Composing Constraint Automata, State-by-State

Sung-Shik T.Q. Jongmans[1] Tobias Kappé[2] Farhad Arbab[1,2]

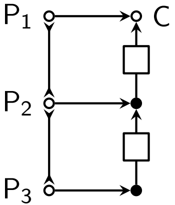
[1]Centrum Wiskunde & Informatica, Amsterdam

[2]Leiden Institute of Advanced Computer Science, Leiden

Formal Aspects of Component Software, 2015

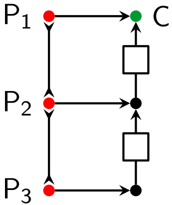
Reo

Reo is a *coordination language*, it facilitates coordinating synchronization and communication among components.



Reo

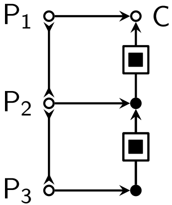
Reo is a *coordination language*, it facilitates coordinating synchronization and communication among components.



input / **output**

Reo

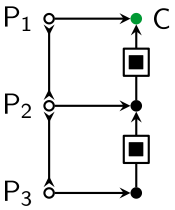
Reo is a *coordination language*, it facilitates coordinating synchronization and communication among components.



input / **output**

Reo

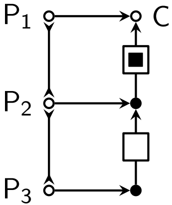
Reo is a *coordination language*, it facilitates coordinating synchronization and communication among components.



input / **output**

Reo

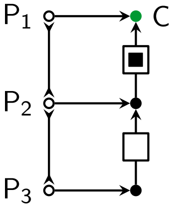
Reo is a *coordination language*, it facilitates coordinating synchronization and communication among components.



input / **output**

Reo

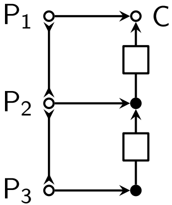
Reo is a *coordination language*, it facilitates coordinating synchronization and communication among components.



input / **output**

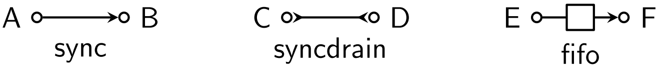
Reo

Reo is a *coordination language*, it facilitates coordinating synchronization and communication among components.



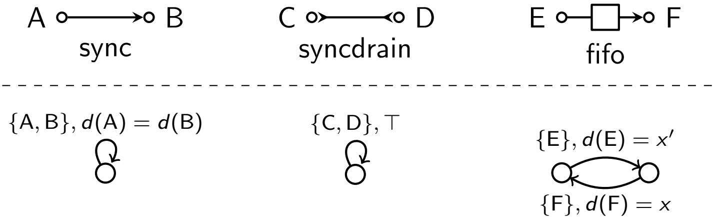
Constraint Automata

Constraint Automata provide a very useful semantics for Reo.



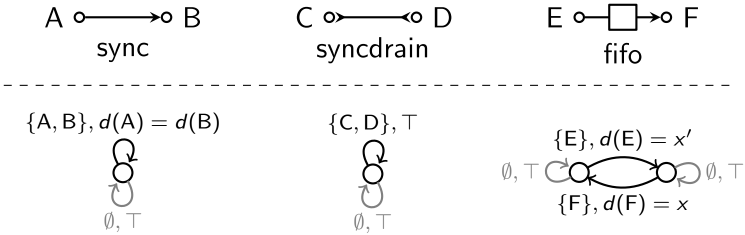
Constraint Automata

Constraint Automata provide a very useful semantics for Reo.



Constraint Automata

Constraint Automata provide a very useful semantics for Reo.



Constraint Automata

Let $q_1 \xrightarrow{P_1,\phi_1} q_1'$ and $q_2 \xrightarrow{P_2,\phi_2} q_2'$ be transitions of α_1 and α_2 respectively. If

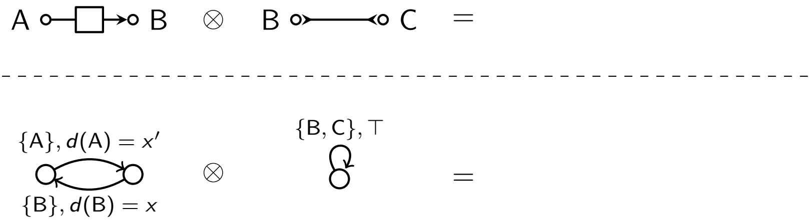
$$P_1 \cap \mathsf{Port}(\alpha_2) = P_2 \cap \mathsf{Port}(\alpha_1)$$

then $(q_1, q_2) \xrightarrow{P_1 \cup P_2, \phi_1 \wedge \phi_2} (q_1', q_2')$ is a transition of $\alpha_1 \otimes \alpha_2$.

Informally: *transitions can be composed if (and only if) they agree on common ports.*

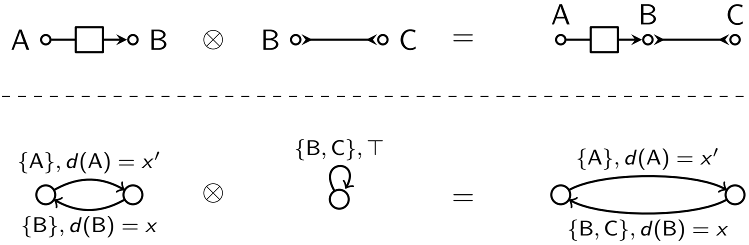
Constraint Automata

As an example, let's compute the following composition:



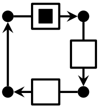
Constraint Automata

As an example, let's compute the following composition:



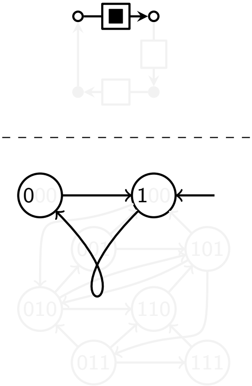
Problematic products

Computation of total semantics sometimes gives rise to exponential growth of intermediary products, such as seen here:



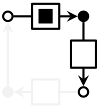
Problematic products

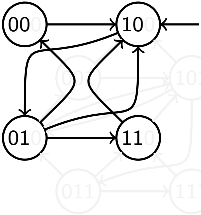
Computation of total semantics sometimes gives rise to exponential growth of intermediary products, such as seen here:



Problematic products

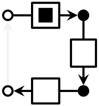
Computation of total semantics sometimes gives rise to exponential growth of intermediary products, such as seen here:

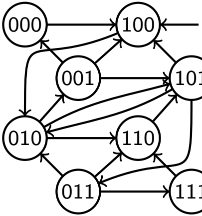




Problematic products

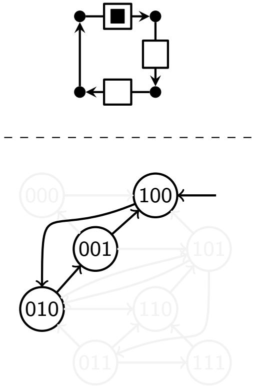
Computation of total semantics sometimes gives rise to exponential growth of intermediary products, such as seen here:





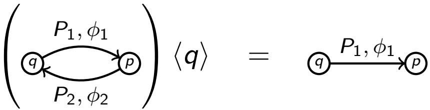
Problematic products

Computation of total semantics sometimes gives rise to exponential growth of intermediary products, such as seen here:



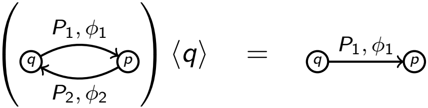
State-by-State Product

Let $\alpha\langle q\rangle$ denote the *state-based decomposition* of α with respect to q.

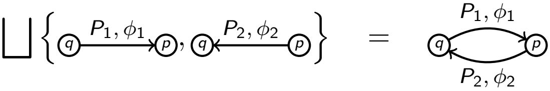


State-by-State Product

Let $\alpha\langle q\rangle$ denote the *state-based decomposition* of α with respect to q.

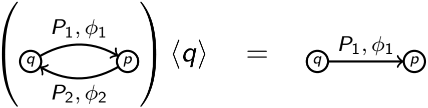


We write $\bigsqcup A$ for the *state-based recomposition* of Constraint Automata $\alpha \in A$.

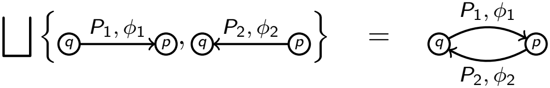


State-by-State Product

Let $\alpha\langle q\rangle$ denote the *state-based decomposition* of α with respect to q.



We write $\bigsqcup A$ for the *state-based recomposition* of Constraint Automata $\alpha \in A$.



A decomposed automaton can be recomposed to obtain the original:

$$\bigsqcup\{\alpha\langle q\rangle : q \in \mathsf{State}(\alpha)\} = \alpha$$

State-by-State Product

More importantly, we show that the product distributes over decomposition:

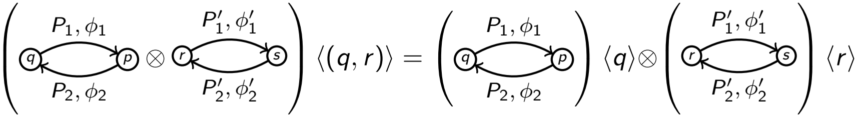
$$(\alpha_1 \otimes \cdots \otimes \alpha_n)\langle(q_1, \ldots, q_n)\rangle = \alpha_1\langle q_1\rangle \otimes \cdots \otimes \alpha_n\langle q_n\rangle$$

State-by-State Product

More importantly, we show that the product distributes over decomposition:

$$(\alpha_1 \otimes \cdots \otimes \alpha_n)\langle(q_1, \ldots, q_n)\rangle = \alpha_1\langle q_1\rangle \otimes \cdots \otimes \alpha_n\langle q_n\rangle$$

As in:

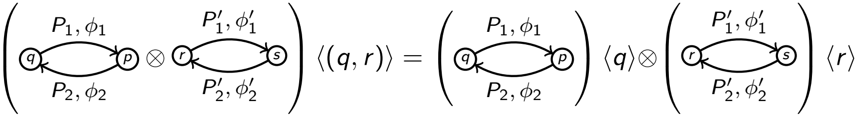


State-by-State Product

More importantly, we show that the product distributes over decomposition:

$$(\alpha_1 \otimes \cdots \otimes \alpha_n)\langle (q_1, \ldots, q_n) \rangle = \alpha_1\langle q_1 \rangle \otimes \cdots \otimes \alpha_n\langle q_n \rangle$$

As in:



This means that we can cheaply calculate part of the composition!

Idea: use this to calculate *only the reachable part* of the composition.

State-by-State Product

Algorithm "computes transitions that exit reachable states":

$A := \emptyset$
$A' :=$
$\{\alpha_1\langle q_1\rangle \otimes \cdots \otimes \alpha_n\langle q_n\rangle : (q_1, \ldots, q_n) \in \mathsf{Init}(\alpha_1) \times \cdots \times \mathsf{Init}(\alpha_n)\}$
while $\alpha \in A' \setminus A$ **for some** α **do**
    $A := A \cup \{\alpha\}$
    $A' :=$
$A' \cup \{\alpha_1\langle q_1'\rangle \otimes \cdots \otimes \alpha_n\langle q_n'\rangle : (q, P, \phi, (q_1', \ldots, q_n')) \in \mathsf{Tr}(\alpha)\}$
end while

We show that, after the loop, $\bigsqcup A = \lfloor \alpha_1 \otimes \cdots \otimes \alpha_n \rfloor$.

Implementation

We implemented the algorithm in Extensible Coordination Tools, a toolchain for Reo.

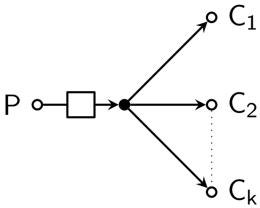
We then compared the composition speed of some circuits parameterized in k.

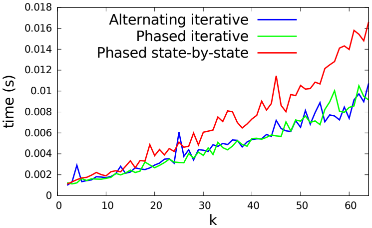
Three methods were compared:

- *Alternating iterative*: composition and abstraction alternated
- *Phased iterative*: abstraction occurs after composition
- *Phased state-by-state*: by our algorithm, abstraction occurs after composition

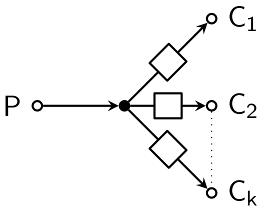
Implementation

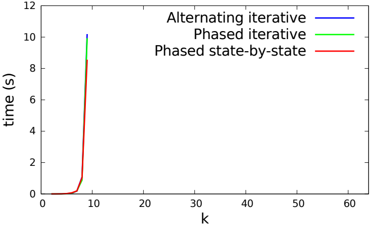
For some circuits, there was no big difference, because the state space is small . . .





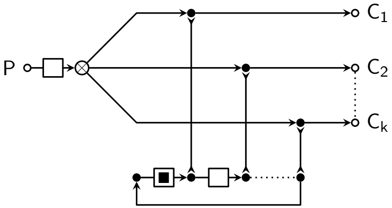
. . . or because the state space is large.

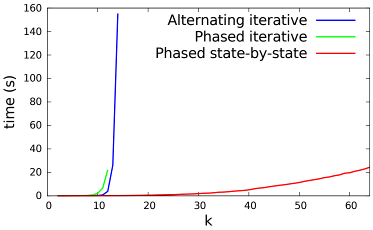




Implementation

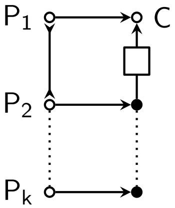
For others, the new algorithm was faster because of a small *reachable* state space.

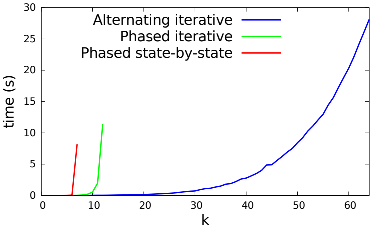




Implementation

And in one case, abstraction turned out to be important to limit the state space:





Conclusion

- New method makes composition of some circuits feasible.
- Method is applicable to other formalisms that use Constraint Automata.
- Some problematic cases remain, notably with relation to abstraction.
- Future research could look into a heuristic as to which method to use.