

# DCR Execution Equivalence

Søren Debois & Axel Christfort

February 6, 2026

## Abstract

We present an Isabelle formalization of the basics of DCR-graphs [1] before defining *Execution Equivalent* markings. We then prove that execution equivalent markings are perfectly interchangeable during process execution, yielding significant state-space reduction for execution-based model-checking of DCR graphs.

## Contents

<b>1</b>	<b>DCR processes</b>	<b>1</b>
1.1	Execution semantics	2
1.2	Execution Equivalence	3

```
theory DCRExecutionEquivalence
  imports Main
begin
```

## 1 DCR processes

Although we use the term "process", the present theory formalises DCR graphs as defined in the original places and other papers.

**type-synonym** *event* = *nat*

The static structure. This encompasses the relations, the set of event *dom* of the process, and the labelling function *lab*. We do not explicitly enforce that relations and marking are confined to this set, except in definitions of enabledness and execution below.

**record** *rels* =

```
cond :: event rel
pend :: event rel
incl :: event rel
excl :: event rel
```

*mist* :: event rel

*dom* :: event set

The dynamic structure, called the marking

**record** *marking* =

*Ex* :: event set

*In* :: event set

*Re* :: event set

It will be convenient to have notation for the events required, excluded, etc. by a given event.

**abbreviation** *conds* :: rels  $\Rightarrow$  event  $\Rightarrow$  event set

**where**

$conds\ T\ e \equiv \{ f . (f,e) \in cond\ T \}$

**abbreviation** *excls* :: rels  $\Rightarrow$  event  $\Rightarrow$  event set

**where**

$excls\ T\ e \equiv \{ x . (e,x) \in excl\ T \wedge (e,x) \notin incl\ T \}$

**abbreviation** *incls* :: rels  $\Rightarrow$  event  $\Rightarrow$  event set

**where**

$incls\ T\ e \equiv \{ x . (e,x) \in incl\ T \}$

**abbreviation** *resps* :: rels  $\Rightarrow$  event  $\Rightarrow$  event set

**where**

$resps\ T\ e \equiv \{ f . (e,f) \in pend\ T \}$

**abbreviation** *mists* :: rels  $\Rightarrow$  event  $\Rightarrow$  event set

**where**

$mists\ T\ e \equiv \{ f . (f,e) \in mist\ T \}$

Similarly, it is convenient to be able to identify directly the currently excluded events.

## 1.1 Execution semantics

**definition** *enabled* :: rels  $\Rightarrow$  marking  $\Rightarrow$  event  $\Rightarrow$  bool

**where**

$enabled\ T\ M\ e \equiv$

$e \in In\ M \wedge$

$(conds\ T\ e \cap In\ M) - Ex\ M = \{\}$   $\wedge$

$(mists\ T\ e \cap In\ M) - (dom\ T - Re\ M) = \{\}$

**definition** *execute* :: rels  $\Rightarrow$  marking  $\Rightarrow$  nat  $\Rightarrow$  marking

**where**

```

execute T M e ≡ (|
  Ex = Ex M ∪ { e },
  In = (In M - excls T e) ∪ incl T e,
  Re = (Re M - { e }) ∪ resps T e
|)

```

## 1.2 Execution Equivalence

**definition** *accepting* :: marking ⇒ bool **where**  
*accepting* M = (Re M ∩ In M = {})

**fun** *acceptingrun* :: rels ⇒ marking ⇒ event list ⇒ bool **where**  
*acceptingrun* T M [] = *accepting* M  
| *acceptingrun* T M (e#t) = (enabled T M e ∧ *acceptingrun* T (execute T M e) t)

**definition** *all-conds* :: rels ⇒ nat set **where**  
*all-conds* T = { fst rel | rel . rel ∈ cond T }

**definition** *execution-equivalent* :: rels ⇒ marking ⇒ marking ⇒ bool **where**  
*execution-equivalent* T M1 M2 = (  
 (In M1 = In M2) ∧  
 (Re M1 = Re M2) ∧  
 ((Ex M1 ∩ *all-conds* T) = (Ex M2 ∩ *all-conds* T))  
 )

**lemma** *conds-subset-eq-all-conds*: conds T e ⊆ *all-conds* T  
**using** *all-conds-def* **by** *auto*

**lemma** *ex-equiv-over-cond*: (Ex M1 ∩ *all-conds* T) = (Ex M2 ∩ *all-conds* T) ⇒  
(Ex M1 ∩ conds T e) = (Ex M2 ∩ conds T e)  
**using** *conds-subset-eq-all-conds* **by** *blast*

**lemma** *enabled-ex-equiv*:  
**assumes** *execution-equivalent* T M1 M2 *enabled* T M1 e  
**shows** *enabled* T M2 e

**proof** –

**from** *assms*(1) **have**  
(Ex M1 ∩ *all-conds* T) = (Ex M2 ∩ *all-conds* T)  
**by** (*simp add: execution-equivalent-def*)

**hence** *ex-eq*:  
(Ex M1 ∩ conds T e) = (Ex M2 ∩ conds T e)  
**using** *ex-equiv-over-cond* **by** *metis*

**from** *assms*(1) **have** *in-eq*:  
In M1 = In M2  
**by** (*simp add: execution-equivalent-def*)

**from** *assms*(2) **have**  
(conds T e ∩ In M1) ⊆ Ex M1  
**by**(*simp-all add: enabled-def*)

**hence**

$(conds\ T\ e \cap In\ M1) \cap (conds\ T\ e) \subseteq Ex\ M1 \cap (conds\ T\ e)$   
**by auto**  
**hence**  
 $(conds\ T\ e \cap In\ M1) \subseteq Ex\ M1 \cap (conds\ T\ e)$   
**by auto**  
**hence**  
 $(conds\ T\ e \cap In\ M2) \subseteq Ex\ M2 \cap (conds\ T\ e)$   
**using ex-eq in-eq by auto**  
**hence**  
 $(conds\ T\ e \cap In\ M2) \subseteq Ex\ M2$   
**by simp**  
**then show ?thesis**  
**using enabled-def assms in-eq execution-equivalent-def by auto**  
**qed**

**lemma execute-ex-equiv:**

**assumes** *execution-equivalent T M1 M2 execute T M1 e = M3 execute T M2 e*  
 $= M4$

**shows** *execution-equivalent T M3 M4*

**proof**–

**from assms have**

$In\ M3 = In\ M4$

**using execute-def execution-equivalent-def by fastforce**

**moreover from assms have**

$Re\ M3 = Re\ M4$

**using execute-def execution-equivalent-def by force**

**ultimately show ?thesis using assms execute-def execution-equivalent-def**  
**by fastforce**

**qed**

**lemma accepting-ex-equiv:** *execution-equivalent T M1 M2  $\implies$  accepting M1  $\implies$  accepting M2*

**by (simp add: accepting-def execution-equivalent-def)**

**theorem acceptingrun-ex-equiv:**

**assumes** *acceptingrun T M1 seq execution-equivalent T M1 M2*

**shows** *acceptingrun T M2 seq*

**using assms**

**proof**(*induction seq arbitrary: M1 M2 rule: acceptingrun.induct*)

**case** (1 *T M*)

**then show ?case**

**by (simp add: accepting-ex-equiv)**

**next**

**case** (2 *T M e t*)

**then show ?case proof**–

**from 2(2) obtain M1e where m1e:**

$M1e = execute\ T\ M1\ e$

**by blast**

**hence m1e-accept:**

```

    acceptingrun T M1e t
using  $\mathcal{Q}(2)$  acceptingrun.simps(2) by blast
obtain M2e where
    M2e = execute T M2 e
by blast
moreover from this m1e have
    execution-equivalent T M1e M2e
using  $\mathcal{Q}(3)$  execute-ex-equiv by blast
moreover from this have
    acceptingrun T M2e t
using  $\mathcal{Q}(1)$  m1e-accept by blast
ultimately show ?thesis using  $\mathcal{Q}(2)$  enabled-ex-equiv  $\mathcal{Q}(3)$  acceptingrun.simps(2)
by blast
qed
qed

end

```

## References

- [1] C. O. Back, T. Slaats, T. T. Hildebrandt, and M. Marquard. Discover: accurate and efficient discovery of declarative process models. *International Journal on Software Tools for Technology Transfer*, pages 1–25, 2021.