Ontological Support for Consistency Checking of Engineering Design Workflows

Franz Maier, Wolfgang Mayer and Markus Stumptner

Advanced Computing Research Centre,
University of South Australia,
Mawson Lakes Campus, Mawson Lakes Boulevard,
Mawson Lakes, SA 5095,
Email: franz.maier|wolfgang.mayer|mst@cs.unisa.edu.au

Abstract

In this paper we describe a novel approach for engineering process representation based on the application of formal ontologies. We illustrate components of a framework that comprises domain abstractions, design interfaces and meta-data in the engineering design domain in the form of process-, artefact- and task ontologies. A concrete application and use case of the framework components is detailed in order to demonstrate the capabilities for representation of design processes on a high level of abstraction. The realm of *Planning* is employed to showcase workflow decomposition. We show, how the framework can be applied to perform workflow consistency checking and point out some scenarios where the developed framework can provide support in the task of engineering design analysis and improvement.

Keywords: process ontology, engineering design, process integration, planning, process modelling and decomposition, consistency checking.

1 Introduction

Data and processes in engineering design environments need to be formalised in an unambiguous way. High level design interfaces and design meta-data are not sufficiently integrated in the daily routine of execution and composition of design workflows (Maier et al., 2008). Ontologies of design knowledge are already part of manufacturing environments with examples being ontology supported checking of consistency requirements of design aspects (Tudorache, 2008), capturing of design rationales (Burge, 2006), feature-based process planning (Jiang and Liu, 2008) and manufacturing planning (Borgo and Leitao, 2007). Further, ontologies can be applied to represent an intermediate language between engineering applications, comprising applications for product lifecycle management (Fenves et al., 2003), enterprise resource planning systems (Borgo and Leitao, 2007) and computer-aided design (Andersen and Vasilakis, 2007).

The major contribution of this paper is the demonstration of a use case that details how the introduced ontology framework can be used to support workflow consistency checking. As a means of process decomposition we will introduce a tool from the planning realm that allows us to perform a refinement and grounding of abstract engineering processes. The use case also shows how historic process execution traces can be utilised to construct new design scenarios. The paper is structured in the following way: first we introduce the general architecture of the framework and briefly comment on the role of each framework layer. Then we explain in detail how each of the framework components have been realised and how layers are linked with each other and comment on the realisation of data- and control- flow. A grounding of the process model will be provided and it is illustrated how existing engineering ontologies can be utilised by the framework. In the second part of the paper we demonstrate the application of the framework concepts in the planning domain and introduce details of our model. Finally, a use case is introduced that illustrates how workflow consistency checking is supported by the framework.

1.1 Research Context

The integration approach was developed in the context of Multidisciplinary Design Optimisation (MDO) in an automotive design environment. In MDO, a design engineer analyses all investigated design disciplines in parallel, instead of optimising each domain separately. The results of this process are prioritised with the intent to obtain the best design alternative as a compromise of all included disciplines. As an example, an automotive design scenario could include disciplines such as noise, vibration, harshness, driving dynamics, fluid dynamics and structural statics in a single MDO experiment. MDO is typically carried out after a basic design of a vehicle is constructed and will be iteratively applied in the early stages of the vehicle design process. As a general context, the trend toward increased virtualisation of the product development process has to be considered as well as the requirement to represent all artefacts of the design optimisation process in virtual form. In the engineering design context, ontologies support the semantic interoperability between participating design disciplines due to the application of meta-models that connect between disciplines. Semantic-preserving translations between design disciplines must be preserved and abstractions of design constraints and simulation results must be mapped with each other.

1.2 Abstract Workflow Refinement

Using automated reasoning technology, process models can automatically be translated into workflow enactment models that are executed. This enables the designer to automatically track, store, and reason about process outcomes that are handled by the MDO

environment. Through common task and artefact ontologies and adapters, simulation inputs and results can be compared and possible changes to the process may be suggested and validated.

The conceptual representation of an MDO scenario allows us to create a workflow description that can subsequently be instantiated and executed. Before such a workflow template can be provided, it is necessary to develop a mapping between a high level conceptual view of the MDO process and concrete parameter bindings that are part of a workflow description. For instance a concept describing a part of the domain model has to be linked to concrete entities such as a geometry-model and meshing-model that can have concrete value assignments. This can be done by assigning each abstract process step to a range of domain concepts that are valid in a given design state. The knowledge about value ranges is derived from past MDO scenarios that could successfully be completed.

2 Related Work

Dartigues et al. investigate into semantic support of computer-aided design (CAD) and computer-aided process-planning (CAPP) activities. In addition to geometry-related product data, they employ a feature-based integration approach that relies on a shared ontology available in the Knowledge Interchange Format (KIF). Domain specific ontologies are developed after an analysis of participating engineering applications is performed. Subsequently, mapping rules are created to link domain specific ontologies to the shared ontology (Dartigues et al., 2007). Their approach focuses on semantically enhanced data exchange between engineering applications, where ontologies on *features* and *feature decomposition* are developed, supported for instance by a constraint classification mechanism and the identification of design specific classes. In contrast to our approach, a task ontology for typical engineering activities is not part of the shared ontology and process planning activities are mainly associated with the manufacturing model of an artefact.

Another related approach is based on a framework to negotiate ontology mediation dynamically (Kannengiesser and Gero, 2006), driven by a knowledge model rooted in the function-behaviour-structure ontology (Gero and Kannengiesser, 2007). In contrast, our work adheres to the knowledge intensive approach, where a mediating ontology is constructed manually.

Kitamura and Mizoguchi provide a framework to systematise functional knowledge of devices in an engineering context, resulting in a comprehensive *device ontology* (Kitamura and Mizoguchi, 2004). The framework has initially been applied to the mechanical domain and its utilisation is subsequently expanded to other application domains. It enables knowledge sharing among engineering artefacts and ensures their respective interoperability. Their major contribution is the provision of a *functional ontology* for *devices*. Although the framework only covers a subset of a complete behavioural model for engineering artefacts, the dimensions that are introduced for classification of functional knowledge can be reused in the engineering design domain. The framework does not include task ontologies for typical engineering activities, however, these can be found in earlier works (Seta et al., 1998) of the same authors.

The Process Specification Language (PSL) (Grüninger et al., 2006) was designed as formal 'interlingua' that captures fundamental concepts of manufacturing processes in order to mediate between process representation formats such that the semantics of interrelated representation formalisms is preserved. We utilise PSL as a formal basis to reason about task ontologies and execution traces, but specialise the ontology to aspects specific to MDO. While PSL focuses on processes, standardised representations of product models such as STEP (International Organization for Standardization, 1994), the Core Product Model and Open Assembly Model (Rachuri et al., 2005) have also been proposed as unifying ontologies. Further, extensions of ISO 10303 (STEP) to represent analysis-driven design activities are the focus of current research activities (Maier and Stumptner, 2007). Here, we extend the approach to integrate artefact and process models into a unified framework.

3 Framework Architecture

For a detailed elaboration of the architecture and comprised components, see (Maier et al., 2008). The approach taken adapts a layered architecture where the MDO meta-model is located at the top, task and artefact ontologies comprise the intermediate layer, and domain-specific ontologies form the bottom layer in the ontology hierarchy.

Concrete executable systems, such as CAD environments, optimisation tools and workflow engines are located below the knowledge representation layers. From analysis of individual domains, ontologies of domain-specific concepts, properties and relations are created, as well as specifications of domain-specific analysis primitives. Process execution environments, for example workflow enactment systems, are treated in the same way. As a result, a set of domain ontologies is obtained. Domain-independent aspects and processes are found by generalisation of domain-specific ontologies to form the intermediate layer. By defining suitable ontology mappings, specific knowledge is mapped into the unified task and artefact-specific ontologies at the intermediate level. Established engineering practices and processes may also be incorporated into the MDO task ontology. Similarly, a generalised artefact ontology is constructed from the artefact domain ontologies.

Task and artefact ontologies at the intermediate layer must conform to the meta-models in the upper layer. The common meta-model allows us to describe and reason about domain-independent and task-independent concepts, such as execution traces and execution histories. Further, task and artefact ontologies need not be expressed in the same formal framework and may need to be reconciled to obtain an inference framework that can handle mixed expressions.

4 Framework Utilisation

The general role of the outlined framework can be summarised in the following way:

- Ensuring interoperability of engineering tools by provision of common terminology employed to describe typical tasks, problem solving methods and artefacts. The MDO meta-model can be used as interchange format for operative applications.
- Enabling communication between knowledge engineers, design engineers and further stakehold-

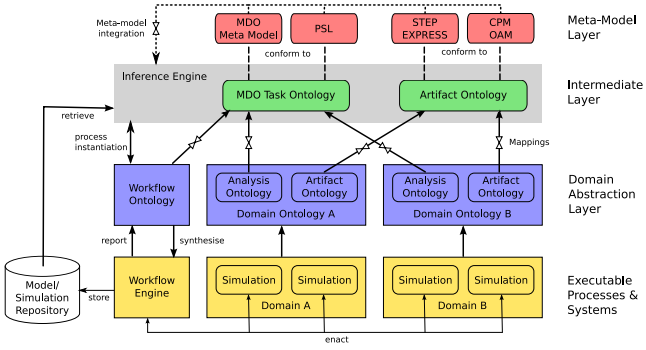

Figure 1: Ontology Architecture

ers of the design process and represents common terminology and language.

- When developing an application in the design environment, concepts can be reused and the development activities can build on existing domain concepts. Reasoning capabilities provided by the framework can support the extension of existing domain concepts.

Beyond general framework roles, we defined concrete use cases that utilise defined high level concepts and their interrelations. In other words, the MDO meta-model represents the basis for more specific ontologies that are required to realise a particular application. Therefore, in the general case, we first have to extend the MDO meta-model by the concepts required for use case realisation before commencing its implementation.

5 Realisation of Framework Components

The structural part of the realised ontologies comprised in the framework is presented here as UML models that define high-level concepts of the MDO domain. We also implemented some of the concepts in *Ontolingua* as a shared ontology available for integration of engineering design concepts. UML can be employed for ontology development (Cranefield and Purvis, 1999) and is able to capture the concepts on the meta-model layer. In the planning use case, explained later in this paper, the MDO meta-models depicted in UML support the decomposition of abstract tasks into executable process instances.

5.1 Domain Model

MDO domain concepts in combination with the MDO meta model represent the core of the ontology framework. In what follows, we introduce concepts from the domain ontology as depicted in Figure 13, shown in the appendix of this paper. The most relevant concepts illustrated in the domain model are elaborated below.

Concept *artefact* is the anchor for a majority of included concepts. In contrast to the CPM ontology (Rachuri et al., 2005), where all aspects of a design artefact are represented, we mainly focus on function and form of an artefact and do not represent behaviour and features of an artefact. The form of an *artefact* represents a potential design solution intended as a result of a simulation scenario. This is in accordance with existing work (Regli et al., 2009) in modelling of the engineering design domain.

Besides the reference to its *format*, an *artefact* includes concepts to describe its *construction* and *configuration* process, the former referring to the creation of an *artefact* achieved for instance via *source-code*, the latter addressing parametrisation of an existing *artefact*. These entities, not included in the top-level of CPM, are introduced due to the emphasis of the process view, inherent in our process meta-model. Both concepts have further sub-concepts such as *meshing-configuration* and *geometry-configuration* as well as *topology-construction* and *geometry-construction*, to make a functional distinction corresponding to a particular MDO task respectively.

The *tool* concept includes attributes as typical characteristics for manipulation of engineering models. Each tool comprises a set of *input* and *output* parameters, a set of accepted *format*s and *documentation* among other properties. Closer attention to the *tool* aspect is given in grounding and execution of a simulation scenario, where a realisation as *workflow engine* or *finite element analysis application* could be necessary.

In order to obtain a physical equivalent to the introduced abstract terms, each concept tagged with the stereotype *domain model* has a *Manifestation* that includes a *URI*—the pointer to the corresponding resource—as well as a *Format* and *Revision* for further specification. *Manifestation* is modelled as a *trait* (Schaerli et al., 2002) that represents an abstraction of properties that are common to multiple disjoint concepts of the domain model. A *trait* supports structuring of the conceptual model as it is a valuable means of concept reuse. The stereotype *domain model* is attached to those domain concepts that occur as input or output of MDO processes. *Domain model* is used in the meta-model layer as a means to establish the link between domain and process model.

Further, the concept *meta-data* provides additional information on an artefact beyond its physical realisation and in most cases is used in conjunction with concept *Manifestation*. Attributes such as *creation-date*, *creator*, *owner*, *comment* and *documentation* are included in the *meta-data* information. In many cases the introduced *artefact* concept will only appear in form of its constituents manifested as *domain models*. All concepts mentioned have further sub-concepts.

5.2 Linking MDO Domain Ontology Concepts with MDO Process Model

The MDO meta-model has its main focus on the process view of MDO. In order to cover the previously detailed design artefacts on the meta-model layer, additional modelling constructs are necessary.

Figure 2 shows the MDO meta-model extended with the domain conceptualisation, represented by the entity *DomainModel*. *DomainModel* has been used as a stereotype in the domain conceptualisation to label input and output parameters of processes. A *DomainModel*, a specialisation of entity *ParameterValue*, is linked to the process model via concept *DataFlow* that appears either as a source or as a target of a *Port*. All *DomainModel*s have a *Manifestation* assigned that points to its physical representation. A *ParameterValue* can be associated with a *Constraint* in a *Process* that represents either pre- or post-condition of a *Process*.

5.2.1 Control and data flow

Dataflow is realised in the process model via *Port*s that are source or target of a *DataFlow*. *DataFlow*

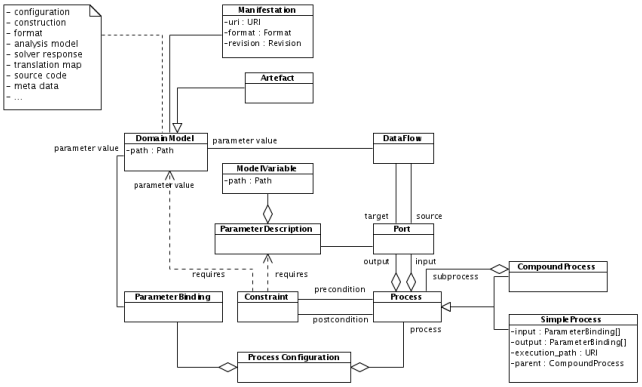

Figure 2: Extension of Process Meta-model with Domain Entities

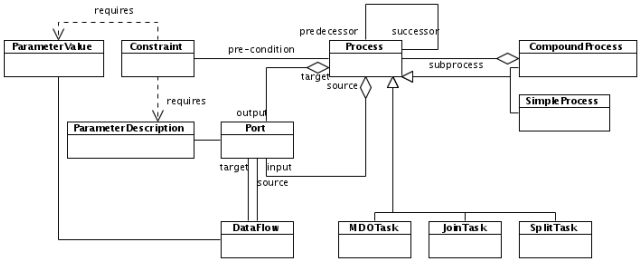

Figure 3: Control and Data Flow in MDO Process Model

represents data channels to route *ParameterValue*s as design artefacts and constraints to MDO processes. The previously described *DomainModel* can either represent primitive values such as strings and integers, or, sophisticated data as for instance an analysis-model or an assembly-model. The concrete realisation of *DataFlow* depends on the *grounding* of the meta-model and different means of distinguishing locations and versions of a *DomainModel* are subsequently added when concrete values are assigned to the abstract entities.
Figures 2 and 3 illustrate the participants necessary for realisation. The model includes process port numbers, labels and identifiers of processes as well as predecessor and successor of a MDO task. *Split* and *Join* tasks are used to model alternative branches in the control flow.

5.2.2 Realisation of Control Flow

Figure 3 shows a lightweight model of control flow to express the role of a *predecessor* and *successor* process and also includes specialisations of the general *Process* entity to distinguish between different types of tasks.

Figure 4 presents a sample overview scenario for the entities illustrated on the meta-model layer. A *MDOTask* and a *Join* process following as a *successor* to the executed *MDOTask*. In context of data flow aspects, addressed in Section 5.2.1, *Split* dispatches input values to MDO tasks and *Join* processes aggregate and forward output values from MDO tasks. In order to express control flow, we introduce a variable *control* that is communicated similarly to a regular data value and is passed on by a *Split* process either to the actual *MDOTask* or the corresponding *substitute process*. A *Join* process on the other hand receives *control* from either of those two processes and

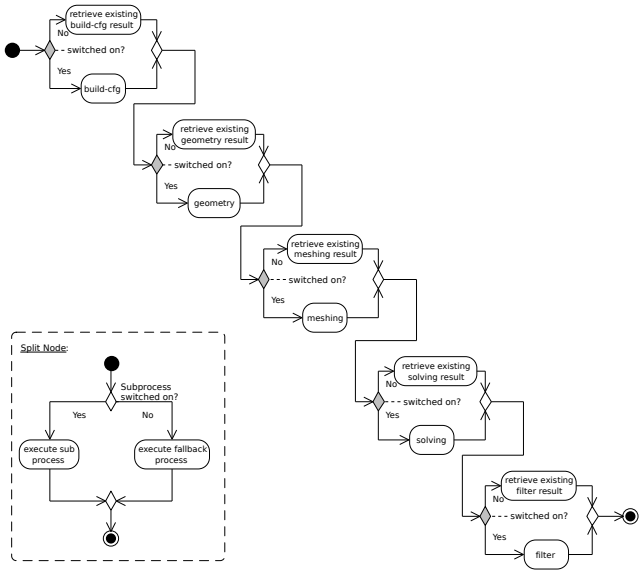

Figure 4: Schema of Control Flow Realisation

forwards it to the succeeding *Split* task. In which of these two possible directions a control variable is routed currently depends on a manual control the designer can manipulate. This dependency is modelled in Figure 4. However, the current realisation does allow knowledge engineers to introduce more sophisticated decision criteria.

5.2.3 Realisation of Grounding

Grounding of the MDO meta-model incorporates concepts required for instance to read and write input and output data of processes and transform abstract processes into executable workflows. To realise a *grounding* for MDO, an extension of the MDO meta-model is specified that assigns concrete values to a subset of the concepts presented on the meta-model layer including their respective slots. It therefore ascertains the execution of our abstract model by provision of the concepts that assure a formalisation of the process execution environment. Figure 5 shows the MDO meta-model including the link to the domain-ontology layer. Additionally, the *Manifestation* concept is further refined via its constituents *URI*, *Format* and *Revision*. The second concept directly associated with grounding is *MDOTool* that can represent for instance a concrete engineering application such as a finite element solver or a complete workflow execution engine. *MDOTool* inherits from *SimpleProcess* the entity that holds the necessary parameter for process execution including *ParameterBinding*s for input and output values, a *URI* denoting to the process execution path, and, a link to the parent process.

We separate *URI* into its constituents 'scheme', 'authority', 'path', 'query' and 'fragment'. This makes a constraint formulation possible that is based on URI attributes and serves the purpose of allocating process resources. The *URI* uniquely *locates* a resource such as a domain-model within a file—achieved via the 'fragment' attribute that is part of a query—and, secondly, *identifies* a resource by providing a unique name for it.

We can obtain a representation of an executable process model that is sufficiently specified for execution by a particular workflow engine, resulting in a workflow instance as for example represented in the

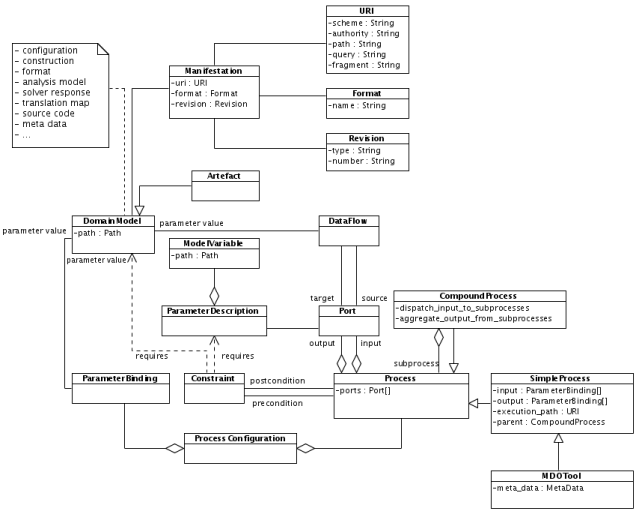

Figure 5: Grounding of MDO Meta Model Layer

Taverna workflow (Muehlenfeld et al., 2008). An executable model is obtained via grounding of domain abstraction layer, in conceptual terms, mapping the domain abstraction layer to the process execution layer.

5.3 Integration of Ontology Layers

Translation between the intermediate layer and the ontologies below is accomplished by adapters that map between domain-independent and domain-specific representations. The same idea is used to instantiate abstract process models at the intermediate level to *generate concrete process specifications* tailored to specific MDO environments that are subsequently enacted using a workflow system. In the context of *grounding* of MDO tasks we discussed adapters as a means of relating *Problem Solving Methods* to MDO tasks (Fensel, 1997), (Chandrasekaran et al., 1998). Here we will employ *adapters* as linking elements between ontologies on different levels of abstraction.

5.3.1 Linking Intermediate and Domain Abstraction Layer

As a means to realise the concept of *adapters* between intermediate and domain abstraction layer, a strategy can be followed that is based on subsumption relations between concepts from these two layers.

To illustrate this mapping mechanism, consider two representative domains such as meshing and solving that are both covered by the optimisation scenario. For the meshing domain we discovered concepts required for resource annotation such as *mesh-format*, *meshing-source-code*, *topology-construction* and *meshing-configuration*. Similarly, for the solving domain we identified domain terms as for instance *solving-format*, *solving-source-code*, *solving-construction* and *solving-parametrisation*. The *adapter* identified by the domain expert can be realised by a *generalisation* relation from concept *mesh-format* at the domain abstraction layer to the more general concept *format* at the intermediate layer, and, likewise, from *solving-format* valid in the solving domain to concept *format* realised on the intermediate layer. As a result, two disjoint concepts—*mesh-format* and *solving-format*—have been mapped via a *subsumption* relation to the concept *format* at the intermediate layer. This approach can be applied immediately to concepts that represent equivalent semantics and only differ in the domain covered, as it is the case for example for *mesh-format* and *solving-format*.

Domain expertise is required when semantically different concepts are to be mapped to a common subsuming concept as it is the case for instance in *meshing-configuration* and *solving-parametrisation*. In this case, a domain engineer has to support in the decision whether, firstly, these concepts can be unified with a common super-concept, and, secondly, what terminology should be used for the unifying concept. For the given example we decided to use the term *configuration* as a subsuming concept, as it was perceived as the more general concept.

In analogy to the adapter mechanism explained above, we can create adapters from a domain-specific MDO task to a generic MDO task on the intermediate layer.

Consider an example where a domain-specific solving task is detailed. The task is defined as a subtask of *Crashworthiness* and requires as input a *mesh* parameter-value and creates *result* as output value. To develop an adapter to the intermediate layer, we need to provide a definition of a finite element solver that is generic enough to represent all possible solver realisations.

The solving process itself is opaque to the user, only input variables can be changed in a great variety and in addition to *?mesh* provided as basic input, parameters such as material properties, section properties, joint stiffness, contacts, boundary conditions, restraints, load curves and further control parameters can be defined.

We define an input variable *?control-parameter* that acts as an aggregate for all possible parameters of a solver task. *?control-parameter* typically will itself represent a domain ontology that includes terminology as mentioned above. It is provided as additional variable of the solver task to complement *?mesh* and *?result*. To complete the generalisation of *LsDyna*, we rename the domain-specific task to *solver* and obtain a comprehensive definition of a generic MDO task allocated at the intermediate layer. As an equivalent to the MDO task presented here, an *ontology of a problem solver* (Chandrasekaran et al., 1998) can be defined where a *second level ontology* denotes to the equivalent of the domain abstraction layer, specifying the concepts defined in the *first level ontology*.

6 Framework Application

We will now outline how the introduced framework can be utilised to support the designer in execution and analysis of MDO scenarios. For this purpose we translate the concepts into a target environment to validate the parametrisation of a MDO process. We transfer framework concepts to the *Hierarchical Task Network* (HTN) target platform, in particular the HTN-like planner Shop2. We provide an implementation of an MDO sample scenario in Shop2 and decompose a high level formulation of a MDO workflow into executable process steps. Finally, we point out lessons learned that have to be considered when providing a mapping from MDO to an execution environment and make comments on formal semantics.

```
(:operator (!join ?predecessor ?current ?successor)
  ((process ?predecessor)(process ?current)(process ?successor))
  ((control-at ?current))
  ((control-at ?successor)))
)
```

Figure 6: Representation of a SimpleProcess in Shop2

6.1 Translation of MDO Framework Components to Shop2 Notation

We provide a mapping from MDO framework concepts to HTN form by detailing how each framework component can be realised in Shop2.

MDO SimpleProcess A *SimpleProcess* in MDO, for instance grounded as a *MDOTool*, will be represented as an *operator* in Shop2. Local preconditions of a *SimpleProcess* correspond to the *operator*'s preconditions, the *operator*'s add list corresponds to the output of a *SimpleProcess*. Input of a *SimpleProcess* is validated and accepted via Shop2 preconditions. All input and output objects of a *SimpleProcess* are available as facts of the knowledge base. Postconditions of a *SimpleProcess* are either part of the *operator*'s add- or delete-list. The link to a parent process is not directly established in Shop2, however, the parent process—a Shop2 *method*—includes a link to its sub-processes. An example for a *SimpleProcess* realised as *operator* is shown in Figure 6. The figure illustrates a simple join process, where the control token is passed between the current and the succeeding process. *SimpleProcess*(es) including their respective preconditions and output artefacts can be translated automatically to Shop2 notation.

MDO CompoundProcess *CompoundProcess*(es) in MDO correspond to *methods* in Shop2. For each *method* we define alternative branches comprising a list of *operator*s, *method*s and *predicate*s that are evaluated given the preceding set of preconditions is true. A *CompoundProcess* in MDO consists of one or more *SimpleProcess*(es) and *CompoundProcess*(es), and, is processed by executing all included *SimpleProcess*(es). In MDO, sub-processes of a *CompoundProcess* can either be executed in parallel or sequential. However, Shop2 does not directly support the notion of concurrency (Wu et al., 2003). We will neglect this fact for now, as the level of granularity modelled for the MDO simulation steps only considers sequential tasks.

The semantics of *method*s in Shop2 prescribes that exactly one branch, satisfying all pre-conditions, is executed. We explain the translation of this MDO component based on the example depicted in Figure 7: Figure 7 contains two branches, where *(trigger ?manual)* constitutes the precondition for the first branch and *(not(trigger ?manual))* represents the precondition of the second branch. If the first precondition is true, all tasks included in the tail of the branch—for instance *!split*, *!create-mesh-failover*, *!join*—are executed.

A *CompoundProcess* typically represents an algorithm or a procedure that comprises a number of dedicated steps to be completed in order to produce a particular result. This algorithm can be implemented in different domain-specific ways dependent on the given context. MDO domains are reflected

```
(:method (simulation-meshing-step ?control ?artefact ?process
    ?manual ?p1 ?p2 ?mesh-format ?mesh-construction
    ?topology-format)
    (trigger ?manual)
        ((!split ?p1 ?p2 ?control)
        (!create-mesh-failover ?mesh-format ?mesh-construction)
        (!join ?p1 ?p2 ?control))

    (not(trigger ?manual))
        ((!split ?p1 ?p2 ?control)
        (!assign-port ?p ?a)
        (!create-mesh ?mesh-format ?topology-format ?artefact)
        (!join ?p1 ?p2 ?control))
 )
```

Figure 7: CompoundProcess in Shop2

```
(:operator(!create-geometry...)
...
;add-list of operator
(depends-on ?geometry-construction ?geometry-source-code)
(depends-on ?geometry-format ?geometry-configuration)
(derived-from ?geometry-construction ?geometry-source-code)
(created-by ?geometry-format ?geometry-construction-tool)
(created-by ?geometry-construction
?geometry-construction-tool)
))
```

Figure 8: Local Constraints as Relation between Input and Output of a MDO Task

as different branches in a Shop2 *method* where exactly one branch—the one corresponding to the given context—is executed. A sequence of simple and further nested MDO tasks can be formulated as a list of *operator*s and *method*s contained in the tail of a particular branch. Automated translation of *CompoundProcess*es into this format can be supported.

Constraints A pivotal part in transferring the MDO domain representation to Shop2 is the translation of diverse requirements, assumptions and constraints identified in MDO and formulated as constraints. Figure 8 shows how local constraints of an MDOTask are expressed as preconditions of a Shop2 *operator*.

Figure 8 also includes constraints that relate input and output of an *operator*. Relations between input and output of tasks support the concept of traceability of MDO artefacts and processes by tracing of dependencies between input and output.

The predicates shown in Figure 8 are part of the add-list of an *operator*.
Further integrity constraints can be expressed through the formulation of axioms to define global restrictions all instances of the problem domain are bound to. Axioms in Shop2 are evaluated either as part of other axioms or in preconditions of Shop2 methods.

Ports Ports are represented in Shop2 by variable names defined to hold particular types of process input and output, typically corresponding to MDO artefacts passed to a process. They can validate process input and only variables of the defined types are accepted when starting the planning process.

```
(:method (mdo-workflow ?control ?process-id
    ?p1 ?p11 ?port1 ?art-1 ?res-1 ?aut-1
    ?p2 ?p22 ?port2 ?art-2 ?res-2 ?aut-2
    ?p3 ?p33 ?port3 ?art-3 ?res-3 ?aut-3
    ?p4 ?p44 ?port4 ?art-4 ?res-4 ?aut-4
    ?p5 ?p55 ?port5 ?art-5 ?res-5 ?aut-5)
    ()
    ((simulation-build-config-step
      ?control ?p1 ?p11 ?port1 ?art-1 ?res-1 ?aut-1 ?process-id)
    (simulation-create-geometry-step
      ?control ?p2 ?p22 ?port2 ?art-2 ?res-2 ?aut-2 ?process-id)
    (simulation-meshing-step
      ?control ?p3 ?p33 ?port3 ?art-3 ?res-3 ?aut-3 ?process-id)
    (simulation-solving-step
      ?control ?p4 ?p44 ?port4 ?art-4 ?res-4 ?aut-4 ?process-id)
    (simulation-filter-response-step
      ?control ?p5 ?p55 ?port5 ?art-5 ?res-5 ?aut-5 ?process-id)
))
```

Figure 9: MDO Workflow as Composite Process

```
(:method (simulation-meshing-step ?control ?p1 ?p2
        ?p ?a ?resource ?aut ?process-id)

    (not(trigger ?aut))
        ((!!init ?process-id)
         (!split ?p1 ?p2 ?control)
         (!create-mesh-failover ?mesh-format
         ?topology-format ?mesh-construction ?resource)
         (!join ?p1 ?p2 ?control)
    )
    (trigger ?aut)
         ((!!init ?process-id)
          (!split ?p1 ?p2 ?control)
          (!assign-port ?p ?a)
          (!create-mesh ?p ?a ?control ?process-id)
          (!join ?p1 ?p2 ?control)
    )
)
```

Figure 10: Simulation Meshing Process in Shop2

DataFlow Process output can be represented in Shop2 via the produced effects, available as facts in the current state of the world. Process input is formalised via preconditions. As a predicate to further support the notion of process input and output we define the primitive *instance-of* that takes as arguments the current operator instance, the created concept or attribute, and, an ID, unique per simple process and used for tracing of output of subsequent process steps.

Artefacts Each input and output of a Shop2 operator corresponds to an artefact from the MDO domain. In addition, each artefact has a manifestation attached, pointing to its physical location. Individual types of artefacts are defined via predicates that are available as logical primitives in operators, methods and axioms. Relations between these artefacts, such as constraints, can be defined in axioms and are added in the body of a Shop2 method definition. Artefacts produced as output of a process are part of the current planning state and each precondition of a following Shop2 method or operator will be evaluated against this globally available state.

6.1.1 Top-Down Formulation of Executable MDO Workflow in Shop2

For a complete formulation of the MDO workflow in Shop2, the translated components are parametrised and combined into a high level method. The resulting construct represent the planning goal of Shop2 that has to be solved by the planner.

The top-level Shop2 method is illustrated in Figure 9. As the complete implementation of this sample spawns several pages, only snippets of the source code for the planning problem are depicted.

The refinement strategy for the abstract workflow is predefined by the decomposition mechanism of the planner. The decomposition of the MDO workflow starts with the most abstract method *mdo-workflow* that is added to the plan and broken down into compound and simple tasks, in Shop2 terminology, methods and operators. The top-level method *mdo-workflow* can be broken down into the following Shop2 methods:

simulation-build-config-step,
simulation-create-geometry-step,
simulation-meshing-step,
simulation-solving-step, and
simulation-filter-response-step

These correspond to the simulation processes, *build-cfg*, *create-geometry*, *meshing*, *solving* and *filter-response*. However, these sub-processes represent a super-set of the basic simulation processes, as control flow as well as a failover process are combined in the high-level process.

The workflow formulation illustrated in Figure 9 comprises all parameters required to execute the simulation, assuming that effects resulting from execution of individual operators are available as state information that is taken as input for the respective subsequent processes.
As an example for the next highest layer of abstraction, the method *simulation-meshing-step* is detailed. The method, allocated at the intermediate layer of the MDO framework, represents a domain independent version of all meshing procedures that can be executed as part of MDO scenarios. For a mapping from the intermediate layer to the domain-abstraction layer we add details such as tool-information and other environment specific knowledge, that establishes a binding to a given execution environment.

The method shows two branches that produce equivalent results. The second branch depicted contains the Shop2 operators *!!init*, *!split*, *!assign-port*, *!create-mesh* and *join*. All tasks listed are either MDO *SimpleProcess*(es) or they are Shop2 internal tasks performing supporting activities. In Shop2 terminology this means all tasks contained in method *simulation-meshing-step* can be directly executed and are not further decomposed. The method displayed is a simplified configuration for an MDO scenario. Further specifications for each included domain have to be added that represent the various domain-specific algorithms and tools that can be configured when using method *simulation-meshing-step*.

6.2 Support of workflow consistency checking by decomposition in Shop2

We now elaborate on a concrete use case with the objective to demonstrate how the ontology framework can be applied to support engineering design scenarios. The addressed problem can be described as follows: an experienced design engineer typically composes a design optimisation scenario by taking ad-

```
<!-- PROCESS configuration formulation -->
<process-configuration process= "create-geometry" cluster = "karros.autocrc.com" platform = "sun-solaris-10.4.2"
 <process name = "create-geometry" parent = "mdo-simulation" simple-process = "false" type = "mdo-task">
  <sub-process name = "read-geometry-definition"/>
  <sub-process name = "read-geometry-configuration"/>
  <sub-process name = "parametrise-geometry"/>

      <port id = "Bi1" process = "create-geometry" type = "input"
            parameter-description = "geometry-configuration ^ geometry-construction ^
            geometry-source-code ^ meta-data ^ manifestation"
            constraints = "/karros.autocrc.com:mdo-simulation-cb1:create-geometry:Bi1"/>

      <port id = "Bi2" process = "create-geometry" type = "input"
            parameter-description = "geometry-configuration ^ meta-data ^ manifestation"
            constraints = "/karros.autocrc.com:mdo-simulation-cb1:create-geometry:Bi2" />

      <port id = "Bo5" process = "create-geometry" type = "output"
            parameter-description = "geometry-format ^ STEP-format ^ solid-model-format ^ CAD-format"
            constraints = "/karros.autocrc.com:mdo-simulation-cb1:create-geometry:Bo5" />

  <error type = "input-mismatch" message = "input mismatch error" />
  <error type = "cluster-unavailability" message = "cluster unavailable error" />
  <error type = "output-mismatch" message = "output mismatch error" />
  <error type = "external-dependency" message = "external dependency error"/>
  </process>
</process-configuration>
```

Figure 11: Executable MDO Process description in internal xml format

vantage of an existing repository of process models that represent successfully performed tasks on design artefacts. However, as the given problem description distinguishes from previous scenarios, these process traces have to be reparametrised in order to suit the given situation. For example, due to a changed model size a different hardware component might be required in order to execute a particular solving application. In another case, a mass optimisation scenario might imply a switch to a different geometry format that is not part of a standard setup. In order to avoid the execution of a design scenario that consumes extended resources and due to constraints described above, finally fails, we want to check a given configuration in advance to improve the probability of a successful workflow execution. For this purpose we utilise Shop2 and realise *workflow consistency checking* by implementing the concepts represented in the MDO ontology framework.

Shop2 will be applied for the decomposition of a reparametrised MDO workflow into executable planning steps, showing that a consistent workflow refinement is possible. The difficulty in this task is the selection of value assignments for the included constraints, i.e., managing the trade-off between creating a too restrictive configuration that implies a low probability that a *plan*—corresponding to a valid workflow decomposition—can be found, and, selecting the assignments for the participating parameters too generic, i.e., risking that too many *plans* are found by the planner. We assume the following setup for a workflow consistency checking scenario:

- An instance and parametrisation of an MDO process is given as shown in Figure 11. The Figure illustrates the individual process steps and the parameter assignments per sub-process.
- For a new similar MDO scenario, a designer can select existing process traces from a repository of process models that have been used previously. The selection of these components is based on the number and type of parameters contained in the process model.
- The new parametrisation for the MDO workflow to be executed is transferred to the selected process models
- The new configuration can be checked for consistency with Shop2 and a reasoner can be applied for detecting violations of integrity constraints.

```
(:method (create-geometry ?control ?cluster ?platform ?workflow-engine ?revision
        ?parent-process ?simple-process ?type ?errors
    ?port-id_1 ?process_1 ?type_1 ?parameter-description_1
        ?constraints_1 ?address_1 ?manual_1
    ?port-id_2 ?process_2 ?type_2 ?parameter-description_2
        ?constraints_2 ?address_2 ?manual_2
    ?port-id_3 ?process_3 ?type_3 ?parameter-description_3
        ?constraints_3 ?address_3 ?manual_3)
    ()
    ((read-geometry-definition ?port-id_1 ?process_1 ?type_1
        ?parameter-description_1 ?constraints_1 ?address_1 ?manual_1)
    (read-geometry-configuration ?port-id_2 ?process_2 ?type_2
        ?parameter-description_2 ?constraints_2 ?address_2 ?manual_2)
    (parametrise-geometry ?port-id_3 ?process_3 ?type_3
        ?parameter-description_3 ?constraints_3 ?address_3 ?manual_3))
```

Figure 12: MDO Sub-Process *create-geometry* as Shop2 method

Assume an instance of the MDO workflow is given by the following generic description - only the subprocess *create-geometry* is illustrated for space reasons: all parameters of the workflow description are grounded, i.e., they possess a valid parameter-binding. Deriving a new workflow instance from an existing one means modifying some of the value-assignments of the original workflow instance to suit the context of the new scenario. However, what typically causes errors are implicit dependencies between parameters, that are not considered when modifying existing value assignments.

Shop2 either decomposes a high level abstract process into executable operators and methods (that is, it checks whether a *plan* can be found), or, alternatively, it can validate whether a high level representation exists that corresponds to a given execution trace of an MDO scenario. For the latter, the execution trace is translated into Shop2 constructs as explained above and correspondences between these constructs and high level Shop2 methods are investigated. Both approaches can be utilised for consistency checking of MDO workflows. We will demonstrate how Shop2 is applied for workflow consistency checking by top-down decomposition of a given Shop2 goal formulation that corresponds to a given MDO execution trace.

After the reconfiguration for the new situation has been completed, the description will be transformed into Shop2 notation and the modified parameters are updated in the high-level representation of the MDO workflow. The corresponding formulation of the new workflow in Shop2 is shown in Figure 12.

We can demonstrate consistency of the new workflow by showing that the planner is able to find a plan that corresponds to an executable workflow. If no plan can be found, an inconsistency in the workflow representation exists.

As an example for an inconsistency that can be detected in this way, consider the modification of parameters that represent the model size of a geometry model. As a typical MDO environment comprises a number of alternative MDO tools for a particular MDO task, a different MDO tool is assigned dependent on model characteristics. For instance the MDO task *create-mesh* can require are different meshing-tool dependent on parameters such as model-size and mesh-size. Using a different meshing tool can also imply that subsequent process steps have to be executed with different software components, dependent on the product used in the meshing step, resulting in an alternative execution path that was not considered by the designer when modifying the model size but keeping all remaining components in place.

To summarise, our reuse scenario is based on historic process execution traces that are parametrised according to the given design task. When reconfiguring a given trace by assigning a different parameter value, an inconsistent parametrisation of a process instance can result due to dependencies not considered by the designer. Shop2 can support in elimination of these inconsistencies by simulating the process execution, i.e., calculating whether a plan can be found for the new configuration.

6.2.1 Summary on Process Refinement

The technical realisation of the MDO workflow refinement in Shop2 demonstrated that the concepts comprised in the ontology framework can support a concrete design scenario. By utilising technologies from the planning domain, MDO process decomposition can be simulated with the objective to support the designer in the configuration process of a design scenario. As a future work, Shop2 models can be considered for translation into an executable workflow model, available in a target workflow language, provided the semantics of the target formalism can be represented in Shop2. Another aspect is the degree of automation that can be achieved when translating MDO tasks into Shop2 notation.

7 Conclusion

The introduced ontology framework can support typical activities of design engineers and enable them to more effectively use and explore results of an MDO scenario. The framework, comprising components such as domain and execution layer, domain abstraction layer as well as an intermediate and meta-model layer, addresses the integration of engineering processes by mapping and interrelating the developed concepts. Techniques from the *planning* realm have been utilised to demonstrate how our ontology framework can be applied to support the detection of workflow inconsistencies.

References

Andersen, O.A. and Vasilakis, G. (2007). Building an ontology of CAD model information. In *Geometric Modelling, Numerical Simulation, and Optimization: Applied Mathematics and SINTEF*, pages 11–40. Springer.

Borgo, S. and Leitao, P. (2007). Foundations for a core ontology of manufacturing. In *Ontologies: A Handbook of Principles, Concepts and Applications in Information Systems*, volume 14, pages 752–776. Springer.

Burge, J. et al. (2006). Enhanced design checking involving constraints, collaboration and assumptions: Ontology-supported rationale for collaborative argumentation. In *Proceedings of Design Computing and Cognition '06*, pages 655–674, Eindhoven, Netherlands. Springer.

Chandrasekaran, B., Josephson, J., and Benjamins, R. (1998). The ontology of tasks and methods. In *Proceedings of the 11th Knowledge Acquisition Modeling and Management Workshop, KAW'98*, Banff, Canada.

Cranefield, S. and Purvis, M. (1999). UML as an Ontology Modelling Language.

Dartigues, C., Ghodous, P., Gruninger, M., Pallez, D., and Sriram, R. (2007). CAD/CAPP Integration using Feature Ontology. In *CE: Concurrent Engineering: Research and Applications*, pages 237–249. SAGE.

Fensel, Dieter (1997). The tower-of-adapter method for developing and reusing problem-solving methods. In Plaza, Enric and Benjamins, V. Richard, editors, *EKAW*, volume 1319 of *Lecture Notes in Computer Science*, pages 97–112. Springer-Verlag.

Fenves, S.J., Sriram, R.D., Sudarsan, R., and Wang, F. (2003). A product information modeling framework for product lifecycle management. In *Proceedings of International Symposium on Product Lifecycle Management*, Bangalore, India.

Gero, J.S. and Kannengiesser, U. (2007). A function-behavior-structure ontology of processes. *Artificial Intelligence for Engineering, Design, Analysis and Manufacturing (AI EDAM)*, 21:379–391.

Grüninger, M., Bock, C., Libes, D., Lubell, J., and Subrahmanian, E. (2006). Evaluating reasoning systems. Technical Report NISTIR 7310, National Institute of Standards and Technology (NIST).

International Organization for Standardization (1994). *ISO 10303-11:1994, Part 11: The EXPRESS language reference manual*. International Organization for Standardization, Geneva, Switzerland.

Jiang, P. and Liu, Z. (2008). Combining and ontology representation with rule-based reasoning for the process planning of bulk silicon micro-manufacturing. *International Journal of Internet Manufacturing and Services*, 1(3):262–277.

Kannengiesser, U. and Gero, J.S. (2006). Towards mass customized interoperability. *Computer-Aided Design*, 38:920–936.

Kitamura, Yoshinobu and Mizoguchi, Riichiro (2004). Ontology-based systematization of functional knowledge. *Journal of Engineering Design*, 15(4):327–351.

Maier, F., Mayer, W., Stumptner, M., and Muehlenfeld, A. (2008). Ontology-based process modelling for design optimisation support. In *Proceedings of the 3rd International Conference on Design Computing and Cognition*, pages 513–532.

Maier, F. and Stumptner, M. (2007). Enhancements and ontological use of ISO-10303 (STEP) to support the exchange of parameterised product data models. In *Proceedings of Seventh International Conference on Intelligent Systems Design and Applications (ISDA'07)*.

Muehlenfeld, A., Mayer, W., Maier, F., and Stumptner, M. (2008). Ontology-based process modelling and execution using STEP/EXPRESS. In *Proceedings of 20th International Conference on Software Engineering and Knowledge Engineering*, San Francisco, USA.

Rachuri, Sudarsan et al. (2005). Information models for product representation: core and assembly models. *International Journal of Product Development*, 2(3):207–235.

Regli, W., Grauer, M., Wilkie, D., Kopena, J., Piecyk, M., and Osecki, J. (2009). Archiving the semantics of digital engineering artifacts in CIBER-U. In *Proceedings of the Twenty-First Innovative Applications of Artificial Intelligence Conference*.

Schaerli, N., Ducasse, S., Nierstrasz, O., and Black, A. (2002). Traits: Composable units of behavior. Technical Report CSE 02-012, Department of Computer Science and Engineering, OGI School of Science & Engineering, Oregon Health & Science University.

Seta, K., Ikeda, M., Shima, T., Kakusho, O., and Mizoguchi, R. (1998). CLEPE: a Task Ontology Based Conceptual Level Programming Environment.

Tudorache, T. (2008). *Ontologies in Engineering: Modeling, Consistency and Use Cases*. VDM Verlag.

Wu, D., Sirin, E., Hendler, J., Nau, D., and Parsia, B. (2003). Automatic Web Services Composition Using SHOP2.

Appendix A: MDO Domain Model

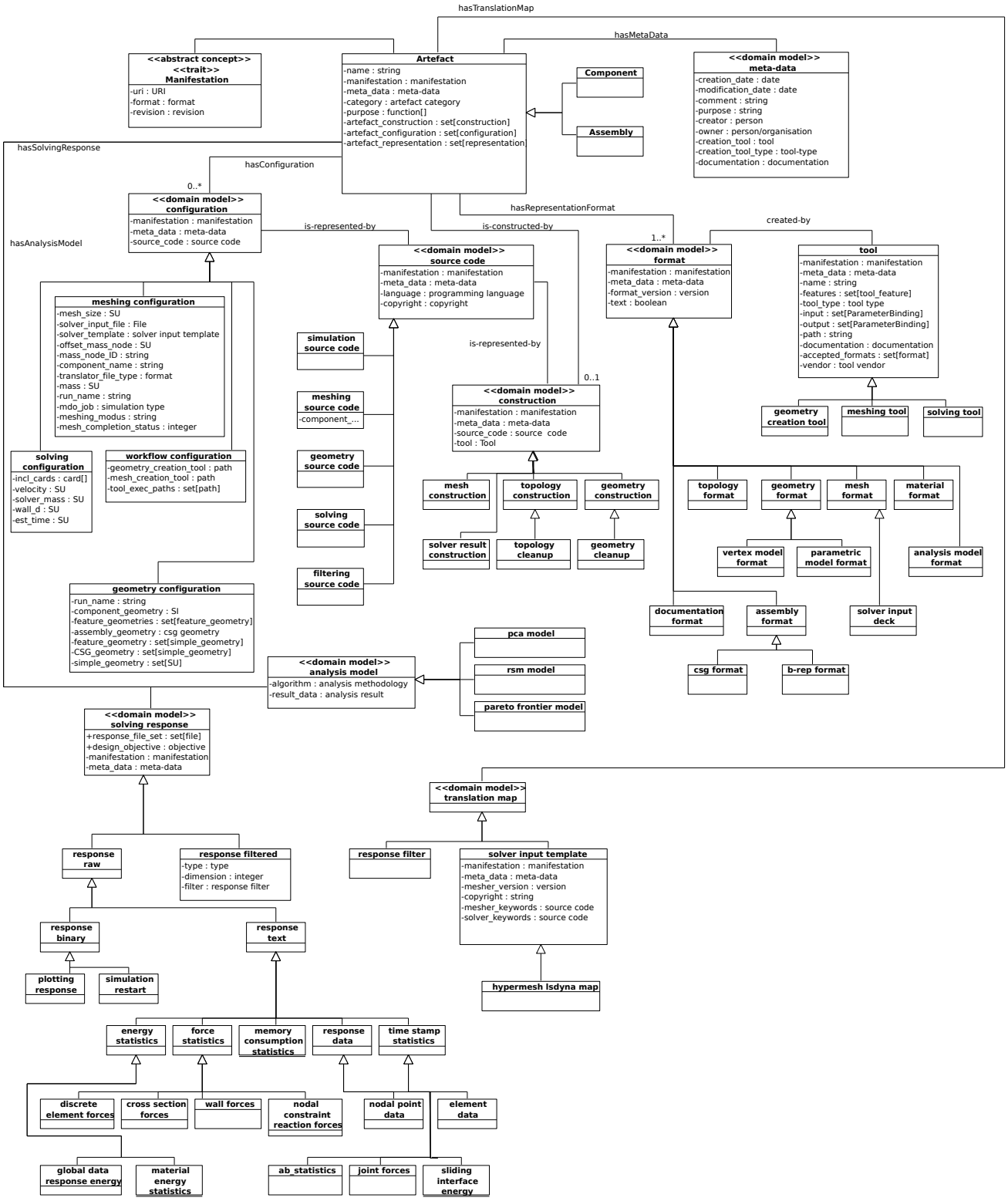

Figure 13: MDO Domain Model