ReGraDa: Reactive Graph Data*

Leandro Galrinho[1], João Costa Seco[1,2], Søren Debois[3,5], Thomas Hildebrandt[4], Håkon Norman[4,5], and Tijs Slaats[4]

[1] School of Science and Technology, NOVA University of Lisbon
[2] NOVA Laboratory for Computer Science and Informatics (NOVA LINCS)
[3] Computer Science Department, IT University Copenhagen
[4] SDPS Section, Department of Computer Science, University of Copenhagen
[5] DCR Solutions (`DCRSolutions.net`)

Abstract. Business processes, data, and run-time control state are all key elements in the design of enterprise applications. However, the different layers for processes, data, and control are usually represented using different technologies that must be explicitly combined and kept in sync. We introduce ReGraDa, a process definition and programming language inspired by the declarative business process language Dynamic Condition Response Graphs and targeting the integrated description of business processes, data and run-time state as reactive graph data. ReGraDa processes compile directly to a data-centric graph-based system (*neo4j/cypher*), allowing for the database system to manage a process without the need for an external execution engine.
The underlying graph structure allows for the definition of native data relations between data elements that subsumes the integrity guaranties of relational and the semi-structured data models. Graph relationships are also used to represent data-dependency and control-flow in the database. We present the reactive semantics of the language, a translation from ReGraDa to *cypher*, evaluate its performance, and briefly discuss future work and applications.

1 Introduction

Process-aware information systems [4] include both control-flow and data. The notions of control and data are, however, often treated separately: process models may refer to specific documents or data values to guide decision making, but data manipulation is largely handled outside the processes control. Moreover, as pointed out by several authors [16,17], even among notations that provide bindings for data and computation, it remains cumbersome or impossible to model complex data models and dependencies between data and activities belonging to multiple instances of processes.

* Supported by Innovation Fund Denmark (EcoKnow.org & DREAM), Independent Research Fund Denmark (PAPRiCAS), EU MSCA-RISE BehAPI (ID:778233), NOVA LINCS UID/CEC/04516/2013, and GOLEM Lisboa-01-0247-Feder-045917.

These difficulties hinder both representation and reasoning about the full process behaviour, and form a practical barrier to adoption of system implementations based on formal workflow models.

Recent formalisms attempt to address these difficulties in combining data-modeling and declarative process-modeling, in various ways. The Case Management Model and Notation (CMMN) [7] makes data a first class citizen and arbiter of activity availability. The Object-Centric Behavioural Constraints (OCBC) [17] formalism cleverly ties together DECLARE and ER-modelling, explaining, e.g., how the satisfaction of a response constraint requires the creation of a data object. Extensions of Petri nets with identifiers [12, 18] model the interplay of information and processes and model the influence of process transitions in the data model using first-order logic over finite sets. Finally, the Reactive Semi-structured Data Formalism removes the distinction between "activity" and "data" in a constraint-based language [13].

The present work advances this research agenda in two directions at once: First, we **introduce graph data and queries**, most notably aggregating queries, into the language of ReSeDa [13] and relaxing the semi-structured data, resulting in the more general language ReGraDa, short for Reactive Graph Data. While other data-centric process models use loosely coupled data relations (values as keys), which provide weak data-integrity guarantees, ReGraDa provides a declarative definition of processes, graph data modeling, queries, and reactive computation. We formally define the syntax and semantics of the language and then focus on its pragmatics. The formal results on liveness properties of processes are out of the scope of this work, as they conservatively extend the results already obtained in ReGraDa.

Second, we **provide a high-performance implementation** of this language via a translation of ReGraDa programs into a contemporary graph database engine query language, specifically the query language *cypher* [6] of the graph database *neo4j*. The database computation capabilities and reactive mechanisms (triggers) are strong enough to not only implement the reactive embedded query language of ReGraDa, but also its process semantics: Process execution is fully and autonomously realised in the database. We provide an initial exploratory empirical study of the performance of translated ReGraDa programs in the latter section of this paper; initial results are encouraging.

ReGraDa (and ReSeDa) are inspired by both the declarative Dynamic Condition Response (DCR) graph process notation [1, 8, 15] and the widely used reactive model of data and behaviour used in spreadsheets [3, 14] and provide a unified specification of data, behaviour, and run-time execution state. However, ReGraDa differs from DCR Graphs in several ways. Firstly, it introduces a distinction between *input* and *computation* events, here called data elements. Secondly, ReGraDa allows data elements to be related in a general labelled graph structure and to be referenced in computation expressions, either directly or by using graph query expressions. Similarly, control relations (edges) between data elements in ReGraDa are dynamic. They are defined between sets of data

elements given by graph query expressions and guarded by boolean expressions over the graph state.

Overview. We proceed as follows. We present ReGraDa informally by example in Sec. 2; then provide formal syntax and semantics in Sec. 3 and 4. In Sec. 5 we present the translation of ReGraDa to cypher/*neo4j*. In Sec. 6 we report on exploratory empirical studies of the performance characteristics of the translation. Finally, we briefly conclude and provide directions for future work in Sec. 7.

2 ReGraDa: Programming with Reactive Graph Data

In this section, we informally illustrate the syntax and semantics of ReGraDa by giving an example of managing authors, books and book loans in a library.

A ReGraDa process defines simultaneously the process, the data and the control flow of a software system. Consider the process below comprising three main sections separated by semicolon symbols. The first section (line 1) declares all instances of data elements in the process, i.e. the nodes of the graph. The second section (line 3-15) defines control relations. The third section after the semicolon in line 16 declares data relations, i.e. the edges in the graph data. This section is initially empty in this example.

```
1  (createAuthor:Input) [?: { authorName:String }]
2  ;
3  createAuthor -->> {
4    (author:Author) [{ name:@trigger.value.authorName }],
5    (createBook:Input) [?:{ bookTitle:String, isbn:String }]
6    ;
7    createBook -->> {
8      (book:Book) [{bookTitle:@trigger.value.bookTitle,
9                    isbn:@trigger.value.isbn,
10                   author:author.value.name }]
11     ; ;
12     author -[:WROTE]-> book
13   }
14   ;
15 }
16 ;
```

Line 1 declares an input data element that accepts values as input of type record `{authorName:String}`. Line 3 defines a spawn relation, which is a control relation that, whenever the input data element in line 1 is executed, triggers the creation of the elements contained in the sub-process defined in line 4-15. The sub-process creates two new data elements (line 4-5): `author` and `createBook`. The two data elements are implicitly associated with each other due to a syntactic dependency through a nested spawn relation (lines 7-13). The expression enclosed in declaration of data element `author`, of type record, denotes the new value given to the new data element, where `@trigger` is evaluated once in a call-by-value strategy (copying the value of the `createAuthor` data-element

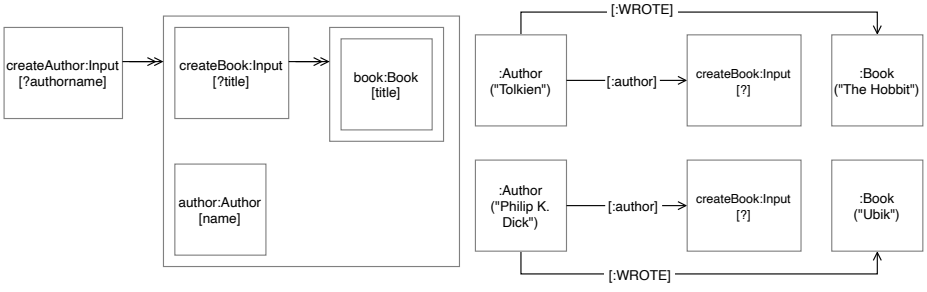

Fig. 1. Process state after the creation of two authors and two books.

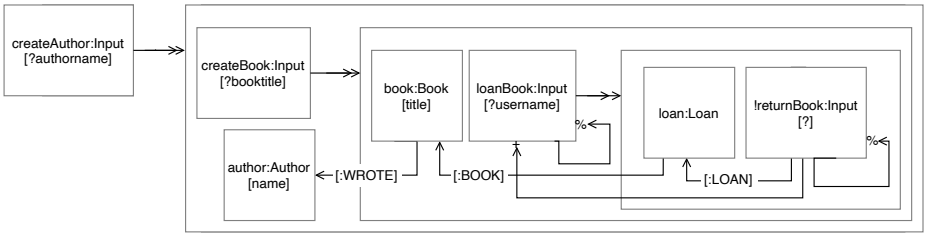

Fig. 2. Complete representation of the example.

triggering the spawn reaction). Notice that each data element `author` will be explicitly associated to all `book` data elements created by its, implicitly associated, `createBook` input data element. From this, we can see that each author will have a distinct and direct entry point in the system to create their own books in the database. In our pragmatic approach, the entry points for input data elements are implemented in a companion system following REST conventions to identify the target element. Also, notice that the value of name `author` in the expression of data element `book` is associated to the data element statically associated with `createBook`. The sub-process spawned by the rule in line 7, when the `createBook` input data element is executed, it introduces a data element `book` and a new data relation, with label `WROTE`, between the data element `author` defined in the outer scope and the new data element `book`. The resulting graph is a flat structure of data elements, all created at the process's top level. So, after executing input data element `createAuthor` twice – with values `"Tolkien"` and `"Philip K. Dick"` – and "adding" one book for each one of the authors, we can observe that the process now includes the elements visually depicted in Figure 1. Notice that the data relation above is introduced between the two newly created instances of the data elements. ReGraDa also allows for the definitions of more general control and data relations between sets of data elements, which are denoted by graph queries.

To illustrate other control flow constraints we define the input data element `loanBook` in the sub-process of each book (line 8), and the sub-process that it triggers (lines 12-17).

```
1  (createAuthor:Input) [?: {authorName:String}] ;
2  createAuthor -->> {
3    (author:Author) [{name:@trigger.value.authorName}],
4    (createBook:Input) [?:{bookTitle:String, isbn:String}] ;
5    createBook -->>{
6      (book:Book) [{bookTitle:@trigger.value.bookTitle,
7                    isbn:@trigger.value.isbn, author:author.value.name}],
8      (loanBook:Input) [?:{username:String}] ;
9      loanBook -->% loanBook,
10     author -[:WROTE]-> book
11     loanBook -->>{
12       (loan:Loan) [{user:@trigger.value.username}],
13       !(returnBook:Input) [?:()] ;
14       returnBook -->% returnBook,
15       returnBook -->+ loanBook ;
16       loan -[:BOOK]-> book,
17       returnBook -[:LOAN]-> loan
18     } ;
19   } ;
20 } ;
```

This inner process makes more use of DCR constructs, in particular the notions of a data element being *included*, *excluded*, and *pending*.

Intuitively, an *excluded* data element is considered temporarily excluded from the process: We pretend that it, its contents, and the relations to or from it are "not there". The notion of exclusion is dynamic: data elements can switch states from included to excluded and back again. In the example, the relation `-->%` indicates such a dynamic exclusion: Whenever the data element on the left is executed, the one on the right is so excluded.

Similarly, a data element may become *pending* as consequence of another executing via the "response" relation `*-->`. Intuitively, a pending element must at some later point execute (or be excluded) in order for the program to terminate.

In this inner process, `loanBook` is executed given a user's name, and it excludes itself from the process (`-->%`, line 9), thus executing only once until it is explicitly included again; it also spawns the elements in a sub-process (line 11): a new data element `loan` that stores the user's name, and an input data element `returnBook` that is used when the user wants to terminate a loan in the library. Notice that data element `returnBook` is declared using an exclamantion mark as annotation. This means that this input data element must be executed, or excluded, in order to complete the process. The example also declares data relations to link data element `loan` to its corresponding `book` element. The sub-process introduces new control relations to self-exclude `returnBook`, i.e. it cannot happen twice, and to include `loanBook` back into the process, allowing for new loans to happen. Figure 2 depicts the final state in our example. For the

$$
\begin{array}{lll}
P & ::= \overline{D}\ ;\ \overline{R}\ ;\ \overline{Y} & \text{Processes} \\
D & ::= (n_\rho : \ell)[?{:}T]{:}(h,i,r,v) \mid (n_\rho : \ell)[E]{:}(h,i,r,v) & \text{Data Elements} \\
R & ::= \phi \text{–}[E]\!\rightarrow\!\bullet\ \phi \mid \phi \text{–}[E]\!\rightarrow\!\diamond\ \phi \mid \phi\bullet\text{–}[E]\!\rightarrow\!\phi & \\
 & \quad \mid \phi \text{–}[E]\!\rightarrow\!+\ \phi \mid \phi \text{–}[E]\!\rightarrow\!\%\ \phi \mid \phi \text{–}[E]\!\twoheadrightarrow P & \text{Control Relations} \\
Y & ::= \phi \text{–}[n : \ell]\!\rightarrow \phi & \text{Data Relations} \\
T & ::= \mathsf{Unit} \mid \mathsf{String} \mid \mathsf{Number} \mid \mathsf{Boolean} \mid \mathsf{List}\ T \mid \{\ \overline{x : T}\ \} & \text{Data Types} \\
\phi & ::= \psi\ \mathsf{RETURN}\ E \mid n & \text{Node Queries} \\
\psi & ::= \ \mathsf{MATCH}\ Q\ \mathsf{WHERE}\ E & \\
 & \quad \mid \ \mathsf{MATCH}\ Q\ \mathsf{WHERE}\ E\ \mathsf{WITH}\ \mathit{pipe}\ \mathsf{WHERE}\ E & \text{Match Expressions} \\
E & ::= c \mid n \mid Q \mid \phi \mid \psi \mid E{:}\mathit{attr} \mid f(E_1, ..., E_n) & \\
 & \quad \mid \{\ \overline{x = E}\ \} \mid E.\ell \mid [\overline{E}] \mid \mathsf{hd}(E) \mid \mathsf{tl}(E) & \text{Expressions} \\
c & ::= \mathit{numbers} \mid \mathit{strings} \mid \mathsf{true} \mid \mathsf{false} \mid \mathbf{1} \mid \bot & \text{Literals} \\
\mathit{pipe} & ::= \mathit{agg}\ \mathsf{AS}\ n \mid n\ \mathsf{AS}\ n & \text{Pipeline Term} \\
\mathit{agg} & ::= n \mid \mathsf{COUNT}(E) \mid \mathsf{MAX}(E) \mid \mathsf{MIN}(E) \mid \mathsf{SUM}(E) & \text{Aggregating Functions} \\
\mathit{attr} & ::= \mathit{value} \mid \mathit{executed} \mid \mathit{included} \mid \mathit{pending} & \text{Attributes}
\end{array}
$$

Fig. 3. Syntax of ReGraDa

sake of space, we omit from this diagram the implicit data dependencies between data elements that are necessary to implement the reactive computation.

This section illustrates, by example, the dynamic spawning of data and control elements, and also the dynamic inclusion and exclusion of data elements in the process. We have omitted the details of how the run-time state of each data element is represented, i.e. the value assigned to an input data element, whether an event is included or excluded and whether it is pending or not. This is explained and made formal in the next two sections.

3 Formal Syntax of ReGraDa

The abstract syntax of ReGraDa is given by the grammar in Figure 3. In this grammar, we assume given enumerable sets of names (ranged over by n, a, b, c and the reserved keyword *trigger*), of unique data element identifiers (ranged over by ρ), and labels for data elements and relations (ℓ). We use the notation $\overline{x}$ to denote a sequence $x_1, ..., x_n$ and elide concrete syntax separators and the number of elements in the sequence. Distinctively, names abstract data elements in the scope of a process, very much like variables in a program. Identifiers are global and uniquely denote data elements in a process, like a key of a record in a database or the memory heap allocated regions. Labels work like classes for data elements, playing an important role in queries on the graph of data elements.

Analysing the syntax topdown, we have that a ReGraDa process P defines data, control relations and runtime state simultaneously as seen in the example using the concrete syntax given in the previous section. It comprises a sequence $\overline{D}$ of data element definitions (D), followed by a sequence $\overline{R}$ of control relation definitions (R) and a sequence $\overline{Y}$ of data relation definitions (Y). Both control and data relation definitions use query expressions over graphs of data elements

denoting sets of data elements (ϕ) and expressions (E) that manipulate the usual set of datatypes, comprising types for unit, strings, numbers, boolean values, lists, and records.

As exemplified in the previous section, ReGraDa processes define a nested structure of process definitions (P), via the spawn control relation (ϕ–[E]↠P). Data elements D declare a local identifier (n) in the scope of the current process definition and its corresponding sub-processes; a label (ℓ), visible globally; and a unique runtime identifier (ρ), the latter is a runtime constant that is added to the data elements when they are created.

Data elements can be of one of two kinds: an input data element or a computation data element. Input data elements, such as the `createAuthor:Book` element in the previous section, has the form $(n_\rho{:}\ell)[?{:}T]{:}(h,i,r,v)$ and define system entry points of type (T), that can be linked, to web services, web forms, or any other form of input to the process.

Computation data elements, of the form $(n_\rho{:}\ell)[E]{:}(h,i,r,v)$, define nodes in the graph that compute and store values denoted by their expressions E. Such expression may refer to values stored in other data elements by using their identifiers or the reserved name `@trigger` that refers to the data element triggering a spawn control relation as used in the `author` and `book` data elements in the previous section. Expressions may also explicitly query the graph. For instance, the expression

```
(authorCount:Stats)[MATCH (l:Author) COUNT(l)]
```

declares a computation data element that if executed computes the number of author elements in the graph.

Following the typical structure of Dynamic Condition Response (DCR) graphs [2, 8, 9, 13], the run-time state of a data element, referred to as the marking, is defined using the four properties (h,i,r,v) associated to each data element (and so far ignored in our examples):

1. a boolean value h (`executed` in the concrete syntax) that signals that the event was previously executed (happened). A data element can be executed multiple times, this property only registers the first execution.
2. a boolean i (`included`) indicating whether the data element is currently included. A data element not included, also referred to as excluded, is considered irrelevant: it cannot execute and cannot prevent execution of others;
3. a boolean r (`pending`) indicating whether the data element is currently pending. A pending element is required to be subsequently updated (or become excluded) at some point in the future to let a process reach a final state; and finally,
4. a value v of any type admissible in the language (`value`) indicating the current value of the data element, or the undefined value ($\perp$) if the data element has not yet been executed/received a value and has no initial value.

For instance, the `createAuthor:Input` data element in the previous section will initially have the state

```
(createAuthor:Input)[?:{authorName:String}](false,true,false,⊥)
```

representing that the element has not been executed, is included, is not pending, and has the value $\perp$ because no value has been given yet. After executing it inputting the value "Tolkien", the data element will have the state

```
(createAuthor:Input)[?:{authorName:String}](true,true,false,"Tolkien")
```

Control relations (R) range over a set of six different kinds of relations stemming from the declarative process model of DCR graphs. All relations are guarded by a boolean expression E, that may refer all identifiers in scope or use queries to refer to the global state of the graph.

1. **condition** (ϕ–[E]→• ϕ) (`-->*` in the concrete syntax), defining that if the expression E evaluates to true, then the data-elements denoted by the query on the right-hand side cannot execute unless all data elements denoted by the query on the left-hand side are either marked not included or executed;
2. **milestone** (ϕ–[E]→◇ ϕ) (`--><>`), defining that if the expression E evaluates to true, then the data elements denoted by the query on the right cannot execute unless all data elements denoted by the query on the left-hand side are marked not included or not pending;
3. **response** (ϕ•–[E]→ϕ) (`*-->`), defining that if the expression E evaluates to true, then whenever some data element denoted by the query on the left-hand side executes, all data elements denoted by the query on the right-hand side become marked pending;
4. **inclusion** (ϕ–[E]→+ ϕ) (`-->+`), defining that if the expression E evaluates to true, then whenever some data element denoted by the query on the left-hand side executes, all data elements denoted by the query on the right-hand side become marked included;
5. **exclusion** (ϕ–[E]→% ϕ) (`-->%`), defining that if the expression E evaluates to true, then whenever some data element denoted by the query on the left-hand side executes, all data elements denoted by the query on the right-hand side become excluded; and finally,
6. **spawn** relation (ϕ–[E]↠ P) (`-->>`), defining that if the expression E evaluates to true, then whenever some data element denoted by the query on the left hand side executes, the data elements and rules in the sub-process P on the right-hand side are instantiated in the current process. The reserved name *trigger* denotes the left-hand side element that caused the spawing of the sub-process, in all expressions in the new elements.

Just like in [8], one important semantic property of data elements is whether they are enabled or not. We say that a data-element is *enabled* iff (i) it is included, (ii) every other element preceding it by a condition is either executed or not included, (iii) every element preceding it by a milestone is either not included or not pending. An enabled element may *execute*, modifying its marking by making it executed and not pending, and possibly assigning it a new value. This execution causes toggling the included state of every element succeeding the executed one by an inclusion or exclusion, and by making pending all elements succeeding the executed one by a response relation. In all cases, both for enabledness and execution, a relation is considered only if it either has no guard, or that guard evaluates to true.

We already saw examples of the spawn, include and exclude relations constraining the `loanBook` and `returnBook` input actions in the previous section. As an example of the guard expression E, we can use a guarded inclusion relation

```
returnBook -[ MATCH (l:Loan)-[r:BOOK]->(book)
              WITH COUNT(l) as n WHERE n<50  ]->+ loanBook
```

to only include it if a book has not reached its lifespan, e.g. 50 loans. If the query returns an empty set of nodes that satisfy the condition, the relation is disabled.

As examples of the condition, milestone and response, consider the relations

```
createAuthor *--> authorCount
createAuthor -->* authorCount
authorCount --<> createAuthor
```

The response control relation means that the `authorCount` computation event becomes pending when a new author is created, and the reactive semantics of ReGraDa will then ensure that the author count is updated whenever an author is added. The condition control relation ensures that the authorCount can not be executed before any author has been added. Finally, the milestone relation ensures that a new author cannot be created if `authorCount` is pending, i.e. it has not yet been updated after the last creation of an author.

The data relations (Y) of the form $\phi \dashv[n : \ell]\rightarrow \phi$ create a native data link between all combinations of nodes resulting from the queries on the left and right-hand side. Data relations (Y) and, in particular, match expressions (ϕ) closely resemble the notation of *cypher* [6] for node relationships and graph queries respectively, identifying nodes and relations via patterns (Q), filtering and aggregating results (ψ). This approach contrasts with, but can encode, the semi-structured data style of ReSeDa, and the relational schema of approaches like [17]. For instance, the example given in the previous section introduced simple data relations like

```
tolkien -[:WROTE]-> hobbit
```

Where `tolkien` and `hobbit` are identifiers denoting data elements. Finally, the expression language in ReGraDa includes the usual set of constructor and destructor expressions for all the base datatypes considered. It also includes the use of match expressions to enable the runtime manipulation of data elements and their attributes as already illustrated above.

4 Semantics

Following the tradition of DCR graphs [2, 8, 10, 13], we define the semantics of ReGraDa by means of a transition system, where states are processes comprising data elements, the corresponding relations, and including their *marking* information. Each transition, written $P \xrightarrow{\rho} P'$, corresponds to the execution (i.e. value update) of a data element ρ. The execution of an input data element $P \xrightarrow{\rho(v)} P'$ requires a value v to be provided in the transition. To formally define

the transition system, we must first define two auxiliary functions. Section 4.1 defines function $\mathsf{enabled}_P(\rho)$, that determines, for a given ReGraDa program state, if a given data element ρ is currently executable (enabled). Section 4.3 defines function $\mathsf{effects}_P(\overline{R})$ that determines, for given a ReGraDa program state and a set of control rules $\overline{R}$, what are the effects of those rules that allow us to compute the next state of the process. For the sake of space, we omit the semantics of expressions, $(\!|\, E \,|\!)_P$, and query expressions, $[\![\phi]\!]_P$, which are straightforwardly defined on the structure of the expression or graph, respectively. We finally define the two allowed transitions in our semantics.

4.1 Enabledness

For a data element ρ in process P to be enabled it must be the case that (i) the data element ρ must itself be included; (ii) every data element ρ' that is a condition (where E evaluates to true) for ρ must be either excluded or previously executed; (iii) every data element ρ' that is a milestone (where E evaluates to true) for ρ must be either excluded or not pending. We define function $\mathsf{enabled}_P(\rho)$, that checks if the data element ρ is enabled in process $P = (\overline{D}; \overline{R}; \overline{Y})$.

$$
\begin{aligned}
\mathsf{enabled}_P(\rho) \triangleq\ & \rho[\mathsf{included}] \\
& \wedge\ \forall \phi \text{–}[E]\!\!\rightarrow\!\!\bullet\ \phi' \in \overline{R}.\ \rho \in [\![\phi']\!]_P \wedge (\!|\, E \,|\!)_P = \mathsf{true} \implies \\
& \qquad ([\![\phi[\mathsf{included}]]\!]_P = [\![\phi[\mathsf{included} \wedge \mathsf{executed}]]\!]_P) \\
& \wedge\ \forall \phi \text{–}[E]\!\!\rightarrow\!\!\diamond\ \phi' \in \overline{R}.\ \rho \in [\![\phi']\!]_P \wedge (\!|\, E \,|\!)_P = \mathsf{true} \implies \\
& \qquad ([\![\phi[\mathsf{included}]]\!]_P = [\![\phi[\mathsf{included} \wedge \neg\mathsf{pending}]]\!]_P)
\end{aligned}
$$

4.2 Effects

The effect of executing a data element ρ in the context of a global process P can be computed from the current state and given a value in the case of input data elements. Effects are gathered by iterating the set of control rules (excluding conditions and milestones) in the process as defined below

$$
\mathsf{effects}_P(\rho) \triangleq \cup_{R \in \overline{R}}\, \mathsf{effects}'_P(R, \rho) \text{ with } P = \overline{D}; \overline{R}; \overline{Y}
$$

$$
\begin{aligned}
\mathsf{effects}'_P(\phi\bullet\text{–}[E]\!\!\rightarrow\!\!\phi', \rho) &\triangleq \{(\mathsf{pend}, \rho') \mid (\rho, \rho') \in [\![\phi]\!]_P \times [\![\phi']\!]_P \wedge (\!|\, E \,|\!)_P = \mathsf{true}\} \\
\mathsf{effects}'_P(\phi\text{–}[E]\!\!\rightarrow\!\!\%\ \phi', \rho) &\triangleq \{(\mathsf{excl}, \rho') \mid (\rho, \rho') \in [\![\phi]\!]_P \times [\![\phi']\!]_P \wedge (\!|\, E \,|\!)_P = \mathsf{true}\} \\
\mathsf{effects}'_P(\phi\text{–}[E]\!\!\rightarrow\!\!+\ \phi', \rho) &\triangleq \{(\mathsf{incl}, \rho') \mid (\rho, \rho') \in [\![\phi]\!]_P \times [\![\phi']\!]_P \wedge (\!|\, E \,|\!)_P = \mathsf{true}\} \\
\mathsf{effects}'_P(\phi\text{–}[E]\!\!\twoheadrightarrow P', \rho) &\triangleq \{(\mathsf{spawn}(P'\{^{\rho}/_{\mathsf{trigger}}\})) \mid \rho \in [\![\phi]\!]_P \wedge (\!|\, E \,|\!)_P = \mathsf{true}\}
\end{aligned}
$$

4.3 Transitions

Finally, we present below the transition rules that define the complete labelled transition system. The execution of a computation data element ($P \xrightarrow{\rho} P'$) is possible from a state where the given element is enabled, and to a state where the value of the element is refreshed based on the values of other references, given

by function $\mathsf{compute}_P(\rho)$, and where the effects of the execution are applied by operation $(P' \triangleleft \delta)$.

$$\frac{\mathsf{enabled}_P(\rho) \quad P' = \mathsf{compute}_P(\rho) \quad \delta = \mathsf{effects}_{P'}(\rho)}{P \xrightarrow{\rho} P' \triangleleft \delta}$$

In the case of an input data element, a transition may occur if the element is enabled, and to a state where the element's value is updated ($\mathsf{update}_P(\rho, v)$) to the value v given in the transition label as input, and after applying the corresponding effects.

$$\frac{\mathsf{enabled}_P(\rho) \quad P' = \mathsf{update}_P(\rho, v) \quad \delta = \mathsf{effects}_{P'}(\rho)}{P \xrightarrow{\rho(v)} P' \triangleleft \delta}$$

We omit the definition of function $\mathsf{compute}_P(\rho)$ which is the straightforward application of the expression semantics to the enclosed expression in computation data elements, with relation to other data elements, and the definition of function $\mathsf{update}_P(\rho, v)$ which simply updates the value of the data element with the incomming value. The application of the effects consists in changing the state of the given data element and the creation of new elements in the case of spawn relations, thus providing fresh identifiers and binding the triggering data element to the new data elements and relations.

$$\begin{gathered}
(\overline{D}, (n_\rho : \ell)[?{:}T]{:}(h, i, p, v)); \overline{R}; \overline{Y} \ \triangleleft \ (\mathsf{pend}, \rho) \triangleq (\overline{D}, (n_\rho : \ell)[?{:}T]{:}(h, i, \mathsf{t}, v)); \overline{R}; \overline{Y} \\
(\overline{D}, (n_\rho : \ell)[E]{:}(h, i, p, v)); \overline{R}; \overline{Y} \ \triangleleft \ (\mathsf{excl}, \rho) \triangleq (\overline{D}, (n_\rho : \ell)[E]{:}(h, \mathsf{f}, p, v)); \overline{R}; \overline{Y} \\
(\overline{D}, (n_\rho : \ell)[E]{:}(h, i, p, v)); \overline{R}; \overline{Y} \ \triangleleft \ (\mathsf{incl}, \rho) \triangleq (\overline{D}, (n_\rho : \ell)[E]{:}(h, \mathsf{t}, p, v)); \overline{R}; \overline{Y} \\
\overline{D}; \overline{R}; \overline{Y} \ \triangleleft \ (\mathsf{spawn}(\overline{D'}; \overline{R'}; \overline{Y'}), \rho) \triangleq \overline{D}, \overline{D'\sigma}; \overline{R}, \overline{R'\sigma}; \overline{Y}, \overline{Y'\sigma}
\end{gathered}$$

Substitution σ, used above, replaces the free names of $\overline{D'}$ with fresh event identifiers, assigns new node identifiers and replaces identifier *trigger* by ρ. The set of free names of $\overline{R'}$ and $\overline{Y'}$, except *trigger*, is a subset of the free names of $\overline{D'}$.

5 From ReGraDa to Cypher

We now define the compilation procedure of ReGraDa to *cypher*, the query language of the *neo4j* database. For want of space, we omit an introduction *cypher* here, and refer the reader to the introductory work [6] and the *neo4j* official documentation[6].

Our encoding uses the native capabilities of the database system as much as possible to allow for an almost standalone execution of the process and embeds the reactive behaviour of the process to independent data modifications. Our approach is to translate a ReGraDa process to a set of update queries and

[6] `https://neo4j.com/`

```
(a:A)[?:Number], (b:B)[a.value+1] ;
b -->% a,
a -->> {
  (c:C)[ {x:a.value+b.value, y:@trigger.value} ] ;
  c -->> { (d:D)[c.value.x+c.value.y+a.value] ; ; } ;
} ;
```

Fig. 4. Toy example illustrating compilation of ReGraDa to *cypher*

triggers in the database and let data elements, their properties, and data relationships act as the data model of an external application to be freely queried and modified. We next present the compilation procedure that systematically transforms a simple ReGraDa process into *cypher*.

The structure of our target code is a list of trigger definitions and a *cypher* script, which is a flat list of node and relationship declarations, graph queries and update commands. To transform the nested structure of ReGraDa processes into the flat structure of *cypher* code we resolve all names with a standard static resolution of names and representing syntactic dependencies with data relationships between nodes in the database graph.

So, in the general case, a ReGraDa process is translated into a four-part *cypher* script containing: (1) a list of queries that is used to fetch and bind all related nodes from other contexts to be used in local (or inner) definitions, this list is empty at the top-level; (2) a set of node definitions that map the definitions in the current process of input and computation data elements; (3) a set of node relationship definitions that map data dependencies, control relations, and data relations defined in the current process; and finally, (4) (in the case of triggers associated to a computation data element), one update command that (re)evaluates the node's expression with relation to the nodes it depends on.

We use a simplified example, in Figure 4, to illustrate the compilation procedure. It contains all main cases of a ReGraDa process (data dependencies, nested processes, and sub-processes). The *cypher* code emitted for this example is presented step by step in this section.

First, the top-level data elements are translated to node definitions that include the defined name (alpha renamed), their labels, and default values for the markings. We add an extra field that identifies the source element in the code (`reda_id`). Notice that node names are only visible in the current script.

```
CREATE (a_0:A{reda_id="a_0", executed:0, included:true, pending:false})
CREATE (b_1:B{reda_id="b_0", executed:0, included:true, pending:false})
```

Both nodes are initialized without an attribute for its value. This attribute is left uninitialized since the data elements were not executed and therefore cannot be referred or evaluated at this stage. Also, we alpha-renamed names and identifiers (`a_0`, `b_1`) to avoid name clashing between different declaration contexts.

Consider the syntactic dependencies between data elements `a` and `b` in Figure 4, created by the expression of data element `b`, and also because `a` spawns a subprocess using `b`. The static resolution of names in ReGraDa is mapped onto

explicit node relations that define a name substitution inside the sub-process in the spawn relation of line 3 in Figure 4. Three relationships are needed in *neo4j*

```
3 CREATE (a_0)-[:a]->(a_0)
4 CREATE (a_0)-[:a]->(b_1)
5 CREATE (b_1)-[:b]->(a_0)
```

Lines 3 and 4 mean that node `a_0` in this context is the correct substitution for the free name `a` in all sub-processes (and expression) of node `a_0` and `b_1`.

The data dependency of `b` on `a` is reified into control relations (condition and response) as follows. Clearly we cannot execute `b` without first having executed—and thus gotten a value for—`a`; and equally clearly, whenever the value of `a` changes, we must re-compute `b` to reflect that change in the value of `b`. That is, we add the following condition and response relations:

```
6 CREATE (a_0)-[:condition]->(b_1)
7 CREATE (a_0)-[:response]->(b_1)
```

These relations establish the essence of the reactive behaviour of ReGraDa, similarly to the semantics of ReSeDa. (The mechanics here is akin to spreadsheet semantics: updating "cell" `a` forces a recompute of the value of `b`.)

We next translate the control relation (excludes) on line 3 almost verbatim.

```
8 CREATE (b_1)-[:excludes]->(a_0)
```

Other control relations are translated directly to relations between node instances. This concludes the translation of node and relation top-level declarations. Next, we present the main trigger that checks for the enabledness of data elements prior to execution, and applies the effects of control relations (inclusion, exclusion, responses) after a successful execution.

```
9  CALL apoc.trigger.add('EVERYWERE',
10 'UNWIND apoc.trigger.propertiesByKey($assignedNodeProperties,"executed")
11   as prop WITH prop.node as n WHERE n.executed>0
12
13   CALL apoc.util.validate(
14     n.included=false, "EVENT IS NOT INCLUDED", [])
15   CALL apoc.util.validate(
16     EXISTS((n)<-[:condition]-({included:true, executed:0})),
17     "EVENT HAS A CONDITION UNSATISFIED", [])
18   CALL apoc.util.validate(
19     EXISTS((n)<-[:milestone]-({included:true, pending:true})),
20     "EVENT HAS A MILESTONE UNSATISFIED", [])
21
22   SET n.pending=false WITH n
23   OPTIONAL MATCH (n)-[:response]->(t) SET t.pending = true WITH n
24   OPTIONAL MATCH (n)-[:excludes]->(t) SET t.included = false WITH n
25   OPTIONAL MATCH (n)-[:includes]->(t) SET t.included = true
26   RETURN 1 ', {phase:'before'});
```

The *enabledness* validation is translated into explicit validations (lines 13-20) that check if any preceding element (using a relation with tag `:condition` or

`:milestone`) is included, and not executed in the case of condition relations or pending in the case of milestone relations. The effects of *execution* via the response, includes and excludes relations are translated to *cypher* update queries that search for this kind of relations between node instances and modifies the marking of the target node accordingly (lines 23-25). The enabledness check and the subsequent execution of effects is performed here in a way that aborts any transaction in case of error.

The remaining behaviour, including the spawning of sub-processes is represented in triggers generated for each one of the data elements statically declared in the program. Such triggers are fired whenever the associated `executed` property is changed. These triggers contain the compiled code for all the actions that need to be executed when related nodes are (re)evaluated. Consider the example of node `a_0`, compiled from the computation node `a` in the example. The trigger is the following

```
27 CALL apoc.trigger.add('When a_0 happens',
28 'UNWIND apoc.trigger.propertiesByKey($assignedNodeProperties,"executed")
29   as prop WITH prop.node as n WHERE n.reda_id="a_0" AND n.executed>0
30
31   MATCH (a_0)-[:a]->(n)
32   MATCH (b_1)-[:b]->(n)
33   CREATE (c_2:C{reda_id="c_2", executed:0, included:true,
34                                pending:false, value_y:n.value})
35   CREATE (a_0)-[:a]->(c_2)
36   CREATE (b_1)-[:b]->(c_2)
37   CREATE (c_2)-[:c]->(c_2)
38   CREATE (b_1)-[:condition]->(c_2)
39   CREATE (b_1)-[:response]->(c_2)
40   CREATE (a_0)-[:condition]->(c_2)
41   CREATE (a_0)-[:response]->(c_2)
42   RETURN 1 ', {phase:'before'});
```

This trigger starts by instantiating the free names of the sub-process of `a_0`, reifying the nested structure of the process. It queries the nodes that represent identifiers `a` and `b` in this context (lines 31-32). These relations match the relations created at the top-level (lines 3-5). Line 33 includes the local definition for data element `c`, here alpha-renamed to `c_2`, and includes the partial evaluation of expression `{x:a.value+b.value, y:@trigger.value}`, in this case for field `y` which depends on the triggering data element. Lines 35-37 repeat the static resolution of names for the inner scope of sub-processes and expressions as described at the top level. Notice that identifiers `a` and `b` cross more than one syntactic context level and direct links are created at all stages. Lines 38-41 create the control relations created by the sub-process.

The trigger that handles node `b` is quite simpler since it is not used to spawn any sub-process. The value of node `b` depends on the value of node `a`, so its trigger basically (re)computes the value of `b` whenever the node is executed.

```
43 CALL apoc.trigger.add('When b_1 happens',
44  'UNWIND apoc.trigger.propertiesByKey($assignedNodeProperties,"executed")
```

```
45   as prop WITH prop.node as n WHERE n.reda_id="b_1" AND n.executed>0
46
47   MATCH (a_0)-[:a]->(n)
48   SET n.value=a_0.value+1
49   RETURN 1 ', {phase:'before'});
```

Line 47 retrieves the substitution for identifier `a_0` in this context and updates the value attribute of node `b_1` (line 45). Notice that each instance of `b` may be associated with a different instance of `a`. We omit the handlers for nodes `c_2` and `d_3`, that are similar to the handlers of nodes `a_0` and `b_1` depicted above (lines 27 and 43). Notice also that field `x` of the record in data element `c_2` should be computed in the corresponding trigger since it depends on other elements.

The resulting target code comprises a set of top-level definitions, and a set of triggers: a main trigger, and a trigger associated to each definition of data elements that contains computations or spawns sub-processes.

6 Empirical experiments

To study the performance and scalability of the resulting code, and thus the translation strategy presented in this paper, we tested the prototype using thousands of data elements and relationships between them. Note that the number of triggers is statically determined by the process definition and remains constant throughout all executions. We designed two case studies for this purpose: an "expected" program and a worst-case scenario. These programs were developed to grow linearly in an experiment with twenty-five executions, each one executed ten times. They were executed using *neo4j* 4.1.1 with APOC 4.1.0.2, Windows 8.1, and an Intel Core i7-4510U CPU @ 2.00GHz 2.60GHz with 8Gb RAM.

The "expected" case scenario consists of a program where nodes and relationships are split amongst different clusters. This case study starts with only two data-elements belonging to one cluster, with each further execution adding an arbitrary number (eighty-seven) of new data-elements that are then split uniformly into three clusters. Each data-element will either have either no relationships, acting as spawner input data-elements, or will have a number of relationships between one and the third of the number of existing data-elements. As depicted in Figure 5, the time difference between the request and the response tends to grow in a somewhat linear to sub-linear fashion with each execution. However, in the worst-case scenario, where we have one giant cluster where data-elements are highly dependent on each other, the time difference between the requests and responses tends to grow in a linear to super-linear fashion, as depicted in Figure 6. Pre-determined sequences of requests were used in the tests.

To conclude, triggers are many times the source of performance issues in database systems. With this in mind, we made every effort to encode all the reactivity and behavior of ReGraDa into static triggers, remaining constant in number throughout all execution. There are at most two triggers activated each time a specific data-element is executed: `(i)` the trigger regarding that specific data-element, containing the specific behavior defined in the process; and

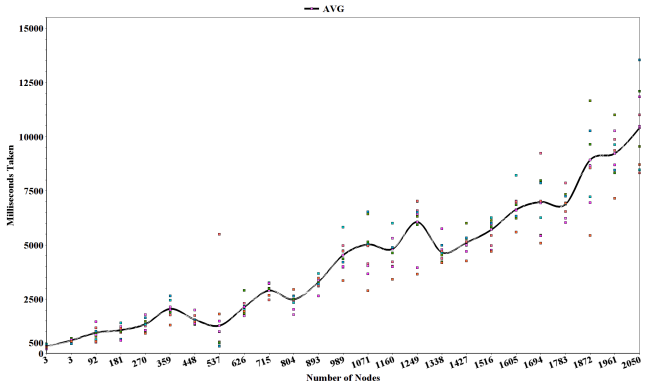

Fig. 5. Expected case scenario.

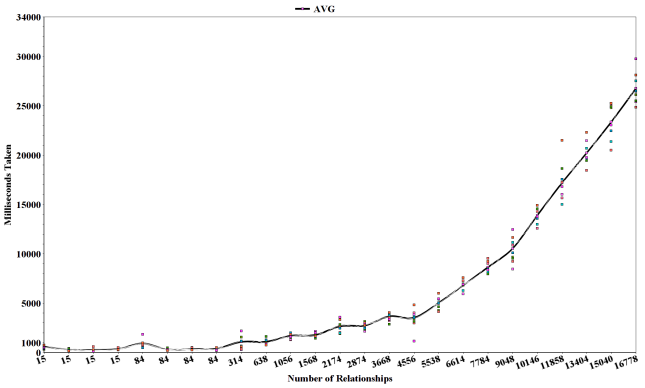

Fig. 6. Worst case scenario.

(ii) the `main_trigger` that is always activated despite the data-element being executed, guaranteeing that the language rules like the `enabledness` verification or the application of DCR effects are being followed. With this information and the results of our case study, we can infer that the main cause for performance deterioration is the number of control relationships on each node being executed. This comes from the need to process each of the relations in the `main_trigger` to reach the next ReGraDa process state.

7 Conclusions and Future Work

We introduced ReGraDa, a language for REactive GRAph DAta based on the declarative DCR graph language, and evaluated the language in a prototype implementation using the graph-database *neo4j*, with promising performance characteristics of the first early tests. As future work we also plan to research the transfer of DCR results to ReGraDa, e.g. refinement [2] and choreographies [11], providing guarantees for deadlock freedom by design. Also, we plan to investigate the use of ReGraDa as target language for multi-instance process mining of complex ERP and EIM systems and the relation between ReGraDa models and the recent work in [5].

References

1. 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.
2. Søren Debois, Thomas T. Hildebrandt, and Tijs Slaats. Replication, refinement & reachability: Complexity in dynamic condition-response graphs. *Acta Informatica*, 55(6):489–520, September 2018.
3. Miguel Domingues and João Costa Seco. Type safe evolution of live systems. In *Workshop on Reactive and Event-based Languages & Systems (REBLS'15)*, 2015.
4. Marlon Dumas, Wil M Van der Aalst, and Arthur H Ter Hofstede. *Process-aware information systems: bridging people and software through process technology*. John Wiley & Sons, 2005.
5. Stefan Esser and Dirk Fahland. Multi-dimensional event data in graph databases. *Journal of Data Semantics*, 2021.
6. Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 1433–1445, New York, NY, USA, 2018. Association for Computing Machinery.
7. Object Management Group. Case Management Model and Notation V 1.1. 2016.
8. Thomas Hildebrandt and Raghava Rao Mukkamala. Declarative Event-Based Workflow as Distributed Dynamic Condition Response Graphs. In *Post-Proceedings of PLACES 2010*, volume 69 of *EPTCS*, pages 59–73, 2010.
9. Thomas Hildebrandt, Raghava Rao Mukkamala, Tijs Slaats, and Francesco Zanitti. Contracts for cross-organizational workflows as timed Dynamic Condition Response Graphs. *JLAP*, 82(5):164–185, 2013.
10. Thomas T Hildebrandt and Raghava Rao Mukkamala. Declarative event-based workflow as distributed dynamic condition response graphs. *arXiv:1110.4161*, 2011.
11. Thomas T. Hildebrandt, Tijs Slaats, Hugo A. López, Søren Debois, and Marco Carbone. Declarative choreographies and liveness. In Jorge A. Pérez and Nobuko Yoshida, editors, *Formal Techniques for Distributed Objects, Components, and Systems*, pages 129–147, Cham, 2019. Springer International Publishing.
12. Artem Polyvyanyy, Jan Martijn E. M. van der Werf, Sietse Overbeek, and Rick Brouwers. Information systems modeling: Language, verification, and tool support. In Paolo Giorgini and Barbara Weber, editors, *Advanced Information Systems Engineering - 31st International Conference, CAiSE 2019, Rome, Italy, June 3-7, 2019, Proceedings*, volume 11483 of *Lecture Notes in Computer Science*, pages 194–212. Springer, 2019.
13. João Costa Seco, Søren Debois, Thomas T. Hildebrandt, and Tijs Slaats. RESEDA: Declaring Live Event-Driven Computations as REactive SEmi-Structured DAta. In *22nd IEEE International Enterprise Distributed Object Computing Conference, EDOC 2018*, pages 75–84, 2018.
14. Peter Sestoft. *Spreadsheet implementation technology. Basics and extensions*. MIT Press, United States, 2014.
15. Tijs Slaats. *Flexible Process Notations for Cross-organizational Case Management Systems*. PhD thesis, IT University of Copenhagen, January 2015.
16. 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*, pages 1–9, 2017.

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, 2017.
18. Jan Martijn E. M. van der Werf and Artem Polyvyanyy. The information systems modeling suite - modeling the interplay between information and processes. In Ryszard Janicki, Natalia Sidorova, and Thomas Chatain, editors, *Application and Theory of Petri Nets and Concurrency - 41st International Conference, PETRI NETS 2020, Paris, France, June 24-25, 2020, Proceedings*, volume 12152 of *Lecture Notes in Computer Science*, pages 414–425. Springer, 2020.