Formal Management of CAD/CAM Processes

Michael Kohlhase, Johannes Lemburg, Lutz Schröder, and Ewaryst Schulz

DFKI Bremen, Germany
`<firstname>.<lastname>@dfki.de`

Abstract. Systematic engineering design processes have many aspects in common with software engineering, with CAD/CAM objects replacing program code as the implementation stage of the development. They are, however, currently considerably less formal. We propose to draw on the mentioned similarities and transfer methods from software engineering to engineering design in order to enhance in particular the reliability and reusability of engineering processes. We lay out a vision of a document-oriented design process that integrates CAD/CAM documents with requirement specifications; as a first step towards supporting such a process, we present a tool that interfaces a CAD system with program verification workflows, thus allowing for completely formalised development strands within a semi-formal methodology.

1 Introduction

Much of our life is shaped by technical artifacts, ranging in terms of intrinsic complexity from ball point pens over pacemakers, automobiles, and nuclear power stations to robots. These artifacts are the result of *engineering design processes* that determine their quality, safety, and suitability for their intended purposes and are governed by best practices, norms, and regulations. The systematic development of products is guided by descriptions of problems and their solutions on different levels of abstraction, such as the requirements list, the function structure, the principle solution, and eventually the embodiment design. The elements of these representations are linked by dependencies within and across the different levels of abstraction. The present state of the art in computer-aided design and manufacture of industrial artifacts (CAD/CAM) does not support this cross-linking of dependencies. Consequently, e.g. non-embodied principle solutions are still often shared and stored in the form of hand-made sketches and oral explanations. In other words, large parts of the engineering process are not completely representable in current CAD/CAM systems, which are focused primarily on the embodiment level.

Contrastingly, software engineers have long acknowledged the need for a formal representation of the entire software development process. In particular, formal specification and verification of software and hardware systems is without alternative in safety-critical or security areas where one cannot take the risk of failure. Formal method success stories include the verification of the Pentium IV arithmetic, the Traffic Collision Avoidance System TCAS, and various security protocols. In many cases, only the use of logic-based techniques has been

able to reveal serious bugs in software and hardware systems; in other cases, spectacular and costly failures such as the loss of the Mars Climate Orbiter could have been avoided by formal techniques. Norms such as IEC 61508 make the use of formal methods mandatory for software of the highest safety integrity level (SIL 3). Thus, formal methods will form an integral part of any systematic methodology for safe system design.

The main goal of the present work is to outline how formal methods, hitherto used predominantly in areas such as software development and circuit design that are inherently dominated by logic-oriented thinking anyway, can be transferred to the domain of CAD/CAM, which is more closely tied to the physical world. In particular, we wish to tie formal specification documents in with a semi-formal engineering design process. Potential benefits for the CAD/CAM process include

- formal verification of physical properties of the objects designed
- tracing of (formalized) requirements across the development process
- improved control over the coherence of designs
- semantically founded change management.

We lay out this vision in some more detail, relating it to an extended discussion of current best practice in engineering design (Section 2), before we proceed to report a first step towards enabling the use of formal methods in engineering design: we describe a tool that extracts formal descriptions of geometric objects from CAD/CAM designs (Section 3). Specifically, the tool exports designs in the CAD/CAM system SolidWorks into a syntactic representation in the wide-spectrum language HasCasl [10], thereby making the connection to a formal semantics of CAD/CAM objects in terms of standard three-dimensional affine geometry as defined in a corresponding specification library. We apply the tool in a case study (Section 4) involving a toy but pioneering example where we prove that a simple CAD drawing implements an abstractly described geometric object, using the semi-automatic theorem prover Isabelle/HOL, interaction with which is via logic translations implemented in the Bremen heterogeneous tool set HeTS [8].

2 A Document-oriented Process for CAD/CAM

Best practices for designing technical artifacts are typically standardized by professional societies. In our exposition here, we will follow the German VDI 2221 [12], which postulates that the design process proceeds in well-defined phases, in which an initial idea is refined step-by-step to a fully specified product documentation. We observe that the process is similar to the software engineering process and that the stages in the design process result in specification documents, as they are e.g. found in the V-model (see Fig. 1). In contrast to software engineering approaches like the V-model, however, VDI 2221 (and actual current practice in engineering design) do not provide a mechanism to ensure consistency between the design stages,

or methods for verifying that products actually meet requirements specified in preceding phases of the development. In fact, the VDI 2221 process corresponds only to the left shank of the process depicted in Fig. 1, while the quality control process (the right shank in Fig. 1 and the main contribution of the V-model) is left unspecified.

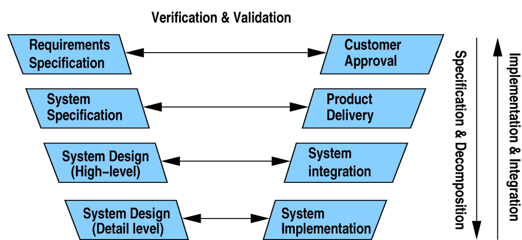


Fig. 1. The V-model of Software Engineering

2.1 The Engineering Design Process

To make the correspondence between VDI 2221 and the V-model explicit we review the six[1] stages and relate them to the V-model before we illustrate them with a simple example.

S1 **Purpose/Problem**: a concise formulation of the purpose of the product to be designed.
S2 **Requirements List**: a list of explicit named properties of the envisioned product. It is developed in cooperation between designer and client and corresponds to the user specification document in the V-model.
S3 **Functional Structure**: A document that identifies the functional components of the envisioned product and puts them into relation with each other.
S4 **Solution in Principle**: a specification of the most salient aspects of the design. It can either be a CAD design like the one in Fig. 2 below or a hand drawing [7].
S5 **Embodiment Design/"Gestalt"**: a CAD design which specifies the exact shape of the finished product.
S6 **Documentation**: accompanies all steps of the design process.

Note that most of these design steps result in informal text documents, with step S5 being the notable exception. In the envisioned document-oriented engineering design process we will concentrate on these documents, enhance them with semantic annotations and link them to background specifications to enable machine support: e.g. requirements tracing, management of change, or verification of physical properties. Before discussing this vision in more detail, let us set up an example by considering a rational reconstruction of the design process of a machinists' hammer according to DIN 1041.

2.2 The Design of a Hammer

[1] In fact, [12] specifies additional stages for determining modular structures and developing their embodiments, which we will subsume in steps S3 and S5

The Purpose of a Hammer The first and most important step in setting up a requirements list is the specification of the purpose of the product. The purpose describes the intended use of the product solution-neutrally. This is the highest level of abstraction within the design process. In the case of a hammering tool, the purpose can be in the form of a very simple definition:

> A **hammer** is an apparatus for transmitting an impulse to an object.

In reference to a hand-tool in contrast to e.g. a hammer mill, the purpose can be narrowed to:

> A **hammer** is an apparatus for the manual generation and transmission a defined impulse to an object.

Ideally, the list of requirements of a product should be unambiguous, clear and complete. This ideal goal is seldom fulfilled in a real life design process. This is due e.g. to implicit customer wishes, which in fact often are more important to the market-success of a product than the well-known and explicitly named requirements. In the case of the hammer, the requirements might include the following.

Explicit Requirements

- E1 The hammer has to fulfill the standard DIN 1041 and all related subsequent standards, namely: DIN 1193, DIN 1195, DIN 5111, DIN 68340 and DIN ISO 2786-1.
- E2 The handle has to be painted with a clear lacquer over all and with colour RAL 7011 (iron grey) at 10 cm from the lower end.
- E3 Two company logos of 20mm length are placed on both sides of the handle.

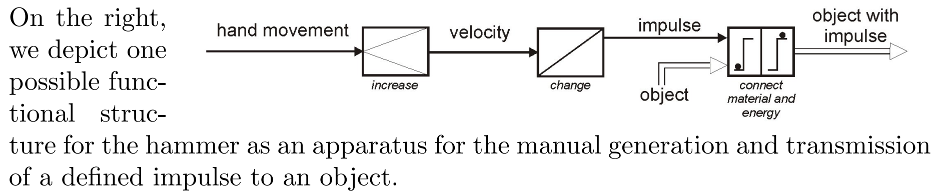
Implicit Requirements

- I1 The hammer must be usable for right-handed and left-handed persons.
- I2 The hammer should be ergonomic.
- I3 The hammer must fit into a standard tool chest.
- I4 The hammer shall look strong and matter-of-fact.

Functional Specification of a Hammer Within the design process, the functional specification is done by setting up the function structure that breaks down the complex problem into manageable smaller sub-functions and represents the logical interrelationships of the given tasks. As in the previous design steps, the function structure is still solution neutral. The aim is to open up a preferably broad solution field, which will be narrowed by explicit named criteria within further steps of the design process.

The function structure is intended to explain interrelationships within the future embodied product; therefore, the connection between function structure and the given product has to be clear. Every sub-function can be found within the product, or the product is not a suitable solution. On the other hand, function structures are not appropriate as a tool for reverse engineering, because the relation between the embodied product and the underlying functions is ambiguous.

On the right, we depict one possible functional structure for the hammer as an apparatus for the manual generation and transmission of a defined impulse to an object.



The Principle Solution for a Hammer From the functional specification, we develop a *principle solution* (see Fig. 2). This solution abstracts from the physical traits of the eventual product and identifies the functional parts. For a hammer, one of these is the handle, here a cylindrical part of the hammer shaft used for gripping. The fact that it is symmetric/cylindrical is a response to the requirement E1. The handle is connected to an inert mass (depicted by a solid ball in Fig. 2) which is again connected to an *active surface* that delivers the impact on the object. The size and form of the active surface will be determined by the requirement E2. In fact, the principle solution reveals that there is a second possible active area of the hammer, opposite to the primary one; Fig. 2 shows three variants of the principle solution with differing secondary active surfaces.

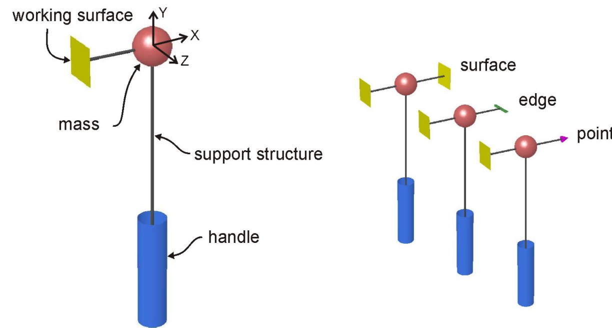


Fig. 2. A principle solution for a Hammer

The Embodiment of a Hammer Note that the principle solution is not a finished design yet, since it abstracts from most of the physical traits of a hammer, e.g. the dimensions of the shaft and the form of the head, which will be specified in the embodiment design step. Here, the ultimate three-dimensional shape and the materials of the product are derived, taking into account material properties, manufacturability constraints, and aesthetic factors. These can lead to the widely differing final designs we see in use today.

2.3 A Document-Oriented Design Process

We propose to reinforce the systematic engineering design process laid out above with technologies and practices from software engineering and Formal Methods to obtain a document-oriented process where designs are semantically enhanced and linked to formal and semi-formal specifications. It is crucial to note that the various design documents necessarily have differing levels of rigour, ranging from

informal and hard-to-quantify requirements like E2 to mathematical proofs of security-relevant properties, e.g. in aerospace applications. Additionally, different product parts and aspects underlie differing economic and security-related constraints, so that design quality control must be supported at various levels of formality. As a consequence, design documents need to be encoded in a document format that supports *flexible degrees of formality*, such as OMDoc (Open Mathematical Documents [6]). The OMDoc format concentrates on structural aspects of the knowledge embedded in documents and provides a markup infrastructure to make it explicit by annotation. Crucially, the format supports a fine-granular mixture of formal and informal elements and thus supports, e.g., the stepwise migration from informal user requirements to specifications expressed in formal logics supported by verification environments like the Bremen heterogeneous tool set HeTS [8]. The format itself is *semi-formal*, i.e. focuses on explicitly structured documents where relevant concepts are annotated by references to *content dictionaries* that specify the meaning of the terms used in design documents. Semi-formal design documents already bring added value to the engineering process by enabling machine support for many common quality control tasks like *requirements tracing* and *management of change* which are based on an explicitly given dependency relation (see [1] for details). Fully formal development strands embedded in a semi-formal process additionally allow for the rigorous verification of critical properties in a design, thus providing a reliable link between various stages of the engineering design process. It is this aspect that we concentrate on in the following.

3 Invading SolidWorks

We now illustrate how the document-oriented formal/semi-formal methodology in engineering design processes laid out in the last section can be supported by means of an integration of formal methods tools with the widely used CAD system SolidWorks [11]. The latter serves mainly as a demonstration platform; our overall approach is sufficiently general to apply equally well to any other CAD system that provides suitable interfaces.

To access concrete CAD designs, we provide a SolidWorks plug-in that extracts the designs as formal specifications, i.e. as lists of terms denoting sketches and features, and as formulas expressing constraints relating these sketches and features. These data are obtained using the SolidWorks API, and are output as a HasCasl [10] specification encoded in an OMDoc file [6].

Overview of HasCasl HasCasl is a higher order extension of the standard algebraic specification language Casl (Common Algebraic Specification Language) [2,9] with partial higher order functions and type-class based shallow polymorphism. The HasCasl syntax appearing in the specifications shown in the following is largely self-explanatory; we briefly recall the meaning of some keywords, referring to [10] for the full language definition. Variables for individuals, functions and types are declared using the keyword **var**. The keyword **type**

declares, possibly polymorphic, types. Types are, a priori, loose; a **free type**, however, is an algebraic data type built from constructor operations following the standard no-junk-no-confusion principle. Types are used in the profiles of operations, declared and, optionally, defined using the keyword **op**. Operations may be used to state axioms in standard higher order syntax, with some additional features necessitated through the presence of partial functions, which however will not play a major role in the specifications shown here (although they do show up in the geometric libraries under discussion). Names of axioms are declared in the form **%**(axiom_name)**%**.

Beyond these *basic specification* constructs, HasCasl inherits mechanisms for structured specification from Casl. In particular, *named* specifications are introduced by the keyword **spec**; specification *extensions* that use previously defined syntactic material in new declarations are indicated by the keyword **then**; and unions of syntactically independent specifications are constructed using the keyword **and**. Annotation of extensions in the form **then %implies** indicates that the extension consists purely of theorems that follow from the axioms declared previously. Named specifications may be parameterized over arbitrary specifications. They may be imported using the given name. Named morphisms between two specifications can be defined using the keyword **view** to express that modulo a specified symbol translation, the source specification is a logical consequence of the target specification. HasCasl is connected to the Isabelle/HOL theorem prover via HeTS [8].

The SolidWorks Object Model In order to obtain a formal representation of CAD designs, we define the SolidWorks object types as algebraic data types in a HasCasl specification[2] following the SolidWorks object hierarchy, using a predefined polymorphic data type *List a* of lists over *a*. (All specifications shown below are abridged.)

spec SolidWorks = AffineRealSpace3DWithSets
then free types
  SWPlane ::= *SWPlane* (*SpacePoint* : *Point*; *NormalVector* : *VectorStar*;
                InnerCS : *Vector*);
  SWArc ::= *SWArc* (*Center* : *Point*; *Start* : *Point*; *End* : *Point*);
  SWLine ::= *SWLine* (*From* : *Point*; *To* : *Point*);
  SWSpline ::= *SWSpline* (*Points* : *List Point*);
  SWSketchObject ::= *type SWArc* | *type SWLine* | *type SWSpline*;
  SWSketch ::= *SWSketch* (*Objects* : *List SWSketchObject*;
                Plane : *SWPlane*);
  SWExtrusion ::= *SWExtrusion* (*Sketch* : *SWSketch*; *Depth* : *Real*);
  ...
  SWFeature ::= *type SWExtrusion* | ...

[2] All mentioned HasCasl specifications can be obtained over the web from: `https://svn-agbkb.informatik.uni-bremen.de/Hets-lib/trunk/HasCASL/Real3D/`

This provides a formal *syntax* of CAD designs, which we then underpin with a formal geometric *semantics*. The constructs are classified as follows.

- *Base objects* are *real numbers*, *vectors*, *points*, and *planes*, the latter given by a point on the plane, the normal vector and a vector in the plane to indicate an inner coordinate system.
- *Sketch objects:* From base objects, we can construct *sketch objects* which are *lines* defined by a start and an end point, *arcs* also given by start and end, and additionally a center point, and *splines* given by a list of anchor points.
- *Sketch:* A plane together with a list of sketch objects contained in it constitutes a *sketch*.
- *Features* represent three dimensional solid objects. They can be constructed from one or more sketches by several *feature constructors*, which may take additional parameters.

We will focus in the following on the *extrusion* feature constructor which represents the figure that results as the space covered by a sketch when moved orthogonally to the plane of the sketch for a given distance.

In order to reason about formal SolidWorks designs, we equip them with a semantics in terms of point sets in three-dimensional affine space (i.e. in $\mathbb{R}^3$ equipped with the standard affine structure). For example, the term $SWLine(A, B)$ is interpreted as a line segment from point A to point B in $\mathbb{R}^3$. Formally, the semantics is based on point set constructors that correspond to the syntax constructors, specified as follows.

spec SolidWorksSemanticConstructors =
    AffineRealSpace3DWithSets
then ops
    $VWithLength(v : Vector; s : Real) : Vector =$
        $v \ when \ v = 0 \ else \ (s \ / \ (|| \ v \ || \ as \ NonZero)) * v;$
    $VPlane(normal : Vector) : VectorSet = \lambda \ y : Vector \bullet orth \ (y, normal);$
    $VBall(r : Real) : VectorSet = \lambda \ y : Vector \bullet || \ y \ || \leq r;$
    $ActAttach(p : Point; vs : VectorSet) : PointSet = p + vs;$
    $ActExtrude(ax : Vector; ps : PointSet) : PointSet =$
    $\lambda \ x : Point \bullet \exists \ l : Real; y : Point$
        $\bullet \ l \ isIn \ closedinterval \ (0, 1) \wedge y \ isIn \ ps \wedge x = y + l * ax;$

Using these semantic constructors, the point set interpretation of, e.g., planes and features is given by the following specification.

spec SolidWorksWithSemantics = SolidWorks
and SolidWorksSemanticConstructors
then ops
    $i : SWExtrusion \rightarrow PointSet;$
    $i : SWPlane \rightarrow PointSet$

```
vars  o, x, y, z : Point; n : VectorStar; ics : Vector;
      l : Real; plane : SWPlane
• i (SWPlane (o, n, ics)) = ActAttach (o, VPlane n);
• i (SWExtrusion (SWSketch ([ SWArc (x, y, z) ], plane), l))
  = (let cp = x;
         r1 = vec (cp, y);
         ball = ActAttach (cp, VBall || r1 ||);
         planeI = i plane;
         scaledAxis = VWithLength (NormalVector plane, l)
     in
        ActExtrude (scaledAxis, ball intersection planeI))
     when y = z else emptySet;
```

In the case study of the next section, we will show a concrete example which illustrates the use of the plug-in in the context of our envisioned development process. The case study is mainly concerned with the verification of designs against abstract requirements. Further potential uses of the invasive approach include *semantic preloading*, i.e. automated rapid prototyping of designs from abstract specifications, as well as requirements tracing and a closer general integration of specifications and designs, e.g. by user-accessible links between specifications and parts in the SolidWorks design.

4 Case Study: Simple Geometric Objects

We will now illustrate what form a formal strand of the integrated formal/semi-formal development process advocated above might take on a very basic case study: we construct a simple object in the CAD/CAM system, specifically a cylinder, export its formal description using our tool, and then formally verify that it implements a prescribed abstract geometric shape, i.e., that it really is a cylinder. Simple geometric objects like cylinders form the basic repertoire from which complex technical artifacts are built up; for instance, the principle solution of a hammer in Fig. 2 represents the handle by a cylinder.

Naively, one would imagine that there is really nothing to verify about a geometric object: a cylinder is a cylinder is a cylinder. But as soon as one starts using a real CAD system, it becomes clear that the situation is actually more complex. The mathematical concept of a three-dimensional geometric object is a set of points in three-dimensional euclidean space, typically described by a general formation principle and a number of parameters. E.g. in the case of a (solid) cylinder, the parameters are

- the coordinates of some anchor point, say the centre of the cylinder,
- the spatial direction of the axis,
- the height h and the radius r,

and the formation principle for cylinders prescribes that these parameters describe the set of points p such that

– p has distance at most r from the axis (regarded as an infinite straight line);
– the orthogonal projection of p onto the axis has distance at most h from the centre point, and
– p lies in the positive half space determined by the base plane.

On the other hand, the design that we extract from our CAD construction takes a totally different form: instead of defining a point set using the above-mentioned parameters, we construct the cylinder as a *feature* by applying a suitable *feature constructor* to more basic two-dimensional objects called *sketches* as laid out in Section 3. Additionally, we may impose *constraints* on the dimensions involved, e.g. equality of two sides in a triangle, a point which we have not explicitly treated in Section 3. Specifically, the construction of a cylinder in SolidWorks would typically proceed as follows.

– Create a plane.
– Insert a circle into the plane, described as a circular arc with coincident start and end points.
– Extrude the circle to a certain depth.

Thus, the cylinder is constructed a feature stemming from the extrusion feature constructor which is anchored in the sketch consisting of one sketch object, the circle. We shall generally refer to a combination of features as described above as a *concrete design*, while a definition via mathematical point sets will be called an *abstract design*. While in the above case it is easy to see intuitively that the concrete design matches the abstract design, i.e. that extruding a circle really yields a cylinder, the formalisation of this intuition is by no means an entirely trivial enterprise, and more complex objects quickly lead to quite challenging verification tasks – imagine e.g. having to check that two given circular extrusions of two circles yield two interlocking chain links. Additional complexity arises from the above-mentioned constraints – e.g. one may initially leave the height of the cylinder open, cut part of the cylinder off using a skewed plane placed at a defined angle to the base plane and touching the perimeter of the bottom circle, and then impose that the height of the short side of the arising cut cylinder is half that of the long side, thus completely determining the height.

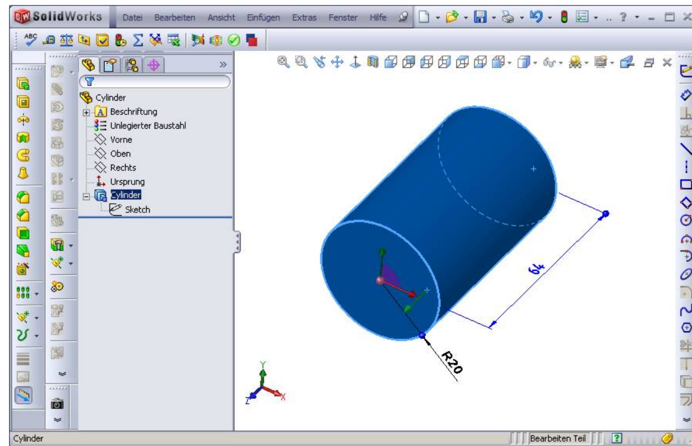


Fig. 3. A cylinder in SolidWorks

It is therefore desirable to have machine support for checking that an abstract design is actually implemented by a given concrete design. Besides the mere fact that one can verify geometric shapes, added benefits include

- Easy proofs of physical and geometric properties of the objects involved – e.g. once one has matched the abstract cylinder to the concrete cylinder, one can now prove *on the abstract side* (much more easily than on the concrete side) that the cylinder adheres to a prescribed surface area, volume, or mass (if the density is known).
- Better control over the cohesion and consistency of the design – e.g. if it turns out that the design fails to match the abstract object, this may mean that the designer has accidentally left extraneous degrees of freedom. Such errors may later lead to blocked designs that cannot be completed due to unsatisfiability of their constraints, a notorious problem in computer-aided construction; verification against abstract designs may help detecting such errors at an early stage of the development process.
- The abstract design may in fact properly abstract from the concrete shape of the final object, e.g. by leaving less relevant dimensions open (within certain ranges) or omitting features that do not play a central role in the present stage of the design process, thus providing for a property-centered approach to evolutionary design.

Further possible semantic services enabled by the connection between abstract and concrete designs within HETS include semantic annotation and requirements tracing as discussed in Section 2. A more visionary potential application of abstract designs is the automated derivation of concrete designs, i.e. rapid prototyping by compilation of abstract designs into preliminary *formally verified* CAD documents.

A collection of geometry libraries The basis of the proposed formal geometric verification framework is a collection of HASCASL specification libraries, structured as follows. The abstract specification of three dimensional basic geometry is contained in a library which provides the fundamental types and objects such as the data types *Point* and *Vector* for points and vectors in $\mathbb{R}^3$, types for point sets and vector sets, and operations on these types. These specifications use specification instantiations from an abstract library of linear algebra and affine geometry which provides the basic notions of a Euclidean vector space such as linear dependency, norm and distance, the inner product and orthogonality, and the operations which relate points and vectors in affine geometry. For instance, the basic definition of an affine space, i.e. intuitively a vector space without origin, is given as follows.

spec AFFINESPACE[VECTORSPACE[FIELD]] =
  type *Point*
  op $__+__ : Point \times Space \to Point$ %(point space map)%
  vars $p, q : Point; v, w : Space$

$\bullet\ p + v = p + w \Rightarrow v = w$ %(plus injective)%
$\bullet\ \exists\ y : Space \bullet p + y = q$ %(plus surjective)%
$\bullet\ p + (v + w) = p + v + w;$ %(point vector plus associative)%
then %implies
$\forall\ p : Point;\ v, w : Space$
$\bullet\ p + v + w = p + w + v;$ %(point vector plus commutative)%
end

spec ExtAffineSpace [AffineSpace[VectorSpace[Field]]] = **%def**
op $vec : Point \times Point \rightarrow Space$
$\forall\ p, q : Point \bullet p + vec\ (p, q) = q;$ %(vec def)%
then %implies
vars $p, q, r : Point;\ v, w : Space$
$\bullet\ vec\ (p, q) + vec\ (q, r) = vec\ (p, r)$ %(transitivity of vec plus)%
$\bullet\ vec\ (p, q) = -\ vec\ (q, p)$ %(antisymmetry of vec)%
$\bullet\ p + v = q \Rightarrow v = vec\ (p, q);$ %(plus vec identity)%
end

(Here, we employ a pattern where specifications are separated into a base part containing only the bare definitions and an extended part containing derived operations, marked as such by the semantic annotation **%def**.)

The libraries for SolidWorks consists of the data types and semantics introduced in section 3 and a library which contains common concrete design patterns such as, e.g., the construction of a cylinder described earlier in this section. They also contain views stating the correctness of these patterns, as exemplified next. Constructions exported from SolidWorks using our tool can then be matched with design patterns in the library via (trivial) views, thus inheriting the correctness w.r.t. the abstract design from the design pattern.

4.1 A proof of a refinement view

We illustrate the verification of concrete design patterns against abstract designs on our running example, the cylinder. The abstract design is specified as follows.

spec Cylinder = AffineRealSpace3DWithSets
then op $Cylinder(offset : Point;\ r : RealPos;\ ax : VectorStar) : PointSet =$
$\lambda\ x : Point \bullet let\ v = vec\ (offset, x)\ in$
$||\ proj\ (v, ax)\ ||\ \leq\ ||\ ax\ ||$
$\wedge\ ||\ orthcomp\ (v, ax)\ ||\ \leq r$
$\wedge\ (v * ax) \geq 0;$

We wish to match this with the concrete design pattern modelling the CAD construction process outlined above (importing the previously established fact that planes in SolidWorks are really affine planes):

```
spec SolidWorksCylByArcExtrusion =
     SolidWorksPlane_is_AffinePlane
then op
     SWCylinder(center, boundarypt : Point; axis : VectorStar): SWFeature =
     let plane = SWPlane (center, axis, V (0, 0, 0));
         arc = SWArc (center, boundarypt, boundarypt);
         height = || axis ||
     in SWExtrusion (SWSketch ([ arc ], plane), height);

view SWCylByAE_IsCylinder : Cylinder to
     {SolidWorksCylByArcExtrusion
     then op
          Cylinder(offset : Point; r : RealPos; axis : VectorStar): PointSet =
          let boundary = λ p : Point • let v = vec (offset, p)
                                       in orth (v, axis) ∧ || v || = r;
              boundarypt = choose boundary
          in i (SWCylinder (offset, boundarypt, axis));
     }
```

The above view expresses that every affine cylinder can be realized by our concrete design pattern. In induces a proof obligation stating that the operation *Cylinder* defined in the view by means of $i \circ SWCylinder$ is equal to the operation *Cylinder* defined in the specification *Cylinder*, the source of the view. Listing 1.1 shows this proof obligation translated to an Isabelle/HOL assertion.

Listing 1.1. Proof obligation as Isabelle/HOL assertion

```
theorem def_of_Cylinder :
"ALL axis offset r.
 Cylinder ((offset , r), axis) =
 (% x. let v = vec(offset , x)
       in ( || proj(v, gn_inj(axis)) || <=' || gn_inj(axis) || &
            || orthcomp(v, gn_inj(axis)) || <=' gn_inj(r)) &
          v *_4 gn_inj(axis) >=' 0'')"
```

We will sketch the corresponding proof in Isabelle/HOL, using a slightly more readable notation than those in the original Isabelle source code[3]. After the unfolding of function definitions such as *SWCylinder*, *SWExtrusion*, *i*, *ActExtrude* and some bookkeeping steps involving let-environments, conditionals, and function equality, we arrive at an equivalence of the form

```
 (1) Exists l:Real, y:Point.
    (1.1) l in [0..1] /\ (1.2) y in (ball intersection plane) /\ (1.3) x = y + l * axis
<=> (2)
    (2.1) ||vp|| <= ||axis|| /\ (2.2) ||vo|| <= r /\ (2.3) v * axis >= 0
```

[3] The Isabelle source code for this proof can be obtained over the web from: `https://svn-agbkb.informatik.uni-bremen.de/Hets-lib/trunk/HasCASL/Real3D/SolidWorks/CylinderView.thy`

with free variables x, offset, r and axis and a local environment containing the following variables with their bindings (see Table 1 for an explanation of the occurring function symbols)

```
(0.1)  boundary = \p. let v=vec(offset, p) in orth(v, axis) /\ ||v|| = r
(0.2)  bp = choose(boundary)
(0.3)  r1 = vec(offset, bp)
(0.4)  pln = SWPlane offset axis 0
(0.5)  arc = SWArc offset bp bp
(0.6)  ht = ||axis||
(0.7)  ball = ActAttach(offset, VBall(||r1||))
(0.8)  plane = i(pln)
(0.9)  v = vec(offset, x)
(0.10)  vp = proj(v, axis)
(0.11)  vo = orthcomp(v, axis)
```

FUNCTION	DESCRIPTION
[0..1]	closed unit interval
intersection	binary set intersection
*	overloaded binary operator (inner product, scalar multiplication, ...)
||_||	norm of a vector
vec	the vector connecting two points
orth	the orthogonality predicate for two vectors
choose	usual choice operator for a predicate
SWPlane	SOLIDWORKS constructor for a plane (see Section 3)
SWArc	SOLIDWORKS constructor for an arc (see Section 3)
VBall	vector set constructor for a ball (see Section 3)
ActAttach	point set constructor adding a vector set to a point (see Section 3)
i	interpretation function (see Section 3)
proj	orthogonal projection of a vector onto another
orthcomp	the orthogonal component of an orthogonal decomposition

Table 1. Function symbols and their meaning

The key to the proof is the relation between v, y and l and the orthogonal decomposition of v along the axis: v = vp + vo. From (0.8) together with (0.4) and the semantics definition for a plane, we obtain plane = offset + VPlane(axis) = offset + {z | orth(z,axis)}, and with (1.2), which gives us y in plane, we have y = offset + y' with y' satisfying orth(y',axis). Similarly we obtain from (0.7) that y = offset + y'' with y'' <= ||r1|| and of course y' = y'' by injectivity of the addition of vectors to points in affine space. Substituting y into (1.3) gives us x = offset + y' + l * axis.

On the other hand, from (0.9) we have x = offset + v = offset + vo + vp with vp a multiple of axis and vo orthogonal to it. Hence we obtain offset + y' + l * axis = offset + vo + vp and thus y' + l * axis = vo + vp. As l * axis and vp are linearly dependent and each side of the equation is the unique orthogonal decomposition of v, we obtain finally our relation as y' = vo and l * axis = vp.

To show (1) => (2) using this relation it remains to establish the following.

```
(1') l in  [0..1]  /\ (1.2') y in  ball
=> (2)
    (2.1') ||l * axis||  <= ||axis||  /\ (2.2')  ||y'||  <= r
/\ (2.3') (vo + l * axis) * axis >= 0
```

The rest is now real arithmetic together with the distributive law of the inner product and some basic facts concerning the inner product and the norm, thus concluding the correctness proof of the concrete design pattern for cylinders.

5 Conclusion and Further Work

We have argued that systematic engineering design processes (as laid down e.g. in VDI 2221) have many commonalities with software engineering. To transfer methods from software engineering to engineering design we have to deal with the fact that engineering design processes are considerably less formal and are geared towards producing CAD/CAM objects instead of program code. We have formulated a semi-formal, document-oriented design process that integrates CAD/CAM documents with specification documents of various degrees of formalisation, up to and including fully formal specification and verification. To support the CAD/CAM parts of this design process, we have extended a widely used CAD system with an interface for exporting CAD objects to the Bremen heterogeneous tool set HeTS, specifically to translate them into specifications in the wide-spectrum language HasCasl. Thereby, we turn CAD documents into fully formal documents within our process, as the export mechanism defines a rigorous geometric semantics for CAD designs. Moreover, we have illustrated the formal proof obligations that may arise in this process, and have presented a sample proof that verifies the implementation of a simple abstract geometric object by a CAD design.

The work reported here forms part of a long-term endeavor where we want to rethink the systematic engineering design process as a whole. Further steps in this program include the systematic development of a library of abstract construction patterns, improved automated proof support for geometric proofs possibly using an integration of computer algebra systems into the HeTS framework, rapid prototyping of CAD/CAM objects from abstract specifications, and verification of CAD/CAM designs against formalised industrial standards. The reasoning support for formalised geometry may eventually profit from existing results on automated theorem proving in geometry including [3,13,4] and, most recently, from the Flyspeck project [5], either by reuse of concepts or by actually importing existing theorems using the heterogeneous mechanisms provided by HeTS.

Acknowledgements

The work reported here was supported by the FormalSafe project conducted by DFKI Bremen and funded by the German Federal Ministry of Education

and Research (FKZ 01IW07002). We gratefully acknowledge discussions with Bernd Krieg-Brückner, Dieter Hutter, Christoph Lüth, and Till Mossakowski, and thank Tanmay Pradhan for his work on the SolidWorks plugin.

References

1. S. Autexier, D. Hutter, T. Mossakowski, and A. Schairer. Maya: Maintaining structured documents. In OMDoc – *An open markup format for mathematical documents [Version 1.2]* [6], chapter 26.12.
2. M. Bidoit and P. D. Mosses. Casl *User Manual*, vol. 2900 of *LNCS*. Springer Verlag, 2004.
3. S.-C. Chou. *Mechanical Geometry Theorem Proving*. Reidel, Dordrecht, 1988.
4. H.-G. Gräbe. The SymbolicData GEO records - a public repository of geometry theorem proof schemes. In *Automated Deduction in Geometry*, vol. 2930 of *LNCS*, pp. 67–86. Springer Verlag, 2004.
5. T. C. Hales. Introduction to the Flyspeck project. In *Mathematics, Algorithms, Proofs*, vol. 05021 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.
6. M. Kohlhase. OMDoc – *An open markup format for mathematical documents [Version 1.2]*. Number 4180 in LNAI. Springer Verlag, 2006.
7. J. P. Lemburg. *Methodik der schrittweisen Gestaltsynthese*. PhD thesis, Fakultät für Maschinenwesen, RWTH Aachen, 2008.
8. T. Mossakowski, C. Maeder, and K. Lüttich. The Heterogeneous Tool Set. In O. Grumberg and M. Huth, eds., *Proceedings of the* 13th *International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS-2007*, number 4424 in LNCS, pp. 519–522, Berlin, Germany, 2007. Springer Verlag.
9. P. D. Mosses, ed. Casl *Reference Manual*. LNCS 2960 (IFIP Series). Springer Verlag, 2004.
10. L. Schröder and T. Mossakowski. HasCasl: Integrated higher-order specification and program development. *Theoret. Comput. Sci.*, 410:1217–1260, 2009.
11. *Introducing SolidWorks*. SolidWorks Corporation, Concord, MA, 2002.
12. VDI-Gesellschaft Entwicklung Konstruktion Vertrieb. *Methodik zum Entwickeln und Konstruieren technischer Systeme und Produkte*, 1995. English title: Systematic approach to the development and design of technical systems and products.
13. W.-T. Wu. *Mechanical Theorem Proving in Geometries*, vol. 1 of *Texts and Monographs in Symbolic Computation*. Springer, 1994.