A Deductive View on Process-Data Diagrams

Manfred A. Jeusfeld

Tilburg University, Warandelaan 2, 5037AB Tilburg, The Netherlands
manfred.jeusfeld@acm.org, http://conceptbase.cc

Abstract. Process-Data Diagrams (PDDs) are a popular technique to represent method fragments and their recombination to new adapted method specifications. It turns out that PDDs are at odds with a strict separation of MOF/MDA abstraction levels as advocated by MOF/MDA. We abandon the restriction and specify PDDs by a metametamodel that supports both process and product parts of PDDs. The instantiation of the process side of PDDs can then the used as the type level for a simple traceability framework. The deductive formalization of PDDs allows to augment them by a plethora of analysis tools. The recombination of method fragments is propagated downwards to the recombination of the process start and end points. The hierarchical structure of the product side of PDDs can be used to detect unstructured updates from the process side.

Keywords: method fragment, deductive rule, traceability, metamodel

1 Introduction

Method Engineering advocates the assembly [12] of adapted information system development methods from a pool of method fragments [3], depending on the development context [4, 13]. One technique for recording re-usable method fragments are process-data diagrams (PDDs) [14]. They integrate the complex development process with the development products, typically models, documents, and code. The process part is represented using an extension of UML activity diagrams, while the product part is represented as a UML class diagram, utilizing the part-of construct to represent document or model composition. Further information about the method fragment, such as motivation, goals of the method fragment, examples and literature, is textually represented in a Wiki style (http://www.cs.uu.nl/wiki/bin/view/MethodEngineering/).

The first goal of this paper is to investigate how PDDs can be represented in a deductive system that axiomatizes the re-combination of method fragments to larger fragments, and ultimately, to complete methods. The formalization yields

- rules for detecting incorrect re-combinations
- rules to detect unreachable method parts (not discussed here)
- rules to detect unstructured writes to the product side

The second goal is to investigate to what extent the PDDs are capable of supporting the traceability of executions of assembled method fragments. We observe that the process part of PDDs itself is a model subject to instantiation and discuss possible extensions to PDDs to allow a limited, but still useful, form of traceability.

The paper is organized as follows. The subsequent section 2 introduces the constructs of PDDs using an example. Section 3 then relates PDDs to the standard abstraction levels of metamodeling. In section 4, PDDs are are formalized using the deductive ConceptBase metamodeling environment [8]. Finally, section 5 relates the structure of the process part structure of the product part of PDDs, and section 6 interprets the execution of a PDD (i.e. process trace) as an instance of the PDD model.

2 Constructs of Process-Data Diagrams

A PDD provides constructs to denote processes similar to UML activity diagrams, constructs to denote the deliverables and data using a variant of UML class diagrams, and a link construct to combine the two sides.

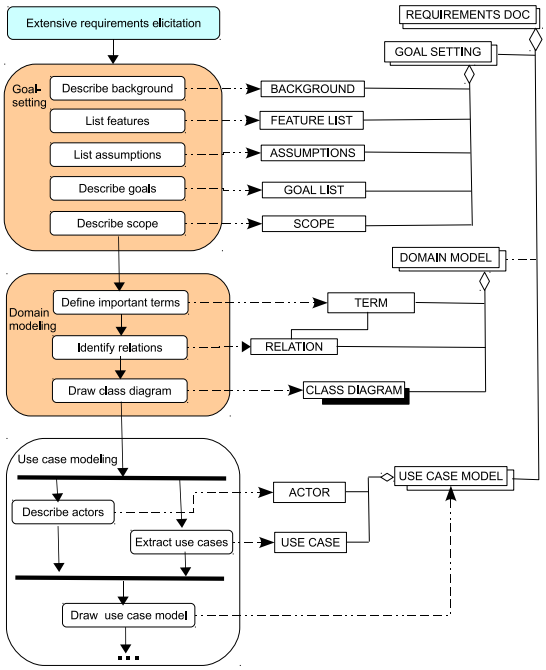

Fig. 1. Example PDD of the Web Engineering Method (excerpted from [15])

Figure 1 shows an example PDD. Activities like "Domain modeling" can have sub-activities like "Identify relations". Activities can also be associated with agents who perform them (not shown in the figure). Activities exist at various aggregations levels: whole projects, phases, larger activities and individual steps. PDDs consider two types of complex activities. Open activities have explicit sub-activities, and closed activities have sub-activities, but they are not made explicit. Activities are routed via decision nodes ("if then else") and parallel splits. There are also parallel joins, all denoted with UML activity diagrams.

The product part of a PDD is a UML class diagram hierarchically organized via composition associations. Open complex concepts are explicitly decomposed into parts, while closed complex concepts are known to consist of parts but the parts are not shown. The decomposition can be down to individual model elements such as an individual actor in a use case diagram. The hierarchical structure of the product part resembles the hierarchical decomposition of activities into sub-activities. However, there is no strict rule that elements of the process part are matched to elements to the product part that have the same decomposition level, e.g. whole methods matched to the top concept in the hierarchy of data concepts. It is assumed – though not enforced – that the process part of a PDD has a unique start and a unique end. The process and product parts are connected by an output link (dashed arrow in fig. 1).

Method fragments are stored in a method base, for example the Complete Definition Phase method fragment of fig. 1 [14, 15]. We shall refer to the method fragment by its name and note that a method fragment is a certain aggregation of an activity, typically covering a phase. The goal of section 4 shall be a logic-based reconstruction of PDDs that allows to formalize syntactic correctness rules for PDDs. The formalization shall also support the automatic recombination of method fragments and a simple form of traceability of a method execution. Specifically, we aim for the following properties:

1. If a method fragment A is defined to be followed by method fragment B then the last activity of method fragment A is followed by the first activity of method fragment B. Note that these two activities can themselves be decomposed. The composition rule then applies to their sub-activities as well.
2. Unstructured writing to data elements should be detectable, i.e. if phase A writes to data elements that are grouped with a complex data element DA, and phase B writes to data elements that are grouped with a complex data element DB, then there should be no activity of A that writes to elements of DB.
3. The origin of actual data elements, i.e. instances of the data element types specified in a PDD should be traceable, i.e. which other data elements were needed in order to produce this data element.

The last function is refers to the execution of a method rather than to its definition in the PDD. We shall introduce the notion of execution as a simple instantiation of a method specified in a PDD.

3 PDDs versus Metamodeling

PDDs combine a product part and a process part. They are in fact workflow models that include the products of the activities inside the workflow model. Still, there is a special clue with PDDs: the product of the activities are usually models, such as a use case diagram. Before formalizing PDDs, we have to understand the abstraction level [6, 10] of PDD elements. Subsequently, we use the abbreviations M3 (metametaclass level), M2 (metaclass level), M1 (class level) and M0 (data/execution level) as explained in [2]. Consider the following three statements:

M0/M0 *Bill changes the delivery address of order 453 to "Highstreet 3".*
M0/M1 *Mary defines ORDER as an entity type within ERD-12.*
M0/M2 *Peter proposes EntityType as modeling construct.*

All three statements are about some process executions involving some products. The first statement is a typical element of a business process trace. The products are data elements (abstraction level M0). The trace statement itself is also at M0 level. The second statement is from a modeling activity. The products are model elements (abstraction level M1), but the trace statement itself cannot be further instantiated: it is at the M0 level. Finally, the product part of the third statement is at M2 level, while the statement itself is at M0 level, since the object 'Peter' cannot be further instantiated. The examples show that the abstraction level of the product part characterizes the nature of the process, i.e. whether it is a business process, a modeling process, or a metamodeling process.

The three statements are all excerpts from an execution of a process. The process definition is one abstraction level higher for both the process and the product parts:

M1/M1 *A customer changes the delivery address of an order to a new value.*
M1/M2 *A data modeler defines entity types in entity-relatiship diagrams.*
M1/M3 *A metamodeler proposes constructs of modeling languages.*

PDDs as a notation could represent all three flavors of statements, i.e. the process part of PDDs are always at M1 level and the product part is either M1, M2, or M3. Since method engineers design the workflow models for modelers, a typical PDD is at M1 level for the process part and at M2 level for the product part. Figure 2 puts both the process part (left) and the product part (right) into this MOF perspective. Example PDDs are at M1 level, the data element that they produce are at M2 level.

The OMG-style use of metamodeling strictly separates abstraction levels: they may only be connected via instantiation. The PDD case shows that this strict separation prohibits combined process and product models targeted to method engineering processes. We can still stick to the abstraction levels and the instantiation link between them when regarding only the product part or only the process part.

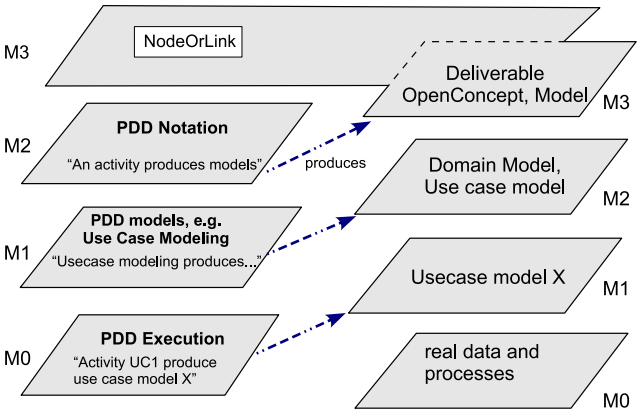

Fig. 2. Putting PDDs into a metamodeling perspective

Another concern for formalizing PDDs is the specification language. As argued, a strict use of MOF leads to a violation of the instantiation rule. The object constraint language (OCL) builds upon the separation of class and instance level. Indeed, one OCL constraint can only link two level pairs [1] and it lacks a fixpoint semantics to follow transitive links in cyclic graphs. We shall therefore use a deductive formalization[1].

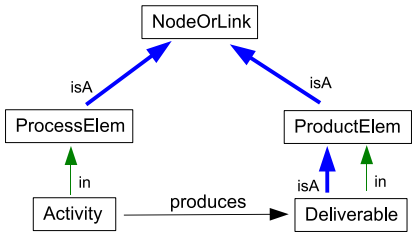

Fig. 3. M3 level for product and process parts

Figure 3 defines the new combined M3 level that can cover both the product and process parts of PDDs. Note that `Deliverable` is both specialization and an instance of `ProductElem`, which itself is a specialization of `NodeOrLink` – the most generic construct of the M3 model used in this paper. Consequently, `Deliverable` can be regarded both as a M3 and M2 object. On the left-hand

[1] Gogolla et al. [5] proposed to represent all abstraction levels into a single instance level and use a generic class level that basically supports the representation of graphs. This representation would consequently allow a use of OCL that is not restricted to just a level pair like M1-M0, M2-M1 etc. As the class level would not contain specific classes, the OCL constraints would be rather complex.

side, `Activity` is an M2 object because it is an instance of the M3 object `ProcessElem`.

4 Deductive Formalization

We use the capabilities of Telos [9] and its implementation in ConceptBase to logically reconstruct the PDD notation and axiomatize its syntax and part of its semantics. ConceptBase implements a dialect of Telos via Datalog-neg, i.e. Horn clauses without function symbols and with stratified negation as failure. This interpretation of a Datalog-neg theory is efficiently computable. We use the following predicates in our formalization:

(x in c) the object x is an instance of the object c, also called the class of x;
(c isA d) the object c is a specialization of object d;
(x m/n y) the object x is associated to object y via a link labeled n; this link has the category m.

Deductive rules are formulated on top of these three predicates, deriving further facts of these predicates. One single base predicate P(o,x,n,y) provides the base solutions for the three predicates [6]. We subsequently formalize PDDs in the frame syntax that aggregates facts of the above three predicates into a textual frame. We use the MOF/MDA abstraction levels in comments to improve readability of the formalizations. They are not part of the formalization. Most of the subsequent formalization is about the structure of PDDs and is represented by facts of the three predicates.

4.1 The Product Part in ConceptBase

The product part of fig. 1 lists models and model elements that are at the M2 MOF level. Hence, to formalize that part, we need to specify its constructs at the M3 level. We formulate it as a specialization of the basic M3 level used in [6].

Constructs of the Product Part of PDDs (M3)

```
NodeOrLink with    {* = (NodeOrLink attribute/connectedTo NodeOrLink) *}
   attribute
    connectedTo: NodeOrLink
end
Node isA NodeOrLink end    {* = (Node isA NodeOrLink) *}
NodeOrLink!connectedTo isA NodeOrLink end
Model isA Node with
   attribute
      contains: NodeOrLink
end
ProcessElem isA NodeOrLink end
ProductElem isA NodeOrLink end
```

```
Deliverable in ProductElem isA ProductElem end
Concept isA Deliverable end
StandardConcept isA Concept end
OpenConcept isA Concept,Model with
  attribute
     contains: Deliverable
end
ClosedConcept isA Concept,Model end
DocumentDeliverable isA OpenConcept end
ModelDeliverable isA OpenConcept end
```

The first constructs are standard constructs for the M3 level: `NodeOrLink` for model elements that are aggregated into models. The second half states that PDD product elements are 'deliverables'. Open concepts are concepts that have other deliverables as parts. The constructs `DocumentDeliverable` and `ModelDeliverable` are introduced to distinguish textual deliverables (e.g. reports) from model deliverables composed of diagrams.

4.2 The Process Part in ConceptBase

The process part in the example of figure 1 is at MOF/MDA M1 level because it can only be instantiated once: its actual execution in the context of some project. Hence, the constructs of the process part to denote such examples are at the M2 level:

Constructs of the Process Part of PDDs (M2)

```
ActivityNode in Node,ProcessElem with
  connectedTo
     next: ActivityNode
end
ActivityDiagram in Model,Class isA Activity with
  contains
     activity: ActivityNode;
     control: ActivityNode!next
end
Phase in Model isA ActivityDiagram end
PDD in Model isA Phase end
PDDLibrary in Model isA PDD end
Agent in connectedTo end
Activity in ProcessElem isA ActivityNode with
  connectedTo
     produces: Deliverable;
     performer: Agent
end
ParallelBranch in ProcessElem isA Activity with
  connectedTo
     branch: ActivityNode
end
ParallelBranch!branch isA ActivityNode!next end
```

```
ParallelJoin in Node isA Activity end
DecisionPoint in ProcessElem isA Activity with
  connectedTo choice: ActivityNode
end
DecisionPoint!choice isA ActivityNode!next end
DecisionJoin in Node isA Activity end
```

Basically, the above definitions are UML activity diagrams augmented with certain extensions for PDDs. Agents are introduced as performers of activities. The control structure of activity diagrams is expressed by the `next` construct of activity nodes (standing for an activity at any aggregation level). We refer to such a link by an expression `ActivityNode!next`. The `produces` construct of `Activity` establishes the link to the data part of PDDs, i.e. the arrows with broken lines in fig. 1.

4.3 Definition of PDD Combination

PDDs follow syntactic rules such as that all activities in the process part must be on the path from the start activity to the end activity (compare also workflow models as presented in [16]). They have a certain semantics such as about the composition of PDDs (method fragments) to larger PDDs or methods. Subsequently we consider our first challenge from section 1: if two PDDs are combined then the combination is inherited downwards to the end and start activities of the participating PDDs. To realize this property, we assume that the basic properties of relations such as transitivity, reflexivity, symmetry etc. are already provided by the ConceptBase system. See [7] for for details. Given these definitions, we specify:

Deductive rules for combining PDDs

```
ActivityDiagram in Model,Class isA Activity with
  reflexive,attribute
     subactivity: ActivityNode
  rule  t1: $ forall ad/ActivityDiagram a/ActivityNode (ad activity a)
                  ==> (ad subactivity a) $;
     t2: $ forall ad1,ad2/ActivityDiagram a/ActivityNode (ad1 activity ad2)
                 and (ad2 subactivity a) ==> (ad1 subactivity a) $
end
StartNode in GenericQueryClass isA ActivityNode with
   parameter,computed_attribute
      diagram: ActivityNode
   constraint isStart: $ ((diagram in ActivityDiagram) and
      Adot(ActivityDiagram!activity,diagram,this) and
      not exists a/ActivityNode
            Adot(ActivityDiagram!activity,diagram,a) and
              (a \= this) and :(a next this):) or
               (not (diagram in ComplexActivity)) and (this=diagram) $
end
```

```
Activity in Class with
  rule d1: $ forall a1/ClosedActivity a2/ComplexActivity s/ActivityNode
              (a1 next a2) and (s in StartNode[a2])  ==> (a1 next s) $;
       d2: $ forall a1/ComplexActivity a2/ClosedActivity e/ActivityNode
              (a1 next a2) and (e in EndNode[a1]) ==> (e next a2) $;
       d3: $ forall a1,a2/ComplexActivity e,s/ActivityNode
              (a1 next a2) and (e in EndNode[a1]) and
              (s in StartNode[a2]) ==> (e next s) $
end
```

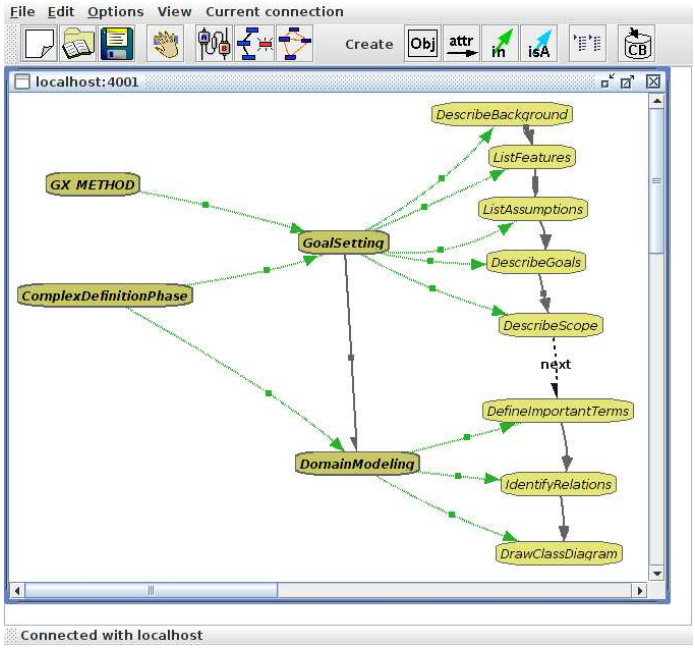

Fig. 4. Combining two PDDs (ConceptBase screenshot)

The concepts `StartNode` and `EndNode`[2] define the first and last activity of a PDD. We also support single activities as (degenerated) PDDs, that are the start and end node of themselves. The main logic is in the deductive rules d1 to d3. The first two special cases are for PDDs that are closed activities. Rule d3 is the general case which takes care that the `next` link is propagated downwards to the start/end nodes. Figure 4 shows a screenshot of an application of the rules. The example is taken from [15] and shows the combination of two PDDs for a Web Engineering Method. The dotted link marked 'next' is inherited via rule d3.

[2] The concept `EndNode` is defined analogously to `StartNode`.

The activity `DescribeScope` is the end activity of `GoalSetting`. The links from left to right are denoting sub-activities. The activity `DefineImportant-Terms` is the first activity of `DomainModeling`. We state `(GoalSetting next DomainModeling)` denoted by the vertical link between the two. This leads to the deduction of the link `(DescribeScope next DefineImportantTerms)`. If `DescribeScope` and/or `DefineImportantTerms` were complex activities themselves, then the `'next'` link would be inherited downwards to their start/end activities. Fig. 4 also displays two complex activities `GX-Method` and `Complex-DefinitionPhase`. Here `GX-Method` stands for a library of reusable PDDs and `ComplexDefinitionPhase` is one phase of the target web engineering method.

5 Detecting Unstructured Data Production

The deductive formalization of PDDs allows to detect certain unstructured accesses from activities to complex data elements. Unstructured writes are characterized by a pattern with two phases that both include activities which write into parts of the same model deliverable aggregating smaller deliverables.

Definition of Unstructured Writing

```
CrossWrittenDeliverable in QueryClass isA ModelDeliverable with
    computed_attribute
       crosswriter: Activity
    constraint
      crossCond: $ exists phase1,phase2/Phase d1,d2/Deliverable
           writer/Activity  (phase1 \= phase2) and
           (phase1 activity writer) and (phase2 activity crosswriter) and
           (writer produces d1) and (crosswriter produces d2) and
           (this contains d1) and (this contains d2) $
end
```

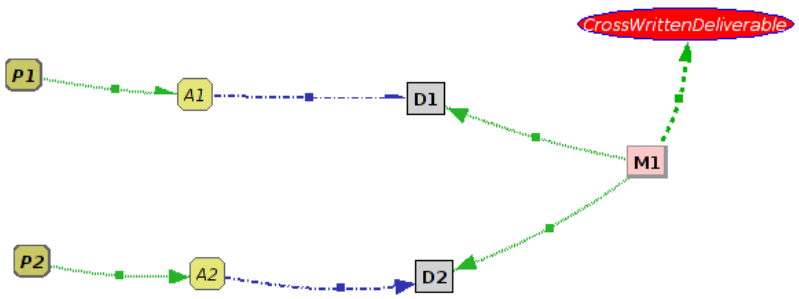

Fig. 5. Cross-written deliverables (screenshot from ConceptBase)

The above query class is returning all model deliverables that are written into by different phases. Hence, in structured PDDs, a phase may not write into

a model deliverable that is also written into by another phase. One can argue that this should not always forbidden. Indeed, the formulation as a query class allows a modeler to tolerate violations but still expose them via the query.

Figure 5 shows a generic example of an unstructured writing. The broken links between the activities A1 and A2 and the deliverables D1 and D2 are 'produces' associations. So logically, we have `(A1 produces D1)` and `(A2 produces D2)`. The model deliverable M1 is exposed as instance of `CrossWrittenDeliverable` (oval node in fig. 5).

6 Realizing Traceability

We observed in section 3 that the product part of PDDs is at M2 level, while the process part is at M1 level. We can instantiate both to yield an actual trace of the execution of the process part (M0) linked to data elements at the M1 level. This is a natural relation since modeling is an activity that creates models rather than data from the reality, see also fig. 2.

In the same way the example PDDs are classified into the PDD Process Notation, we can also instantiate them to form a process trace (M0). On the product part, the corresponding instantiation is from a model type (M2) to an example model (M1), e.g. a specific use case diagram. The existing PDD notation only specifies which activity has *produced* a certain product, e.g. a model. It does not specify which products were *required* in order to create it. We extend the PDD notation to include this "input" link as follows:

Extending the PDD (M2)

```
Activity in ProcessElem isA ActivityNode with
  attribute
     retrieves: Deliverable;
     produces: Deliverable;
     performer: Agent
end
```

There is just one additional `retrieves` attribute of `Activity`. The augmented definition now allows us to define coarse-grain traceability on the level of deliverables:

Simple Traceability model (M3-M1)

```
Deliverable in Class with
  rule dr1: $ forall D/Deliverable d/VAR
              (d in D)  ==> (d in DeliverableInstance) $
end
Activity in Class with rule
  ar1: $ forall A/Activity a/VAR (a in A) ==> (d in ActivityInstance) $
end
ActivityInstance in Activity end
```

```
DeliverableInstance in Class with
  attribute
    depOnDirectly: DeliverableInstance;
    depOn: DeliverableInstance
  rule
    depRule1: $ forall d1,d2/DeliverableInstance a/ActivityInstance
      (a [retrieves] d1) and (a [produces] d2) ==> (d2 depOnDirectly d1) $;
    depRule2: $ forall d1,d2,d3/DeliverableInstance
       (d1 depOnDirectly d2) and (d2 depOn d3) ==> (d1 depOn d3) $
end
```

The construct `Deliverable` is at the M3 level. However, we are interested in traceability at the level of example deliverables (M1) such as an example use case model X. To do so, rules dr1 and ar1 ensure that any M1 deliverable is also an instance of `DeliverableInstance`, and that any M0 activity instance is an instance of M1 `ActivityInstance`. This axiomatization allows us to realize traceability *regardless* of the specific PDDs in our library. The rules work with all PDDs.

7 Conclusions

This paper applies a deductive metamodeling approach to the the PDD notation used to represent method fragments. We found that the challenges mentioned in the introduction can be addressed rather easily. The main result is a different one: the rule that only allows instantiation links between abstraction levels is too strict and prevents a proper formalization of PDDs and similar techniques. The rule is neither necessary nor useful. There are some open problems and shortcomings:

- The combination of method fragments fails if there is not a unique start and end activity of the participating method fragments. One may want to support multiple such activities and combine them with decision points (if-then-else) or parallel branches/joins.
- The traceability model neglects the decomposition of deliverables. If a part of a deliverable depends on some part of some other deliverable, then the aggregated deliverables should also depend on each other.
- The formalization is represented by a deductive database, more precisely Datalog with negation. The fixpoint semantics compute the unique minimal Herbrand interpretation under closed-world assumption. This allows direct implementation and use of the formalization but is weaker than a full first-order logic specification.

The formalization is embedded into an existing M3 model. Analysis techniques developed for that M3 model are directly applicable, for example the analysis of connectivity between model elements. The integration with Graphviz allows us to generate diagrams with a reasonable layout (see appendix) from

the PDD represesentation in ConceptBase. The re-combination of PDDs is governed by deductive rules that automatically connected the correct ends of the participating PDDs, even if they are deeply decomposed.

Acknowledgments. This paper has been motivated by a challenge formulated by Inge van Weerd when she gave a guest lecture on PDDs in the method engineering course in Tilburg.

References

1. Baar, T.: The definition of transitive closure with OCL - limitations and applications. In: Broy, M., Zamulin, A.V. (eds.) Ershov Memorial Conference. Lecture Notes in Computer Science, vol. 2890, pp. 358–365. Springer (2003)
2. Bézivin, J., Gerbé, O.: Towards a precise definition of the OMG/MDA framework. In: ASE-2001. pp. 273–280. IEEE Computer Society (2001)
3. Brinkkemper, S.: Method engineering: engineering of information systems development methods and tools. Information & Software Technology 38(4), 275–280 (1996)
4. Brinkkemper, S., Saeki, M., Harmsen, F.: Assembly techniques for method engineering. In: Pernici, B., Thanos, C. (eds.) CAiSE. Lecture Notes in Computer Science, vol. 1413, pp. 381–400. Springer (1998)
5. Gogolla, M., Favre, J.M., Büttner, F.: On squeezing M0, M1, M2, and M3 into a single object diagram. In: Proceedings Tool-Support for OCL and Related Formalisms - Needs and Trends (2005)
6. ISO: ISO/IEC 10027: Information technology - information resource dictionary system (irds) - framework (1990), `http://www.iso.org/iso/catalogue_detail.htm?csnumber=17985`
7. Jeusfeld, M.A.: Partial evaluation in meta modeling. In: Ralyté et al. [11], pp. 115–129
8. Jeusfeld, M.A.: Metamodeling and method engineering with ConceptBase. In: Metamodeling for Method Engineering, pp. 89–168. The MIT Press (2009)
9. Mylopoulos, J., Borgida, A., Jarke, M., Koubarakis, M.: Telos: Representing knowledge about information systems. ACM Trans. Inf. Syst. 8(4), 325–362 (1990)
10. Object Management Group: Meta object facility (mof) core specification (2006), `http://www.omg.org/spec/MOF/2.0/PDF/`
11. Ralyté, J., Brinkkemper, S., Henderson-Sellers, B. (eds.): Situational Method Engineering: Fundamentals and Experiences, Proceedings of the IFIP WG 8.1 Working Conference, 12-14 September 2007, Geneva, Switzerland, IFIP, vol. 244. Springer (2007)
12. Ralyté, J., Rolland, C.: An assembly process model for method engineering. In: Dittrich, K.R., Geppert, A., Norrie, M.C. (eds.) CAiSE. Lecture Notes in Computer Science, vol. 2068, pp. 267–283. Springer (2001)
13. Rolland, C.: Method engineering: Trends and challenges. In: Ralyté et al. [11], p. 6
14. van den Weerd, I.: Advancing in Software Product Management - A Method Engineering Approach. Ph.D. thesis, Utrecht University (2009)
15. van den Weerd, I.: Guest lecture on meta-modeling for method engineering (2010)
16. van der Aalst, W.M.P., van Hee, K.M.: Workflow Management: Models, Methods, and Systems. MIT Press (2002)

Appendix: Graphviz Visualization

The PDD visual notation can be approximated by converting the PDD representation of ConceptBase into a format that can be processed by Graphviz[3]. Figures 6 and 7 show two example PDDs excerpted from ConceptBase and layed out by Graphviz. Figure 8 aggregates them with others to a whole phase.

The complete specification of the formalization including the Graphviz integration is available on `http://merkur.informatik.rwth-aachen.de/pub/bscw.cgi/3045636`. It contains also a couple of additional analysis queries that were not included in this paper due to space limitations.

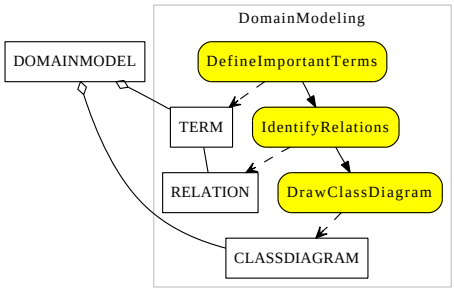

Fig. 6. Domain Modeling PDD layed out by Graphviz

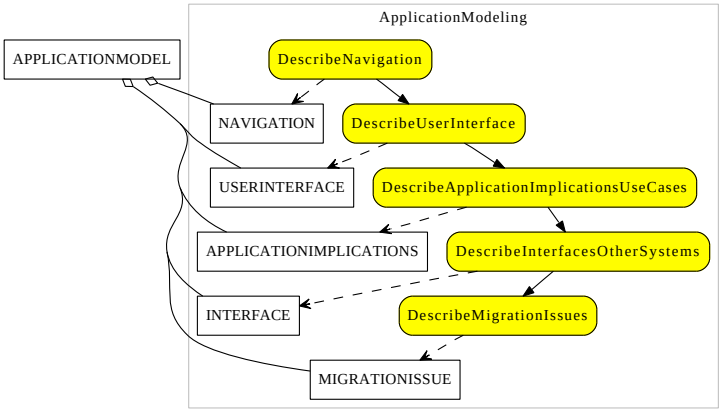

Fig. 7. Application Modeling PDD layed out by Graphviz

[3] See `http://graphviz.org`

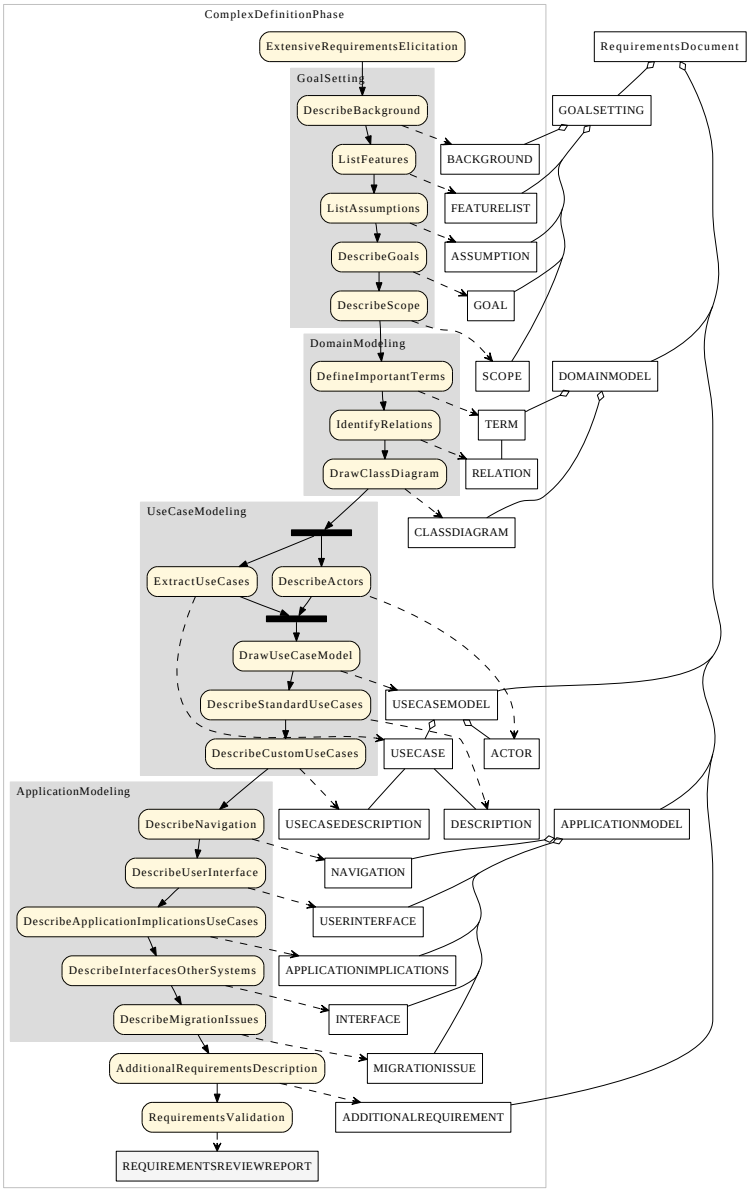

Fig. 8. Graphviz visualization CompleteDefinitionPhase