Declarative Process Models: Different Ways to be Hierarchical

Riccardo De Masellis[1], Chiara Di Francescomarino[1], Chiara Ghidini[1], and Fabrizio M. Maggi[2]

[1] FBK-IRST, Italy {r.demasellis,dfmchiara,ghidini}@fbk.eu
[2] University of Tartu, Estonia f.m.maggi@ut.ee

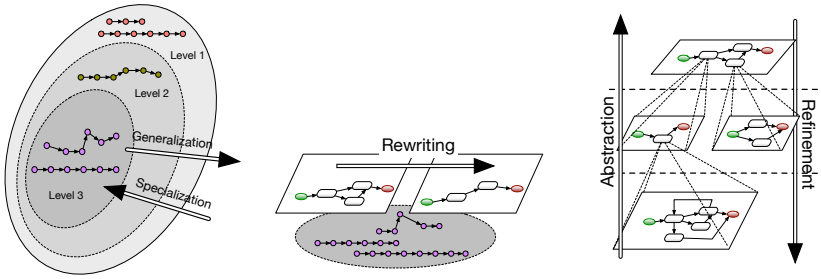
Abstract. In the literature, hierarchical dimensions for *procedural* process models have been widely investigated as they provide different ways to relate, organize and classify models. Such a categorization is based on the dimensions of *inheritance*, behavioral *equivalence*, and *modularization* and can be used to better understand and modify models as well as handle their complexity. Unfortunately, in the context of *declarative* process models hierarchical dimensions have been sparsely investigated. This paper addresses such a research gap. More specifically, we study a formal semantics for the dimensions above and show how they naturally induce hierarchies on a declarative process language based on DECLARE.

Keywords: Hierarchical Process Model, Linear Temporal Logic, Declare

1 Introduction

Hierarchical relations have been widely investigated and adopted in practice in the context of business process modeling as a key mechanism to handle complexity, organization and categorization. Concepts such as *modularization*, *decomposition*, *refinement*, *inheritance*, *reduction* and so on have been introduced, not infrequently with conflicting meanings, to hierarchically structure and relate business processes along different dimensions. Despite the terminological differences, we follow the analysis of [15] and identify three different dimensions along which processes can be arranged in a hierarchy.

The first dimension aims at achieving *inheritance* among process models. Inheritance is often interpreted with a behavioral connotation, wherein a child model enables a restrictive set (i.e., a subset) of its parents behaviors (see Figure 1a). This can also be achieved, in some special cases, through syntactical inheritance, where parts of the parent model are borrowed by the child. However, this dimension cannot be characterized from a syntactic viewpoint, e.g., through specific constructs. We adopt the UML terminology, and use *specialization* and *generalization* to indicate the top-down and bottom-up directions of the inheritance relation. The second dimension can be seen as a special case of the first one, as it categorizes equivalent process models, that is, those that differ from the syntactic point of view but accept the same set of behaviors. However, it differs from the inheritance dimension by being "horizontal" and, as such, it is not characterized by bottom-up and top-down directions but rather realized through *rewriting* of the processes, as depicted in Figure 1b. Finally, the third



(a) (behavioral) Inheritance. (b) Rewriting processes. (c) Decomposable modules.

Fig. 1: Three ways to be hierarchical.

hierarchical dimension involves the process-subprocess decomposition, and its graphical representation is provided in Figure 1c, where increasing levels specify more and more process details. This dimension is essential to obtain *modularization* through decomposable modules, or subsystems, as observed in [15] and [13], and is usually achieved, in a procedural process specification, by means of ad hoc syntactic constructs, e.g., the BPMN subprocess construct. In this paper, we adopt the terminology from [13] and use *refinement* and *abstraction* to indicate the top-down and bottom-up directions of modular decomposition.

While the notion of hierarchy is well investigated and supported in procedural approaches to business process modeling, it is less understood and used in the context of declarative models. This can be ascribed to the intrinsic difficulty of characterizing and supporting these dimensions in declarative process models. In this work, we aim at addressing such a gap by investigating, in a precise and logic-based manner, the formalization and the effects of *inheritance*, *rewriting*, and *modularization* in the context of the DECLARE modeling language [14]. We chose DECLARE as it is the most popular declarative language for modeling business processes, and because it grounds on a formal, logic-based, semantics. This latter fact enables us to embed the conceptual understanding of the three hierarchical relations into the modeling language in a precise manner and to characterize them by means of logical properties holding between the hierarchically related processes. This formal investigation highlights the following: first, DECLARE, in its current form, fails to support hierarchies in full. Thus, Section 5 introduces HIDEC, a DECLARE extension which fills this gap. Second, each dimension is indeed represented by means of a hierarchy, i.e., a computationally decidable (partial) order among HIDEC processes (Section 6). The formal analysis provided Sections 5 and 6 is grounded in a literature review (Section 2) and in a conceptual investigation of how to interpret the three dimensions above in a declarative setting (Section 4).

2 Related Work

Hierarchies of procedural process models have been widely investigated. The classification provided in this paper is inspired by the work proposed in [15] where the research on procedural process model hierarchies is recognized to provide contributions to *inheritance*, *rewriting* and *modularization*. Most of the work in

the literature on hierarchical procedural models fall in one of these categories and even if there are much less contributions in the declarative settings, the same concepts/categories also apply. In the following, we analyze the literature according to its contribution to each dimension.

In [1], the authors study inheritance of behavior in a simple process-algebraic setting as well as in a Petri-net framework. The approach in [11] groups together similar process models where similarity measures are (also) based on the concept of inheritance borrowed by object-oriented programming. Inheritance is also used in the context of process configuration to customize generic process model "templates" instead of building one from scratch. From a template, several process variants can be derived by means of a restricted set of change operations [23]. In the last decade, configurability of procedural process models has been widely investigated [16], while for declarative ones the approach in [21] has been recently proposed.

Concerning rewriting, several works provide reduction rules to support the analysis of procedural process models. In [24], reduction rules translating reset-nets to much smaller ones (whilst preserving the original properties) are shown. Analogously, [25] presents reduction rules for YAWL workflows with cancelation regions and OR-joins. Also, [17] introduces a set of graph reduction rules as a verification mechanism to identify structural conflicts in a procedural process model. In the context of declarative process models, reduction rules to remove redundancies in DECLARE models obtained from process discovery are investigated in [10].

Coming to modularization, in [2], the authors provide guidelines to select parts of procedural process models, represented as meta-graphs, for modularization purposes. Good candidates for subprocesses are fragments with a single input and a single output control flow arc. Other work [22, 9] provide recommendations regarding the size of a subprocess in a process model. To assess which modularization should be preferred starting from the characteristics of a complex process model, quality metrics are proposed in [12]. In [23], the ability to extract a subprocess from a process model has been described as a change pattern for process-aware information systems. In the context of declarative process models, in [27] the understandability of hierarchical declarative models is discussed and how subprocesses enhance the expressiveness of declarative modeling languages is shown. Differently form what we propose, this paper discusses a cognitive-psychology-based framework to assess the impact of hierarchy on the understandability of declarative models, rather than formally studying its properties. A different approach to modularization for declarative process models has been presented in [5], where a definition of hierarchical declarative process models based on Dynamic Condition Response (DCR) graphs is given, and can be used for incremental refinement, adaptation, and dynamic creation of subprocesses. However, besides being based on a semantics different from LTL, the focus of the paper is providing constructs that support modularization, rather than studying the different dimensions of hierarchy. Lastly, also the recently-introduced Declarative Process Intermediate Language (DPIL) [26] allows for modeling sub-processes, and it focuses not only on the model behavioral dimension, i.e., the traditional control-flow, but also on other perspectives such as the organizational perspective (tasks may be assigned/performed by specific roles/groups) and the informational perspective (resources/entities accessed by

activities). Given the expressivity of DPIL, several hierarchies can be defined by considering the different perspectives, e.g., hierarchies between roles [18]. Our analysis focus on the core traditional control-flow perspective only so as to first have a formal basis on top of which possibly many other constructs/extensions can be considered.

Aspect	**Inheritance**	**Rewriting**	**Modularization**
Understandability		[24, 10]	[12, 15, 22, 9, 27]
Reusability	[11, 16, 21, 11]		[23, 15]
Scalability		[25, 17]	[15]
Analysis		[25, 17]	

Table 1: Aspects of hierarchical dimensions.

The importance of these dimensions in the context of hierarchical declarative process models is also demonstrated by the number of properties and aspects that they affect. Table 1 maps the three dimensions of hierarchical models (columns) to the corresponding affected aspects (rows). The table shows that for instance, *understandability* is affected by both modularization and equivalence. As shown in several studies, indeed, understandability is affected by modularization, because smaller modules are usually easier to understand for humans (see, e.g., [12]). On the other hand, also different representations of the same set of behaviors (equivalence) impact on understandability [10]. *Reusability* is affected by both inheritance and modularization. Inheritance, for example, enables the reuse of (process) variants [21] while modularization the reuse of (process) modules [23]. Rewriting and modularization, enabling respectively the optimized [25, 17] and distributed [15] execution of process models impact on *scalability*. Finally, rewriting also supports formal *analysis* of models [25, 17], as conflict detection.

From the above literature review, we can conclude that contributions on hierarchical declarative process models have been sparse, often ad hoc, and, in the vast majority of cases ([5] is a notable exception) the proposed semantics is not formally grounded. In addition, they usually deal with a subset of dimensions thus not providing a comprehensive interpretation/semantics for all of them, which we tackle in this work.

3 Preliminaries

Our choice of using DECLARE [14] grounds on the fact that it adopts the semantics of a well-known and well-studied temporal logics. This not only paves the way for a mathematical characterization of dimensions and hierarchy, but also allows us to borrow some of the theoretical results originally developed for that logics.

A DECLARE model is a set of constraints that must hold in conjunction during the process execution, declaratively setting the boundaries that process instances must not overcome. Each constraint is chosen among a set of predefined "templates" that express different (partial) orders on the activities the process is intended to perform.

Definition 1 (DECLARE). *Given a finite alphabet of activities Σ, a* DECLARE *process Φ is a set of constraints, intended to be in conjunction, inductively defined*

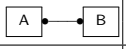
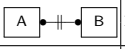
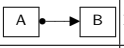
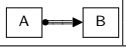
TEMPLATE	FORMALIZATION	NOTATION	DESCRIPTION
existence(A)	$\Diamond A$		A has to occur at least once
absence(A)	$\neg\Diamond A$		A has to never occur
exactly1(A)	$\Diamond A \wedge \neg(A \wedge \bigcirc(\Diamond A))$		A has to occur exactly once
coexistence(A,B)	$(\Diamond A \rightarrow \Diamond B) \wedge \neg(\Diamond B \rightarrow \Diamond A)$		If A occurs, B must occur and viceversa
not coexistence(A,B)	$(\Diamond A \rightarrow \neg\Diamond B)$		If A occurs, B must not occur and viceversa
response(A,B)	$\Box(A \rightarrow \Diamond B)$		If A occurs, B must eventually follow
chain response(A,B)	$\Box(A \rightarrow \bigcirc B)$		If A occurs, B must occur next

Table 2: Graphical notation and LTL formalization of some DECLARE templates.

as follows:

$$\varphi ::= \textsc{decUn}(\mathsf{A}) \mid \textsc{decBin}(\mathsf{A}_1, \mathsf{A}_2)$$
$$\Phi ::= \{\varphi\} \mid \Phi_1 \cup \Phi_2$$

where $\mathsf{A}, \mathsf{A}_1, \mathsf{A}_2 \in \Sigma$, $\textsc{decUn}(\cdot)$ *is a unary* DECLARE *template and* $\textsc{decBin}(\cdot,\cdot)$ *is a binary* DECLARE *template. We denote with* $\mathcal{D}$ *the set of* DECLARE *processes and with* $\Sigma(\Phi)$ *the set of activities occurring in* Φ*.*

Together with a mnemonic name for specific LTL formulas, DECLARE also offers a graphical representation for each template. Table 2 reports the graphical notation, the formalization and a brief description of the DECLARE templates that we use in this paper. As examples, the binary *response*(A, B) template says that each occurrence of activity A must be eventually followed by activity B, and it indeed represents the LTL formula $\Box(\mathsf{A} \rightarrow \Diamond\mathsf{B})$, where $\Box$ is the LTL "always" temporal operator and $\Diamond$ is the "eventually" temporal operator, while the unary template *existence*(A) (resp., *absence*(A)) says that activity A must be eventually performed (resp., never be performed), and its LTL formula is $\Diamond\mathsf{A}$ (resp., $\neg\Diamond\mathsf{A}$).

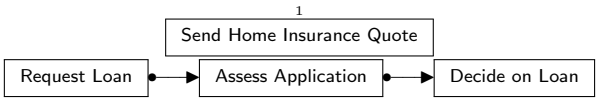
Given a DECLARE process Φ, its semantics is given in terms of *finite* sequences of activities, also called traces, satisfying Φ, which we denote by $\mathcal{L}(\Phi)$, where only one activity is performed at a time. This is formally achieved by taking the finite-trace semantics of LTL [3, 4] and by adding an (implicit) global constraint in each process expressing the mutual exclusion among activities. From a practical viewpoint, the reasoning tasks on processes Φ, $\Psi \in \mathcal{D}$, namely, satisfiability (is $\mathcal{L}(\Phi) \neq \emptyset$?), validity (is every trace in $\mathcal{L}(\Phi)$?) and logical implication (is $\mathcal{L}(\Phi) \in \mathcal{L}(\Psi)$?), reduce to each other and can be solved by building the so-called *automaton* for Φ (and Ψ) [3], which we denote by $A(\Phi)$. We observe that when adopting the LTL finite-trace semantics, automata for formulas are actually *finite-state machines* and as such, they can be manipulated by using well-known and optimized algorithms (we exploit some of them in Section 6). In the remainder, we refer to LTL by implicitly meaning LTL with finite-trace semantics, which allows for using the term *automata* and *finite-state machine* as synonyms. We stress that the result presented here *do not* carry to traditional (infinite-trace semantics) LTL (see [4] for a dissertation on the difference between finite- and infinite-trace semantics).

4 Conceptual Investigation

In this section, we provide a conceptual investigation of *inheritance*, *rewriting* and *modularization* with the help of a running example inspired by [7].

In a typical *loan application* (LA) scenario, after a customer has requested a loan (Request Loan (RL)), the customer application is assessed (Assess Application (AA)) and, once assessed, a decision about the loan (Decide on Loan (DL)) is taken. This is modeled in DECLARE with the two *response* constraints (see Section 3): $response(\mathsf{RL}, \mathsf{AA})$ and $response(\mathsf{AA}, \mathsf{DL})$, whose graphical representation is:

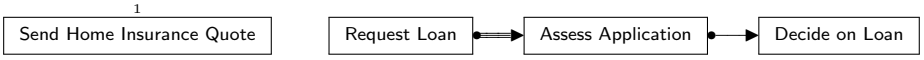
The first dimension we study is process *inheritance*, whose bottom-up and top-down directions are defined as *specialization* and *generalization*, respectively. Intuitively, a process model *specializes* another (parent) process model if the behaviors allowed by the specialization are a subset of the behaviors allowed by the parent. Generalization is defined symmetrically. For instance, let us consider the *mortgage loan* (ML) specializing the behavior of LA in that it restricts the behaviors of the latter (i.e., those satisfying the two *response* constraints) to those containing exactly one occurrence of the activity Send Home Insurance Quote (SHIQ), i.e., those focusing on house loans.



As the above graphical representation of the DECLARE constraints shows, in this case, we have not only a behavioral inheritance between the two process models but also a syntactical one, as the specialized process model is obtained by adding constraints (e.g., the $exactly1(\mathsf{SHIQ})$ in the example) to the set of constraints of the parent.

A slightly different example of specialization of the LA process is the loan application process for fidelity customers FCL which restricts the behaviors allowed by the general LA process by imposing that fidelity customers are served immediately after the loan request is presented:

As the above constraints shows, FCL does not inherit the syntactical description of LA, as the $response(\mathsf{RL}, \mathsf{AA})$ is replaced by a $chain\_response(\mathsf{RL}, \mathsf{AA})$, indicating that Assess Application has to be executed immediately after Request Loan. Nonetheless, it accepts a subset of the parent's behaviors. In general, we can have several layers of inheritance. We can think for instance at the process for the mortgage loan for fidelity customers:



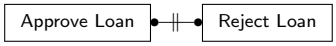
which is (*i*) a specialization of the mortgage loan process, restricting its behaviors to those dedicated to fidelity customers, (*ii*) a specialization of the fidelity customers loan, restricting its behaviors to those related to the mortgage, and

(*iii*) a specialization of the loan process, restricting its behaviors to mortgage processes and fidelity customers.

A second hierarchical dimension relates to *rewriting*. Intuitively, rewriting refers to the case in which two (declarative) process models describe exactly the same set of behaviors although their representation is different and, in particular, one is more compact than the other. In other terms, the process models are only semantically but not syntactically equivalent. Let us consider again the DECLARE model of the fidelity customer loan `FCL` and the one below (called `FCLRed`):

Although the two DECLARE descriptions are different (`FCLRed` contains an extra-constraint with respect to the `FCL`) the behaviors they allow are exactly the same ones since according to the DECLARE order relationships [20], the *chain response* is stronger than the *response* constraint. We can conclude that original `FCL` is more compact, and indeed is also the *minimal* one, i.e., it cannot be further reduced.

The third dimension of hierarchical models identified in Section 2 is *modularization*, whose top-down direction is called *refinement*, and represents the process-subprocess relation, whilst the bottom-up direction, called *abstraction*, is the vice versa. Let us consider the `LA` model presented before: the behavior of the Decide on Loan activity can be *refined*, by detailing that the loan decision consists of either a loan request approval or a loan request rejection. The following figure shows the refinement of the Decide on Loan activity, imposing a *not coexistence* between the Approve Loan and the Reject Loan activities.



We notice that, while inheritance can easily be defined in DECLARE, modularization, although being very common and well-investigated in procedural languages, cannot. In order to overcome this limitation, we extend the traditional DECLARE language by defining and investigating the properties of the different hierarchical dimensions it enables.

5 HiDec: Hierarchical Declarative Processes

We introduce HiDec, a DECLARE extension that allows for the formal definition and implementation of the three hierarchical dimensions.

Definition 2 (HiDec). *Given a finite set of activities Σ, a* HiDec *process Φ is a set of constraints, intended to be in conjunction, inductively defined as follows:*

$$\begin{aligned}\varphi ::= {} & \textsc{decUn}(\mathsf{A}) \mid \textsc{decUn}(\varphi) \mid \\ & \textsc{decBin}(\mathsf{A}_1, \mathsf{A}_2) \mid \textsc{decBin}(\varphi_1, \varphi_2) \mid \textsc{decBin}(\mathsf{A}, \varphi) \mid \textsc{decBin}(\varphi, \mathsf{A}) \\ \Phi ::= {} & \{\varphi\} \mid \{\mathsf{A} \leftrightarrow \varphi\} \mid \Phi_1 \cup \Phi_2\end{aligned}$$

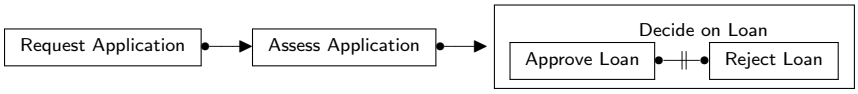
Where $\mathsf{A}, \mathsf{A}_1, \mathsf{A}_2 \in \Sigma$, $\textsc{decUn}(\cdot)$ is a unary DECLARE *template and $\textsc{decBin}(\cdot,\cdot)$ is a binary* DECLARE *template. We denote with $\mathcal{C}$ the set of* HiDec constraints*, i.e., the φ formulas of the above grammar, with $\mathcal{H}$ the set of* HiDec *processes, i.e., the Φ formulas, and with $\Sigma(\Phi)$ the set of activities occurring in Φ.*

HiDec allows us to represent all the three different types of hierarchies, included modularization, which we would not have been able to represent with traditional declare. Theorem 1 shows indeed that HiDec is more expressive than declare.

Theorem 1. HiDec *is more expressive than* declare.

Proof (Sketch). Since Σ is finite, the number of different declare constraints is finite, and, as a consequence, the number of (syntactically) different processes of $\mathcal{D}$ is finite as well. This in turn implies that the number of semantically different processes is finite (some syntactically different processes may, in fact, be equivalent). Conversely, the number of HiDec syntactically different processes is (countably) infinite, given that an arbitrary nesting of sub-processes is allowed. Also, the capability of having an arbitrary nesting of temporal operators allows us to express a (countably) infinite number of semantically different processes. As an example, for each $n \in \mathbb{N}$, it is possible to express a formula Υ_n saying: "Activity A occurs at least n times", and for each $i, j \in \mathbb{N}$, $\mathcal{L}(\Upsilon_i) \neq \mathcal{L}(\Upsilon_j)$.

By means of such an extended language, we are able to represent also process `RefLA`, a refinement of `LA` described in Section 4, where activity Decide on Loan is defined as the subprocess Decide on Loan $\leftrightarrow$ ($\Diamond$Accept Loan $\rightarrow \neg(\Diamond$Reject Loan)). The Figure below shows a graphical representation of the above constraint.



6 Hierarchies in Declarative Models

In this section, we provide a precise mathematical structure to concepts introduced before. Such a formalization is naturally originated by the definition of *hierarchy*: an *arrangement* or classification of things according to some dimension. In what follows, indeed, the previous ideas take shape into formal relations between processes which we prove to be (partial) orderings. We recall that a partial order is a set equipped with a binary relation R among its elements which satisfies the property of reflexivity ($R(a, a)$), antisymmetry ($R(a, b)$ and $R(b, a)$ entails $a = b$) and transitivity ($R(a, b)$ and $R(b, c)$ entails $R(a, c)$). Our goal is therefore to define such relations so that they reflect the informal intuitions as well as provide fine properties on the set of HiDec formulas.

6.1 The inheritance hierarchy

The inheritance dimension we explore is purely semantic. We say that process Ψ specializes Φ if the set of traces accepted by Ψ is a subset of those accepted by Φ, or, equivalently, if Ψ logically implies Φ.

Definition 3. *Let Σ be a set of activities, $\mathcal{H}$ the* HiDec *language over Σ and $\Phi \in \mathcal{H}$. Process $\Psi \in \mathcal{H}$ is a* specialization *of Φ, if $\mathcal{L}(\Psi) \subseteq \mathcal{L}(\Phi)$.*

Generalization is defined symmetrically: a process generalizes another one if the former is logically implied by the latter. Being semantic, such a definition applies to traditional DECLARE as well. However, since HIDEC subsumes DECLARE, we stick to HIDEC to be consistent with the definitions of the other hierarchical dimensions.

Unfortunately, relation $\subseteq$ on the set of traces does not order the set $\mathcal{H}$. Specifically, the antisymmetry fails, as there are formulas Φ, Ψ for which $\mathcal{L}(\Phi) \subseteq \mathcal{L}(\Psi)$ and $\mathcal{L}(\Psi) \subseteq \mathcal{L}(\Phi)$ holds, i.e., $\mathcal{L}(\Phi) = \mathcal{L}(\Psi)$ but $\Phi \neq \Psi$. Intuitively, logical languages contain *synonyms*, i.e. syntactically different formulas semantically describing the same set of accepted traces. Next section formalizes this concept and provides a way out.

6.2 The rewriting hierarchy

Definition 4. *Let Σ be a set of activities, $\mathcal{H}$ the* HIDEC *language over Σ and $\Phi, \Psi \in \mathcal{H}$. Processes Φ and Ψ are* equivalent*, written $\Phi \sim \Psi$,if $\mathcal{L}(\Psi) = \mathcal{L}(\Phi)$.*

It is immediate to verify that relation $\sim \subseteq \mathcal{H} \times \mathcal{H}$ is an equivalence relation, i.e., it is reflexive, transitive and symmetric and as such, it partitions the set $\mathcal{H}$ into *equivalence classes*. The equivalence class of an element $\Phi \in \mathcal{H}$ is denoted by $[\Phi]$, and contains all synonyms of Φ: this formally underpins the intuitive notion of "horizontal" hierarchy among processes that can be obtained by rewriting a model into an equivalent one. The set of all equivalence classes of $\mathcal{H}$ by $\sim$ is called the quotient set, and denoted by $\mathcal{H}/\sim$. Since formulas in an equivalence class are satisfied by the same traces, i.e., they are semantically indistinguishable, they all behave the same with respect to the specialization relation: if Ψ is a specialization of Φ, then for any other $\Psi' \in [\Psi]$ and $\Phi' \in [\Phi]$ we have that Ψ' is a specialization of Φ'. This motivates the extension of the specialization definition to the set $\mathcal{H}/\sim$.

Definition 5. *Let Σ, $\mathcal{H}$, Φ and Ψ as before. We define relation $\sqsubseteq \subseteq \mathcal{H}/\sim \times \mathcal{H}/\sim$ as follows: $[\Psi] \sqsubseteq [\Phi]$ if Ψ is a specialization of Φ.*

Theorem 2. *Relation $\sqsubseteq$ is a partial order.*

Proof (Sketch). The definition of $\sqsubseteq$ grounds on the language inclusion $\subseteq$ relation, hence its reflexivity, antisymmetry and transitivity properties trivially follow from those of $\subseteq$, which is a partial order for any set.

This result establishes that $\sqsubseteq$ induces a hierarchy among equivalence classes, which can be used to order HIDEC processes, from the more "permissive" ones, i.e, those that allow for more behaviors, to the "stricter" ones, which accept only few traces.

We notice that equivalence classes are countably infinite and that there is always a *least element* Ψ_ℓ, i.e., the most specialized process which does not allow for any behavior ($\mathcal{L}(\Psi_\ell) = \emptyset$). Also, for any two processes Φ and Ψ one of the following holds:

1. $[\Psi] \sqsubseteq [\Phi]$ meaning that Ψ is more restrictive than Φ (or, equivalently, Φ is more permissive than Ψ);
2. the other way around;

3. both $[\Phi] \sqsubseteq [\Psi]$ and $[\Psi] \sqsubseteq [\Phi]$ meaning that $\Phi \sim \Psi$ (or, equivalently $[\Phi] = [\Psi]$), i.e, they belong to same equivalence class; y
4. they are incomparable, because neither $[\Phi] \sqsubseteq [\Psi]$ nor $[\Psi] \sqsubseteq [\Phi]$ hold.

Example 1. The `LA` process in Section 4 is parent of both `ML` and `FCL` process, namely, $[\texttt{ML}] \sqsubseteq [\texttt{LA}]$ and $[\texttt{FCL}] \sqsubseteq [\texttt{LA}]$, and `ML` and `FCL` are incomparable. Also, `FCL` and `FCLRed` belongs to the same equivalence class, i.e., $[\texttt{FCL}] = [\texttt{FCLRed}]$.

Notice that in HiDec we can express unsatisfiable processes, which all belong to the most restrictive $[\bot]$ equivalence class (no trace is accepted), but the $[\top]$ class is missed, as there is no way to describe processes accepting all traces.

We conclude the section by remarking that relation $\sqsubseteq$ is decidable. Indeed, checking whether $[\Psi] \sqsubseteq [\Phi]$ amounts to checking if Ψ logically implies Φ, which is known to be a Pspace-complete problem [3].

On process rewriting. Once the equivalence relation $\sim$ on $\mathcal{H}$ has been defined, one is typically interested in electing a *representative* of each equivalence class, i.e., a formula which has the same semantic properties of any other in the same class, but that is different in other aspects. One interesting metric can be the "compactness" of the formula (which can be simply defined as its length) or the "understandability" of the process. As pointed out in Section 2, the literature usually considers these two aspects related.

Unfortunately, it is very hard to find a procedure to transform a process into an equivalent one that is, e.g., more compact, for at least two reasons. The first one is that there is no effective way to syntactically reduce an ltl formula by being sure it keeps the same semantic properties (apart from the trivial well-known equivalences, such as $\Box\Box\Phi \sim \Box\Phi$). Existing works in the literature address the problem specifically for declare formulas either by dropping redundant constraints [6] or finding ad-hoc "reduction rules" for declare patterns [10]. These works cannot therefore be used for HiDec. Moreover, there is no guarantee, in general, to find the *minimal* formula. The second motivation concerns the intrinsic difficulty of the problem, as each equivalence class of HiDec (as well as ltl), has in general (countably) infinite cardinality, thus ruling out any brute-force approach.

Our motivation is foundational: do we really need to transform the *syntactic* model? declare, HiDec and ltl formulas are just the "front-end" of a process/dynamic system, and, as such, they are used for modeling purposes only. The whole reasoning machinery behind, which is highly affected, performance-wise, by unnecessary redundancies, actually uses *automata*. We therefore move to the semantic level and, as representative of each class, we find the *minimal* and *unique* automaton accepting all and only the traces of that class, and use it for the actual reasoning tasks. In this way we decouple the representation layer from the semantic layer and leave the modeler free of choosing the representation he prefers. We believe that the process' representation, i.e., the HiDec (or declare, or ltl) constraints can possibly be redundant, as long as the reasoning services are guaranteed to be efficient.

By notational abuse, let $\mathcal{L}(\mathcal{H})$ be the set of all possible traces that can be represented by using formulas in $\mathcal{H}$, $\mathcal{A}$ the set of all possible automata (we stress again, on *finite* traces), A an automaton in $\mathcal{A}$ and $\mathcal{L}(A)$ the set of traces recognized by A.

Definition 6. *Let A, $A' \in \mathcal{A}$ two automata. We say that A and A' are* equivalent*, written $A \approx A'$ if $\mathcal{L}(A) = \mathcal{L}(A')$.*

Trivially, $\approx$ is an equivalence relation, partitioning the set $\mathcal{A}$ into equivalence classes. The quotient set of $\mathcal{A}$ by $\approx$ is $\mathcal{A}/\approx$. Notice that we can actually consider the automaton A for a formula $\Phi \in \mathcal{H}$ as a function $A : \mathcal{H} \to \mathcal{A}$ transforming formulas into automata.

Theorem 3. *Let Φ, $\Psi \in \mathcal{H}$. Function $A : \mathcal{H} \to \mathcal{A}$ is an* equivalence-preserving *function, i.e., if $\Phi \sim \Psi$ then $A(\Phi) \approx A(\Psi)$. Moreover, for each $A' \in [A(\Phi)]$, we have that $\mathcal{L}(A') = \mathcal{L}(\Phi)$.*

Proof. If $\Phi \sim \Psi$, then, by Definition 4, $\mathcal{L}(\Phi) = \mathcal{L}(\Psi)$. By the correctness of automata construction in [3], it follows that $\mathcal{L}(A(\Psi)) = \mathcal{L}(A(\Phi))$, and hence $A(\Psi) \approx A(\Phi)$. By Definition 6 and from $\approx$ being an equivalence relation, it also follows that $\mathcal{L}(A') = \mathcal{L}(\Phi)$.

This result allows us to use, as a representative of a class $[\Phi]$, the minimum automaton $min(A(\Phi))$ recognizing the language $\mathcal{L}(\Phi)$, that can be obtained by using any automata minimizing algorithm on $A(\Phi)$ based on the Myhill-Nerode Theorem (see, e.g., [8]), which guarantees $min(A(\Phi))$ to be:

- *sound*, i.e., $\mathcal{L}(min(A(\Phi))) = \mathcal{L}(\Phi)$;
- the *smallest* automaton for $\mathcal{L}(\Phi)$, i.e., for each $A' \in [A(\Phi)]$, $|min(A(\Phi))| \leq |A'|$ (where $|A|$ measures number of states and transitions) and
- *unique*, i.e., for each $A' \in [A(\Phi)]$, $min(A') = min(A(\Phi))$ (modulo isomorphisms, namely, renaming of states).

Example 2. The (non-trimmed) automaton for the redundant process `FCLRed` in Section 4 obtained with the algorithm[3] in [3], has 14 states and 85 transitions, while after the minimization, it has 6 states and 24 transitions.

6.3 The modularization hierarchy

The last dimension we study is syntactical, and covers the intuitive process/subprocess relation. We propose a step-by-step methodology to *refine* a HiDec process model by specifying its subprocesses, which notably defines a (partial) order.

Definition 7. *Let Σ be an alphabet of activities, and let $\Phi \in \mathcal{H}$. Process $\Psi \in \mathcal{H}$ is a* refinement *of Φ, written $\Psi \preceq \Phi$, if Ψ can be obtained from Φ by applying $n \geq 0$* refinement steps *$\Phi_0 \Rightarrow \ldots \Rightarrow \Phi_n$ where:*

- *$\Phi_0 = \Phi$ and*
- *$\Phi_n = \Psi$ and*
- *each Φ_i $i \in \{1, \ldots n-1\}$ is such that either:*
  - *$\Phi_i = \Phi_{i-1} \cup \{\mathsf{A} \leftrightarrow \varphi\}$ with $\mathsf{A} \in \Sigma(\Phi_{i-1})$ and $\varphi \in \mathcal{C}$ (recall $\mathcal{C}$ is the set of* HiDec *constraints as in Definition 2) or*
  - *Φ_i can be obtained from Φ_{i-1} by applying a* partial *function $r_i : \Sigma(\Phi_{i-1}) \to \mathcal{C}$ which intuitively substitutes (some) activities occurring in Φ_{i-1} with a constraint in $\mathcal{C}$.*

[3] An ongoing implementation is available at: https://github.com/RiccardoDeMasellis/FLLOAT

Abstraction can be defined analogously, with an abstraction step consisting in either removing a $\mathsf{A} \leftrightarrow \varphi$ constraint or applying function $\mathsf{r}_i^{-1} : \mathcal{C} \to \Sigma$ which substitutes a HiDec constraint with an activity. Intuitively, a process is refined when a single activity, say A, is "expanded" in a complex subprocess φ. Such an expansion can take place either by adding a constraint $\mathsf{A} \leftrightarrow \varphi$[4] or by substituting all occurrences of A with φ. This two variants are worth to be discussed. First of all, we observe that the two procedures are semantically different, as the following example illustrate.

Example 3. Let $\Phi = \{\neg\Diamond\mathsf{A}\}$, and let us assume we want to refine A with subprocess $\neg\Diamond\mathsf{A}$. By adding the (unsatisfiable) constraint $\mathsf{A} \leftrightarrow \neg\Diamond\mathsf{A}$, the whole process Φ becomes unsatisfiable. Conversely, by using the substitution $\mathsf{r}(\mathsf{A}) = \neg\Diamond\mathsf{A}$, the refined process $\Psi = \{\Box\Diamond\mathsf{A}\}$ is still satisfiable.

Furthermore, the two choices covers different practical needs. The first option is more suitable for refining a process with a bottom-up approach, as it follows the natural human procedure of specifying a process from a more abstract level to a more specific one, still allowing a comprehensive view of all levels, being "conservative". The second one is instead a more "destructive" option for refinement, as, after few steps, the structure of the original process is lost. However, it is more appropriate for abstraction, as when a complex process contains no $\mathsf{A} \leftrightarrow \varphi$ constraints, it can only be abstracted by applying the r^{-1} function.

The notion of refinement (and abstraction) naturally defines an ordering $\preceq$ on set $\mathcal{H}$.

Theorem 4. *Relation $\preceq \subseteq \mathcal{H} \times \mathcal{H}$ is a partial order.*

Proof (Sketch). Reflexivity is trivial, given that every process is a refinement of itself (by applying 0 steps). Transitivity is also immediate since the composition of refinement steps is a refinement. Antisymmetry is proven by noticing that each step increases the length of the process. Since processes of different lengths are necessarily syntactically different, the only way to have $\Phi \preceq \Psi$ and $\Psi \preceq \Phi$ is when both refinements consists of 0 steps, thus entailing $\Phi = \Psi$.

Relation $\preceq$ therefore induces a hierarchy among HiDec processes, from the more "abstract" ones, i.e, those providing a high-level view of the process, to the more "refined" ones, i.e., those showing the details. We notice that HiDec allows for expressing countably many syntactically different processes and that $\preceq$ does not define a greatest nor least element. However, given Φ and Ψ one of the following holds:

1. $\Psi \preceq \Phi$, that is, Ψ is more refined than Φ (equivalent., Φ more abstract than Ψ);
2. $\Phi \preceq \Psi$;
3. both $\Phi \preceq \Psi$ and $\Psi \preceq \Phi$ hold, hence $\Phi = \Psi$;
4. they are incomparable, because neither $\Phi \preceq \Psi$ nor $\Psi \preceq \Phi$ hold.

[4] To match the intended semantics of the "$\leftrightarrow$" used for expanding a process, we relax the assumption of one activity true at the time by leaving out activities occurring in the left-hand part of double implication constraints, which, after the refinement, do not intuitively represent activities anymore, but rather "placeholders" or syntactic "shortcuts" for sub-processes.

Example 4. Process `RefLA` in Section 5, obtained from `LA` by adding the constraint $\mathsf{Decide\ on\ Loan} \leftrightarrow (\Diamond \mathsf{Accept\ Loan} \rightarrow \neg(\Diamond \mathsf{Reject\ Loan}))$, is therefore a child of `LA` according to the modularization hierarchy, i.e., $\texttt{RefLA} \preceq \texttt{LA}$.

Given a set of processes, a refinement/abstraction hierarchy can be built in practice: given Φ and $\Psi \in \mathcal{H}$, checking whether $\Psi \preceq \Phi$ is decidable. Refinement steps can be indeed seen as grammar production rules which never decrease the length of the process. Since Σ is finite, to check whether $\Psi \preceq \Phi$ we start from Φ, we apply the production rules in all possible ways and we stop when the current process exceeds the length of Ψ.

On the satisfiability of the refined process. The refinement relation is syntactical. As such, it is of interest to study syntactical restrictions on refinements which guarantee semantic properties of the refined processes. One of such semantic properties may be inheritance itself (see [19] for a similar analysis in the context of object-oriented systems). Given the lack of space, here we focus on a more basic yet useful semantic property: satisfiability. Given $\Psi \preceq \Phi$, are there straightforward restrictions on refinements that guarantee the (semantic) satisfiability of Ψ? We provide a negative answer.

Definition 8. *Let $\Phi, \Psi \in \mathcal{H}$ and let Ψ be a refinement of Φ, i.e., $\Phi_0 \Rightarrow \ldots \Rightarrow \Phi_n$, with $\Phi_0 = \Phi$ and $\Phi_n = \Psi$. We define the set of constraints introduced by the refinement as the set $\Delta = \bigcup_{i \in 0,\ldots n} \Delta_i$ where $\Delta_0 = \emptyset$ and each Δ_i with $i \in 1 \ldots n$ is the set of constraints (in $\mathcal{C}$) introduced by the $i-th$ refinement step, namely:*
- *$\Delta_i = \varphi$ if the step added a formula $\mathsf{A} \leftrightarrow \varphi$ to Φ_i or*
- *$\Delta_i = \Im(r_i)$, where $\Im(r_i)$ is the image of r_i, otherwise.*

As a first remark, we observe that the satisfiability of both Φ and Δ is not a sufficient condition to establish the satisfiability of Ψ, as the following counterexample shows.

Example 5. Let $\Phi = \{\neg\Diamond\mathsf{A}, \Diamond\mathsf{B}\}$ and $\Psi = \{\neg\Diamond\mathsf{A}, \Diamond(\Diamond\mathsf{A}))\}$ obtained from Φ by applying one refinement step with $\mathsf{r}(\mathsf{B}) = \Diamond\mathsf{A}$. Clearly, Φ is satisfiable, as well as $\Delta = \{\Diamond\mathsf{A}\}$, but Ψ is not.

This is not surprising, as, intuitively, new constraints in Δ may generate inconsistencies with other constraints in the original process Φ. It is also interesting that the unsatisfiability of some constraints in Δ does not entail the unsatisfiability of Ψ.

Example 6. Let $\Phi = \{\Diamond\mathsf{A}, \neg\Diamond\mathsf{B}\}$ and $\Psi = \{\Diamond\mathsf{A}, \neg\Diamond\mathsf{B}, \mathsf{B} \leftrightarrow (\Diamond\mathsf{C} \wedge \neg\Diamond\mathsf{C})\}$. Set $\Delta = \{\Diamond\mathsf{C} \wedge \neg\Diamond\mathsf{C}\}$ is unsatisfiable, but Ψ is not, as it is satisfied by every trace that eventually contains A but never contains B (which would imply the inconsistency).

Given the above results, we investigate a reasonable restrictions on refinements. The intuition suggests that inconsistencies are typically generated by adding constraints which include activities that are already mentioned in other, existing, constraints. We follow this idea and study special refinements in which each refinement step talk about "fresh" activities only, i.e., activities not that do not appear where else in the process, in order to understand if this is a sufficient condition to guarantee the satisfiability of the refined process. Unfortunately, this is not the case, as the following Theorem prove.

Theorem 5. *Let Φ and Ψ as in Definition 8. Let Φ and each Δ_i be satisfiable and such that for each $i \neq j$ we have $\Sigma(\Delta_i) \cap \Sigma(\Phi) = \emptyset$ and $\Sigma(\Delta_i) \cap \Sigma(\Delta_j) = \emptyset$. Then Ψ can be unsatisfiable.*

Proof. By using the r function is easy to nest temporal operators to generate a formula that can only be satisfied by a trace where two (or more) activities must be true at the same time, which clashes with the assumption of only one activity performed at the time (see Section 3). An example follows. Let $\Phi = \{\neg\Diamond\mathsf{A}, \neg\Diamond\mathsf{B}\}$ and let $\Psi = \{\neg\Diamond(\neg\Diamond\mathsf{C}), \neg\Diamond(\neg\Diamond\mathsf{D})\}$ obtained from Φ by using the refinement function $\mathsf{r}(\mathsf{A}) = \neg\Diamond\mathsf{C}$ and $\mathsf{r}(\mathsf{B}) = \neg\Diamond\mathsf{D}$. Using the well-known equivalence rules we get $\Psi = \{\Box\Diamond\mathsf{C}, \Box\Diamond\mathsf{D}\}$, which is true only if in the last instant both C and D are true.

7 Concluding Remarks

The formal investigation about declarative hierarchies carried out in this work allows us to provide a number of interesting results. First of all, the inheritance, rewriting and modularization dimensions, widely investigated for procedural models, are tailored to fit the declarative setting, thus providing a comprehensive perspective on hierarchical dimensions on declarative processes. We concretize such a conceptual view in HiDec, a language extending DECLARE that, beyond the formalization of inheritance, rewriting and modularization, supports the following results. The mathematical definition of the inheritance dimension based on logical implication allows us to carry any formal property entailed by a specialized process to all its parents, and provide a concrete way for optimizing reasoning tasks on redundant models while preserving the representation designed by the modeler. Finally, the definition of refinement (and abstraction) offers an actual methodology to refine (abstract) any HiDec model, which is an essential feature when dealing with complex processes.

As future work, we plan to empirically investigate a suitable graphical notation for specifying modular HiDec processes, which is only sketched here, as well as to develop a tool for supporting modelers in defining reduction/abstraction steps.

References

1. Basten, T., van der Aalst, W.M.P.: Inheritance of behavior. J. Log. Algebr. Program. 47(2), 47–145 (2001)
2. Basu, A., Blanning, R.W.: Synthesis and decomposition of processes in organizations. Information Systems Research 14(4), 337–355 (2003)
3. De Giacomo, G., De Masellis, R., Grasso, M., Maggi, F.M., Montali, M.: Monitoring business metaconstraints based on LTL and LDL for finite traces. In: Proc. of BPM 2014, Israel. pp. 1–17 (2014)
4. De Giacomo, G., De Masellis, R., Montali, M.: Reasoning on LTL on finite traces: Insensitivity to infiniteness. In: Proc. of AAAI 2014. pp. 1027–1033 (2014)
5. Debois, S., Hildebrandt, T.T., Slaats, T.: Hierarchical declarative modelling with refinement and sub-processes. In: Proc. of BPM 2014, Israel. pp. 18–33 (2014)
6. Di Ciccio, C., Maggi, F.M., Montali, M., Mendling, J.: Ensuring model consistency in declarative process discovery. In: Proc. of BPM'15. pp. 144–159 (2015)

7. Dumas, M., Rosa, M.L., Mendling, J., Reijers, H.A.: Fundamentals of Business Process Management. Springer (2013)
8. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley Longman Publishing Co., Inc. (2006)
9. Jr, N.F.K., McQueen, R.J.: Product flow, breadth and complexity of business processes: An empirical study of 15 business processes in three organizations. Business Process Re-engineering & Management Journal 2(2), 8–22 (1996)
10. Maggi, F.M., Bose, R.P.J.C., van der Aalst, W.M.P.: A knowledge-based integrated approach for discovering and repairing declare maps. In: Proc. of CAiSE 2013. pp. 433–448 (2013)
11. Malone, T.W., Crowston, K., Lee, J., Pentland, B., Dellarocas, C., Wyner, G., Quimby, J., Osborn, C.S., Bernstein, A., Herman, G., Klein, M., O'Donnell, E.: Tools for inventing organizations: Toward a handbook of organizational processes. Management Science 45(3), 425–443 (1999)
12. Mendling, J., Reijers, H.A., Cardoso, J.: What makes process models understandable? In: Proc. of BPM 2007, Australia. pp. 48–63 (2007)
13. Moody, D.L.: The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. IEEE TSE 35(6), 756–779 (2009)
14. Pesic, M., van der Aalst, W.: A declarative approach for flexible business processes management. In: Proc. of BPM2006 Workshops. vol. 4103, pp. 169–180 (2006)
15. Reijers, H.A., Mendling, J.: Modularity in process models: Review and effects. In: Proc. of BPM 2008, Italy. pp. 20–35 (2008)
16. Rosa, M.L., van der Aalst, W.M., Dumas, M., Milani, F.P.: Business process variability modeling : A survey. Tech. rep. (2013), http://eprints.qut.edu.au/61842/
17. Sadiq, W., Orlowska, M.E.: Analyzing process models using graph reduction techniques. Inf. Syst. 25(2), 117–134 (2000)
18. Schönig, S., Cabanillas, C., Jablonski, S., Mendling, J.: A framework for efficiently mining the organisational perspective of business processes. Dec. Supp. Sys. (2016)
19. Schrefl, M., Stumptner, M.: Behavior-consistent specialization of object life cycles. ACM Trans. Softw. Eng. Methodol. 11(1) (Jan 2002)
20. Schunselaar, D.M.M., Maggi, F.M., Sidorova, N.: Patterns for a log-based strengthening of declarative compliance models. In: Proc. of IFM 2012. pp. 327–342 (2012)
21. Schunselaar, D.M.M., Maggi, F.M., Sidorova, N., van der Aalst, W.M.P.: Configurable declare: Designing customisable flexible process models. In: On the Move to Meaningful Internet Systems: OTM 2012, Italy. Proceedings. pp. 20–37 (2012)
22. Sharp, A., McDermott, P.: Workflow Modeling: Tools for Process Improvement and Application Development. Artech House, Inc., Norwood, MA, USA, 1st edn. (2001)
23. Weber, B., Reichert, M., Rinderle-Ma, S.: Change patterns and change support features - enhancing flexibility in process-aware information systems. Data Knowl. Eng. 66(3), 438–466 (2008)
24. Wynn, M.T., Verbeek, H.M.W., van der Aalst, W.M.P., ter Hofstede, A.H.M., Edmond, D.: Soundness-preserving reduction rules for reset workflow nets. Inf. Sci. 179(6), 769–790 (2009)
25. Wynn, M.T., Verbeek, H.M.W.E., van der Aalst, W.M.P., ter Hofstede, A.H.M., Edmond, D.: Reduction rules for YAWL workflows with cancellation regions and or-joins. Information & Software Technology 51(6), 1010–1020 (2009)
26. Zeising, M., Schönig, S.S., Jablonski, S.: Towards a common platform for the support of routine and agile business processes. In: CollaborateCom 2014. pp. 94–103 (2014)
27. Zugal, S., Soffer, P., Haisjackl, C., Pinggera, J., Reichert, M., Weber, B.: Investigating expressiveness and understandability of hierarchy in declarative business process models. Software and System Modeling 14(3), 1081–1103 (2015)

De Masellis R., Di Francescomarino C., Ghidini C., Maggi F. M., Declarative Process Models: Different Ways to Be Hierarchical. *In proc. of* Service-Oriented Computing Volume 9936 of the series Lecture Notes in Computer Science pp 104-119. 10.1007/978-3-319-46295-0_7.


```
@inproceedings{DeMasellisFGM16,
  author    = {Riccardo {De Masellis} and
               Chiara Di Francescomarino and
               Chiara Ghidini and
               Fabrizio Maria Maggi},
  title     = {Declarative Process Models: Different Ways to Be Hierarchical},
  booktitle = {Service-Oriented Computing - 14th International Conference, {ICSOC}
               2016, Banff, AB, Canada, October 10-13, 2016, Proceedings},
  pages     = {104--119},
  year      = {2016},
  crossref  = {DBLP:conf/icsoc/2016},
  url       = {http://dx.doi.org/10.1007/978-3-319-46295-0_7},
  doi       = {10.1007/978-3-319-46295-0_7}
}
```