Using process algebra to statically analyze incremental propagation graphs

Théo Le Calvar[1 2], Fabien Chhel[2], Frédéric Jouault[2], and Frédéric Saubion[1]

[1] LERIA, Université d'Angers, France
{firstname.lastname}@univ-angers.fr
[2] ERIS, ESEO-TECH, Angers, France
{firstname.lastname}@eseo.fr

Abstract. Active Operations are a set of operations that can be composed to build incremental bidirectional OCL-like expressions on collections. Each operation is capable of updating its result (resp. source) when a change occurs on its source (resp. result). The current implementation of active operations relies on the Observer design pattern to propagate changes from each operation to its successors. These relations form an implicit directed acyclic propagation graph. Previous work showed that this approach is limited and alignment issues appear in some situations. Several workarounds were proposed to mitigate these issues. In this work we present a new relational notation to describe propagation graphs. Along with this notation, we also present a new static analysis method of the propagation graph based on process algebra. This new method enables optimizations of the propagation graph not achievable with previous approaches, such as detection of parallelizable sections of the propagation graph or cache optimizations in specific situations.

CEUR-WS.org/Vol-2245/ocl_paper_6.pdf

1 Introduction

In Model-Driven Engineering (MDE) [10], model transformation is used to bridge the gap between design and implementation by providing tools to express links between different models. These models are typically subject to constant changes, in particular due to requirements changes. In this case, dependent models must be updated appropriately. When considering small and numerous changes, the execution of the whole transformation process is costly. Therefore, it is important to be able to focus only on the elements of the models that have been changed, which is usually referred to as incremental transformation.

The incremental evaluation of transformations has many advantages since it reduces the amount of computation required to obtain a new target model, after a developer has updated a source model, or even the reverse case. The main purpose is then to devise more responsive systems for the model designer. Another purpose is to avoid creating new target elements, but rather to update existing ones, which may be connected to other kinds of objects (e.g., view objects). Creating new elements would break such connections. For instance,

an incremental execution can be used to update target models in-place and to update the target model's visual representation automatically.

Active operations [5,1] have been defined to ensure incremental model transformations, considering even bidirectional transformation processes when relevant. In the present work, we will only consider forward change propagation. In active operations, mutable values from the models (i.e., values that can change) are wrapped into observable boxes. Operations can then be applied to these boxes in order to compute initial values and also to propagate changes. Each operation ensures the change of its result boxes when changes occur on its source boxes. These operations can be combined in order to achieve complex transformations. They thus form a directed acyclic propagation graph[3] that connects source boxes to target boxes via active operations and intermediate boxes [7]. Active operations have been used for incremental OCL (Object Constraint Language) evaluation in the Active Operations Framework (AOF), which has been shown to scale to relatively large models [6]. However, in AOF operations observe their source boxes in order to be notified of changes, resulting in an implicit propagation graph. We observed in [7] that implicit propagation graphs can lead to change alignment issues, which can result in invalid propagations, and useless computations. These problems occur when an operation can be notified several times during propagation of a single change. This can notably happen to operations having several source boxes.

In [7] several approaches have been presented to mitigate these alignments issues. It has been shown that decoupling propagation graphs from propagation algorithms allows a better handling of complicated situations (e.g., alignment issues with `zip` operation). In this paper, we present a formalization of propagation graphs, as well as a static analysis approach that is based on process algebra. This leads to more in-depth analysis of the propagation graphs. For instance it can be used to discover parallelizable parts of the propagation.

The paper is organized as follows. section 2 describes the problems due to implicit propagation and previously presented approaches. section 3 defines the main concepts used in our approach. section 4 presents the method used to build a process algebra formula from a given propagation graph. Then, section 5 presents several possible outcomes of our analysis method. We discuss about other interesting properties of the approach in section 6 and section 7 provides some concluding remarks.

2 Context and problem

2.1 Active operations

Active Operations consist of a set of OCL-like operations operating on collections (e.g., `collect`, `select` but also `zip` and `zipWith`[4]). After initialization, each operation maintains synchronization between its source and its result. If the

[3] Called propagation graph in the rest of the paper.
[4] Borrowed from functional languages like Haskell.

source is modified, the operation incrementally computes how the result should be changed to maintain synchronization. By composing these simple operations in a functional way it is possible to build complex incremental expressions. Active Operations can operate in both unidirectional and bidirectional mode. In this work we focus on unidirectional transformations but similar concepts could be applied to reverse change propagation.

The current implementation of Active Operations relies on the well-known *Observer* design pattern. Each value is boxed so that it can be observed. Operations observe changes on their sources and react whenever an update occurs. An operation then computes the corresponding target updates and notifies its successors (i.e., next operations). This process is called *Observer-based propagation*. Source boxes are boxes that are not computed by operations and thus that correspond to entry-points of the transformations (i.e., properties of source model elements).

2.2 Problem statement illustrated on motivating examples

We use a graph to represent the propagation process. Nodes correspond to operations and boxes[5]. Small black dots correspond to inputs and outputs of operations, which can have several inputs and outputs. Arrows connect outputs of upstream operations to inputs of downstream operations.

For instance, the graph in Figure 1 corresponds to the following expression, which is the motivating example from [7]:

```
def: f(a) =
  let b = a->collect(λ1) in
  let c = a->collect(λ2) in
  let d = b->zip(c) in
  d
```

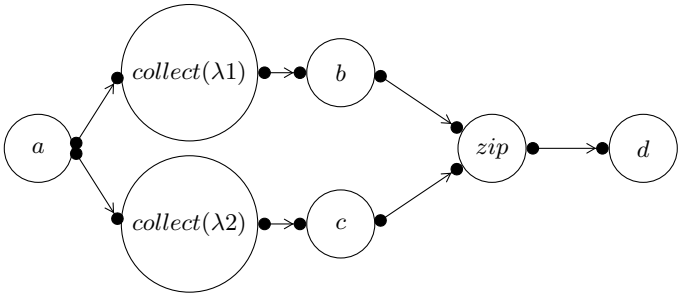


Fig. 1: Simple example with alignment issues

[5] Boxes can be assimilated to identity operations, which store intermediate results and forward changes. In complex examples we omit them for clarity.

A more complex example with multiple source boxes is presented in Figure 2, which corresponds to the following expression:

```
def: g(a, b) =
  let v1 = a->A() in
  let v2 = a->B(b) in
  let v3 = v1->C(v2) in
  let v4 = v2->D() in
  let c = v3->E(v4) in
  c
```

Note that in this example we used dummy operation names (as our approach only cares about the shape of the graph) and removed intermediate boxes from the graph representation.

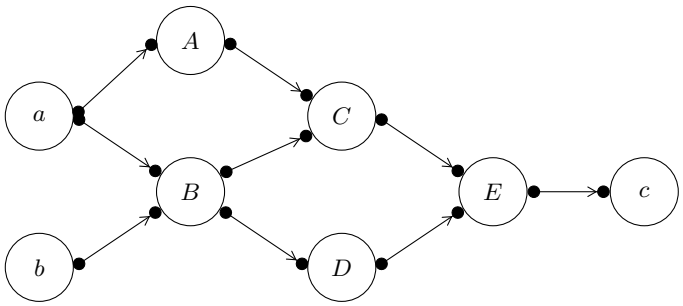


Fig. 2: Another example with alignments issues

In [7], it has been pointed out how the observer-based approach is flawed when operations with several inputs depend on the same input box, directly or indirectly. Several possible mitigations have been proposed.

These problems occur when there are several paths between a given source box and operation. For instance, with a depth-first traversal of the propagation graph, the expression depicted in Figure 1 would notify the `zip` operation twice, which would also notify `d` twice. In [7] a simple workaround has been proposed that consists in adding an option to the `zip` operation so that it can ignore one of its inputs. This workaround is used in our current implementation of Active Operations. However, it has proven to be complex to use for users without a solid understanding of the internal propagation algorithms.

Another possible approach, presented in [7], consists in taking advantage of an explicit propagation graph. Using an explicit propagation graph, it is possible to perform a breadth-first propagation instead of a depth-first one. The breadth-first propagation does not exhibit the same problematic behavior than the depth-first propagation.

In this paper, we present a new approach also based on an explicit propagation graph but that uses the process algebra formalism to define valid propagation ordering.

3 Background and definitions

3.1 Relational model for active operations

Active Operations are operations (such as `bind`, `collect`, `select`, `union` or `zip`) over collections (e.g., singletons, sets or ordered sets). After an initialization phase, which ensures that the inputs and outputs are in a coherent state, changes on the inputs are transformed into changes on the outputs so that both stay coherent. When combined, these operations constitute complex expressions.

For instance, let $a = [1, 2, 3]$ (using the Haskell notation for lists), then the following expression multiplies each value of a by 2 and then selects the multiples of 3, $h(a) = [6]$.

```
def: h(a) =
  let b = a->collect(e | e * 2)->filter(e | (e mod 3) = 0) in
    b
```

Based on these expressions we derive the following relational notation. Let us consider the following sets:

- O is a set of (active) operations (e.g. collect or filter in h)
- Π is a set of ports that represent inputs and outputs of operations

Given O and Π, a transformation is defined by the following relations :

- $\Pi_{in} \subseteq O \times \Pi$ describes the inputs ports of an operation
- $\Pi_{out} \subseteq O \times \Pi$ describes the outputs ports of an operation
- $L \subseteq \Pi \times \Pi$ represents the links between the ports that describe the different interactions between operations in order to achieve the transformation.

The following conditions are required:

- $\forall (p, p') \in L, (\exists o \in O, (o, p) \in \Pi_{out}) \wedge (\exists o' \in O, (o', p') \in \Pi_{in})$ (elements of L are used to link outputs of an operation with the input of another operation.)

A transformation is a tuple $Trans = (O, \Pi_{in}, \Pi_{out}, L)$. However this tuple alone cannot ensure the transformation is correct. Hence a specification of existing operations is needed.

3.2 Operation specification

Let us consider $\Sigma = \{\sigma_1, \sigma_2, \cdots\}$ a set of sorts and a set $P = \{p_1, p_2, \cdots\}$ of port variables. Given an operation $o \in O$ we consider the two following functions :

- $P_{in} : O \to 2^{(P \times \Sigma)}$
- $P_{out} : O \to 2^{(P \times \Sigma)}$

such that $P_{in}(o)$ (resp. $P_{out}(o)$) define the set of sorted input (resp. output) ports of o.

Back to our `zip` example, we would have $P_{in}(\texttt{zip}) = \{\texttt{left} : l, \texttt{right} : r\}$ and $P_{out}(\texttt{zip}) = \{\texttt{out} : (l, r)\}$. Let us remark that (1) we denote pairs (p, s) of

the Cartesian product $(P \times \Sigma)$ as $p : s$ and (2) that such a `zip` operation should be formally defined for all relevant sorts $s \in \Sigma$.

An operation specification is thus a tuple $Spec = (O, P_{in}, P_{out}, \Sigma)$.

We can list other operations such as:

- `collect`: $P_{in}(\texttt{collect}) = \{\texttt{in} : i\}$, $P_{out}(\texttt{collect}) = \{\texttt{out} : o\}$
- `select`: $P_{in}(\texttt{select}) = \{\texttt{in} : i\}$, $P_{out}(\texttt{select}) = \{\texttt{out} : i\}$
- `concat`: $P_{in}(\texttt{concat}) = \{\texttt{in}_1 : s, \texttt{in}_2 : s\}$, $P_{out}(\texttt{concat}) = \{\texttt{out} : s\}$

This list is not complete but we can generalize operations based on their arity. Let o_1 be an operations with arity 1, it is defined as $P_{in}(\texttt{o}_1) = \{\texttt{in} : i\}$, $P_{out}(\texttt{out} : o)$. For o_2 a binary operation we have, $P_{in}(\texttt{o}_2) = \{\texttt{in}_1 : i_1, \texttt{in}_2 : i_2\}$, $P_{out}(\texttt{out} : o)$, and so on for arities greater than 2.

Some special operations may have ports named differently for semantic reasons such as `zip`.

3.3 Validation of a transformation with respect to a specification

Given an operation specification $Spec = (O, P_{in}, P_{out}, \Sigma)$, and a transformation $Trans = (O, \Pi_{in}, \Pi_{out}, L)$, we need to check if $Trans$ is a valid instance with respect to $Spec$.

A transformation mapping from $Trans$ to $Spec$ consists in defining the following mappings :

- $\chi_{ports} : \Pi_{in} \cup \Pi_{out} \to P_{in} \cup P_{out}$
- $\chi_{sorts} : \Pi_{in} \cup \Pi_{out} \to \Sigma$

A transformation mapping $\chi_{Trans,Spec}$ is thus a pair $(\chi_{ports}, \chi_{sorts})$.

A transformation mapping is valid if the following conditions are satisfied :

- $\forall o \in O, \forall p \in \Pi, (O, p) \in \Pi_{in} \Rightarrow \chi_{ports}(p) : \chi_{sorts}(p) \in P_{in}(o)$
- $\forall o \in O, \forall p \in \Pi, (O, p) \in \Pi_{out} \Rightarrow \ \chi_{ports}(p) : \chi_{sorts}(p) \in P_{out}(o)$

These conditions just ensure that the input/output specifications of the operations are satisfied.

3.4 Graphical view of relational notation

To ease the visualization of propagation in these expressions we introduce a graphical notation. It consists of a directed graph, nodes represent operations and arcs are links betweens them. An incoming (resp. outgoing) arc is an input (resp. output) parameter of an operation. If an operation has multiple inputs or outputs, each distinct input/output corresponds to a distinct black dot outside the operation. Arguments can be named to remove any ambiguity for operations with multiple inputs or outputs. An operation can have multiple inputs but only one incoming arrow for each of its input. An operation may have multiple arrows going out from a single output.

Figure 3 corresponds to the expression shown in subsection 3.1.

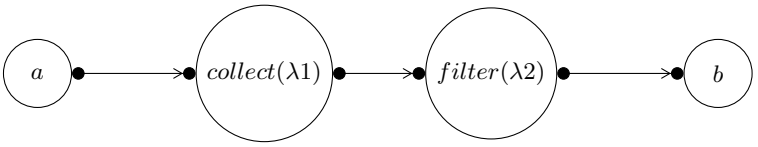


Fig. 3: Propagation graph corresponding to expression $h(a)$

The previous formalism allows us to represent an instance of a transformation that should be clearly related to a set of given specified operations, which have a specific profile (signature). For instance, a `zip` operation is represented by the graph shown in Figure 4. It should be noted that both inputs can have different types, which induce the type of the result.

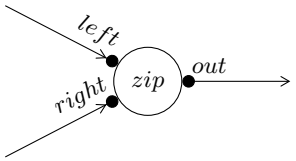


Fig. 4: Graphical representation of the zip operation.

3.5 Process algebra

The foundations of process algebra have been developed by Milner [9] and Hoare [4]. Process algebra constitutes a framework for reasoning about processes and data, focusing on processes that are executed concurrently. In particular, process algebra can be used to formally derive properties of a system specification. A system is expressed by a process term, using operators. It is then possible to force actions into communication using input/output relations.

We recall here the basic material related to process algebra that will be useful in our context (we refer the reader to [3] for more details). In this work, we use a variant of process algebra called Algebra for Communicating Processes (ACP) introduced in [2].

- **Actions:** we consider processes that are composed of atomic actions $a, b, c, \cdots$, or steps. Let $A = a, b, c, \cdots$ be the set of atomic actions.
- **Operations on actions:** atomic actions can be combined by means of operators in order to define more complex processes. The sequential composition, is denoted . (i.e., a dot, which can be omitted when there is no ambiguity). The alternative composition is denoted $+$. For instance, $a.(b+c)$ is a process

that executes the action a, then executes b or c and then stops. These two operators are the building blocks for complex processes.

- **Deadlock:** we define the special action δ, called the deadlock. δ represents failure: once δ is reached, the process is blocked. In the previous example, $a(b+c)$, once the process has executed a then b or c it gracefully finishes. If we consider $a.b.\delta$, once a and b have been executed, the process fails since it cannot escape from the deadlock state.
- **Parallel execution:** processes can be executed in parallel thanks to parallel composition operators. We note $x\|y$ the process that starts with x or y and then executes the remaining of x and y in parallel. We introduce an auxiliary operator $\lfloor\!\lfloor$ such that $x\lfloor\!\lfloor y$ behaves like $x\|y$ but forces to start with x. Therefore we have $x\|y = x\lfloor\!\lfloor y + y\lfloor\!\lfloor x$.
  While $\|$ is sufficient to describe basic parallelization, it does not allow us to describe situations where two processes have to communicate.

To model communications we define a partial binary function $\gamma : A \times A \to A$, where $\gamma(a, b)$ represents the result of the communication between a and b. If $\gamma(a, b)$ is not defined then it means that a and b do not communicate. We use this function to define a new parallel composition operator $|$, such that $a|b = \delta$ if $\gamma(a, b)$ is not defined and $a|b = \gamma(a, b)$ when it is defined. Using this new operator we get $x\|y = x\lfloor\!\lfloor y + y\lfloor\!\lfloor x + x|y$.

Finally, we add the encapsulation operator ∂_H, $H \subseteq A$. This operator replaces every action in H by δ. For instance $\partial_H(a + b)$, with $H = \{a, c, d\}$, is equivalent to $(\delta + b)$.This operator is useful to force communication between two processes. The axioms corresponding to the resulting algebra, defined by A and the previously described operators and functions, are defined below:

A1 $x + y = y + x$	CM4 $(x+y)\lfloor\!\lfloor z = x\lfloor\!\lfloor z+y\lfloor\!\lfloor z$	D1 $\partial_H(a) = a$ if $a \notin H$
A2 $(x+y)+z = x+(y+z)$	CM5 $ax\|b = (a\|b)x$	D2 $\partial_H(a) = \delta$ if $a \in H$
A3 $x + x = x$	CM6 $a\|bx = (a\|b)x$	D3 $\partial_H(x+y)=\partial_H(x)+\partial_H(y)$
A4 $(s + y)z = xz + yz$	CM7 $ax\|by = (a\|b)(x\|y)$	D4 $\partial_H(xy) = \partial_H(x).\partial_H(y)$
A5 $(xy)z = x(yz)$	CM8 $(x + y)\|z = x\|z + y\|z$	SC1 $x\|y = y\|x$
A6 $x + \delta = x$	CM9 $x\|(y + z) = x\|y + x\|z$	SC2 $x\|y = y\|x$
A7 $\delta x = \delta$	C1 $a\|b = b\|a$	SC3 $x\|(y\|z) = (x\|y)\|z$
CM1 $x\|y = x\lfloor\!\lfloor y+y\lfloor\!\lfloor x+x\|y$	C2 $(a\|b)\|c = a\|(b\|c)$	SC4 $(x\lfloor\!\lfloor y)\lfloor\!\lfloor z = x\lfloor\!\lfloor (y\|z)$
CM2 $a\lfloor\!\lfloor x = ax$	C3 $\delta\|a = \delta$	SC5 $(x\|ay)\lfloor\!\lfloor z = x\|(ay\lfloor\!\lfloor z)$
CM3 $ax\lfloor\!\lfloor y = a(x\|y)$	HA $x\|y\|z = \delta$	SC6 $x\|(y\|z) = (x\|y)\|z$

4 Translating propagation graph into process algebra

In order to generate an ACP formula corresponding to the propagation graph, we first need to enrich the relational representation with intermediary operations.

Isolating the sub-propagation graphs

First, we define $Start = \{o \in O | P_{in}(o) = \emptyset\}$, the set of starting operations, these operations are the entry-points of the transformations.

To generate an ACP formula that is equivalent to the transformation, we need to split the transformation for each of its entry-points. To this aim, we need to define the notion of path between two operations.

There is a path from operation a to operation b, noted $Path(a, b)$ if any is verified:

- $\exists (p, p') \in L, (\exists (a, p) \in \Pi_{out}) \wedge (\exists (b, p') \in \Pi_{in})$
- $\exists o \in O, Path(a, o) \wedge Path(o, b)$

The sub-propagation graph of a transformation $t = \{O, \Pi_{in}, \Pi_{out}, L\}$ for a given entry-point s, $PTrans(t, s) = \{O', \Pi'_{in}, \Pi'_{out}, L'\}$, is derived from t with :

- $O' = \{s\} \cup \{o \in O | Path(s, o)\}$
- $\Pi'_{in} = \{(o, p) \in \Pi_{in} | o \in O'\}$
- $\Pi'_{out} = \{(o, p) \in \Pi_{out} | o \in O'\}$
- $L' = \{(p, p') \in L | (p \in \Pi'_{in}) \wedge (p' \in \Pi'_{out})\}$

The sub-propagation graph of an entry-point keeps only the operations, ports and arcs that depends on the entry-point. This sub-propagation graph of an entry-point corresponds to the part of the propagation graph involved in the propagation resulting from a change on that entry-point.

Adding the synchronization operations

We need to add special synchronization operations in the transformations. They are required when an operation can be reached from an entry-point by more than one path. An operation o needs synchronization if $\exists o' \in O, \exists o'' \in O, o' \neq o'', Path(s, o') \wedge Path(o', o) \wedge Path(s, o'') \wedge Path(o'', o)$.

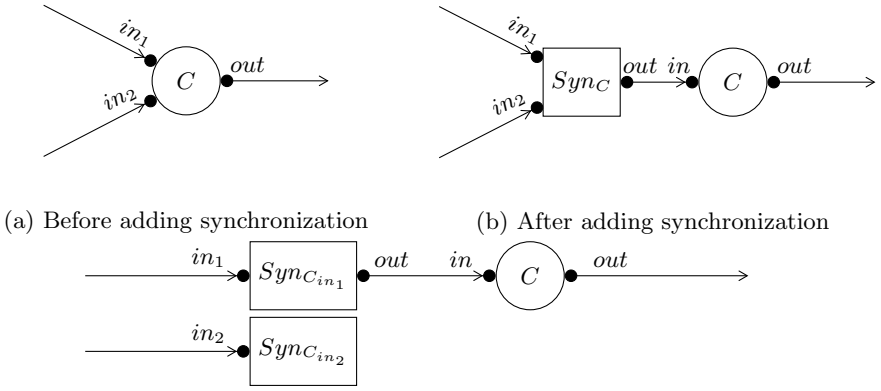
Adding a synchronization operation is achieved by adding an operation and modifying the operation o that needs synchronization.

A synchronization operation, Syn_o, that takes all the inputs of the former operation o, $P_{in}(Syn_o) = P_{in}(o)$ and has one output, is created. o is modified so that it takes only one input. Then a link is added between Syn_o and o. Figures 5a and 5b illustrate this step. Remark: because this transformation is only for static analysis, there is no need to consider how the actual data that flows from the inputs is merged into a single output, but one can imagine that the synchronization operation performs some kind of pairing.

Split of the synchronization operations

Considering now propagation graphs with only one input and synchronization operations when they are needed, we can divide each synchronization operation that has been added in the previous step.

Each synchronization operation is split into multiple ones, based on its inputs. Let s be a synchronization operation. For each $i \in P_{in}(s)$, we create an operation s_i with $P_{in}(s_i) = i$ and $P_{out}(s_i) = \emptyset$. One of the newly created operations, s' is selected to receive the output pin of s, $P_{out}(s') = P_{out}(s)$ and finally s is removed. We define $Split(s)$ as the set of all the new operations intruded while splitting a synchronization node s, $Split(s) = \{s_i | \forall i \in P_{out}(s)\}$.



(a) Before adding synchronization

(b) After adding synchronization

(c) After splitting synchronization operation

Fig. 5: Graphical representation of an operation needing synchronization

We also need to define the ACP communication function for each pair of the synchronization operations that have been created in this step (i.e., we define $\gamma(o, o')\ \forall o, o' \in Split(s), o \neq o'$). This allows synchronization operations to communicate together.

Finally, we keep track of all the operations that were added in $Syncs(o)$, so that we can use them later, with o the entry-point of the partial transformation being processed.

This step is illustrated in Figure 5c, in this example, $Syn_{C_{in_1}}$ is the operation that was selected to receive the output.

Building the process algebra formula

Since the propagation graph has now been processed, it can easily be converted to an ACP formula. We define the function $Next : O \to O$ that returns all the operations directly connected to an output port of an operation. Let o be an operation. $Next(o)$ is defined as:

$$Next(o) = \{o' \in O | \forall p \in P_{out}(o) \wedge \exists p' \in P_{in}(o') \wedge (p, p') \in L\}$$

Finally, we define $R2ACP(o)$, a function that takes an operation and transcribes it to an ACP formula.

$$R2ACP(o) = \begin{cases} Next(o) = \emptyset, & o \\ Next(o) = o', & o \ . \ R2ACP(o') \\ Next(o) = \{o_1, ..., o_n\}, & o \ . \ \Big(R2ACP(o_1) \parallel ... \parallel R2ACP(o_n)\Big) \end{cases}$$

This function returns an ACP formula for any given operation: the complete formula can be created by using the alternative composition between the formula

of each entry-points of the transformation. We also need to add encapsulation to force a synchronization operation to wait for its counterpart.

$$\sum_{o \in Start} \partial_{\{s \in Syncs(o)\}}\big(R2ACP(o)\big)$$

This method can be seen as a tree traversal with sequential composition of operations when going down the tree and parallel composition between the children of an operation.

A complete example is shown in Figure 6. Labels on the ports were removed to make the reading easier. The graph in Figure 6a has only one entry-point, so we can skip the isolation of the sub-propagations graphs. In Figure 6b synchronization operation $Sync_C$ is added before C because it can be reached from B and D. Then, in Figure 6c the synchronization operation $Sync_C$ is split into two new synchronization operations $Sync_{C_B}$ and $Sync_{C_D}$ and the first one is chosen to keep the output of $Sync_C$. Finally, in Figure 6d a corresponding formula is given. Below the formula all allowed communications are listed (only $Syn_{C_B}|Syn_{C_D}$ in this example).

5 Possible analysis outcomes

Once the propagation graph has been transformed into a tree and after a corresponding ACP formula has been generated, the formula can be used to infer several interesting facts about the transformation.

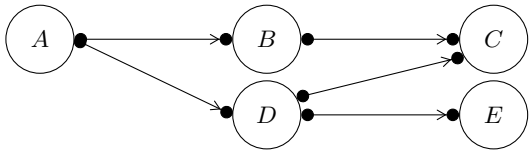
The generated formula defines all valid operations orderings. Thus it can be used to check if existing approaches generate correct orderings. It can also be used to generate new propagation orderings.

During the graph transformation, synchronization operations are added when needed. Adding these synchronization operations generates situations where synchronization is needed explicitly unlike the classical approach that consists in ignoring the first notification. Therefore, it does not require a deep understanding of the propagation. Moreover, adding this operation also means that the `zip` and `zipWith` operations do not need to use a specific algorithm to deal with alignments problems.

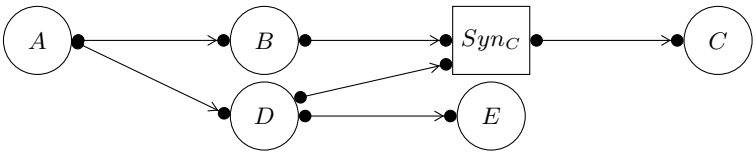
As observed in section 4, propagation graphs are split into sub-propagation graphs and a formula is created for each of these graphs. This means that only relevant operations are considered during propagation. This results in a specialized formula for each of the entry-points of the transformation.

Unlike topological sorting that can be used in breadth-first propagation (as proposed in [7]), generated formulas also contain explicit parallel sections. Each time a $\|$ is found, actions in both its operands can safely be executed in parallel[6]. For instance, with the formula in Figure 6d it is possible to infer that both operations B and D can be executed in parallel (it is also true for C and E).

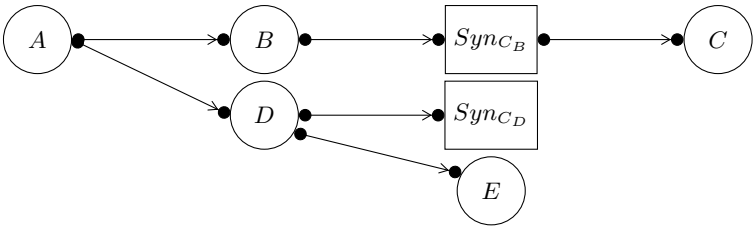
[6] Assuming lambdas given to operations do not have any side effects.



(a) Propagation graph



(b) Adding synchronization nodes



(c) Splitting the synchronization nodes

$$F = \partial_{\{Syn_{C_B},\ Syn_{C_D}\}}\Big(A \cdot \big((B \cdot Syn_{C_B} \cdot C) \parallel (D \cdot (Syn_{C_D} \parallel E))\big)\Big)$$

$$Syn_{C_B}|Syn_{C_D} = S$$

(d) Corresponding ACP formula and communication function

Fig. 6: Illustrated example of transformation from propagation graph to ACP formula.

Exploiting these parallel sections would not be difficult with a central algorithm responsible for dispatching operation execution as proposed in [7].

Finally, with an explicit graph transformation and a corresponding formula, it is possible to detect patterns in the transformation. These patterns could be used to infer parts of the transformation where operations do not need to keep a cache. For instance, with a `zip` operation, it is usually convenient to keep a cache of its inputs in case a change arrives only on one of these inputs. In the current implementation it is not possible to detect situations where it can be proven that notifications will always arrive in pair (e.g., Figure 1). Notifications are said to arrive in pair if a change in a source box leads to two changes in the sources of a `zip` or `zipWith` operation, and if both changes of the pair always have the same index (e.g., both add an element at index 3). In this case, the operation does not need to keep a cache of its inputs because all information

needed to compute the resulting output change is known to be available after both inputs have been notified.

6 Discussion

In the previous section we presented several possible analyses offered by the process algebra formula generated by our approach. In this section we discuss about other usages of the formula not directly related to propagation optimization.

In Section 3.5 we presented a small subset of process algebra. This subset is sufficient to express formulas corresponding to current propagation graphs. However, there are many extensions and features that could be used to describe currently forbidden propagation graphs, such as propagation graphs containing cycles. In the current implementation, propagation graphs containing cycles are not allowed. With a depth-first propagation, such graphs can lead to infinite propagation. Nevertheless, process algebra does support recursive formula. Process algebra could thus provide an appropriate representation for such propagation graphs. However this does not solve the termination problem of such propagation. In order to solve this issue, it would be necessary to add a mechanism to detect when the propagation should be ended (such as reaching a fixed point). This mechanism may be usable to represent transformations combining classical transformations with constraints solving [8].

Another interesting point we did not investigate is the relationship between the source propagation graph and the generated formulas. During processing of the sub-propagation graph, after a synchronization operation is split, the rest of the propagation is attached to only one of the newly created synchronization operations. Each choice leads to a different formula. We believe that these possible formulas are equivalent and result in similar execution orders. This might be provable by showing that all possible formulas are bisimilar.

Moreover, due to the transformation process of the propagation graph, we believe that there is a possibility to rebuild a propagation graph from a given ACP formula. This would enable rewriting of the formula to be forwarded back to the propagation graph. The optimizations applied on the formula could be visualized on the propagation graph.

7 Conclusion

This paper is a follow-up to [7], which presented alignment issues appearing in implicit propagation graphs. This previous work suggested the use of explicit propagation graphs to solve alignments issues. In this paper, we kept the idea of explicit propagation graph and developed another analysis method. This new method is based on a new relational notation for propagation graph and on a transformation from this new notation to process algebra formulas. These formulas can be used to derive operation execution orders. By correctly building these formulas, it is possible to prevent alignment issues.

On top of correct ordering, process algebra comes with many interesting properties for propagation. One of them is the possibility to safely parallelize sections of the propagation.

Acknowledgments

Work partially founded by Angers Loire Métrople and RFI Atlanstic 2020.

References

1. Beaudoux, O., Blouin, A., Barais, O., Jézéquel, J.: Active operations on collections. In: Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part I. Lecture Notes in Computer Science, vol. 6394, pp. 91–105. Springer (2010). https://doi.org/10.1007/978-3-642-16145-2_7, https://doi.org/10.1007/978-3-642-16145-2_7
2. Bergstra, J., Klop, J.: Process algebra for synchronous communication. Information and Control **60**(1), 109 – 137 (1984). https://doi.org/https://doi.org/10.1016/S0019-9958(84)80025-X, http://www.sciencedirect.com/science/article/pii/S001999588480025X
3. Bergstra, J.A., Ponse, A., Smolka, S.A.: Handbook of process algebra. Elsevier, Amsterdam (2001)
4. Hoare, C.A.R.: Communicating sequential processes. Commun. ACM **21**(8), 666–677 (1978). https://doi.org/10.1145/359576.359585, http://doi.acm.org/10.1145/359576.359585
5. Jouault, F., Beaudoux, O.: On the use of active operations for incremental bidirectional evaluation of OCL. In: Proceedings of the 15th International Workshop on OCL and Textual Modeling co-located with 18th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2015), Ottawa, Canada, September 28, 2015. CEUR Workshop Proceedings, vol. 1512, pp. 35–45. CEUR-WS.org (2015), http://ceur-ws.org/Vol-1512/paper03.pdf
6. Jouault, F., Beaudoux, O.: Efficient ocl-based incremental transformations. In: OCL@ MoDELS. pp. 121–136 (2016)
7. Jouault, F., Beaudoux, O., Brun, M., Chhel, F., Clavreul, M.: Improving incremental and bidirectional evaluation with an explicit propagation graph. In: Seidl, M., Zschaler, S. (eds.) Software Technologies: Applications and Foundations. pp. 302–316. Springer International Publishing, Cham (2018)
8. Le Calvar, T., Chhel, F., Jouault, F., Saubion, F., Groupe, E.A.: Transformation de modèles et contraintes pour l'ingénierie dirigée par les modèles. In: Journées Francophones de Programmation par Contraintes 2018. p. 93 (Jun 2018)
9. Milner, R.: A Calculus of Communicating Systems, Lecture Notes in Computer Science, vol. 92. Springer (1980). https://doi.org/10.1007/3-540-10235-3, https://doi.org/10.1007/3-540-10235-3
10. da Silva, A.R.: Model-driven engineering: A survey supported by the unified conceptual model. Computer Languages, Systems & Structures **43**, 139 – 155 (2015). https://doi.org/https://doi.org/10.1016/j.cl.2015.06.001, http://www.sciencedirect.com/science/article/pii/S1477842415000408