A Graph-Based Formal Semantics of Reactive Programming from First Principles

Bjarno Oeyen
bjarno.oeyen@vub.be
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium

Joeri De Koster
joeri.de.koster@vub.be
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium

Wolfgang De Meuter
wolfgang.de.meuter@vub.be
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium

ABSTRACT
In recent years, stream processing has become the de facto paradigm to process any kind of real-time data in many kinds of applications. Different libraries, frameworks and techniques exist which aim to make it easy to build stream processing applications in many modern programming languages. Libraries such as Reactive Extensions, Akka Streams, or web frameworks such as React and Vue are all based on the idea of data streams that are connected to graphs to model the flow of data in applications. To the best of our knowledge, there exist no formalism which captures the essential core semantics of these approaches in a straightforward, easy to understand, manner: namely its graph-based program structure and the turn-based propagations of values through this graph. In this paper, we present Karcharias, a formalisation of reactive programming (a paradigm that shares many core ideas found in the various aforementioned libraries and frameworks) that is built from first principles. Instead of extending an existing language with a graph-based stream processing framework, and formalising this integrated language, we formalised the reactive programming paradigm without relying on a base language (e.g., the \( \lambda \)-calculus). Using our formalism, we show how reactive programs (and thus, stream-based programs in general) need a way to construct a graph and to propagate events through that graph, even in the absence of a base language.

CCS CONCEPTS
• Software and its engineering → Data flow languages; • Theory of computation → Operational semantics.

KEYWORDS
Reactive Programming, Operational Semantics

ACM Reference Format:

1 INTRODUCTION
Stream processing has become a popular paradigm to process data in various kinds of applications. One approach of stream processing that has gained popularity in recent years is called reactive programming (RP) [2, 11]. Reactive programs declare the dependencies (i.e. constraints) between the time-varying signals that make up the program. Whenever a signal updates its value, all signals that depend on that signal are updated by recomputing their values, according to the specified constraints.

In this paper, we make a distinction between two different implementation styles (strains) for reactive programming languages (RPLs). Function-based reactive programming languages usually model reactive programs using signal functions. At each turn (i.e. change of an external, with respect to the RP program, input source that causes a re-computation), a value is propagated through a signal function (which may be composed out of different, smaller, signal functions) to not only produce an output, but also the signal function to use in the next turn. Examples of languages that implement function-based RP with signal functions, and variations thereof, are Yampa [24], SFRP [4], and Dunai [27].

Graph-based reactive programming languages model reactive programs as graphs. During the evaluation of graph-based reactive programs, a graph data structure is constructed (often referred to as the dependency graph) where nodes correspond with the signals and (directed) edges between the nodes correspond with the data dependencies between these edges. Instead of applying functions that produce values and (updated) signal functions, turns in graph-based RP are performed by propagating values along the edges of the dependency graph. A node (signal) is updated when one of its dependent nodes changes. This update is performed once all the dependent nodes have had a chance to update (to ensure the absence of glitches [5]), e.g., by performing the updates in the dependency graph in topological order (an approach often used when the dependency graph is static) or by using a height-ordered priority queue. Examples of RPLs that implement graph-based RP are FrTime [5], Frappé [6], REScala [28], and Flapjax [23].

Roughly speaking, each strain finds its origin in a different research community. Function-based RPLs find their origins in the world of functional programming (and are often implemented in functional programming languages like Haskell and Agda). They encourage programmers to compose programs in a pointfree style using operators like \( \gg \gg \gg \gg \), \&\&\& and *** [4], often by making use

---

1 Before signal functions, many function-based RPLs modeled signals as pure functions that return, given a timestamp [11] or a list of timestamps, e.g., as in [34], the corresponding value(s) of a signal. We ignore their existence in the rest of the paper since they have been mostly superseded by signal functions [20].
We first present a brief informal overview of Haai, the reactive programming language which we have based our formalisation on. Inspired by Lisp and Scheme, Haai uses s-expression syntax to denote reactors. Reactors are the core abstraction of Haai, they describe the structure of a reactive program. For example, the reactive program

\[
\text{(defr} \ (\text{sum-and-product} \ x \ y)\\\text{(out} \ (+ \ x \ y)\\(* \ x \ y)))
\]

defines a reactor (as a DAG, see Figure 1) that computes both the sum and product of two numbers and "returns" both of them (out is a special syntactic construct used to define multiple sink nodes; it is similar to values in Scheme). Instead of applying a procedure or a function as in non-reactive languages, reactors in Haai are deployed (i.e. instantiated) on signals whose value may change over time. We call an instantiation of reactor a deployment. For example, assuming that there are two signals time and velocity, the sum-and-product reactor can be deployed as (sum-and-product time velocity). This produces two new signals, containing respectively the sum and product of the two inputs, which will be updated whenever either time or velocity (or both) changes. The exact approach taken to update signals in an efficient manner is out of the scope of this paper.

Haai is a higher-order reactive language which allows dynamic reconfigurations of the dependency graph, without the need for carefully-designed switching operators. For example, in

\[
\text{(defr} \ (\text{twice} \ r \ x)\\(r \ (r \ x)))
\]

a higher-order reactor is defined which deploys a reactor given as input twice, connecting the sinks of one deployment to the sources of the second. As reactors are first-class values, a signal can carry them as their current value. For example, the expression \(\text{(if} \ (\text{even?} \ \text{time}) \ + \ -)\) corresponds with a signal whose value is either the + reactor or the - reactor, depending on whether time is even or odd. In other words, deployments of higher-order reactors contain holes which, at run-time, are filled in with the deployment of the reactor carried by the operator signal.

Deployments in Haai are disabled if they are not in use, which happens if the operator signal carries a different reactor w.r.t. a previous turn. Deployments are thus re-enabled when the operator signal carries the same reactor again. We call this kind of semantics toggle semantics.

Finally, Haai has support for anonymous reactors. For example, in

\[
\text{(defr} \ (\text{make-twice} \ r)\\(\text{rho} \ (x)\\(r \ (r \ x)))
\]

uses the rho syntactic form to create an anonymous reactor. Reactors employ lexical scoping. In other words, the two occurrences of \(r\) in \((r \ (r \ x))\) refer to the signal node \(r\) as defined in the first line. When the anonymous reactor is deployed, both \(r\)s refer to the \(r\) signal from its lexical scope. We call these reactors with lexical scope captures as they capture the signals from their environment (captures are similar to closures in non-reactive languages).

This concludes our brief overview of Haai. We refer to earlier work on Haai for more details about the language: e.g., the ability to recursively generate reactive programs is discussed in [25] and state management with deployments is discussed in [26].

3 OPERATIONAL SEMANTICS

This section presents the small-step operational semantics of graph-based reactive programming. An implementation of these semantics, using PLT Redex [12], can be found online.²

3.1 Syntax

Figure 2 presents an overview of the syntax of Karcharias. In Karcharias, a reactive program \(p\) consists of a set of reactor definitions. Each reactor definition gives a name to a graph. In graph-based reactive programs, the order of the connections in the graph matters. Inspired by A-Normal Form [13], we encode reactors (i.e.

²https://gitlab.soft.vub.ac.be/boeyen/karcharias/
DAGs) as a set of nodes \( (N) \) where each node is a triple containing the inputs (either names referring to signals defined in the lexical environment or defined locally by another node, or constant values that are in \( V \))\(^3\), the type of the node \( nt \), and the outputs (which are the names to define new signals with).

The reactive program in Figure 1 can be modelled as only two nodes: one for each deployment expression \( ([1, in_1, out_1], DEPLOY, [out_1, 0]) \) and \( ([1, in_1, out_2], DEPLOY, [out_2, 0]) \). The names \( in_{1, j} \) and \( out_{1, j} \) are special as they denote the input and output signals of reactors: the first index denotes the index of the source or sink signal (1-based indexing), and the second index denotes the scoping level (similar to De Bruijn Indices which were originally invented for \( \lambda \)-calculi \(^8\); using 0-based indexing) in case of nested reactor definitions (without nested reactor definitions or lexical scoping there is no need for the second index). Note that the operator itself is one of the inputs of the \( DEPLOY \), in order to support dynamic reactive programs (programs whose dependency graph can change at run-time).

There are two supported node types (\( nt \)) in Karcharias. The aforementioned \( DEPLOY \) nodes denote deployments of DAGs (i.e. holes in one graph that have to be filled in, at run-time, by another DAG). \( RHO(N) \) nodes denotes in-line DAG (reactor) definitions that have access to their lexical scope. I.e. a signal defined in the right-hand side of a node \( n \) is also accessible by the \( RHO(N) \)s of the same reactor. To disambiguate between the source signals of the nested and the surrounding signal DAGs, the source signal accessible as \( in_{1, 0} \) in the surrounding DAG is accessible as \( in_{1, 1} \) in the nested DAG. The same applies to sink signals (\( out_{1, 0} \) becomes \( out_{1, 1} \) in the nested DAG).

In the rest of the paper, we assume that programs are well-formed: we assume that the inputs and outputs are correct for every node\(^9\), that reactors are acyclic, that there are no free variables (except for globally-defined signals and operators such as \( \text{time} \) and \( + \), which are discussed later), and that each reactor has at least one sink node (i.e. \( out_{1, 0} \) has to be defined in every \( R(x, N) \)).

### 3.2 Semantic Entities

Reactive programs in Karcharias are executed in turns. In each turn, the values of a certain set of output signals are computed in terms of the values of certain external time-varying signals (such as \( \text{time} \)). We first explain the intra-turn semantics in Sections 3.2.3.4, and later use the intra-turn semantics to define in the inter-turn semantics in Section 3.5.

The semantic entities needed to express the intra-turn semantics are shown in Figure 3. Remember that, at run-time, a reactive program (i.e. a capture) is instantiated into a number of deployments, which has (usually) created new signals. These signals can then produce values in every turn. We describe these semantic entities in order, before discussing the configurations and the primitive operations.

#### 3.2.1 Captures

Captures are represented as a tuple containing a unique identifier \( c \), a set of nodes \( N \), and a lexical environment \( \Sigma \), which contains the bindings for the captures signals. Named reactors (those defined in \( p \)) have, among other entities, the external signals (such as \( \text{time} \)) in scope. We will explain this in further detail in Section 3.3.

#### 3.2.2 Deployments

Deployments are represented by two separate entities: a deployment wiring, and a deployment snapshot. Information about the structure of the reactive program’s dependency graph is maintained in a deployment wiring. Each wiring consists of an unique identifier \( I_d \) (which identifies the deployment), a set of uninstantiated nodes \( N \) and a signal environment \( \Sigma \). Information about the current values of the signals of a deployment are stored in a deployment snapshot. Each snapshot consist of a unique identifier \( I_d \) (which is always shared with a deployment wiring) and a signal environment \( \Sigma \). In Section 3.4, we will describe the reduction rules that operate on these wirings (i.e. to go from a set of nodes \( N \) to a signal environment \( \Sigma \)) and snapshots (i.e. to convert the signal environment \( \Sigma \) into a value environment \( E \)). The main idea here is that the deployment wirings can be retained across turns as they represent the structure of the dependency graph, independent of the values of the time-varying values.

When the program is running, the \( DEPLOY \) nodes will need to be replaced with instantiated deployments. To avoid needlessly re-creating parts of the dependency graph, it is important that the deployments that have filled in the holes introduced \( DEPLOY \) are kept in-between turns. Thus, every instantiated deployment node is represented by a branching points from \( I_d \), and all information regarding the location is stored in \( D \) (a toggle environment).

#### 3.2.3 Signals

There are four types of signals in Karcharias. 1) \( o \) signals are constant signals whose value never changes over time, which does not only include the domain values from Section 3.1,

![Figure 2: Syntax of Karcharias.](image)

![Figure 3: The semantic entities of Karcharias used in the operational semantics.](image)
but also primitive operation objects (see Section 3.2.5), captures, deployment identifiers \((i_d)\) as well as sequences of values (which is used to represent the sink values of any of the primitive operations). 2) \(S_{GLB}(x)\) signals are references to external signals, whose current value is stored in the configuration. 3) \(S_{REF}(\sigma, x)\) references a signal with a particular name (second element) as stored in a deployment’s (first element) signal environment. 4) And finally, \(S_{DEP}(\langle \sigma, \sigma' \rangle)\) corresponds with an (internal) signal that represents the creation of a new deployment (the \(\sigma\) represents a unique identifier for every branching point, \(\sigma\) represents the operator signal, and \(\sigma\) the operand signals).

### 3.2.4 Configurations
The state of a turn of a reactive program is thus modelled as a configuration \(k\) which consists of a value environment \(E\) containing the values of the external signals, a set of deployment identifiers that are active in the current turn \(I_d\) (i.e. to model toggle semantics), a set of deployment wirings \(W\), a set of deployment snapshots \(S\) (of the active deployments) and a toggle environment \(D\) (which stores, for every branching point \(i_b\) the captures, as identified by \(i_c\) that it has already instantiated into a deployment \(i_d\)).

### 3.2.5 Primitive Operations
Finally, we assume a set of primitives that are available to the operational semantics. Every primitive (denoted as a \(p\), has a corresponding function \(\delta_p\). We assume that every \(\delta_p\) is total (w.r.t. the values \(v\)).

### 3.3 Initial Configuration
Figure 4 shows how the initial configuration is constructed, given a reactive program \(p\). The initial configuration \(k_{init}\) contains an initial wiring in \(W_{init}\) that deploys the main reactor (as defined in \(p\)) to bootstrap the reactive program. It also contains a value environment, which contains the bindings for the captures for the reactors defined in \(p\) (these are created by applying the function ntoc on each \(N_i\)), as well as the initial values of the external signals (such as \(time, \ldots\)). The mechanism that determines the later values will be defined later as described in Section 3.5.

In the definition of \(W_{init}\) and ntoc an initial signal environment \(\Sigma_{init}\) is used. This environment contains bindings for: 1) The primitive operations. To disambiguate between the symbol + and the object representing its computation, the latter is typeset as \(\langle + \rangle\). We only provide +, but this set can easily be extended with more. 2) The primitive reactors in \(p\) (binding their name to an \(S_{GLB}\). And 3) the external time-varying signals. Similar to the primitive operations, we only assume the existence of time. However, this can also easily be extended.

### 3.4 Reduction Rules
After having presented the syntax and the semantic entities, we present the small-step operational intra-turn semantics. We make a distinction between two types of rules: wiring rules (or w-rules for short) that reconfigure the dependency graph and snapshot rules (or s-rules for short) that determine the values of the signals in a deployment during a given turn. Semantically, both kinds of rules can be executed non-deterministically in different regions of the reactive programs graph. Thus, there is only one reduction relation describing the intra-turn semantics \((\rightarrow_k)\). However, as most w-rules only operate on \(W\) objects, and a large part of the s-rules only on \(S\) objects: we define two helper reduction relations \((\rightarrow_w\) and \(\rightarrow_s)\) that operate only on a single \(W\) or \(S\) object.

#### 3.4.1 Wiring Rules
The reduction rules that perform wiring-level operations are presented in Figure 5. In short, the goal of these rules is to go from a set of nodes \(N\) to a signal environment \(\Sigma\) containing all the signals available to a deployment of a capture.

The \(w\)-ref rule replaces one of the named inputs of a node (left-hand side) with the actual signal as stored in \(\Sigma\). This rule ensures that all the other rules can operate with only signals (including constant signals) as inputs.

The purpose of the \(w\)-deploy rule is to create a hole that will be filled in by the s-rules with the actual deployment. In summary, \(w\)-deploy makes the following changes to \(\Sigma\). First, it binds \(S_{DEF}(\langle i_b, \sigma, \sigma' \rangle)\) to a unique name \(x\). This \(x\) will be reduced, by the s-rules, into a \(i_d\). The \(i_b\) uniquely identifies the hole. And secondly, for every output \(o_i\) in \(\Sigma\), \(o_i\) will be bound to \(S_{REF}(\langle i_p, x, out_{iA} \rangle)\). The inner \(S_{REF}\) will later be reduced to the \(i_d\) that the \(S_{DEF}\) bound to \(x\) reduces to, and the outer \(S_{REF}\) will then be reduced into the correct outgoing (sink) value of that deployment.

The \(w\)-rho rule creates a new capture (with a fresh, unique, \(i_c\)) that contains the current environment (with bindings for the signals that \(N\) depends on). To let the nested DAGs have access to the sources and sinks of the surrounding environment, the \(w\)-rho rule renames all input and output signals in \(\Sigma\) by incrementing its second index (which denotes the nesting level).

Finally, the \(w\)-congruence rule allows for capturing a local name (\(\langle \sigma, \sigma' \rangle\)) to the global reduction relation \((\rightarrow_k)\). This is the only global w-rule as the purpose of the w-rules is to configure the dependency graph without using the current values of any signals, and that no w-rule needs to interact with another deployment. Thus, there is no need for other global w-rules.

#### 3.4.2 Snapshot Rules
Figure 6 shows the snapshot reduction rules. These rules describe how a signal environment of a deployment is reduced into a value environment. Most of these rules make use of \(E\) which defines an evaluation context for signal environments. I.e. the reduction rules listed in Figure 6 are performing a kind of graph reduction. The evaluation context \(E\) therefore looks for a (sub)expression for a signal present in \(\Sigma\) which can then be reduced using the s-rules.

The \(s\)-self-ref rule replaces a \(S_{REF}\) referring to a local name with the result of looking up that name in its own environment (only if it is a value, and not a yet-to-be-reduced signal).

The \(s\)-deploy-primitive and \(s\)-tuple-ref rules are used for the primitive operations. The functions that model the primitive operations (i.e. the \(\delta_p\) functions) always return multiple values. Thus the \(s\)-deploy-primitive rule produces a list of values (\(v \rightarrow v\)) and the \(s\)-tuple-ref is therefore used to get a value from the list (depending on the index \(i\) of \(out_{iA}\)).

Similar to \(w\)-congruence, the \(s\)-congruence rule connects the local s-rules to the global reduction relation \((\rightarrow_k)\).
Given \( p = \{ R(x_1, N_1), \ldots, R(x_p, N_p) \} \) (\( p \) is the total number of user-defined reactor definitions).

\[
\begin{align*}
\delta_{\text{init}} &= \{ (i_1, \frac{\delta_{\text{init}}}{\delta_{\text{main}}}, W_{\text{main}}, \emptyset, 0) \} \\
\delta_{\text{init}} &= \{ (i_1, \text{no}(N_1), \forall i \in [1..p]) \} \\
W_{\text{init}} &= \{ W(i_1, \text{main}, (\{\text{main}\}, \text{DEPLOY}^2, \text{out}_{i_1}) | \forall i \in [1..|\text{main}|]), \delta_{\text{init}}) \} \\
\Sigma_{\text{init}} &= \{ x \mapsto S_{\text{DEPLOY}}(x_1) \} \cup (\delta_{\text{init}} \text{ time} + S_{\text{GLOB}}(\text{time}..))
\end{align*}
\]

Where \( |\text{main}| \) is the number of outputs defined in the main reactor in \( p \) and \( \text{no}(N) = C(i_c, N, \Sigma_{\text{init}}) \) (where \( i_c \) fresh)

**Figure 4:** Initial configuration, given a reactive program \( p \).

**Local Wiring Rules (\( \to_w \))**

\[
\begin{align*}
\text{(W-REP)} \quad & W(, ((\{x\} \cdot | \sigma |, n, \sigma)) \cup N, \Sigma) \to_w W(, ((\{x\} \cdot | \sigma |, n, \sigma)) \cup N, \Sigma) \\
\text{(W-DEPLOY)} \quad & x, i \text{ fresh} \\
& Y' = \Sigma[x \mapsto S_{\text{DEPLOY}}(i_p, | \sigma |)](\sigma_1 \to S_{\text{DEPLOY}}(i_p, x, \text{out}_{i_1}) | \forall i \in [1..|\Sigma|]) \\
& W(, ((\{x\} \cdot | \sigma |, \text{DEPLOY}, \sigma)) \cup N, \Sigma) \to_w W(, ((\{x\} \cdot | \sigma |, \text{DEPLOY}, \sigma)) \cup N, \Sigma) \\
\text{(W-BIO)} \quad & i \text{ fresh} \\
& \Sigma = \text{shift io}(\Sigma) \\
& W(, ((\{\Sigma \cdot | \sigma |, \text{io}(N), \emptyset)) \cup N, \Sigma) \to_w W(, ((\{\Sigma \cdot | \sigma |, \text{io}(N), \emptyset)) \cup N, \Sigma) \\
\text{Where shift io}(\Sigma) = (x' \mapsto \sigma | \forall \alpha \to \sigma \in \Sigma, x' = \{ m_{i,j} \} \\
& \text{other otherwise} \\
\text{Global Rules (\( \to_g \))} \\
& \text{w} \to_w \text{w'} \\
& K(, I_p, \{w\} \cup W, S, D) \to_g K(, I_p, \{w'\} \cup W, S, D)
\end{align*}
\]

**Figure 5:** Wiring (graph construction) rules of Kchararias. The \( \Sigma \) symbol denotes concatenation.

In the presence of toggle semantics, deployments can only be activated (i.e. creating a \( S \) from a \( W \)) if and only if the corresponding \( W \) is complete (i.e. its signal environment \( \Sigma \) has been fully populated, which is the case once \( N \) is empty) and the deployment is supposed to be active in the current turn. The s-activate rule does exactly that: when \( i_d \in I_d \) (the set of active deployments), and no existing \( S \) exists for \( i_d \) (in the current turn), a new \( S \) is created and added to \( S \) if the corresponding \( W \) is complete. Note that it is impossible for \( i_d \) to be in \( I_d \) without being a corresponding \( W \) (see the description of the s-deploy-new and s-deploy-existing rules).

The s-ref rule (the global counterpart of s-self-ref) replaces a \( S_{\text{GLOB}} \) with a value from a different deployment.

The s-global rule replaces a \( S_{\text{GLOB}} \) with its value as stored in \( E \).

The s-deploy-new is the most important rule of our formalisation. Its purpose is to deploy: i.e. create a new \( W \) for a \( S_{\text{DEPLOY}} \) that is present in a signal. It does so by first checking if the capture has already been deployed earlier for the same branching point (i.e. \( (i_p, c) \in \text{dom}(D) \)). If so, the s-deploy-existing rule will be used instead (which is explained next). If not, it is going to extend the lexical environment stored in the capture with bindings for the source signals as present in \( S_{\text{DEPLOY}} \) and then use it to create a \( W \) for that deployment. \( D \) is also updated to remember the deployment.

The final rule is the s-deploy-existing which is used if there exists already a deployment given the branching point and capture stored in the operator context. Instead of allocating a new deployment, it looks up the old deployment \( i_d \). The old wiring and any accumulated state will thus be reused.

In these last two rules, we show that deployments are instantiated with the argument signals (\( \Sigma \)) and not the current values of these signals. This mechanism ensures that in a next turn, no additional work is necessary to ensure that a deployment is reacting to the correct values, as in the new turn the lookup is performed again if the deployment is active. Note that this is not the case for
We now focus our attention to the inter-turn semantics. In short, that a configuration does not get stuck prematurely, this is the case for turn configurations.

Errors, type errors, arity errors… are currently not detected in our formalisation. In the future, we may prove this formally. Remark that the outputs of the main reactor are not used by \( \rightarrow \). While an actual implementation would need to use these as output (e.g., to actuate), this has not been formalised as we do not consider it part of Karcharias’ core computational model.

4 DISCUSSION

Our formalisation of graph-based RP provides a clear semantic for deployments: i.e. instantiated parts of (dependency) graphs. Deployment nodes in the dependency graph, which can be filled in at run-time, make it possible for a developer to compose reactive programs out of smaller programs, similar to the composition of signal functions in function-based RP. An interesting future research avenue is to further compare the semantics of graph-based RP with the semantics (and implementations) of function-based RP: e.g., to establish whether or not they are computationally equivalent (i.e. can they implement the same RP programs).

Our work presented here does not formalise all possible aspects of an RPL. One noteworthy omission are conditional deployments created using if. We argue that semantics are already captured by the higher-order deployments. By “delaying” a graph using \( \text{s-tuple-Ref} \) and providing an eager \( \delta_f \), one can easily model conditional deployments (and thus, conditional deployments) using our current formalisation. In the future, we may prove this formally.

Another noteworthy omission is the support for stateful computations (e.g., using operators like foldp [7] or using other state management mechanisms [26]). We argue that doing so is rather trivial. In short, such functionality can be formalised by providing a heap-like store to store turn-transient information in.

Finally, graph-based RPLs usually update their signals incrementally: only signals affected by an (external) change are usually recomputed. This is completely absent from our formalisation as at the start of each turn the next-turn rule throws away all SS. However, incrementality is, in essence, only an optimisation. It thus should have no impact on the semantics. Hence we argue that there was also no need to explicitly formalise this.

5 CONCLUSION

This paper presented Karcharias, a small-step operational semantics of a graph-based reactive programming language. While our formalisation, as presented in this paper, mainly focuses on the reactive programming paradigm, we have observed many similarities with other streaming-based solutions (e.g., Reactive Extensions [22] and Akka Streams [33]). We hypothesise that our formalisation can be generalised such that it also captures other graph-based reactive-like languages.

ACKNOWLEDGMENTS

Bjarno Oeyen is funded by the Research Foundation - Flanders (FWO) under grant number 1S93822N.