahead). The element label is the only required part of a data element specification.

In this example, we model two data elements with label `customer`, which contain data elements modelling the fields `customer_id` and `customer_name`. The structure of data items and associated inner data items and values can be interpreted as an object structure found in document- or XML-based databases. Notice that the data element `customer` has structuring purposes only — intuitively it does not have a value, formally we model it holding the value unit ($\mathbf{1}$).

2.2 Dynamically created data

The key mechanism for obtaining dynamic behaviour in RESEDA is that whenever the value of a data element is updated, that update may have side effects triggered by reaction rules inherited from the DCR* process language [5]. A rule can e.g. specify the dynamic creation of new data elements or requiring other data elements to be updated in response.

We next illustrate how to declare a rule for dynamically creating instances of the class Customer of Figure 1.

Example 2 (Dynamically created customers collection).

```
customers[]{
    create_customer[?]
    ;
    create_customer -->> {
        customer[]{
            !customer_id[freshid()],
            !customer_name[?:string]
        }
    }
}
```

The dynamically created collection of customers is enclosed by a `customers[]` data element, that nests a list of data elements, like the ones defined in Example 1. Initially, the element contain no such customer data elements. Instead, we have introduced an input data element `create_customer`. When the corresponding input event happens, an additional `customer` data element is created as a side effect. This side-effect is specified via a rule using the "spawn relation" (`-->>`) introduced in [5]. Syntactically, we distinguish between the part of a process that specifies data elements, and the part that specifies rules by a `;` symbol.

The reaction rule declares that, whenever the left operand — a data item named `create_customer` — is provided a value, a new data item named `customer` is created in the local context, i.e. inside the collection `customers`. When no data type is provided, as for the `create_customer` data item, we assume it is of unit type, which in the analogy of the web forms would just be a button being clicked. The value of the computation data element `customer_id` inside each new `customer` data element is given by the predefined function `fresh_id` similar to the the `generate_id` function in XPath [7]. The input data item `customer_name` gets its value of string type from the outside context. The result of executing a `create customer` event will thus be the RESEDA program shown in Example 3.

Example 3 (Freshly created customer instance).

```
[customers[]{
    create_customer[?],
      customer[]{
      !customer_id[freshid()],
      !customer_name[?:string]
    }
    ;
    create_customer -->> {
        customer[]{
            !customer_id[freshid()],
            !customer_name[?:string]
        }
    }
}
```

Notice that for any newly created `customer` record, the computation data element `customer_id` needs to be updated and the input data element `customer_name` element is missing a data value. We indicate that the events providing values to the data elements inside a new `customer` data element are initially *pending* as in DCR Graphs, by the exclamation mark `!` used as prefix. A pending event is an event that is required to happen eventually. This means, for computation data elements like `customer_id`, that they are staged to be automatically computed. The corresponding computation event will happen unless it depends on some other data elements for which values have not yet been computed or provided, or other behavioural constraints have been defined as exemplified in later examples. Since there are no dependencies for the computation event `customer_id`, the event will be triggered by the RESEDA processer, thereby computing a fresh id and assigning it as value to the data element, resulting in the RESEDA program shown in Example 4.

Example 4 (Customer id computed).

```
# This file intentionally left blank
```

For input data elements, the pending state that the input should be done *eventually* for the process to progress. Using the analogy of web forms, this would correspond to an input field being marked as mandatory.

The result of executing a `customer_name` event providing the string `John` as value will be the RESEDA program shown in Example 5

Example 5 (Customer name provided as input event).

```
customers[]{
    create_customer[?],
      customer[]{
      customer_id(0)[freshid()],
      customer_name('John')[?:string]
    }
    ;
    create_customer -->> {
        customer[]{
            !customer_id[freshid()],
            !customer_name[?:string]
        }
    }
}
```

Note that a new `customer_name` input event can happen again, analogous to a web form field being editable, but not mandatory.

The same structure can be followed to create the collection corresponding to class Product depicted in Figure 1, by the code of Example 6.

Example 6 (Dynamically created products and trigger path expression).

```
products[]{
    create_product[?:string]
    ;
    create_product -->>{
        product[]{
            !product_id[freshid()],
            !product_name[@trigger:value]
        }
    }
}
```

Here we also illustrate how to access the data element whose update caused the dynamic creation of data using the reserved word `trigger` in a *path expression* (identified by the prefix `@`) for a computation data element `product_name`. We get the value(s) of the element(s) identified by the path expression by adding the suffix `:value`.

The result of executing a `create_product` event providing the string `iPhone X` as value will be the RESEDA program shown in Example 7. Note that the two computation data elements are pending and will thus the corresponding computation events subsequently be computed and assign values to the data elements.

Example 7 (Freshly created iPhone X product).

```
products[]{
    create_product[?:string],
     product[]{
            !product_id[freshid()],
            !product_name['iPhone X']
        }
    ;
    create_product -->>{
        product[]{
            !product_id[freshid()],
            !product_name[@trigger:value]
        }
    }
}
```

We can observe by now that RESEDA expressions represent both the state and code of the system, which relates to live programming and execution environments such as smalltalk virtual machines.

2.3 Regulating behaviour: Condition and exclusion rules

Next, to define the collection of orders we must establish relations between classes Order, Customer and Product (via class OrderLine) as depicted in Figure 1. Consider Example Example 8 below introducing data element `orders` with an input data element `create_order`, used to introduce new elements in the collection.

Example 8 (Orders collection definition).

```
1  orders[]{
2    create_order[
3      ?:@/customers/customer/customer_id:value
4    ]
5    ;
6    create_order -->>{
7      order[]{
8        !order_id[freshid()],
9        !customer_id[@trigger:value],
10       !pick_item[
11          ?:@/products/product/product_id:value
12       ]
13       ;
14       pick_item -->>{
15         order_line[]{
16           !order_line_id[freshid()],
17           !item[@trigger:value],
18           !wrap_item[
19             ?:@/deliveries/
20                  delivery[
21                     not(deliver_items:executed)
22                  ]/delivery_id:value
23           ]
24           ;
25           wrap_item -->* order_line_id,
26           item -->* order_line_id,
27           wrap_item -->% wrap_item
28         }
29       }
30     }
31   }
32 }
```

Observe the type annotation for the input of data element `create_order` in line 2, given by the path expression `@/customers/customer/customer_id:value`. The expression is evaluated with relation to its definition context to a set of data items and dereferenced by the attribute label `value` to obtain all values carried by the selected data items. The initial `/` of the path expression means that the expression is evaluated at the root of the process.

This kind of annotations in input fields thereby acts as a *dependent type* [12] for the input, with the meaning, that only values in the computed set are accepted as valid input value for the data element `create_order`. Further, the input event updating the data element is only possible if there is at least one valid data value. The type in this case thereby ensures that a valid customer id is provided as input for the create order data element. In the analogy of the web form, this would correspond to a data-dependent input validation of the input field or the values in a dropdown UI widget.

As in the previous examples, the unique identifier of an order is automatically computed using the `freshid()` expression in line 6, and the customer explicitly identified by the input value for the `create_order` event is copied to a locally available computation data element (`customer_id`) using the `@trigger:value` path expression in line 7. We thereby establish the relation between a customer and an order.

The pending input data element `pick_item` in line 8 allows items to be added to a created order. The fact that it is initially pending means that at least one item must eventually be added to the order. Combined with the fact that the input data element is created with the new order we fulfil the constraints (5) and (6) in Figure 2. Again, every time a value is provided as input for the data element `pick_item`, by the user using a web form or an external service, the dependent type in line 8 guarantees that the value is a valid product identifier value.

Now, the rule for dynamic creation of data (line 10-21) triggered by the input event for the `pick_item` data element specifies that an `order_line` data element is created inside the structure of the current order. Using the `@trigger` path expression in line 13, the product identifier provided as value to the `pick_item` is copied to the `item` data element of the order line and we thereby establish the relation between an order line and a product. The `wrap_item` input data element in line 14 takes an input value which is again constrained by a path expression. In this case, the path expression constrains the values to be delivery identifiers for deliveries, for which the event corresponding to the `deliver_items` data element has not yet been executed. This implements that the wrapped item is associated with a delivery which has not yet been delivered. We will look into the details of the delivery data class and behaviour in the next example.

Notice the nested structure created here to represent the containment relation of objects from classes Order and OrderLine from Figure 1 and the use of identifiers to represent relations from class Order to class Customer and from class OrderLine to the classes Product and Delivery.

The example introduces two new kinds of behavioural rules in line 18-20 in the definition of `order_line`, inherited from DCR Graphs. The first two behavioural rules (line 18-19) are standard condition constraint rules, as also found in DECLARE, specifying that the `order_line_id` computation data element can only be updated after `wrap_item` and `item` have been updated. As we will see in the next example, the reason for this constraint is that the update of the `order_line_id` has the side effect in the `delivery` class that the item is added to the delivery identified by the `wrap_item` data element. For this reason we want it only to happen once all data elements has been computed. Finally, the rule `wrap_item -->% wrap_item` (line 20), causes the data element `wrap_item` to be dynamically excluded from future computation once a value has been provided to it. In the analogy of the web form, the button will no longer be visible, but its data can still be accessed by the code. This ensures that the `wrap_item` input event can only happen once. Combined with the fact that the data element is initially pending implements constraint (7) in Figure 2.

2.4 Initially excluded data elements aka hidden fields

We now explain how to specify the collection of `delivery` data elements corresponding to the class Delivery from Figure 1. Refer to Example 9 below.

Example 9 (Deliveries collection definition).

```
1  deliveries[]{
2    create_delivery[
3      ?:@/customers/customer/customer_id:value
4    ]
5    ;
6    create_delivery -->> {
7     delivery[]{
```

```
8         !delivery_id[freshid()],
9         !customer_id[@trigger:value],
10        !%deliver_items[?]
11        ;
12        /orders/order[
13           @customer_id:value==@trigger:value
14        ]/order_line[
15           @wrap_item:value==@rule/delivery_id:value
16        ]/order_line_id
17            -->> {
18                 !item_id[@trigger:value]
19                 },
20        item_id -->+ deliver_items,
21        deliver_items -->% deliver_items
22       }
23     }
24   }
```

As for the previous examples, a `delivery` data element is created by assigning a value, in this case a valid customer id, to an input data element `create_delivery` (line 2). The delivery has data elements, just as the other classes, updated with a fresh id (line 6) and a copy of the customer id (line 7) provided when it was created. Each `delivery` data element moreover contains a `deliver_items` input data element (line 8), which is initially both pending and excluded, the exclusion being declared by the prefix %. Initially excluded data elements are another feature inherited from DCR Graphs. Excluded data elements may be included using an *include* reaction rule. In the analogy with web forms, excluded data elements correspond to hidden fields, which may be made dynamically visible.

The Path expression in line 10-11 introduce the use of filters constraining which data elements will trigger the rule when they are updated by a computation event. The first filter, `@customer_id:value==@trigger:value` ensures that only orders of the customer with id matching the id of the delivery are considered. The second filter, `@wrap_item:value==@rule/delivery_id:value` ensures that only items in the order line wrapped for that delivery are considered. (Recall that a delivery id was added when an item was wrapped). The path expression thereby make sure that only order lines with the designated delivery id and for the customer assigned to the delivery triggers the creation of an `item_id` data element. Once the computation event for the created pending `item_id` computation data element is executed, the rule `item_id -->+ deliver_items` in line 15 causes the `deliver_items` data element to be dynamically included. Finally, once the computation event for the `deliver_items` element happens, the `deliver_items` data element is again excluded by the `deliver_items -->% deliver_items` rule. These rules together with the constraint of the `wrap_item` input events that items can only wrapped for deliveries that have not been delivered implement the constraints (8) and (9) of Figure 2, i.e. that a delivery can only happen after an item has been added, and that an item can only be delivered once.

This example demonstrates that behavioural constraints of business processes as well as their underlying data model can be declared harmoniously specified together in RESEDA. In the next sections we provide the formal syntax and semantics.

3 Syntax

In this section we introduce the formal syntax of the RESEDA calculus, leaving out types to simplify the exposition. RESEDA *processes* are the syntactic entities generated by the syntax in Figure 3. (We will relate the formal syntax to the concrete ASCII-syntax used in the previous section shortly.)

A RESEDA process P consists of a vector $\vec{D}$ of (structured) *data terms*, followed by a vector $\vec{R}$ of *rule terms*. Both data terms and relations are built over an *expression language*. As already described above, in the ASCII syntax, we separate data terms and rules are separated by a symbol `;`, and subprocesses are enclosed in braces.

3.1 Expression language

Expressions in RESEDA are built from the mutually recursive classes of *data expressions* E and *path expressions* ϕ.

$$\begin{array}{rll}
P & ::= \vec{D};\vec{R} & \text{processes} \\
D & ::= n[E]{:}M\{P\} \mid n[?]{:}M\{P\} & \text{structured data} \\
M & ::= (h, i, r, v) & \text{markings} \\
R & ::= \phi \rightarrow\!\bullet\ \phi \mid \phi \rightarrow\!\diamond\ \phi \mid \phi \bullet\!\!\rightarrow \phi & \\
 & \mid \phi \rightarrow\!+\ \phi \mid \phi \rightarrow\!\%\ \phi \mid \phi \twoheadrightarrow P & \text{relations} \\
\phi & ::= /\psi \mid \psi & \text{path expressions} \\
\psi & ::= \alpha \mid \psi/\alpha \mid \psi[E] & \\
\alpha & ::= . \mid .. \mid n \mid * & \\
E & ::= c \mid f(E\_1, \ldots, E\_n) \mid \phi \mid \mathsf{trigger} & \\
 & \mid \mathsf{this} \mid \mathsf{rule} \mid E{:}\mathit{attr} \mid \bot \mid \mathbf{1} & \text{data expressions} \\
\mathit{attr} & ::= \mathsf{value} \mid \mathsf{executed} & \\
 & \mid \mathsf{included} \mid \mathsf{pending} &
\end{array}$$

Figure 3: Syntax of RESEDA.

Data expressions E are built over an unspecified set of constants c, assumed to include the positive integers; function symbols f; the undefined value $\bot$ and the unit value $\mathbf{1}$, and three constructs with particular meaning in RESEDA. Data expressions are meant to be evaluated always in the context of a given subterm of a process p. Given such a context, the remaining constructs are:

1. ϕ, a path expression, identifying a set of subterms of P;
2. this, the subterm identified by the current context; and
3. rule, if used in a rule, the subterm identified by the context in which the rule is defined; and
4. trigger, the subterm identifying the data element that caused a spawning effect; and
5. $E{:}\mathit{attr}$, the (set of) attribute values at the current context.

Path expressions ϕ navigate between subprocesses in a manner that should be superficially familiar from, e.g., the UNIX filesystem or XPath expressions [7]: A path expression is a name (n), an instruction to move up (..), a wildcard ($*$) matching any child process, or the concatenation of such steps. Moreover, like in XPath, a path can be filtered by an expression along the way ($\psi[E]$).

In the concrete syntax, we distinguish path expressions from data expressions by prefixing the former with an `@`, and we allow abbreviations of the attribute selectors `value`, `executed`, etc. into simply `v`, `e`, etc.

Example 10. In Example 9, we see path expressions on lines 3, 9, 12-16 and 18. In line 3, for instance, the path expression `@/customers/customer/customer_id:value` starts with a / meaning that it selects data elements in the upper context (see Example 8). In line 12-16 we have an example of a filtered path expression, that only matches `order_line` data elements inside orders with a particular value for the `customer_id` and `wrap_item`. The path expression `customer_id:value` used in the first filter (line 13) matches data elements in deeper context levels, in this case the nested elements of the order element, while the path expression `rule/delivery_id:value` used in the second filter (line 15) matches in the context where the rule is defined, i.e. inside the `delivery` data element.

3.2 Structured data

A data term d consists of either a *computation data element* $n[e]{:}m\{p\}$ or an *input data element* $n[?]{:}M\{P\}$. In either case, the data element is associated with a *marking* M and a subprocess P. Computation data elements further have an associated expression E, indicating what computation they carry out when executed (updated). Input data elements are syntactically distinguished from computation data elements by having a question mark '?' instead of such an expression.

A marking M is a 4-tuple comprising:

1. a boolean h, indicating whether the data element has been executed at least once before,
2. a boolean i indicating whether the data element is currently included,
3. a boolean r indicating whether the data element is currently pending (i.e. required to be updated again if included) and finally
4. a value v indicating the current value of the data element.

A special undefined value is used when a data element that has yet to compute/receive a value for the first time and has no initial value.

In the ASCII variant of this syntax used in previous sections and in the online tool, we admit a number of short-hands:

1. Markings are by default $(\mathsf{false}, \mathsf{true}, \mathsf{true}, \bot)$, that is, unless otherwise specified, data elements that have not been executed, are included, not pending, and have the "undefined" value.
2. Data elements are marked "executed", "excluded" respectively "pending" in the ASCII syntax by prefixing them with `:`, `%`, respectively `!`.
3. Data elements with no square brackets are considered computation data elements computing the unit value, $\mathbf{1}$.

Example 11. In Example 9, we see on lines 8–10 pending data elements, and in line 10 also an excluded data element.

3.3 Relations

A relation definition R consists of one of six relations:

1. the condition $\rightarrow\!\bullet$ / `-->*`, indicating that data elements on the right cannot execute unless each data on the left is either marked not included or executed;
2. the milestone $\rightarrow\!\diamond$ / `--<>`, indicating that elements on the right cannot execute unless each element on the left is marked not included or not pending;
3. the response $\bullet\!\rightarrow$ / `*-->`, indicating that whenever some data element on the left executes, all data elements on the right become marked pending;
4. the dynamic inclusion $\rightarrow\!+$ / `-->+`, indicating that whenever some element on the left executes, all elements on the right become marked included;
5. the dynamic exclusion $\rightarrow\!\%$ / `-->%`, indicating that whenever some element on the right executes, all elements on the right become excluded; and finally,
6. the spawn $\twoheadrightarrow$ / `-->>`, indicating that whenever some data element on the left execute, new sub processes (data elements and rules) as indicated on the left are merged into the current process.

For the first five relations, both the domain and range of the relation is a path expression indicating which data elements (left-hand side) are being related to which other data elements (right-hand side). For the latter relation, the right-hand side indicates the process to be spawned.

Example 12. We see an example of the spawn relation in Example 9 on lines 6 and 12, where a process is spawned for each execution of respectively the `create_delivery` data element inside the `delivery` data element and the `order_line_id` data element.

$$
\begin{array}{rcll}
V & ::= & \rho \mid () \mid \vec{V} \mid \bot \mid \mathit{number} \mid \mathit{string} \mid \mathsf{true} \mid \mathsf{false} & \text{(Values)} \\
\mathcal{E} & ::= & \mathsf{pend} \mid \mathsf{incl} \mid \mathsf{excl} \mid \mathsf{spawn}(T, x.V) & \text{(Effects)}
\end{array}
$$

Figure 4: Auxiliary Syntax for Semantics

4 Semantics

The semantics of RESEDA is given as a transition system, where states are the data elements combined with information about which elements are pending for update and currently included. The transitions are events corresponding to executions (i.e. value updates) of either input data elements or computation data elements, specified as a path to the corresponding data element, and, if the event is an input event, a value to input. To define this transition system, we must define two functions:

1. One that determines, for given a RESEDA term, which data elements are currently executable in that term; and
2. One that determines, for given a RESEDA term and a data element of that term, what is the next state (term) after executing this data element.

Because data elements in a process are not necessarily identified uniquely by name, we identify a data element by a path from the root to it. Such a path can be uniquely represented as a sequence of integers, each integer representing the left-to-right index of the subterm to descend into.

Example 13. Consider the following process (using the shorthand that no brackets indicate computation data elements computing the unit value).

```
A, B { A, A }, B { A, A }
```

To identify the data element corresponding to the fourth-from-the-right `A`, we follow the path $2, 0$, that is, first descend into the third child (right-most `B`), then into the first child (left-most `A` inside that `B`).

In the sequel, we let ρ range over such sequences of numbers, calling such a ρ a *location*. A path expression evaluates to the set of locations it identifies; for a path expression ϕ evaluated from the location ρ in a process P, we write this set $[\![\phi]\!]^P_\rho$. Similarly, the value of a data expression E we write $(\!| E |\!)^P_\rho$. It is straightforward to give a mutually recursive definition of these two evaluation functions; refer to Appendix A.3.

Knowing how to evaluate data and path expressions in a process P, we move on to computing which effects are at all enabled.

4.1 Enabledness

The notion of *enabled* data element ρ is defined in Figure 7. For a data element ρ to be enabled, its parents must also be, viz. the recursive definition of the $\mathsf{enabled}$ function. Moreover, the data element ρ that we are considering for enabledness must satisfy that ($\mathsf{enabled}'$):

1. every data element ρ' that is a condition for ρ must be either excluded or previously executed,
2. every data element ρ' that is a milestone for ρ must be either excluded or not pending, and finally,
3. the data element ρ must itself be included.

The $\mathsf{enabled}'$ function makes this check. Note how in Figure 7 it is always invoked by $\mathsf{enabled}$ at the root P: Both for the data element ρ we are checking for enabledness and *all of its parents*, we must traverse the entire process, looking for relations that might prevent it from being enabled.

4.2 Transitions

Finally, we define the transition system of a process P. Recall that we intend states to be processes P and transitions to be either locations ρ (identifying data elements), or pairs of locations ρ and (input) values v.

We begin by defining the set of *effects* executing a data element ρ on a process P' in the context of a global process P; we write this set $\mathsf{effects}_P(P', \rho)$, defined by cases in Figure 5. The first case described defines that the effects are computed from the relations present in the term and inductively computed in the structure of the data element. Next, we treat each base case separately, by registering which data element may cause what particular effect (pend, incl, excl), and to what other data elements it is applied to.

Finally, we define the labelled transition system by the transition rules in Figure 6, making a distinction between computation events (rule COMPUTE) and input events (rule UPDATE). The former transitions are labelled by a data element location ρ, whereas the latter is labelled $\rho(v)$ by both a data element location ρ and an input value v. In both cases the selected data element ρ must be enabled, and the effects of the data element's execution δ are applied to a process P' where the target data element is already (re)computed/updated.

Computing a transition involves in either case three steps:

1. In either case, the data element in question must be *enabled* as defined in Figure 7 for the transition to apply.
2. Then, for an input data element ρ, the value of ρ is updated with the value v input $(\mathsf{compute}^P_\rho(P, \rho))$. For a computation data element, the computation is executed $(\mathsf{update}^P_\rho(P, \rho, v))$.
3. Finally, the effects of executing ρ are computed and applied $(\mathsf{effects}_{P'}(P', \varepsilon)(\rho))$.

See Appendix A.3 for full definitions of $\mathsf{compute}$, update, and $\mathsf{effects}$.

This concludes our introduction of RESEDA semantics. We proceed in the following Sections to study static properties of the language: its expressiveness and its relation to the notion of liveness.

5 Expressive power

We report in this section that Reseda processes embed DCR* processes [5][1].

Theorem 1. *Let G be a DCR* process. Then there exists a Reseda process $[\![G]\!]$ s.t. the languages of G and $[\![G]\!]$ are identical. Moreover, every data expression in $[\![G]\!]$ is trivial.*

Proof. It is sufficient to find an encoding of the "new name" construct of DCR*. Define $[\![G]\!]$ by observing that for each mention of an event e in a relation of P, there is a unique path expression ϕ_e identifying syntactically the location of e. This insight gives us a way to convert translate a relation $e \mathrel{\mathcal{R}} f$ of DCR* to a Reseda relation $\phi_e \mathrel{\mathcal{R}} \phi_f$. □

We can use this encoding to study the expressive power of RESEDA. In [5] it is shown how to encode counter machines in DCR* and from that proving that the problem determining whether a given sequence of events is accepted by a DCR* process is undecidable. We obtain via the encoding of DCR* in RESEDA that RESEDA is Turing complete and the problem of whether a sequence of events leads to an accepting state is undecidable even in the fragment where every data expression (but not path expression) is trivial.

Corollary 1. RESEDA *can encode counter machines using only trivial data expressions, proving that this fragment of* RESEDA *is Turing complete.*

Proof. Immediate from Turing-completeness of DCR* and Theorem 1 above. □

Corollary 2. *The problem of determining whether the language of a* RESEDA *process P includes a string s of events is undecidable. This problem is undecidable already in the fragment of* RESEDA *where every data expression is trivial.*

[1] Restating the syntax and semantics of DCR* is out of scope for the present paper; we refer instead the reader to [5].

6 Liveness

We aim for the central property that a pending data-element can always be either executed or excluded in the future. When this is the case we say that the program is *live*.

Definition 1. Let P be a process. We say that P is *live* iff whenever Q is reachable from P and an event ρ is included and pending in Q, then there exists an R reachable from Q s.t. either ρ was executed on the way from Q to R, or ρ is not included in R.

We will not be able to achieve this property for arbitrary RESEDA processes. In particular, we rule out RESEDA processes with subprocesses whose bodies spawn copies of themselves. We formalise processes not including this behaviour as *bounded* processes.

Definition 2 (Runs)**.** Let P be a process. The set $R(P)$ of *runs* of P is the set

$$R(P) = \{\rho_1, \ldots, \rho_n \mid P = P_1 \xrightarrow{\rho_1} P_2 \xrightarrow{\rho_2} \cdots \xrightarrow{\rho_n} P_n\} \, .$$

The *non-repeating runs* $R^!(P)$ of P is the subset of $R(P)$ defined by $\rho_1, \ldots, \rho_n \in R^!(P)$ iff the ρ_i are pairwise distinct, i.e., for $1 \leq i < j \leq n$ we have $\rho_i \neq \rho_j$.

Definition 3 (Bounded process)**.** Let P be a process. We say that P is *bounded* iff for every Q reachable from P, $R^!(Q)$ is finite.

Example 14. Despite appearances, the following process *is* bounded:

```
A ; A -->>  B
```

Consider the following particularly instructive run of this process (writing for brevity only the Structured Data part of the process, the only thing that changes):

$$A \xrightarrow{A} AB \xrightarrow{B/0} AB \xrightarrow{A} ABB \xrightarrow{B/1} ABB \cdots$$

Clearly, this run is both not only infinite, but mentions also infinitely many distinct events ($B/0, B/1, \ldots$). So in what sense is this process bounded? Well, observe that the run in question is not repeating, in fact, it repeats A infinitely. It is in fact impossible to construct an infinite run of this process *without repeatedly executing the A event*. It is in this sense that the process is bounded.

In contrast, here is an example of a process that is not bounded:

```
A ; A -->>  A
```

This one has the run (again omitting relations):

$$A \xrightarrow{A[0]} AA \xrightarrow{A[1]} AAA \xrightarrow{A[2]} AAAA \xrightarrow{A[3]} \cdots$$

Definition 4. Let P be a process, and suppose P contains as subterm a relation $\phi_b \twoheadrightarrow_{x \in E} Q$, and in turn that Q contains as subterm an event a. Let b be any name mentioned in ϕ. In this case we say there is a *potential spawn dependency* (PSD) from b to a in P.

Example 15. In the preceding example, the process `A ; A -->>  B` has PSD from `A` to `A`; and the process `A ; A -->>  A` has a PSD from `A` to itself. Notice that the only the latter, which exhibits unbounded behaviour, has a cyclic PSD.

The following Lemma is then immediate.

Lemma 1. *Let P be a process and suppose the graph of PSDs in P is acyclic. Then P is bounded.*

Now that we have identified the class of bounded processes, we show how to ensure liveness . The basic idea follows that of [2] in constructing a static approximation of the dependencies between events that may be incurred by conditions and milestones, referred to as the *static inhibitor graph*, and then require the static inhibitor graph to be acyclic. To execute a given event, we may simply execute "from the bottom" of that acyclic graph, eventually reaching the desired event at the top.

Definition 5 (Static inhibitors)**.** Let P be a process, and let ρ be an event of P. The *static inhibitors of* ρ *in* P, written $SI_P(\rho)$ is the set of events ρ' of P s.t. there is in P a condition or milestone $\rho' \rightarrow\!\bullet\ \rho$ or $\rho' \rightarrow\!\diamond\ \rho$. The *static inhibitor graph of* ρ *in* P, written $SI^*_P(\rho)$ is the graph arising as the closure under $SI_P(-)$ of ρ, further closed under the rule that if $\rho' \bullet\!\!\rightarrow \rho''$ or $\rho' \rightarrow\!+\ \rho''$ in P with $\rho', \rho'' \in SI^*_P(\rho)$ then there is an edge ρ', ρ'' in $SI^*_P(\rho)$.

While the core of the idea of [2] remains in the present development, we also need to account for events that are generated dynamically as a result of spawn relations. We do so by requiring that the static inhibitor graph can at any time be embedded in a fixed, "global" graph, referred to as the *dependency control*.

Definition 6 (Dependency control)**.** Let P be a process, and let ρ be an event of P. A *dependency control* (G_ρ, ι^-_ρ) for ρ is a finite non-empty acyclic graph G_ρ and a family of graph homomorphims $\iota^Q_\rho : SI^*_Q(\rho) \to G$, for Q reachable from P, s.t. for any Q reachable from P and Q' with $Q \xrightarrow{\rho''} Q'$:

1. if $\iota^Q_\rho(x)$ defined, then $\iota^Q_\rho(x) = \iota^{Q'}_\rho(x)$, and
2. for $\rho' \in SI^*_{Q'}(\rho) \setminus SI^*_Q(\rho)$ we have $\iota^{Q'}_\rho(\rho') > \iota^{Q'}_\rho(\rho'')$.

Given a dependency control, we can find appropriate first actions to execute. Notice that, exactly because events can be dynamically generated, the planned actions to execute *must* be derived from the global dependency control, and not from the particular static inhibitors graph: It may be that a dependency has yet to materialise; in that case, it will not be present in the static inhibitors graph, but it *will* be represented in the dependency control.

Definition 7 (Planned action)**.** Let P be a process, ρ an event of P, and (G_ρ, ι^-_ρ) a dependency control for ρ. Suppose Q is reachable from P. Define Λ to be the subset of events of $SI_Q(\rho)$ that are enabled and non-executed or pending that are minimal in G_ρ, i.e.,

$$\Lambda = (\iota^Q_\rho)^{-1}(\min\{\iota^Q_\rho(\rho') \mid \rho' \in SI^*_Q(\rho) \land \rho' \text{ included and (non-executed or pending)}\})$$

Assume an arbitrary but fixed choice of minimal elements for any such Λ, and define the *planned action* $\lambda_\rho(Q)$ to be to be that choice when Λ is non-empty, undefined otherwise.

We can now prove that for graphs with dependency control, we can always find a planned action when we need to.

Lemma 2. *Let P be a process, let ρ be an event of P, and suppose (G_ρ, ι^-_ρ) is dependency control for ρ. Then at each Q reachable from P where ρ is included, either (a) $\lambda_\rho(Q)$ is defined and enabled or (b) ρ is enabled.*

Proof. Suppose Q is reachable from P with ρ included. If ρ is enabled, we are done, so suppose not. In this case ρ has an unfulfilled milestone or condition, so $SI^*_Q(\rho)$ is non-empty. Because $\iota^{SI^*_Q(\rho)}_\rho$ is acyclic so is $SI^*_Q(\rho)$; it follows that $\lambda_\rho(Q)$ is defined. By definition, $\lambda_\rho(Q)$ has no included unsatisfied conditions or milestones and hence is enabled. □

The difficult bit is establishing that undertaking a finite amount of planned actions is enough. The following Lemma establishes the key properties of planned actions to that end: actions are necessarily planned "bottom-up" (1), and you never plan the same action twice (2). We shall see that for bounded processes, these two properties are sufficient to ensure existence and termination of execution plans.

Lemma 3. *Let P be a process, let ρ be an event of P, and suppose (G_ρ, ι^-_ρ) is dependency control for ρ. Let Q be a process reachable from P, and suppose we have a sequence $Q = Q_1, \dots, Q_n$ s.t.*

$$Q = Q_1 \xrightarrow{\lambda_\rho(Q_1)} Q_2 \xrightarrow{\lambda_\rho(Q_2)} \cdots \xrightarrow{\lambda_\rho(Q_{n-1})} Q_n\ .$$

Then for each $1 \leq i < j \leq n$ we have:

1. $\iota_\rho^{Q_j}(\lambda_\rho(Q_i)) \not> \iota_\rho^{Q_j}(\lambda_\rho(Q_j))$.

2. $\lambda_\rho(Q_i) \neq \lambda_\rho(Q_j)$.

Proof. Part (1). By induction on n. The base case is trivial; so suppose the theorem holds for k, consider the case $k+1$, and suppose for a contradiction that for some $i < k+1$ we have

$$\iota_\rho^{Q_{k+1}}(\lambda_\rho(Q_{k+1})) > \iota_\rho^{Q_{k+1}}(\lambda_\rho(Q_i)) \tag{1}$$

For convenience write $\rho_i = \lambda_\rho(Q_i)$ and $\rho_{k+1} = \lambda_\rho(Q_{k+1})$, whence (1) becomes the slightly more palatable

$$\iota_\rho^{Q_{k+1}}(\rho_{k+1}) > \iota_\rho^{Q_{k+1}}(\rho_i) \tag{2}$$

Because ρ_i was minimal (Definition 7) at Q_i, ρ_{k+1} was at that point either excluded, or included, executed and not pending. It follows that at some l with $i < l < k+1$, ρ_{k+1} became included and not executed or included and pending. But then $\rho_l = \lambda_\rho(Q_l)$ had inclusion and/or response relations to ρ_{k+1}, so by Definition 5 (left-most comparison) and assumption (right-most comparison) we must have:

$$\iota_\rho^{Q_{k+1}}(\rho_l) < \iota_\rho^{Q_{k+1}}(\rho_{k+1}) < \iota_\rho^{Q_{k+1}}(\rho_i) \ . \tag{3}$$

But this contradicts the IH that the theorem holds at $l < k+1$.

Part (2). Assume for a contradiction that for some i, j we have $x = \lambda_\rho(Q_i) = \lambda_\rho(Q_j)$. As in the previous case, clearly x was made included and/or pending by some l in $i < l < j$, and we find

$$\iota_\rho^{Q_n}(\lambda_\rho(Q_i)) = \iota_\rho^{Q_n}(\lambda_\rho(Q_j)) > \iota_\rho^{Q_n}(\lambda_\rho(Q_l)) \ ,$$

contradicting the just-established Part (1). □

We can now prove that if an event ρ has dependency control, we can always find a way to execute ρ.

Lemma 4. *Let P be a bounded process, let ρ be an event of P, and suppose (G_ρ, ι_ρ^-) is dependency control for ρ. Then at each Q reachable from P with ρ included, there exists a sequence*

$$Q = Q_1 \xrightarrow{\lambda_\rho(Q_1)} \cdots \xrightarrow{\lambda_\rho(Q_{n-1})} Q_n$$

such that at Q_n the event ρ is either enabled or excluded.

Proof. Let Q be reachable from P with ρ included in Q. Inductively construct the sequence $Q_1, \ldots, Q_n$ as follows: Set $Q_1 = Q$. When Q_i is defined, by Lemma 2, either ρ enabled, in which case we are done, or $\lambda_\rho(Q_i)$ is defined and enabled; obtain Q_{i+1} by executing that event. This process terminates because by Lemma 3 Part (2) each $\lambda_\rho(Q_i) \neq \lambda_\rho(Q_j)$ so by boundedness of P and Lemma 1 this process terminates. □

Using Lemma 4, we can find execution plans for an event ρ *provided* we have dependency control. The question remains: How do we find a dependency control, i.e., a graph G_ρ and an associated family of graph homomorphisms ι_ρ^-? We shall see that simply unfolding recursively every spawn relation in a given process P once will be almost enough; "almost" because we will need to also artificially add edges from any event ρ' to any event ρ'' spawned by ρ'.

Definition 8. Let P be a process. The *unfolding of P* is the process P' obtained from P by recursively unfolding every spawn in P once. For an event ρ of P', the *dependency graph G_ρ* is the graph $SI^*_{P'}(\rho)$, adding an edge ρ', ρ'' whenever ρ'' was spawned by ρ'.

Theorem 2. *Let P be a bounded process, let Q be a process reachable from P, and let ρ be an event of Q. If for all $\rho' \in \mathsf{events}(P)$ the dependency graphs $G_{\rho'}$ of P is acyclic, then so is the dependency graph for ρ in Q.*

Proof. By construction of $G_{\rho'}$ as the unfolding of P. □

Corollary 3. *Let P be a bounded process, and let P' be the unfolding of P. If for every node ρ of P' the dependency graph G_ρ of P' is acyclic, then P is live.*

We conclude by noting that the prerequisites of Corollary 3 are efficiently computable; the following proposition forms the basis of the on-line implementation, available at `http://dcr.itu.dk/reseda`.

Proposition 1. *Let P be a process. We can compute in polynomial time (a) whether P is bounded, (b) the unfolding P' of P, (c) the dependency graphs for the events of P and (d) whether these dependency graphs are acyclic.*

7 Discussion & Related work

So-called data-centric process notations, including artefact- and object-centric notations [9,17,18], giving more equal emphasis on the modelling of data and behaviour. However, as pointed out in [17,18], the artefact-centric process notations tend not to provide a good overview of the overall data or behavioural model, since the behavioural modelling is distributed on artefacts. The authors of [17] propose instead the Object-Centric Behavioural Constraints (OCBC) approach to provide combined presentations of the data-model and behavioural constraints. The OCBC notation also include relations between the two models (not shown in Figure 1 and Figure 2). In this example, the cross model relations indicate a one-to-one relationship between activities *create order* and *pick item* (marked by (1) and (2) respectively) and the elements of the classes Order and Order Line, indicating that these activities represent the activities for creating elements of these classes. This is reflected in the RESEDA process (cf. Example 8) by the spawn rules triggered by the `create_order` and `pick_item` input data elements.

Other important related approaches include *case handling* [19], *business artefacts* [3,11] and the Guard-Stage-Milestone (GSM) approach [8]. The case handling paradigm is similar to RESEDA by being data-driven, based on forms, and distinguishing mandatory and optional data fields. Case handling approaches does however normally not support complex data modelling.

XML-based approaches that follow [1] do build interesting dynamic and nested databases by extending a local schema with external sources. Although structurally similar, our approach provides a fully expressive language to build and change the structureof nested data objects.

The business artefacts approach models life cycles of data objects as finite state machines and not initially supported by an operational semantics for execution [9,16]. This was provided with the GSM approach, for instance. The operational semantics for GSM is however considerably more complex than the one defined for RESEDA in the present paper and it is non-trivial to relate GSM to the declarative constraints present in DECLARE and DCR on which RESEDA and OCBC is based, e.g., see [8] for a comparison between GSM and DCR.

8 Conclusions

We introduced a novel executable, declarative data-centric process language RESEDA unifying data and behavioural modelling as reactive semi-structured data, inspired in equal parts by declarative business process modelling notations and reactive spreadsheet programming. We demonstrated the use of RESEDA on a process given in [17,18]. We provided a formal semantics for RESEDA defining an execution semantics as rewrite rules of processes and outlined the proof of a sufficient, polynomially decidable criteria for liveness of bounded processes. Finally we noted that Turing-completeness of RESEDA can be derived from the fact that RESEDA embeds the DCR* language. We leave for future work to investigate generalising the results of safe refinement in [6] to RESEDA. The RESEDA language is supported by a prototype implementation available both in source form and on-line at `http://dcr.itu.dk/reseda`. All major examples of the present paper are included in the tool.

$$\begin{array}{rcl}
\mathsf{effects}_P(\vec{D};\vec{R},\rho) & \triangleq & \bigcup_{R\in\vec{R}}\ \mathsf{effects}'_P(R,\rho)\cup\bigcup_{1\leq i<n}\ \mathsf{effects}_P(D_i,\rho i) \\
 & & \text{where } P_i = n_i[.]{:}M_i\{P_i\} \\
\mathsf{effects}'_P(\phi \bullet\!\!\rightarrow \phi',\rho) & \triangleq & \{(\rho',(\mathsf{pend},\rho''))\mid(\rho',\rho'')\in[\![\phi]\!]^P_\rho\times[\![\phi']\!]^P_\rho\} \\
\mathsf{effects}'_P(\phi \rightarrow\!+ \phi',\rho) & \triangleq & \{(\rho',(\mathsf{incl},\rho''))\mid(\rho',\rho'')\in[\![\phi]\!]^P_\rho\times[\![\phi']\!]^P_\rho\} \\
\mathsf{effects}'_P(\phi \rightarrow\!\% \phi',\rho) & \triangleq & \{(\rho',(\mathsf{excl},\rho''))\mid(\rho',\rho'')\in[\![\phi]\!]^P_\rho\times[\![\phi']\!]^P_\rho\} \\
\mathsf{effects}'_P(\phi \twoheadrightarrow P',\rho) & \triangleq & \{(\rho',(\mathsf{spawn}(P'\{^{(\!|\rho':v|\!)^P_\rho}/_{\mathsf{trigger}:v}\},\rho'))\mid\rho'\in[\![\phi]\!]^P_\rho\}
\end{array}$$

Figure 5: Effects of Terms

$$\frac{\mathsf{enabled}_P(\rho)\quad P'=\mathsf{compute}^P_\rho(P,\rho)\quad \delta=\mathsf{effects}_{P'}(P',\varepsilon)(\rho)}{P\xrightarrow{\rho}P'\lhd\delta}\ (\textsc{Compute})$$

$$\frac{\mathsf{enabled}_P(\rho)\quad P'=\mathsf{update}^P(P,\rho,v)\quad \delta=\mathsf{effects}_{P'}(P',\varepsilon)(\rho)}{P\xrightarrow{\rho(v)}P'\lhd\delta}\ (\textsc{Update})$$

Figure 6: Labelled Transition System

$$\begin{array}{l}
\mathsf{enabled}'(\rho,P,\vec{D};\vec{R},\rho') = \forall\phi\rightarrow\!\bullet\ \phi'\in R.\ \rho\in[\![\phi']\!]^P_{\rho'}\implies[\![\phi[\mathsf{included}]]\!]^P_{\rho'}=[\![\phi[\mathsf{included}\wedge\mathsf{executed}]]\!]^P_{\rho'} \\
\qquad\wedge\forall\phi\rightarrow\!\diamond\ \phi'\in R.\ \rho\in[\![\phi']\!]^P_{\rho'}\implies[\![\phi[\mathsf{included}]]\!]^P_{\rho'}=[\![\phi[\mathsf{included}\wedge\neg\mathsf{pending}]]\!]^P_{\rho'} \\
\qquad\wedge\forall i\in\mathsf{size}(\vec{D}).\ \mathsf{enabled}'(\rho,P,P'_i,\rho' i)(\text{ where }\vec{D}=P_1,\ldots,P_n\text{ and }P_i=n_i[.]{:}M_i\{P'_i\}\ )
\end{array}$$

$$\mathsf{enabled}_P(\rho i)=\mathsf{enabled}_P(\rho)\wedge\mathsf{enabled}'(\rho i,P,P,\epsilon)\qquad\text{and}\qquad\mathsf{enabled}_P(\epsilon)=\mathsf{true}$$

Figure 7: Enabledness.

References

[1] Serge Abiteboul, Omar Benjelloun, and Tova Milo. The active xml project: an overview. *The VLDB Journal*, 17(5):1019–1040, Aug 2008.

[2] David A. Basin, Søren Debois, and Thomas T. Hildebrandt. In the nick of time: Proactive prevention of obligation violations. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*, pages 120–134, 2016.

[3] K. Bhattacharya, R. Hull, and J. Su. A data-centric design methodology for business processes. pages 503–531, 2009.

[4] Søren Debois, Thomas Hildebrandt, and Tijs Slaats. Hierarchical declarative modelling with refinement and sub-processes. In *International Conference on Business Process Management*, pages 18–33. Springer, Cham, 2014.

[5] Søren Debois, Thomas T. Hildebrandt, and Tijs Slaats. Safety, liveness and run-time refinement for modular process-aware information systems with dynamic sub processes. In *FM 2015*, pages 143–160, 2015.

[6] SÃ¸ren Debois, Thomas Hildebrandt, and Tijs Slaats. *Safety, Liveness and Run-time Refinement for Modular Process-Aware Information Systems with Dynamic Sub Processes*, pages 143–160. Springer, Germany, 2015.

[7] Denise Draper, Peter Fankhauser, Mary Fernandez, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jarome Simon, and Philip Wadler. XQuery 1.0 and XPath 2.0 formal semantics. *W3C recommendation*, 23, 2007.

[8] R. Eshuis, S. Debois, T. Slaats, and T. Hildebrandt. Deriving consistent gsm schemas from dcr graphs. In *Service-Oriented Computing. ICSOC 2016.*, volume 9936 of *Lecture Notes in Computer Science*. Springer, Cham, 2016.

[9] Hull R. et al. Introducing the guard-stage-milestone approach for specifying business entity lifecycles. In *Web Services and Formal Methods. WS-FM 2010*, volume 6551 of *Lecture Notes in Computer Science*, Berlin, Heidelberg, 2011. Springer.

[10] Thomas T Hildebrandt and Raghava Rao Mukkamala. Declarative event-based workflow as distributed dynamic condition response graphs. *arXiv:1110.4161*, 2011.

[11] Richard Hull. Artifact-centric business process models: Brief survey of research results and challenges. In *OTM '08. Part II on On the Move to Meaningful Internet Systems*, OTM '08, pages 1152–1163. Springer, 2008.

[12] James McKinna. Why dependent types matter. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 1–1, New York, NY, USA, 2006. ACM.

[13] M. Pesic, H. Schonenberg, and W.M.P. van der Aalst. DECLARE: Full Support for Loosely-Structured Processes. In *EDOC '07*, pages 287–. IEEE, 2007.

[14] Peter Sestoft. *Spreadsheet implementation technology. Basics and extensions*. MIT Press, United States, 2014.

[15] Jianwen Su, Lijie Wen, and Jian Yang. From data-centric business processes to enterprise process frameworks. In *21st IEEE International Enterprise Distributed Object Computing Conference, EDOC 2017, Quebec City, QC, Canada, October 10-13, 2017*, pages 1–9, 2017.

[16] Yutian Sun, Wei Xu, and Jianwen Su. Declarative choreographies for artifacts. In *Service-Oriented Computing - 10th International Conference, ICSOC 2012, Shanghai, China, November 12-15, 2012. Proceedings*, pages 420–434, 2012.

[17] W. van der Aalst, A. Artale, Montali, M., and S. Tritini. Object-centric behavioral constraints: Integrating data and declarative process modelling. In *DL 2017, International Workshop on Description Logics*, volume 1879 of *CEUR Workshop Proceedings*, 2017.

[18] Wil M. P. van der Aalst, Guangming Li, and Marco Montali. Object-centric behavioral constraints. *CoRR*, abs/1703.05740, 2017.

[19] Wil M. P. van der Aalst and Mathias Weske. Case handling: A new paradigm for business process support. *Data Knowl. Eng.*, 53(2):129–162, May 2005.

A Appendix

A.1 Path, data, and expression evaluation

We give a mutually recursive semantics of path expressions and data expressions in Figures 9 and 8.

We write $[\![\phi]\!]^P_\rho$ to denote the exact set of locations that the path expression ϕ corresponds to when evaluated from location ρ. This set is necessarily also dependent on the global process P, because the path expression may both "move up" ($\phi = \phi'/..$) and require starting from the root ($\phi = /\phi'$). We remark on this evaluation function:

1. The evaluation of the root path expression, $[\![/\psi]\!]^P_\rho$, resets the context of evaluation (ρ) to ε in $[\![\psi]\!]^P_\varepsilon$.
2. The evaluation of the base cases of path expressions, . and .., is also defined by manipulation of the location context of the semantic function ρ.
3. The next two cases both select events in the local context based on a name (n) or a wildcard ($*$) and return the corresponding indexes.
4. In the case of a filter on events based on the denotation of an expression in the local context, the concrete set of events identified by the path expression ($[\![\psi]\!]^P_\rho$) is collected and the expression is evaluated for each one of its elements ($(\!| E |\!)^P_{\rho'}$).
5. A similar situation occurs when considering a composed path expression with base path elements as suffixes. The denotation is obtained by evaluating the last component of the path in all possible contexts denoted by the prefix expression.

Next, we define in Figure 9 the semantics of expressions E, similarly in the context of a global process P and a starting point ρ. We write in this case $(\!| E |\!)^P_\rho$ for the evaluation of E. Notice that locations and sets of locations are manipulated by the language as first-class values. We remark on the definition:

1. As mentioned, we assume the existence of constants c and abstract all possible operations on basic type values by means of a function f, for which we admit a native semantics restricted to basic type values for the arguments.
2. The constant this denotes the index of the event given as current location.
3. Event attributes accessed through expression modifiers (e.g. $(\!| E\text{:executed} |\!)^P_\rho$) are directly selected from the marking of the concrete event $P(\rho)$, for all the indexes of target events. We also assume an implicit coercion from singleton lists of elements to their element.

A.2 Substitution

We define substitution in Figure 10 below.

A.3 Compute and update functions

The auxiliary functions in Figure 11 and 12 are both defined inductively on the structure of the process. They both traverse the process, destructuring the location ρ. In the case of computation and update functions, Figure 11, we have the use of expression semantic function in the base case. In the case of the application of effects, Figure 12, the marking of the events are modified accordingly. The spawn effect, $\mathsf{spawn}(\vec{D}';\vec{R}', x.v{:}\vec{v})$ where x is the cursor of the iteration and $v{:}\vec{v}$ is the list of elements, is operated by iterating the elements in the list, in the last two cases of Figure 12.

$$
\begin{array}{rr}
[\![/\psi]\!]^P_\rho \triangleq [\![\psi]\!]^P_\varepsilon & [\![n]\!]^P_\rho \triangleq \{\rho i \mid 0 \le i < \mathsf{size}(P(\rho)),\ P(\rho i) = n[]{:}M\{P\}\} \\
[\![.]\!]^P_\rho \triangleq \{\rho\} & [\![*]\!]^P_\rho \triangleq \{\rho i \mid 0 \le i < \mathsf{size}(P(\rho))\} \\
[\![..]\!]^T_{\rho i} \triangleq \{\rho\} & [\![\psi[E]]\!]^P_\rho \triangleq \{\rho' \mid \rho' \in [\![\psi]\!]^P_\rho, (\!| E |\!)^P_{\rho'} \neq \emptyset\} \\
 & [\![\psi/\alpha]\!]^P_\rho \triangleq \bigcup_{\rho' \in \chi} [\![\alpha]\!]^P_{\rho'} \text{ where } \chi = [\![\psi]\!]^P_\rho
\end{array}
$$

Figure 8: Semantics of Path expressions.

$$
\begin{array}{rr}
(\!| c |\!)^P_\rho \triangleq c & (\!| f(E_1,\dots,E_n) |\!)^P_\rho \triangleq f((\!| E_1 |\!)^P_\rho,\dots,(\!| E_n |\!)^P_\rho) \\
(\!| \phi |\!)^P_\rho \triangleq [\![\phi]\!]^P_\rho & (\!| E{:}\mathsf{executed} |\!)^P_\rho \triangleq \{h \mid \rho \in (\!| E |\!)^P_\rho \wedge P(\rho) = n[.]{:}(h,i,p,v)\{P'\}\} \\
(\!| \mathsf{this} |\!)^P_\rho \triangleq \rho &
\end{array}
$$

(Clauses for E:included(i), E:pending(p) and E:value(v) omitted to conserve space.)

Figure 9: Semantics of Data expressions.

A.4 Expressiveness

of Theorem 1. It is sufficient to find an encoding of the "new name" construct of DCR*. Define $[\![G]\!]$ by observing that for each mention of an event e in a relation of P, there is a unique path expression ϕ_e identifying syntactically the location of e. This insight gives us a way to convert translate a relation $e\ \mathcal{R}\ f$ of DCR* to a Reseda relation $\phi_e\ \mathcal{R}\ \phi_f$. □

Definition 9. Let P be a RESEDA process. Define for a label $\alpha = \rho$ or $\alpha = \rho(v)$ of a transition out of P the $\mathsf{symbol}(\alpha)$ to be the event identified by the underlying location ρ. Define for a run

$$P = P_1 \xrightarrow{\lambda_1} P_2 \xrightarrow{\lambda_2} \cdots$$

the (data-free) trace to be the string $\langle \mathsf{symbol}(\lambda_1), \mathsf{symbol}(\lambda_2), \dots\rangle$. Define then the *language* of P to be the set of strings generated by the finite and infinite runs of P.

of Corollary 2. Immediate from Turing-completeness of DCR* and Theorem 1. □

$(\vec{P};\vec{R})\{^v/_x\} = (P\{^{\vec{v}}/_x\});(R\{^{\vec{v}}/_x\})$

$(n[E]{:}M\{T\})\{^v/_x\} = n[E\{^v/_x\}]{:}M\{(T\{^v/_x\}\})$

$(n[?]{:}M\{T\})\{^v/_x\} = n[?]{:}M\{(T\{^v/_x\}\})$

$(\phi \rightarrow\!\bullet\ \phi)\{^v/_x\} = (\phi\{^v/_x\}) \rightarrow\!\bullet\ (\phi\{^v/_x\})$
...
$(\phi \twoheadrightarrow_{x\in E} T)\{^v/_x\} = (\phi\{^v/_x\}) \twoheadrightarrow_{x\in(E\{^v/_x\})} (T\{^v/_x\})$

$(/\psi)\{^v/_x\} = /(\psi\{^v/_x\})$

$\alpha\{^v/_x\} = \alpha$

$(\psi/\alpha)\{^v/_x\} = (\psi\{^v/_x\})/\alpha$

$(\psi[E])\{^v/_x\} = (\psi\{^v/_x\})[E\{^v/_x\}]$

$\mathsf{this}\{^v/_x\} = \mathsf{this}$

$\mathit{number}\{^v/_x\} = \mathit{number}$

$\mathit{string}\{^v/_x\} = \mathit{string}$

$()\{^v/_x\} = ()$

$\mathsf{true}\{^v/_x\} = \mathsf{true}$

$\mathsf{false}\{^v/_x\} = \mathsf{false}$

$(E + E)\{^v/_x\} = E\{^v/_x\} + E\{^v/_x\}$

$\phi\{^v/_x\} = \phi$

$(E{:}\mathit{attr})\{^v/_x\} = (E\{^v/_x\}){:}\mathit{attr}$

Figure 10: Definition of substitution

$$\mathsf{compute}^{P}_{\rho'}((\vec{D}, n_j[E]{:}(h,i,p,v)\{\,P'\,\}, \vec{D'})\ \vec{R}, j\rho)$$
$$\triangleq (\vec{D}, n_j[E]{:}(h,i,p,v)\{\,\mathsf{compute}^{P}_{\rho'}(P',\rho)\,\}, \vec{D'})\ \vec{R}$$

$$\mathsf{compute}^{P}_{\rho'}((\vec{D}, n_j[E]{:}(\,h\,,i,\,p\,,\,v\,)\{P\}, \vec{D'})\ \vec{R}, j)$$
$$\triangleq (\vec{D}, n_j[E]{:}(\,\mathtt{t}\,,i,\,\mathtt{f}\,,\,(\!|E|\!)^{P}_{\rho'})\,\{P\}, \vec{D'})\ \vec{R}$$

$$\mathsf{update}^{P}((\vec{D}, n_j[E]{:}(h,i,p,v)\{\,P'\,\}, \vec{D'})\ \vec{R}, j\rho, v')$$
$$\triangleq (\vec{D}, n_j[E]{:}(h,i,p,v)\{\,\mathsf{update}^{P}(P',\rho,v')\,\}, \vec{D'})\ \vec{R}$$

$$\mathsf{update}^{P}((\vec{D}, n_j[E]{:}(\,h\,,i,\,p\,,\,v\,)\{P\}, \vec{D'})\ \vec{R}, j, v')$$
$$\triangleq (\vec{D}, n_j[E]{:}(\,\mathtt{t}\,,i,\,\mathtt{f}\,,\,v'\,)\{P\}, \vec{D'})\ \vec{R}$$

Figure 11: Computing and updating values.

$$\begin{aligned}
((\vec{D}, n_j[E]{:}(h,i,p,v)\{P\}, \vec{D'})\ \vec{R}) \lhd (\mathcal{E}, j\rho) &\quad\triangleq\quad ((\vec{D}, n_j[E]{:}(h,i,p,v)\{P\} \lhd (\mathcal{E},\rho)\,, \vec{D'})\ \vec{R}) \\
((\vec{D}, n_j[E]{:}(h,i,\,p\,,v)\{P\}, \vec{D'})\ \vec{R}) \lhd (\mathsf{pend}, j) &\quad\triangleq\quad ((\vec{D}, n_j[E]{:}(h,i,\,\mathtt{t}\,,v)\{P\}, \vec{D'})\ \vec{R}) \\
((\vec{D}, n_j[E]{:}(h,\,i\,,p,v)\{P\}, \vec{D'})\ \vec{R}) \lhd (\mathsf{incl}, j) &\quad\triangleq\quad ((\vec{D}, n_j[E]{:}(h,\,\mathtt{t}\,,p,v)\{P\}, \vec{D'})\ \vec{R}) \\
((\vec{D}, n_j[E]{:}(h,\,i\,,p,v)\{P\}, \vec{D'})\ \vec{R}) \lhd (\mathsf{excl}, j) &\quad\triangleq\quad ((\vec{D}, n_j[E]{:}(h,\,\mathtt{f}\,,p,v)\{P\}, \vec{D'})\ \vec{R}) \\
(\vec{D};\vec{R}) \lhd (\mathsf{spawn}(\vec{P'};\vec{R'}, x.\varepsilon), \varepsilon) &\quad\triangleq\quad \vec{D};\vec{R} \\
(\vec{D};\vec{R}) \lhd (\mathsf{spawn}(\vec{D'};\vec{R'}, x.v{:}\vec{v})), \varepsilon) &\quad\triangleq\quad D, \overrightarrow{D'\{{}^{v}\!/_{x}\}};R, \overrightarrow{R'\{{}^{v}\!/_{x}\}} \lhd (\mathsf{spawn}(\vec{D'};\vec{R'}, x.\vec{v}), \varepsilon)
\end{aligned}$$

Figure 12: Application of Effects.