Open to Change: A Theory for Iterative Test-Driven Modelling*

Tijs Slaats[1], Søren Debois[2], and Thomas Hildebrandt[1]

[1] University of Copenhagen, Denmark
slaats@di.ku.dk, hilde@di.ku.dk
[2] IT University of Copenhagen, Denmark
debois@itu.dk

Abstract. We introduce *open tests* to support iterative test-driven process modelling. Open tests generalise the trace-based tests of Zugal et. al. to achieve *modularity*: whereas a trace-based test passes if a model exhibits a particular trace, an open test passes if a model exhibits a particular trace *up to* abstraction from additional activities not relevant for the test. This generalisation aligns open tests better with iterative test-driven development: open tests may survive the addition of activities and rules to the model in cases where trace-based tests do not. To reduce overhead in re-running tests, we establishing sufficient conditions for a model update to preserve test outcomes. We introduce open tests in an abstract setting that applies to any process notation with trace semantics, and give our main preservation result in this setting. Finally, we instantiate the general theory for the DCR Graph process notation, obtaining a method for iterative test-driven DCR process modelling.

Keywords: Test-driven Modelling, Abstraction, Declarative, DCR Graphs

1 Introduction

Test-driven development (TDD) [4,17] is a cornerstone of agile software development [8] approaches such as Extreme programming [3] and Scrum [24]. In TDD, tests drive the software development process. Before writing any code, developers gather and translate requirements to a set of representative tests. The software product is considered complete when it passes all tests.

In [27,28] Zugal et al. proposed applying the TDD approach to process modelling, introducing the concept of *test-driven modelling* (TDM). Like in TDD, the modeller in TDM first defines a set of test cases then uses these test cases to guide the construction of the model. A test case in this setting consists of a trace of activities expected to be accepted by the model.

Specifically, Zugal et al. proposed a *test-driven modelling methodology* where the model designer first constructs a set of process executions (traces) that will

* Work supported in part by the Innovation Fund project EcoKnow (7050-00034A); the first author additionally by the Danish Council for Independent Research project *Hybrid Business Process Management Technologies* project (DFF-6111-00337).

be used as test cases. The model designer then constructs the process model by repeatedly updating the model to make it satisfy more tests. Once all tests pass, the model is complete.

This methodology *benefits* the end-user by allowing him to focus on specific behaviours of the model that should be allowed in isolation, without having to immediately reason about all possible behaviours. By eventually arriving at a model where all tests pass, he is ensured that all desired behaviour is supported; and should a previously passing test fail after a model update, he knows that this update is wrong.

Process modelling notations can be roughly divided into two classes: declarative notations [21,22,14,10,16,19], which model *what* a process should do, e.g. as a formal description the rules governing the process; as opposed to imperative process models, which model *how* a process should proceed, e.g. as a flow between activities. Zugal et. al. argued [27,28] that TDM is particularly useful for the declarative paradigm, where understanding exactly which process executions the model allows and which it does not requires understanding potential non-trivial interplay of rules. In this setting, TDM is helpful to both constructing the model in a principled way (incrementally add declarative rules to satisfy more tests), as well as to recognize when the model is becoming over-constrained (when previously passing tests fail after a model extension). The commercial vendor DCR Solutions has implemented TDM in this sense in their commercial DCR process portal, `dcrgraphs.net`.

Unfortunately, TDM falls short of TDD in one crucial respect: its test cases are insufficiently modular and may cease to adequately model requirements as the model evolves. Consider a requirement "payout can only happen after manager approval", and suppose we have a model passing the test case:

$$\langle \mathsf{Approval}, \mathsf{Payout} \rangle \tag{1}$$

Now suppose that, following the iterative modelling approach, we refine the model to satisfy also the only tangentially related requirements that "approval requires a subsequent audit" and "payout cannot happen before an audit". That is, the model would have a trace:

$$\langle \mathsf{Approval}, \mathsf{Audit}, \mathsf{Payout} \rangle \tag{2}$$

Crucially, while the *requirement* "payout can only happen after manager approval" is still supported by the refined model, the *test case* (1) intended to express that requirement no longer passes.

In the present paper we propose *open tests*: A generalisation of the "test cases" of [27,28] that is more robust under evolution of the model. Open tests formally generalise [27,28]: An open test comprises a trace as well as a *context*, a set of activities relevant to that test. This context will always contain the activities of the trace, and will often (but not always) be the set of activities known when the test was defined. For example, if we generalise the test (1) to an open test with the same trace and context $\{\mathsf{Approval}, \mathsf{Payout}\}$, then the extended

model which has the trace (2) *passes* this open test, because, *when we ignore the irrelevant activity* Audit, the traces (1) and (2) are identical.

In practice, one will use open tests in a similar way as regular trace-based tests; the core TDM methodology does not change. The difference comes when the specification of a model changes, for example because of changes in real-world circumstances such as laws or business practices, or because one wants to add additional detail to the model. Using simple trace-based tests, one would need to update each individual test to make sure that they still accurately represent the desired behaviour of the system. With open tests one only needs to change those tests whose context contains activities that are directly affected by the changes. Because of the modularity introduced by open tests, any test not directly affected by the changes will continue to work as expected. This also means that open tests are much better suited for *regression testing*: it is possible to make small changes to a model and continue to rely on previously defined tests to ensure that unrelated parts continue to work as intended.

We define both positive and negative open tests. A positive open test passes iff *there exists a model trace whose projection to the context is identical to the test case*. A negative test passes iff *for all model traces, the projection to the context is different from the test case*. Note that positive open tests embody existential properties, and negative open tests universal ones.

We instantiate the approach of open tests for the Dynamic Condition Response (DCR) Graph process notation [14,10] and provide polynomial time methods for approximating which open tests will remain passing after a given model update. This theoretical result makes the approach practical: When the relevant activities are exactly those of the model, an open test is the same as a test of [27,28]. From there, we iteratively update the model verifying at each step in low-order polynomial time that the open tests remain passing.

Altogether, we provide the following contributions:

1. We give a theory of process model testing using open tests (Section 2). This theory is general enough that it is applicable to all process notations with trace semantics: it encompasses both declarative approaches such as DECLARE or DCR, and imperative approaches such as BPMN.
2. We give in this general theory sufficient conditions for ensuring preservation of tests (Proposition 16). This proposition is key to supporting iterative test-driven modelling: it explains how a test case can withstand model updates.
3. We apply the theory to DCR graphs, giving a sufficient condition ensuring preservation of tests across model updates (Theorems 30 respectively 33).

Related Work Test-driven modelling (TDM) was introduced by Zugal et al. in [27,28] as an application of test-driven development to declarative business processes. Their studies [27] indicate in particular the that simple sequential traces are helpful to domain experts in understanding the underlying declarative models. The present approach generalises that of [27,28]: We define and study preservation of tests across model updates, alleviating modularity concerns while preserving the core usability benefit of defining tests via traces.

Connections between refinement, testing-equivalence and model-checking was observed in [6]. But where we consider refinements guaranteeing preservation of the projected language, the connection in [6] uses that a refinement of a state based model (Büchi-automaton) satisfies the formula the state based model was derived from. Our approach (and that of [12]) has strong flavours of refinement. Indeed, the iterative development and abstract testing of system models in the present paper is related to the substantial body of work on abstraction and abstract interpretation, e.g., [1,7,9]. In particular, an open test can be seen as a test on an abstraction of the system under test, where only actions in the context of the test are visible. In this respect, the abstraction is given by string projection on free monoids. We leave for future work to study the ramifications of this relationship and the possibilities of exploiting it in employing more involved manipulations than basic extensions of the alphabet in the process of iterative development, such as, e.g., allowing splitting of actions.

The synergy between static analysis and model checking is also being investigated in the context of programming languages and software engineering [13]. In particular there have been proposals for using static analysis to determine test prioritisation [26,18] when the tests themselves are expensive to run. Our approach takes the novel perspective of analysing the adaptations to a model (or code), instead of analysing the current instance of the model.

2 Open Tests

In this section, we introduce open tests in the abstract setting of trace languages. The definitions and results of this section apply to any process notation that has a trace semantics, e.g., DECLARE [22], DCR graphs [14,10,12], or BPMN [20]. In the next section, we instantiate the general results of this section specifically for DCR graphs.

Definition 1 (Test case). *A* test case (c, Σ) *is a finite sequence* c *over a set of activities* Σ*. We write* $\mathsf{dom}(c) \subseteq \Sigma$ *for the set of activities in* c*, i.e., when* $c = \langle e_1 \ldots . e_n \rangle$ *we have* $\mathsf{dom}(c) = \{e_1, \ldots, e_n\}$*.*

As a running example, we iteratively develop a process for handling reimbursement claims. The process we eventually develop conforms to §42 of the Danish Consolidation Act on Social Services (Serviceloven) [5]. The process exists to provide a citizen compensation for lost wages in the unfortunate circumstances that he must reduce his working hours to care for a permanently disabled child.

Example 2. Consider the set of activities $\Sigma = \{\mathsf{Apply}, \mathsf{Document}, \mathsf{Receive}\}$, abbreviating respectively "Apply for compensation", "Document need for compensation", and "Receive compensation". We define a test case t_0:

$$t_0 = (\langle \mathsf{Apply}, \mathsf{Document}, \mathsf{Receive} \rangle, \Sigma) \tag{3}$$

Intuitively, this test case identifies all traces, over *any* alphabet, whose projection to Σ is exactly $\langle \mathsf{Apply}, \mathsf{Document}, \mathsf{Receive} \rangle$.

Open test cases come in one of two flavours: a *positive* test requires the presence of (a representation of) a trace, whereas a *negative* test requires the absence of (any representation of) a trace.

Definition 3 (Positive and negative open tests). *An* open test *comprises a test case* $t = (c, \Sigma)$ *and a* polarity ρ *in* $\{+, -\}$*, altogether written* t^+ *respectively* t^-*.*

Example 4. Extending our previous example, define a positive and negative open test as follows:

$$t_0^+ = (\langle \mathsf{Apply}, \mathsf{Document}, \mathsf{Receive} \rangle, \{\mathsf{Apply}, \mathsf{Document}, \mathsf{Receive}\}^+ \quad (4)$$

$$t_1^- = (\langle \mathsf{Receive} \rangle, \{\mathsf{Document}, \mathsf{Receive}\})^- \quad (5)$$

Intuitively, the positive test t_0^+ requires the presence of some trace that projects to exactly $\langle \mathsf{Apply}, \mathsf{Document}, \mathsf{Receive} \rangle$. The negative requires that no trace, when projected to $\{\mathsf{Document}, \mathsf{Receive}\}$, is exactly $\mathsf{Receive}$, that is, this test models the requirement that one may not $\mathsf{Receive}$ compensation without first providing $\mathsf{Document}$ation.

To formalise the semantics of open tests we need a system model representing the possible behaviours of the system under test. In general, we define a system as a set of sequences of activities, that is, a language.

Definition 5. *A* system $S = (L, \Sigma)$ *is a language* L *of finite sequences over a set of activities* Σ*.*

We can now define under what circumstances positive and negative open tests pass. First we introduce notation.

Notation Let ϵ denote the empty sequence of activities. Given a sequence s, write s_i for the ith element of s, and $s|_\Sigma$ defined inductively by $\epsilon|_\Sigma = \epsilon$, $(a.s)|_\Sigma = a.(s|_\Sigma)$ if $a \in \Sigma$ and $(a.s)|_\Sigma = s|_\Sigma$ if $a \notin \Sigma$. E.g, if $s = \langle \mathsf{Apply}, \mathsf{Document}, \mathsf{Receive} \rangle$ is the sequence of test t_0 above, then $s|_{\{\mathsf{Document}, \mathsf{Receive}\}} = \langle \mathsf{Document}, \mathsf{Receive} \rangle$ is the projection of that sequence. We lift projection to sets of sequences point-wise.

Definition 6 (Passing open tests). *Let* $S = (L, \Sigma')$ *be a system and* $t = (c, \Sigma)$ *a test case. We say that:*

1. S passes *the open test* t^+ *iff there exists* $c' \in L$ *such that* $c'|_\Sigma = c$*.*
2. S passes *the open test* t^- *iff for all* $c' \in L$ *we have* $c'|_\Sigma \neq c$*.*

S fails *an open test* t^ρ *iff it does not pass it.*

Notice how activities that are not in the context of the open test are ignored when determining if the system passes.

Example 7 (System S, Iteration 1). Consider a system $S = (L, \Sigma)$ with activities $\Sigma = \{\mathsf{Apply}, \mathsf{Document}, \mathsf{Receive}\}$ and as language L the subset of sequences of Σ^* such that the Receive is always preceded (not necessarily immediately) by Document, and Apply is always succeeded (again not necessarily immediately) by Receive.

Positive tests require existence of a trace that projects to the test case. This system S passes the test t_0^+ for $t_0 = (\langle \mathsf{Apply}, \mathsf{Document}, \mathsf{Receive} \rangle, \Sigma)$ as defined above, since the sequence $c' = \langle \mathsf{Apply}, \mathsf{Document}, \mathsf{Receive} \rangle$ in L has $c'|_\Sigma = \langle \mathsf{Apply}, \mathsf{Document}, \mathsf{Receive} \rangle$.

Negative tests require the absence of any trace that projects to the test case. S also passes the test t_1^- for $t_1 = (\mathsf{Receive}, \{\mathsf{Document}, \mathsf{Receive}\})$ since if there were a $c' \in L$ s.t. $c'|_{\{\mathsf{Document},\mathsf{Receive}\}} = \mathsf{Receive}$ that would contradict that Document should always appear before any occurrence of Receive.

Finally, consider the following positive test.

$$t_2^+ = (\mathsf{Apply}, \{\mathsf{Apply}, \mathsf{Receive}\})^+$$

The System S fails this test t_2^+, because every sequence in L that contains Apply will by definition also have a subsequent Receive, which would then appear in the projection.

We note that a test either passes or fails for a particular system, never both; and that positive and negative tests are dual: t^+ passes iff t^- fails and vice versa.

Lemma 8. *Let $S = (L, \Sigma)$ be a system and t a test case. Then either (a) S passes t^+ and fails t^-; or (b) S fails t^+ and S passes t^-.*

Example 9 (Iteration 2, Test preservation). We extend our model of Example 7 with the additional requirement that some documentation of the salary reduction is required before compensation may be received. To this end, we refine our system (L, Σ) to a system $S' = (L', \Sigma' = \Sigma \cup \{\mathsf{Reduction}\})$ where Reduction abbreviates "Provide documentation of salary reduction", and L' is the language over Σ'^* that satisfies the original rules of Example 7 and in addition that Receive is always preceded by Reduction.

The explicit context ensures that the tests t_0^+, t_1^-, t_2^+ defined in the previous iteration remain meaningful. The system S' no longer has a trace

$$\langle \mathsf{Apply}, \mathsf{Document}, \mathsf{Receive} \rangle$$

because Reduction is missing. Nonetheless, S' *still* passes the test t_0^+, because S' *does* have the trace:

$$c' = \langle \mathsf{Apply}, \mathsf{Document}, \mathsf{Reduction}, \mathsf{Receive} \rangle \in L'$$

The projection $c'|_\Sigma = \langle \mathsf{Apply}, \mathsf{Document}, \mathsf{Receive} \rangle$ then shows that t_0^+ passes S'.

Similarly, S' still passes the test t_1^- since for any $c' \in L'$, if $c'|_\Sigma = \langle \mathsf{Receive} \rangle$ then $c' = \langle c_0, \mathsf{Receive}, c_1 \rangle$ for some $c_0, c_1 \in \Sigma' \backslash \Sigma$, but that contradicts the requirement that Document must appear before any occurrence of Receive.

We now demonstrate how open tests may be preserved by model extensions where the trace-based tests of Zugal et. al. [27,28] would not be.

Example 10 (Non-preservation of non-open tests). We emphasize that if we interpret the trace $s = \langle \mathsf{Apply}, \mathsf{Document}, \mathsf{Receive} \rangle$ underlying the test t_0^+ as a test in the sense of [27,28], that test is *not* preserved when we extend the system from (L, Σ) to (L', Σ'): The original system L has the behaviour s, but the extension L' does not.

Example 11 (Iteration 2, Additional tests). We add the following additional tests for the new requirements of Iteration 2.

$$t_3^- = (\langle \mathsf{Apply}, \mathsf{Document}, \mathsf{Receive} \rangle, \Sigma')^-$$

Note that the trace of t_3 is *the same* as the original test t_0; the two tests differ only in their context. This new test says that in a context where we know about the Reduction activity, omitting it is not allowed.

In these particular examples, the tests that passed/failed in the first iteration also passed/failed in the second. This is not generally the case; we give an example.

Example 12 (Iteration 3). The reduction in salary may be rejected, e.g. if the submitted documentation is somehow unsatisfactory. In this case, compensation must be withheld until new documentation is provided. We model this by adding an activity Rejection to the set of activities Σ' and constrain the language L' accordingly. Now the system will pass the test $t_2^+ = (\mathsf{Apply}, \{\mathsf{Apply}, \mathsf{Receive}\})^+$ defined above, because it has a trace that contains Apply but no Receive: the sequence in which the documentation of reduced salary is rejected.

We now turn to the question how to "run" an open test. Unlike the tests of Zugal et. al., running an open test entails more than simply checking language membership. For positive tests we must find a trace of the system with a suitable projection, and for negative tests we must check that no trace has the test trace as projection.

First, we note that if the context of the open test contains all the activities of the model under test, it is in fact enough to simply check language membership.

Lemma 13. *Let $S = (L, \Sigma)$ be a system, let $t = (c, \Sigma')$ be a test case, and suppose $\Sigma \subseteq \Sigma'$. Then: 1. t^+ passes iff $c \in L$. 2. t^- passes iff $c \notin L$.*

Second, we show how checking whether an open test passes or fails in the general case reduces to the language inclusion problem for regular languages.

Proposition 14. *Let $S = (L, \Sigma)$ be a system, let $t = (\langle c_1, \ldots, c_n \rangle, \Sigma')$ be a test case. Define the set of irrelevant activities $I = \Sigma \setminus \Sigma'$ as those activities in the system but not in the test case. Assume wlog $I = \{i_1, \ldots, i_m\}$, and let $\mathsf{ri} = (i_1 | \cdots | i_m)^*$ be the regular expression that matches zero or more irrelevant activities. Finally, define the regular expression $\mathsf{rc} = \mathsf{ri}\ c_1\ \mathsf{ri}\ \cdots\ \mathsf{ri}\ c_n\ \mathsf{ri}$. Then:*

1. *S passes the positive test t^+ iff* $\mathsf{lang}(\mathsf{rc}) \cap L \neq \emptyset$, *and*
2. *S passes the negative test t^- iff* $\mathsf{lang}(\mathsf{rc}) \cap L = \emptyset$.

Example 15. Consider again S' of Example 9, and the test case t_0^+ of Example 2:

$$S' = (L', \Sigma' = \{\mathsf{Apply}, \mathsf{Document}, \mathsf{Receive}, \mathsf{Reduction}\})$$
$$t_0^+ = (\langle \mathsf{Apply}, \mathsf{Document}, \mathsf{Receive} \rangle, \{\mathsf{Apply}, \mathsf{Document}, \mathsf{Receive}\}^+$$

In the notation of Proposition 14, we have $I = \{\mathsf{Reduction}\}$, $\mathsf{ri} = \mathsf{Reduction}^*$ and by that Theorem, t_0^+ passes the system S' because S' has non-empty intersection with the language defined by the regular expression:

$$\mathsf{Reduction}^* \ \mathsf{Apply} \ \mathsf{Reduction}^* \ \mathsf{Document} \ \mathsf{Reduction}^* \ \mathsf{Receive} \ \mathsf{Reduction}^*$$

Lemma 13 and Proposition 14 explain how to "run" open tests, they apply directly for any process notation with trace semantics. Language inclusion for regular languages is extremely well-studied: practical methods exist for computing such intersections from both the model-checking and automata-theory communities. For certain models these methods may be sufficient; for example in the case of BPMN where models tend to be fairly strict and allow little behaviour, it could be feasible to always rely on model checking. However, for models that represent large state spaces, which is particularly common for declarative notations, this will not suffice and we will need to reduce the amount of model checking required.

The key insight of open tests is that oftentimes, changes to a model will preserve open test outcomes, obviating the need to re-check tests after the change. The following Proposition gives general conditions for when outcomes of positive (resp. negative) tests for a system S are preserved when the system is changed to a new system S'.

Proposition 16. *Let $S = (L, \Sigma)$ and $S' = (L', \Sigma')$ be systems, and let $t = (c_t, \Sigma_t)$ be a test case. Assume that $\Sigma' \cap \Sigma_t \subseteq \Sigma' \cap \Sigma$. Then:*

1. *If $L'|_\Sigma \supseteq L$ and S passes t^+, then so does S'.*
2. *If $L'|_\Sigma \subseteq L$ and S passes t^-, then so does S'.*

In words, the assumption $\Sigma' \cap \Sigma_t \subseteq \Sigma' \cap \Sigma$ states that the changed system S' does not introduce activities appearing in the context of the test that did not already appear in the original system S. Condition 1 (resp. 2) expresses that positive (resp. negative) tests are preserved if the language of the original system S is included in (resp. including) the language of the changed system S' projected to the activities in the original system. Now, if one can find static properties of changes to process models for a particular notation that implies the conditions of Proposition 16 then these properties can be checked instead of relying on model-checking to infer preservation of tests. We identify such static properties for the Dynamic Condition Response (DCR) Graphs [14,10,12] process notation in Section 4. First however, we recall the syntax and semantics of DCR graphs in the next section.

3 Dynamic Condition Response Graphs

DCR Graphs is a declarative notation for modelling processes superficially similar to DECLARE [21,22] or temporal logics such as LTL [23] in that it allows for the declaration of a set of temporal constraints between activities.

One notable difference is that DCR graphs model also the run-time state of the process using a so-called marking of activities. The marking consists of three finite sets (Ex,In,Re) recording respectively which activities have been executed (Ex), which are currently included (In) in the graph, and which are required (Re) to be executed again in order for the graph to be accepting, also referred to as the pending events. The marking allows for providing semantics of DCR Graphs by defining when an activity is enabled in a marking and how the execution of an enabled activity updates the marking. Formally, DCR graphs are defined as follows.

Definition 17 (DCR Graph [14][3]). *A* DCR graph *is a tuple* $(\mathsf{E}, \mathsf{R}, \mathsf{M})$ *where*

- E *is a finite set of* activities*, the nodes of the graph.*
- R *is the edges of the graph. Edges are partitioned into five kinds, named and drawn as follows: The* conditions $(\rightarrow\bullet)$*,* responses $(\bullet\rightarrow)$*,* inclusions $(\rightarrow+)$*,* exclusions $(\rightarrow\%)$ *and* milestones $\rightarrow\diamond$*.*
- M *is the* marking *of the graph. This is a triple* $(\mathsf{Ex}, \mathsf{Re}, \mathsf{In})$ *of sets of activities, respectively the previously executed* (Ex)*, the currently pending* (Re)*, and the currently included* (In) *activities.*

Next we recall from [14] the definition of when an activity is enabled.

Notation. When G is a DCR graph, we write, e.g., $\mathsf{E}(G)$ for the set of activities of G, $\mathsf{Ex}(G)$ for the executed activities in the marking of G, etc. In particular, we write $\mathsf{M}(e)$ for the triple of boolean values $(e \in \mathsf{Ex}, e \in \mathsf{Re}, e \in \mathsf{In})$. We write $(\rightarrow\bullet\, e)$ for the set $\{e' \in \mathsf{E} \mid e' \rightarrow\bullet\ e\}$, write $(e\bullet\rightarrow)$ for the set $\{e' \in \mathsf{E} \mid e \bullet\rightarrow e'\}$ and similarly for $(e\rightarrow+)$, $(e\rightarrow\%)$ and $(\rightarrow\diamond e)$.

Definition 18 (Enabled activities [14]). *Let* $G = (\mathsf{E}, \mathsf{R}, \mathsf{M})$ *be a DCR graph, with marking* $\mathsf{M} = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In})$*. An activity* $e \in \mathsf{E}$ *is* enabled*, written* $e \in \mathsf{enabled}(G)$*, iff (a)* $e \in \mathsf{In}$ *and (b)* $\mathsf{In} \cap (\rightarrow\bullet e) \subseteq \mathsf{Ex}$ *and (c)* $(\mathsf{Re} \cap \mathsf{In}) \cap (\rightarrow\diamond e) = \emptyset$*.*

That is, enabled activities (a) are included, (b) their included conditions have already been executed, and (c) have no pending included milestones.

Executing an enabled activity e of a DCR Graph with marking $(\mathsf{Ex}, \mathsf{Re}, \mathsf{In})$ results in a new marking where (a) the activity e is added to the set of executed activities, (b) e is removed from the set of pending response activities, (c) the responses of e are added to the pending responses, (d) the activities excluded by

[3] In [14] DCR graphs model constraints between so-called events labelled by activities. To simplify the presentation, we assume in the present paper that each event is labelled by a unique activity and therefore speak only of activities.

e are removed from included activities, and (e) the activities included by e are added to the included activities.

From this we can define the language of a DCR Graph as all finite sequences of activities ending in a marking with no activity both included and pending.

Definition 19 (Language of a DCR Graph [14][4]). *Let* $G_0 = (\mathsf{E}, \mathsf{R}, \mathsf{M})$ *be a DCR graph with marking* $\mathsf{M}_0 = (\mathsf{Ex}_0, \mathsf{Re}_0, \mathsf{In}_0)$. *A* trace *of* G_0 *of length* n *is a finite sequence of activities* $e_0, \ldots, e_{n-1}$ *such that for* $0 \leq i < n$, *(i)* e_i *is enabled in the marking* $M_i = (\mathsf{Ex}_i, \mathsf{Re}_i, \mathsf{In}_i)$ *of* G_i, *and (ii)* G_{i+1} *is a DCR Graph with the same activities and relations as* G_i *but with marking* $(\mathsf{Ex}_{i+1}, \mathsf{Re}_{i+1}, \mathsf{In}_{i+1}) = (\mathsf{Ex}_i \cup \{e_i\}, (\mathsf{Re}_i \backslash \{e_i\}) \cup (e_i \bullet\!\!\rightarrow), (\mathsf{In}_i \backslash (e_i \rightarrow\!\!\%)) \cup (e_i \rightarrow\!\!+))$.

We call a trace of length n accepting *if* $\mathsf{Re}_n \cap \mathsf{In}_n = \emptyset$. *The* language $\mathsf{lang}(G_0)$ *of* G_0 *is then the set of all such accepting traces. Write* $\hat{G}$ *for the corresponding system* $\hat{G} = (\mathsf{lang}(G), \mathsf{E})$ *(viz. Definition 5). When no confusion is possible, we denote by simply* G *both a DCR graph and its corresponding system* $\hat{G}$.

Example 20 (DCR Iteration 1). We model the §42 process of Example 2 in Figure 1a as a DCR graph. This model is simple enough that it uses only the response and condition relations which (in this case) behave the same as the response and precedence constraints in DECLARE. The condition from Document to Receive models the requirement that documentation must be provided before compensation may be received. The response from Apply to Receive models that compensation must eventually be received after it has been applied for. The marking of the graph is $(\emptyset, \emptyset, \{\mathsf{Document}, \mathsf{Receive}\})$, i.e. no activities have been executed, no activities are yet pending responses and all activities are included. (The activity Receive is grey to indicate that it is not enabled).

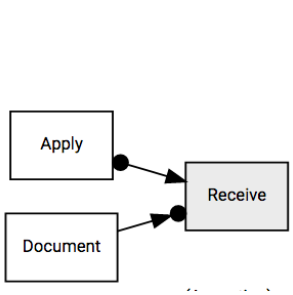
Example 21 (DCR Iteration 2). Following Example 9, we extend the iteration 1 model of Figure 1a to the iteration 2 model in Figure 1b. We model the new requirement that documentation must be provided before compensation may be received by adding a new activity Reduction and a condition relation from Reduction to Receive.

Example 22 (DCR Iteration 3). Following Example 12, we extend the iteration 2 model of Figure 1b to the iteration 3 model in Figure 1c. To model the rejection of documentation, we add the activity Rejection and exclude-relations between Rejection and Receive. This models the choice between those two activities: Once one is executed, the other is excluded and no longer present in the model. To model subsequent re-submission, we add an include relation from Reduction to Rejection and Receive: if new documentation of salary is received, the activities Rejection and Receive become included once again, re-enabling the decision whether to reject or pay.

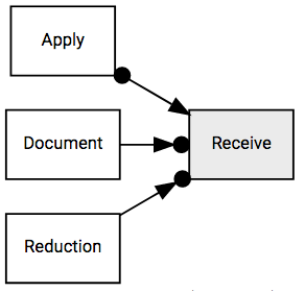
Example 23 (DCR Iteration 3, variant). To illustrate the milestone relation, we show an alternative to the model of Figure 1c in Figure 1d. Using a response

[4] In [14] the language of a DCR graph consists of both finite and infinite sequences. To simplify the presentation, we consider only finite sequences in the present paper.

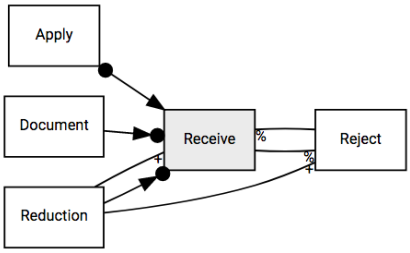
relation from Receive to Reduction we model that after rejection, documentation must be resubmitted; and by adding a milestone from Reduction to Receive we model that compensation may not received again while we are waiting for this new documentation.



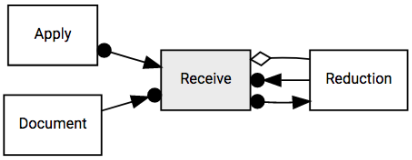
(a) I_1: Iteration 1



(b) I_2: Iteration 2



(c) I_3: Iteration 3



(d) I_3': Variant iteration 3

Fig. 1: DCR Graph models of the §42 of Examples 2–12.

4 Iterative Test Driven Development for DCR Graphs

In this Section, we show how Proposition 16 applies to DCR graphs, and exemplify how the resulting theory supports iterative test-driven DCR model development by telling us which tests are preserved by model updates. We consider the situation that a graph G' extends a graph G by adding activities and relations. Recall the notation $\mathsf{M}(e) = (e \in \mathsf{Ex}, e \in \mathsf{Re}, e \in \mathsf{In})$.

Definition 24 (Extensions). *Let* $G = (\mathsf{E}, \mathsf{R}, \mathsf{M})$ *and* $G' = (\mathsf{E}', \mathsf{R}', \mathsf{M}')$ *be DCR graphs. We say that* G' statically extends G *and write* $G \sqsubseteq G'$ *iff* $\mathsf{E} \subseteq \mathsf{E}'$ *and* $\mathsf{R} \subseteq \mathsf{R}'$. *If also* $e \in \mathsf{E}$ *implies* $\mathsf{M}(e) = \mathsf{M}(e')$, *we say that* G' dynamically extends G *and write* $G \preceq G'$.

Our main analysis technique will be the application of Proposition 16. To this end, we need ways to establish the preconditions of that Theorem, that is:

$$\mathsf{lang}(G')|_{\mathsf{E}} \supseteq \mathsf{lang}(G) \qquad (\dagger)$$

$$\mathsf{lang}(G')|_{\mathsf{E}} \subseteq \mathsf{lang}(G) \qquad (\ddagger)$$

Example 25. Consider the graphs I_1 of Figure 1a and I_2 of Figure 1b. Clearly $I_1 \sqsubseteq I_2$ since I_2 contains all the activities and relations of I_1. Moreover, since the markings of I_1 and I_2 agree, also $I_1 \preceq I_2$. Similarly, $I_2 \sqsubseteq I_3$ and $I_2 \sqsubseteq I'_3$, where I_3 and I'_3 are the graphs of Figures 1c and 1d. On the other hand, neither $I_3 \sqsubseteq I'_3$, since the former graph has activities not in the latter, nor $I'_3 \sqsubseteq I_3$, since the former graph has relations (e.g., the milestone) not in the latter.

We note that DCR activity execution preserves static extensions, i.e. if $G \sqsubseteq G'$ and an activity e is enabled in both G and G' then $G_1 \sqsubseteq G'_1$, if G_1 and G'_1 are the results of executing e in G and G' respectively. Dynamic extension is generally *not* preserved by execution, because an execution might make markings between the original and extended graph differ *on the original activities*, e.g., if G' adds an exclusion, inclusion or response constraint between activities of E.

4.1 Positive tests

We first establish a syntactic condition for a modification of a DCR graph to preserve positive tests. The condition will be, roughly, that the only new relations are either (a) between new activities, or (b) conditions or milestones from new to old activities. For the latter, we will need to be sure we can find a way to execute enough new activities to satisfy such conditions and milestones. To this end, we introduce the notion of dependency graph, inspired by [2].

Definition 26 (Dependency graph). *Let $G = (\mathsf{E}, \mathsf{R}, \mathsf{M})$ be a DCR graph, and let $e, f \in \mathsf{E}$ be activities of G. Write $e \to f$ whenever $e \to\bullet f \in \mathsf{R}$ or $e \to\diamond f \in \mathsf{R}$. The* dependency graph *$D(G, e)$ for e is the directed subgraph of G which has nodes $\{f \mid f \to^* e\}$ and an edge from node f to node f' iff $f \to f'$.*

With the notion of dependency graph, we can define the notion of "safe" activities, intuitively those that can be relied upon to be executed without having undue side effects on a given (other) set of nodes X. The principle underlying this definition is inspired by the notion of dependable activity from [2].

Definition 27 (Safety). *Let $G = (\mathsf{E}, \mathsf{R}, \mathsf{M})$ be a DCR graph, let $e \in \mathsf{E}$ be an activity of G, and let $X \subseteq \mathsf{E}$ be a subset of the activities of G. We say that e is* safe *for X iff*

1. *$D(G, e)$ is acyclic,*
2. *no $f \in D(G, e)$ has an include, exclude, or response relation to any $x \in X$.*
3. *for any $f \in D(G, e)$, if f has a condition or milestone to some $f' \in \mathsf{E}$, then f' is reachable from f in $D(G, e)$.*

The notion of safe activity really captures activities that can reliably be discharged if they are conditions or milestones for other activities. We use this to define a notion of transparent process extensions: a process extension which we shall see preserves positive tests.

Definition 28 (Transparent). *Let* $G = (\mathsf{E}, \mathsf{R}, \mathsf{M})$ *and* $G' = (\mathsf{E}', \mathsf{R}', \mathsf{M}')$ *be DCR graphs with* $G \sqsubseteq G'$*. We say that* G' *is* transparent *for* G *iff for all* $e, f \in \mathsf{E}$ *and* $e', f' \in \mathsf{E}'$ *we have:*

1. *if* $e'\mathcal{R}f' \in \mathsf{R}'$ *for* $\mathcal{R} \in \{\rightarrow\bullet, \rightarrow\diamond\}$ *then* either $e'\mathcal{R}f' \in \mathsf{R}$ or *(a)* $e' \notin \mathsf{E}$*, (b)* e' *is safe for* E*, and (c)* $\mathsf{E}(D(G', e')) \subseteq \mathsf{E}' \setminus \mathsf{E}$*,*
2. *for* $\mathcal{R} \in \{\rightarrow+, \rightarrow\%, \bullet\rightarrow\}$ *we have* $e\mathcal{R}f \in \mathsf{R}'$ *iff* $e\mathcal{R}f \in \mathsf{R}$*.*
3. *for* $\mathcal{R} \in \{\rightarrow+, \bullet\rightarrow\}$ *we have if* $e\mathcal{R}e' \in \mathsf{R}'$ *or* $e' \in \mathsf{Re}(G')$ *then* $e' \in \mathsf{E}$

We rephrase these conditions more intuitively. Call an activity $e \in \mathsf{E}$ an *old* activity, and an activity $e' \in \mathsf{E}' \setminus \mathsf{E}$ a *new* activity. The first item then says that we can never add conditions or milestones from old activities and only add a condition or milestone to an old activity when the new activity is safe, that is, we can rely on being able to discharge that milestone or condition. The second item says that we cannot add exclusions, inclusions or responses between old activities. The third says that we also cannot add inclusions or responses from old to new activities, or add a new activity which is initially pending in the marking, which could cause the new graph to be less accepting than the old. Inclusions, exclusions and responses may be added from a new to an old activity; the interplay of condition 1 of Definition 28 and condition 2 of Definition 27 then implies that this can only happen if the new activity is not in the dependency graph of any old activity. The reason is, that such constraints can be *vacuously* satisfied since the new activity at the source of the constraint is *irrelevant* with respect to passing any of the positive tests.

Example 29. It is instructive to see how violations of transparency may lead to non-preservation of positive tests. Consider a DCR graph with activities A, B and no relations. Note that this graph passes the open test $t^+ = (\langle A, B\rangle, \{A, B\})^+$. Consider two possible updates. (i) Adding a relation $B \rightarrow\bullet\ A$ (a condition between old activities) would stop the test from passing. So would adding an activity C and relations $B \rightarrow\bullet\ C \rightarrow\bullet\ A$. (ii) Adding a relation $B \bullet\rightarrow A$ causes t to end in a non-accepting state, stopping the test t from passing.

Theorem 30. *Let* $G \preceq G'$ *with* G' *transparent for* G*, and let* $t^+ = (c_t, \Sigma_t)^+$ *be a positive test with* $\Sigma_t \subseteq \mathsf{E}$*. If* G *passes* t *then so does* G'*.*

Example 31 (Preservation). Consider the change from the graph I_1 of Figure 1a to the graph I_2 of Figure 1b: We have added the activity Reduction and the condition Reduction $\rightarrow\bullet$ Receive. In this case, I_2 is transparent for I_1: The new activity Reduction satisfies Definition 28 part (1c): even though a new condition dependency is added for Receive, the dependency graph for the new Receive remains acyclic. By Theorem 30, it follows that *any* positive test whose context is contained in {Apply, Document, Receive} will pass I_2 if it passes I_1. In particular, we saw in Example 7 that I_1 passes the test t_0^+, so necessarily also I_2 passes t_0^+.

4.2 Negative tests

For negative tests we must establish the inclusion (‡). This inclusion was investigated previously in [11,12], with the aim of establishing more general refinement

of DCR graphs. Definition 24 is a special case of refinement by merging, investigated in the above papers. Hence, we use the sufficient condition for such a merge to be a refinement from [12] to establish a sufficient condition, *exclusion-safety* for an extension to preserves negative tests.

Definition 32 (Exclusion-safe). *Suppose* $G = (\mathsf{E}, \mathsf{R}, \mathsf{M})$ *and* $G' = (\mathsf{E}', \mathsf{R}', \mathsf{M}')$ *are DCR graphs and that* G' *dynamically extends* G*. We say that* G' *is* exclusion-safe *for* G *iff for all* $e \in \mathsf{E}$ *and* $e' \in \mathsf{E}'$ *we have that:*

1. *if* $e' \rightarrow\% \; e \in \mathsf{R}'$ *then* $e' \rightarrow\% \; e \in \mathsf{R}$.
2. *if* $e' \rightarrow+ \; e \in \mathsf{R}'$ *then* $e' \rightarrow+ \; e \in \mathsf{R}$.

Using [11, Theorem 4.10], we arrive at the following Theorem.

Theorem 33. *Suppose* $G \preceq G'$ *are DCR graphs with* G' *exclusion-safe for* G*, and suppose* $t^- = (c_t, \Sigma_t)^-$ *is a negative test with with* $\Sigma_t \subseteq E$*. If* G *passes* t *then so does* G'*.*

Example 34 (Application). Consider again the change from I_1 to I_2 in Figure 1a and 1b. Since neither contains inclusions or exclusions, clearly I_2 is exclusion-safe for I_1. By Theorem 33 it follows that any negative test whose context is contained in $\{\mathsf{Apply}, \mathsf{Document}, \mathsf{Receive}\}$ which passes I_1 will also pass I_2. In particular, the negative test $t_1^- = (\langle \mathsf{Receive} \rangle, \{\mathsf{Document}, \mathsf{Receive}\})^-$ of Example 7 passes I_1, so by Theorem 33 it passes also I_2. Similarly but less obviously, any negative test with context included in $\{\mathsf{Apply}, \mathsf{Document}, \mathsf{Reduction}, \mathsf{Receive}\}$ which passes I_2 must also pass I_3' (Figure 1d).

Example 35 (Non-application). The changes two from I_1 to I_2 and from I_2 to I_3 (Figures 1a, 1b and 1c), where amongst other changes we have added an activity $\mathsf{Rejection}$ and a relation $\mathsf{Rejection} \rightarrow\% \; \mathsf{Receive}$, both violate exclusion-safety.

In this case, we can find a negative test that passes I_1 and I_2 but not I_3: $t_2^- = \langle \mathsf{Apply} \rangle, \{\mathsf{Apply}, \mathsf{Receive}\}^-$. It passes both I_1 and I_2, because in both of these, Apply leaves $\mathsf{Receive}$ pending, whence one needs to execute also $\mathsf{Receive}$ to get a trace of the process. But in I_3, we can use $\mathsf{Rejection}$ to exclude the pending $\mathsf{Receive}$. So I_3 has a trace $\langle \mathsf{Apply}, \mathsf{Rejection} \rangle$, and the projection of this trace to the context $\{\mathsf{Apply}, \mathsf{Receive}\}$ of our test is the string $\langle \mathsf{Apply} \rangle$: The test fails in I_3.

4.3 Practical use

To perform iterative test-driven development for DCR graphs using open tests, we proceed as follows. When tests are defined, we normally include all activities of the model under test in the context, and will be able to run them as standard tests, cf. Lemma 13. As we update the model, we verify at each step that the update preserves existing tests using Theorem 30 or 33. Should a model update fail to satisfy the prerequisites for the relevant Theorem, we "re-run" tests using model-checking techniques such as Proposition 14. We refer the reader to [25,15] for details on model-checking for DCR Graphs.

The prerequisites of both Theorem 30 and 33 are effectively computable.

Theorem 36. *Let $G \preceq G'$ be DCR graphs. It is decidable in time polynomial in the maximum size of G, G' whether (1) G' is exclusion-safe for G and (2) G' is transparent for G.*

5 Conclusion and Discussion

We introduced a general theory for testing abstractions of process models based on a notion of open tests, which extend the test-driven modelling methodology of [27,28]. In particular, we gave sufficient conditions for ensuring preservation of open tests across model updates. We applied the theory to the concrete declarative notation of DCR graphs and gave sufficient static transparency conditions on the updates of a DCR graph, ensuring preservation of open tests.

While the general theory applies to any process notation with trace semantics, the static conditions for transparency will need to be defined for the particular process notation at hand. Consider for example DECLARE [22]. The monotonicity of the semantics implies that adding more constraints will only remove traces from the language, and removing constraints will only add traces to the language. It is thus straightforward to prove that, if the set of activities is not changed, then adding respectively removing a constraint will satisfy part (1) respectively (2) of Proposition 16 and consequently positive respectively negative tests will not need to be re-checked. Further static conditions can be obtained by considering the constraints individually. Here, one source of complexity is the fact that DECLARE allows constraints that implicitly quantify over all possible activities. For instance, if a DECLARE model has the chain succession relation between two activities A and B, then A and B always happen together in the exact sequence $A.B$ with no other activities in-between. Now, if the model is extended by adding condition constraints from A to a new activity C and from C to B, then the test $A.B$ will fail even when considered in the open context $\{A, B\}$. Another source of complexity is simply that DECLARE allows many more constraints than DCR. We leave for future work to further investigate sufficient conditions for transparency for DECLARE.

Acknowledgements: We are grateful to the reviewers for their help not only to improve the presentation but also to identify interesting areas of future work.

References

1. Baeten, J.C., van Glabbeek, R.J.: Another look at abstraction in process algebra. In: International Colloquium on Automata, Languages, and Programming. pp. 84–94. Springer (1987)
2. Basin, D.A., Debois, S., Hildebrandt, T.T.: In the Nick of Time: Proactive Prevention of Obligation Violations. In: Computer Security Foundations. pp. 120–134 (2016)
3. Beck, K.: Extreme programming explained: embrace change. addison-wesley professional (2000)
4. Beck, K.: Test-driven development: by example (2003)

5. Bekendtgørelse af lov om social service (Aug 2017), Børne- og Socialministeriet
6. Bushnell, D.M.: Research Conducted at the Institute for Computer Applications in Science and Engineering for the Period October 1, 1999 through March 31, 2000. Technical Report NASA/CR-2000-210105, NAS 1.26:210105, NASA (2000)
7. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM transactions on Programming Languages and Systems (TOPLAS) 16(5), 1512–1542 (1994)
8. Cockburn, A.: Agile software development, vol. 177. Addison-Wesley Boston (2002)
9. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. pp. 269–282. ACM (1979)
10. Debois, S., Hildebrandt, T.: The DCR Workbench: Declarative Choreographies for Collaborative Processes. In: Behavioural Types: from Theory to Tools, pp. 99–124. River Publishers (2017)
11. Debois, S., Hildebrandt, T.T., Slaats, T.: Hierarchical Declarative Modelling with Refinement and Sub-processes. In: Business Process Management. pp. 18–33 (2014)
12. Debois, S., Hildebrandt, T.T., Slaats, T.: Replication, Refinement & Reachability: Complexity in Dynamic Condition-Response Graphs. Acta Informatica (2017)
13. Ernst, M.D.: Static and dynamic analysis: Synergy and duality. In: ICSE Workshop on Dynamic Analysis. pp. 24–27 (2003)
14. Hildebrandt, T., Mukkamala, R.R.: Declarative Event-Based Workflow as Distributed Dynamic Condition Response Graphs. In: Post-proceedings of PLACES 2010. EPTCS, vol. 69, pp. 59–73 (2010)
15. Hildebrandt, T., Mukkamala, R.R., Slaats, T., Zanitti, F.: Contracts for cross-organizational workflows as timed dynamic condition response graphs. The Journal of Logic and Algebraic Programming 82(5-7), 164–185 (2013)
16. Hull, R., et al.: Introducing the guard-stage-milestone approach for specifying business entity lifecycles. In: WS-FM. vol. 6551, pp. 1–24. Springer (2010)
17. Janzen, D., Saiedian, H.: Test-driven development concepts, taxonomy, and future direction. Computer 38(9), 43–50 (2005)
18. Mei, H., Hao, D., Zhang, L., Zhang, L., Zhou, J., Rothermel, G.: A static approach to prioritizing junit test cases. IEEE Transactions on Software Engineering 38(6), 1258–1275 (2012)
19. Object Management Group: Case Management Model and Notation. Tech. Rep. formal/2014-05-05, Object Management Group (May 2014), version 1.0
20. Object Management Group BPMN Technical Committee: Business Process Model and Notation, Version 2.0 (2013)
21. Pesic, M., Van der Aalst, W.M.: A declarative approach for flexible business processes management. In: Business Process Management. pp. 169–180 (2006)
22. Pesic, M., Schonenberg, H., Aalst, W.M.P.v.d.: DECLARE: Full Support for Loosely-Structured Processes. In: Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference. pp. 287–300. IEEE (2007)
23. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science (FOCS). p. 46?57 (1977)
24. Schwaber, K., Beedle, M.: Agile software development with Scrum, vol. 1. Prentice Hall Upper Saddle River (2002)
25. Slaats, T.: Flexible Process Notations for Cross-organizational Case Management Systems. Ph.D. thesis, IT University of Copenhagen (January 2015)
26. Zhang, L., Zhou, J., Hao, D., Zhang, L., Mei, H.: Prioritizing junit test cases in absence of coverage information. In: Software Maintenance. pp. 19–28. IEEE (2009)

27. Zugal, S., Pinggera, J., Weber, B.: The impact of testcases on the maintainability of declarative process models. Enterprise, Business-Process and Information Systems Modeling pp. 163–177 (2011)
28. Zugal, S., Pinggera, J., Weber, B.: Creating declarative process models using test driven modeling suite. In: CAiSE Forum 2011. pp. 16–32 (2012)