Formalizing and Operationalizing Industrial Standards*

Dominik Dietrich, Lutz Schröder, and Ewaryst Schulz

DFKI Bremen, Germany
`{firstname.lastname}@dfki.de`

Abstract. Industrial standards establish technical criteria for various engineering artifacts, materials, or services, with a view to ensuring their functionality, safety, and reliability. We develop a methodology and tools to systematically formalize such standards, in particular their domain specific calculation methods, in order to support the automatic verification of functional properties for concrete physical artifacts. We approach this problem in the setting of the Bremen heterogeneous tool set HETS, which allows for the integrated use of a wide range of generic and custom-made logics. Specifically, we (i) design a domain specific language for the formalization of industrial standards; (ii) formulate a semantics of this language in terms of a translation into the higher-order specification language HASCASL, and (iii) integrate computer algebra systems (CAS) with the HETS framework via a generic CAS-Interface in order to execute explicit and implicit calculations specified in the standard. This enables a wide variety of added-value services based on formal reasoning, including verification of parameterized designs and simplification of standards for particular configurations. We illustrate our approach using the European standard EN 1591, which concerns calculation methods for gasketed flange connections that assure the impermeability and mechanical strength of the flange-bolt-gasket system.

1 Introduction

Industrial standards are documents that establish uniform engineering or technical criteria of an item, material, component, system, or service, and are designed to ensure its safety and reliability. To that end, they introduce a precise nomenclature for a limited domain and provide an explicit set of requirements that the item at hand has to satisfy, together with methods to check these requirements. E.g., the European standard EN 1591 [18] defines design rules for gasketed flange connections (see Fig. 1) that guarantee impermeability and mechanical strength of the flange-bolt-gasket system in the form of numerical constraints as well as explicit calculation methods to compute or approximate physical quantities. Performing these calculations in order to verify the target properties for a concrete object can be highly time-consuming and costly; hence, several ad hoc software solutions have been developed to support such calculations (e.g., [4]).

We develop a methodology and tools to systematically formalize such standards, in particular their domain specific computation strategies, to support the automatic verification of the requirements for a concrete physical object. We are particularly interested

---
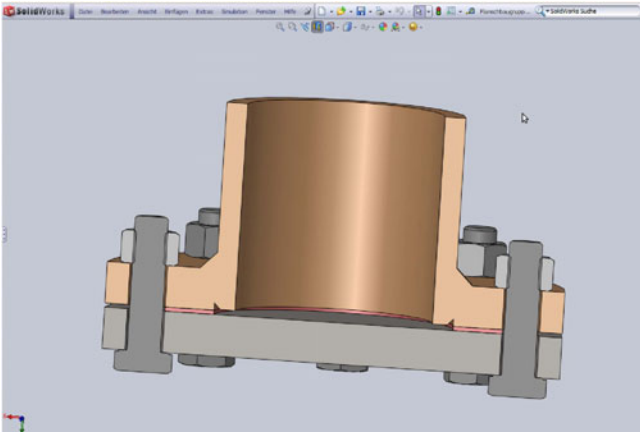
Fig. 1. A flange-bolt-gasket system

in standards giving a guarantee for some functional properties of the system by providing a calculation method. It suffices then to satisfy the criteria of this method in order to ensure these functional properties.

Instead of developing a new system from scratch, our approach consists of designing a new specification language for industrial standards and embedding this language in the institution-based heterogeneous tool integration framework HETS [14], which provides sound and general semantic principles for the integration and translation between different specification languages/logics. This not only allows the reuse of the existing generic machinery provided by the HETS framework, but also gives us direct access to all systems that have already been integrated into the framework, such as the theorem prover Isabelle [15].

The structure of this paper is as follows: Sec. 2 introduces the standard EN 1591. Sec. 3 gives a short overview of relevant parts of the HETS system. The specification language EnCL used to formalize industrial standards is described in Sec. 4 where we also illustrate the architecture of our verification framework for EnCL specifications. We conclude the paper in Sec. 5.

Related work. While there is scattered work on ontological approaches to engineering artifacts, in particular CAD objects (e.g., [1,6]), there is to the best of our knowledge only little existing work on actually formalizing the content of industrial standards, in particular as regards calculation methods. In [12], a more global viewpoint on our overall research goals is given, while the technical results are more oriented towards the representation of CAD geometry. A knowledge-based approach ensuring the safety of pressure equipment is presented in [5]; the formalization in this approach requires more effort than in our framework due to the granularity of the ontology in question. Our approach to formalizing calculations in a logical framework is to some extent related to biform theories [7], the main differences being that we refrain from explicit manipulation of syntax (which instead is left to the CAS operating in the background)

$$W_F = (\pi/4) \times \{f_F \times 2 \times b_F \times e_F^2 \times (1 + 2 \times \Psi_{opt} \times \Psi_Z - \Psi_Z^2) + f_E \times d_E \times e_D^2 \times c_M \times j_M \times k_M\}$$

$$e_D = e_1 \times \left\{1 + \frac{(\beta - 1) \times l_H}{\sqrt[4]{(\beta/3)^4 \times (d_1 \times e_1)^2 + l_H^4}}\right\}$$

$$f_E = \min (f_F; f_S)$$

$$\delta_Q = P \times d_E/(f_E \times 2 \times e_D \times \cos\varphi_S); \quad \delta_R = F_R/(f_E \times \pi \times d_E \times e_D \times \cos\varphi_S)$$

Fig. 2. Some typical definitions from the EN 1591

and moreover work with a loose coupling of algorithmic and axiomatic content via the HETS framework, rather than join the two types of information in mixed theories.

2   Industrial Standards

We proceed to give an overview of the calculation method of the European standard EN 1591. A calculation method is a set of equations, constraints, value tables and instructions. These instructions comprise iterative approximations of a given quantity up to a target accuracy and specify the computation of all data relevant for the checking of the constraints from a set of initial data, which needs to be provided by the user. Figure 3 gives a rough overview of the control flow of the calculation method with an inner and an outer loop. Figure 4 gives a glimpse of how the corresponding instructions look like. It is rather uncommon in mechanical engineering to specify information other

Fig. 3. A control flow diagram from the EN 1591

than mathematical formulas such as equations and inequalities in a formal way, but we can easily represent this instruction as a repeat-until command.
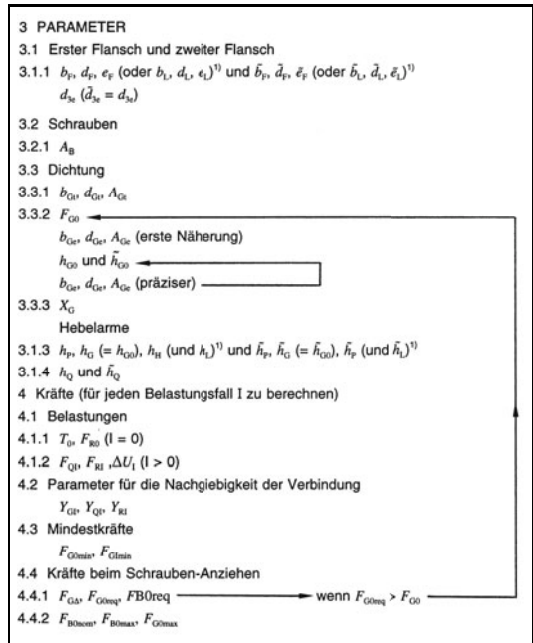
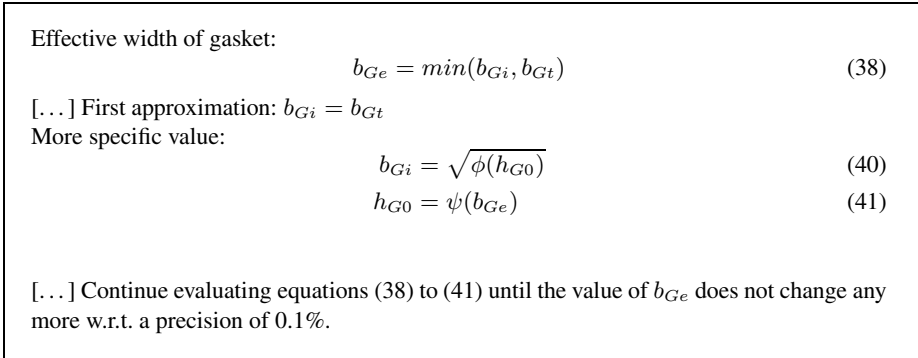Further notable features of the calculational structure of the standard are:

- Most definitions of constants in the standard are given explicitly by equations (see Fig. 2) and in rare cases by value tables corresponding to conditional definitions.

- Constants are sometimes used in the document before they are defined.
- Constants are interpreted as real numbers, and the defining terms contain real constants (π), roots ($\sqrt[n]{}$), elementary functions such as sin and cos, and other functions, e.g., absolute value, min and max, to name only a few.
- The standard contains implicit definitions which can be expressed as optimization problems. This renders the evaluation of the calculation method more challenging.

Effective width of gasket:
$$b_{Ge} = min(b_{Gi}, b_{Gt}) \tag{38}$$
[...] First approximation: $b_{Gi} = b_{Gt}$
More specific value:
$$b_{Gi} = \sqrt{\phi(h_{G0})} \tag{40}$$
$$h_{G0} = \psi(b_{Ge}) \tag{41}$$

[...] Continue evaluating equations (38) to (41) until the value of b_{Ge} does not change any more w.r.t. a precision of 0.1%.

Fig. 4. Iterative Approximation in the EN 1591

3   The Heterogeneous Tool Set

The strategy we pursue to provide formal modelling support for industrial standards and more generally for numerical calculations in engineering is to embed a suitable domain-specific language, to be described in Sec. 4, into the *Bremen Heterogeneous Tool Set (*HETS*)* [14], which integrates a wide range of logics, programming languages, and tools that enable formal reasoning at various levels of granularity. This includes, e.g., ontology languages, equipped with efficient decision procedures, and first-order languages, for which some degree of automated proof support is available, as well as higher-order languages and interactive provers. HETS follows a multi-lateral philosophy where logics and tools are connected via a network of translations, instead of embedding everything into a central interchange logic. The unifying semantic bracket underlying these translation mechanisms and acting as an abstraction barrier in the implementation framework is the theory of *institutions*; we shall not repeat the formal definitions of the concepts of this theory here, but will provide some intuitions concerning its core points.

Our approach will specifically focus on three items in the HETS network:

- the above-mentioned domain-specific language for engineering calculations, EnCL (***En**gineering **C**alculation **L**anguage*), which has an intermediate character between a logic and programming language;
- a computer algebra system (CAS) as an execution engine for EnCL; and
- the higher-order wide-spectrum language HASCASL [16], which serves to make the formal semantics of calculations explicit and to enable advanced reasoning on standards, calculations and concrete engineering designs.

The heterogeneous logical approach involves various translations between these nodes, in particular the following:

- a translation (a so-called *theoroidal comorphism*, a concept discussed in some more detail below) from EnCL to HASCASL which identifies EnCL as sublogic of HAS-CASL;
- a converse translation from the relevant sublogic of HASCASL to EnCL, which enables the use of the CAS as a lemma oracle for HASCASL; and
- the translation from EnCL into the input language of the CAS.

Fig. 5 shows the structure of the HETS logic graph including the above-mentioned features. In the present work, we focus on using the last translation, whose implementation in HETS is already available; implementation of the other two translations is under way.
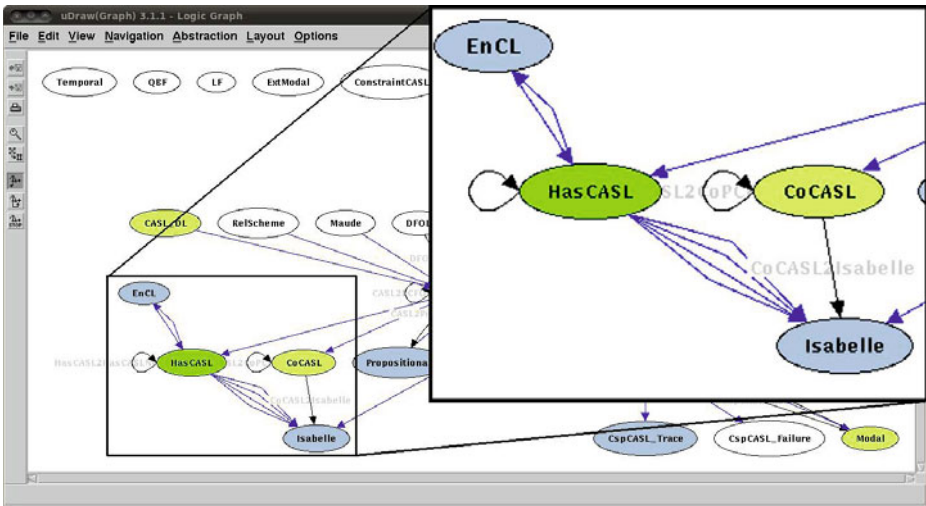

Fig. 5. The HETS logic graph

We now give a brief and informal overview over the concepts of institution and institution comorphism to pave the ground for the description of the respective features in the following sections.

To begin, an *institution* is an abstract logic, construed in a broad sense. It consists of

- a category of *signatures* Σ and *signature morphisms* $\sigma : \Sigma_1 \rightarrow \Sigma_2$ that specify languages of individual theories in the given logic, typically to be thought of as consisting of structured collections of symbols, and their translations, typically renamings and extensions of the set of symbols;
- for each signature Σ, a class of Σ-models interpreting its symbols and a set of Σ-sentences formed using these symbols, together with a *satisfaction relation* $\models$ between models M and sentences ϕ;
- for each signature morphism $\sigma : \Sigma_1 \rightarrow \Sigma_2$, a *sentence translation* map σ that applies the corresponding symbol renaming to Σ_1-sentences and a *model reduction*

that reduces Σ_2-models M to Σ_1-models $M|_\sigma$, typically by interpreting a Σ_1-symbol s in the same way as M interprets $\sigma(s)$.

Moreover, one requires the *satisfaction condition* to hold which essentially states that satisfaction is invariant under signature morphisms, i.e. $M|_\sigma \models \phi$ iff $M \models \sigma(\phi)$.

One of the generic notions built on top of this structure is that of a *theory* $\mathcal{T} = (\Sigma, \Phi)$ consisting of a signature Σ and a set Φ of Σ-formulas. The formulas in Φ are standardly regarded as *axioms*. HETS offers the additional facility to mark formulas in the theory as *implied*, i.e. as logical consequences of the axioms; this gives rise to proof obligations which can be discharged using proof tools associated to the current logic node. A further source of proof obligations are *theory morphisms*, specified as *views* in HETS; here, a theory morphism $(\Sigma_1, \Phi_1) \rightarrow (\Sigma_2, \Phi_2)$ is a signature morphism $\sigma : \Sigma_1 \rightarrow \Sigma_2$ that transforms all axioms of Φ_1 into logical consequences of Φ_2.
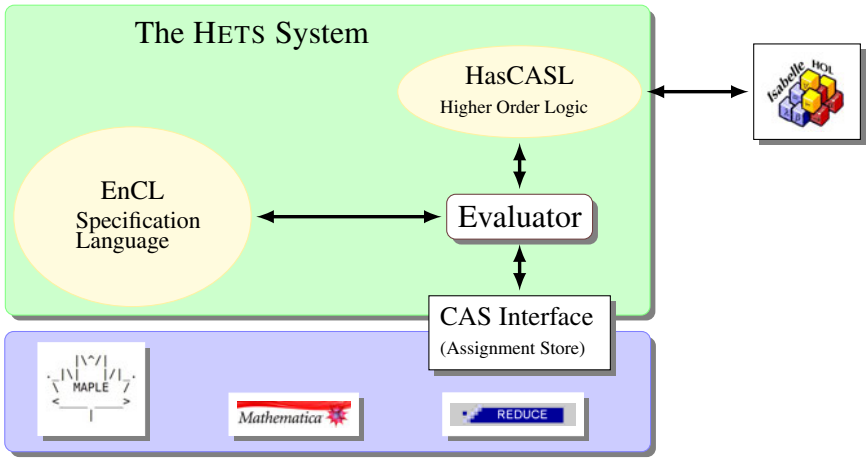

Fig. 6. Architecture of the EnCL Framework

A further logic-independent feature is the ability to build specifications in a modular way by naming, translating, combining, parameterizing and instantiating specifications. These mechanisms are collectively referred to by the term *structured specification*. We will briefly explain some of these mechanisms when they appear in our examples later.

The primary mechanisms used to relate institutions in HETS are *comorphisms* which formalize the notion of logic translation. A comorphism between institutions I and J consists of

- a translation Φ of signatures in I to signatures in J; and
- for each signature Σ in I, a translation α of Σ-sentences into $\Phi(\Sigma)$-sentences and a reduction β of $\Phi(\Sigma)$-models to Σ-models,

again subject to a *satisfaction condition*, in this case that $\beta(M) \models \phi$ iff $M \models \alpha(\phi)$. Below, we will explain how EnCL is cast as an institution, and hence integrated as a logic node in HETS, and how the various translations work.

4   A Domain-Specific Language for Engineering Calculations

Next, we describe the specification language EnCL, which is primarily designed to represent the calculation method of industrial standards but can be used for engineering calculations in general. Our design goals for EnCL are (1) staying close to the informal original description of the standard to minimize the formalization effort and to be readable for engineers and (2) having a precise semantics which allow the formal treatment of the specifications particularly formal verification. Before going into the details we describe briefly the architecture of the EnCL framework for operationalizing industrial standards. EnCL specifications are formalizations of a calculation method, which, given an appropriate set of input values, can be executed. This happens through an evaluator component (see Fig. 6) which communicates with a computer algebra system to evaluate EnCL terms and generates verification conditions that are sent to formal proof assistants for proving.

$<$ Assignment $>$	$::=$ **c :=** $<$ Term $>$ $\mid$ **f**$(x_1, ..., x_n)$ **:=** $<$ Term $>$
$<$ Case $>$	$::=$ **case** $<$ Boolterm $>$ **:** $<$ Program $>$
$<$ Cases $>$	$::=$ $<$ Case $>^+$ **end**
$<$ Sequence $>$	$::=$ **sequence** $<$ Program $>$ **end**
$<$ Loop $>$	$::=$ **repeat** $<$ Program $>$ **until** $<$ Boolterm $>$
$<$ Command $>$	$::=$ $<$ Assignment $>$ $\mid$ $<$ Cases $>$ $\mid$ $<$ Sequence $>$ $\mid$ $<$ Loop $>$
$<$ Program $>$	$::=$ $<$ Command $>^+$
$<$ Range $>$	$::=$ **vars v in** $<$ Set $>$
$<$ Spec $>$	$::=$ $(<$ Command $>$ $\mid$ $<$ Range $>)^+$

Fig. 7. Syntax of EnCL

4.1   Syntax

The EnCL language consists of two layers, one for the definition of the constants used in the calculation and the other for the specification of the calculation method itself. Fig. 7 gives an extended BNF-like grammar of the EnCL language omitting the details of the very standard term-language which is inspired by input languages of computer algebra systems such as Reduce [10], Maple [13] and Mathematica[1]. Following our observations of Sec. 2 the language provides simple assignments, conditional statements and unbound conditional iteration.

Given a signature Σ of predefined constants, functions (see Sec. 2 for some examples) and predicates (essentially comparison operators) we define the following language constructs:

Terms are built as usual over Σ and user defined symbols, i.e., constants and functions. We support also special functions which are binders such as $maximize(t, x)$ defined as the value x' maximizing the term t considered as a function depending on the variable x. This is because we do not support lambda abstraction as this concept is mostly

---

[1] http://reference.wolfram.com/mathematica/guide/Mathematica.html

unused in the mechanical engineering community. We also make a difference between Boolean valued terms and numerical terms and do not allow constants to be assigned to a Boolean value because we do not need it in our current setting.

Assignments of user defined symbols, which are constants c or function patterns $f(x_1, ..., x_n)$, to terms t.

Conditionals. We support conditionals on the command-level, i.e., not inside terms. We will distinguish later between conditionals in programs and conditional assignments which contain only assignments after being flattened, i.e., conditionals which contain only conditional assignments and assignments.

Sequences allow the marking of a sequence of commands, typically assignments, explicitly as a program sequence instead of treating them as assignments. This is important for the evaluation of a specification as described in the next section.

Loops may contain the *c*onvergence predicate in the exit condition beside the predefined comparison operators. Let t be a term which is meant to converge inside a *r*epeat loop and e an expression denoting the acceptable tolerance of t.

convergence(e, t) is defined to be true if and only if the difference of the value of t before the current loop evaluation and afterwards is in the interval $[-|e|, |e|]$.

Fig. 8 shows an excerpt of the specification of the EN 1591. The assignments from Fig. 2 are mainly kept unchanged with the exception that W_F is represented as a function with two arguments. This was necessary because further in the standard W_F is really used as a function, i.e., the arguments k_M and Ψ_Z are used as variables and not as constants.

library ENCL/EN1591
logic ENCL
spec EN1591[FLANGEPARAMETER] = . . .

$\quad$ *W_F(k_M', Psi_Z') := Pi / 4 * (f_F * 2 * b_F * e_F ˆ 2*
$\qquad\qquad\qquad * (1 − Psi_Z'ˆ2 + 2 * Psi_opt * Psi_Z')*
$\qquad\qquad\qquad + f_E * d_E * e_Dˆ2 * c_M * j_M * k_M')$ $\qquad$ %(eq74)%
$\quad$ *e_D := e_1 * (1 + (beta − 1) * l_H / fthrt((beta / 3)ˆ4*
$\qquad\qquad\qquad * (d_1 * e_1)ˆ2 + l_Hˆ4))$ $\qquad$ %(eq75)%
$\quad$ *f_E := min(f_F, f_S)* $\qquad$ %(eq76)%
$\quad$ *delta_Q := P * d_E / f_E * 2 * e_D * cos(phi_S)* $\qquad$ %(eq77.1)%
$\quad$ *delta_R := F_R / f_E * Pi * d_E * e_D * cos(phi_S)* $\qquad$ %(eq77.2)%

Fig. 8. EnCL specification of the standard EN 1591

4.2   Semantics

There are two notions of semantics to be distinguished here. First, we want to represent the background theory of the objects dealt with in EnCL specifications, i.e., real numbers and real functions, and second we want to give a meaning to a EnCL specification as a whole in order to talk about the execution and the correctness of a EnCL specification.

Theory of Real Functions

The reason why we need a formalization of the theory of real functions is that we want to prove the correctness of computations which are specified in a EnCL specification and carried out by the evaluator component of our framework. The expressions in the computations refer to elementary real functions such as *cos*; hence we need to do the proofs in the context of a theory specifying those functions. Rather than formalize the required portion of analysis from scratch, which is itself a time-consuming endeavor [3], we base our approach on formal tools that provide a library containing an appropriate theory in our case Isabelle/HOL [15] and MetiTarski [2]. We focus on the Isabelle/HOL system because HETS already provides an interface to Isabelle and the interactive setting in Isabelle seems better suited for first experiments, in particular the `Approximation` theory[2] developed by Hoelzl providing proof support for inequalities over the reals [11]. In a later stage of the project where we aim at full automation of the correctness proofs, however, we plan to integrate MetiTarski. Both systems formalize elementary functions such as sine and cosine by Taylor series (e.g. in Isabelle/HOL in the `Transscendental` theory[3]).

Evaluation

The EnCL specification language contains as sub-language the language of assignments, i.e., specifications which consist only of assignments and conditional assignments as defined in Sec. 4.1. We will call commands of this sub-language simply assignments. A specification can hence be divided into a program skeleton containing only non-assignments, e.g., repeat-loops and sequences, and assignments. We require that for the assignments (1) there is at most one assignment for each constant and (2) there is no cyclic dependency between two constants, i.e., the dependency graph for the assignments has no cycles. We split the evaluation according to the divided specification into the evaluation of the program skeleton and an assignment store. This assignment store supports requests to evaluate a term containing constants which are defined by some assignments in the store. The result is a fully evaluated expression, where all defined constants were recursively substituted by their assigned value in the current environment. Typically, a specification splits into a small program containing only a few assignments (see Fig. 9 for an example) and a big assignment store. Fortunately many computer algebra systems support exactly the feature of full evaluation required from an assignment store and can hence be used as such in our framework. Currently we support the computer algebra systems Reduce, Maple and Mathematica.

Verification

Within our framework, the evaluator component interprets EnCL programs and uses an external assignment store with which it communicates via a generic interface, the CAS-Interface. The complicated parts of the evaluation, namely the evaluation and computation of the terms, are outsourced to the CAS, and are in general not guaranteed to be carried out correctly. There are many examples for erroneous computations in CAS [11]

---

[2] http://isabelle.in.tum.de/dist/library/HOL/Decision_Procs/Approximation.html
[3] http://isabelle.in.tum.de/dist/library/HOL/Transcendental.html

spec EN1591[FLANGEPARAMETER] = ...
    repeat
     . *repeat*
       . *b_Gi := sqrt(e_G / Pi * d_Ge * E_Gm*
                */ (h_G0 * Z_F / E_F0 + h_G0 * Z_F / E_F0)*
                *+ (F_G / Pi * d_Ge * Q_maxy)ˆ2)*
     until convergence(1.0e−3, b_Ge)
    until convergence(1.0e−3, F_G) and F_G0req <= F_G   **%**(eq54)**%**

Fig. 9. EnCL program skeleton of the standard EN 1591

which justify our prudence in this respect even if most of the computations give correct results.

We verify a computation as follows (see Sec. 4.4 for an example). Before we start the evaluation we mark all constants in the assignment store. A *marked constant* stands for the fact that its value has *changed*. We have two rules for marking and unmarking constants:

1. When the evaluator is at the position of an assignment, all constants affected by this assignment are marked.
2. When we generate a verification condition for an assignment, the assigned constant is unmarked.

We trigger the generation of verification conditions when the evaluator is at one of the following three positions: (1) at an assignment, (2) at an until condition of a repeat loop or (3) at a case condition. Depending on the position of the evaluator we generate a verification condition for the assignment in the first case and for the condition in the other cases. In addition we generate in all three cases verification conditions for the assignments of all marked constants which affect the value of the expression in question, namely the assignment in case (1) and the condition in case (2) and (3). This method guarantees that we do not produce obvious copies of already generated verification conditions. For verification condition generation purposes we can treat a condition Φ which is a Boolean term exactly as we treat assignments: we consider Φ as the assignment $b := \Phi$ with an auxiliary constant b, therefore we will only describe the generation of verification conditions for the assignment case. Given an assignment $y := t(x_1, ..., x_n)$ depending on the constants $x_1, ..., x_n$ we generate the verification condition as follows. For each x_i we request its value v_i from the assignment store. We then request the value w for the expression $t(x_1, ..., x_n)$. The condition that the result is correct w.r.t. the current environment is now expressed as the equation $t(v_1, ..., v_n) = w$. If we can prove this equation in the context of the background theory, then we have proved the correctness of the computation of y.

In addition to the verification of isolated assignments we want to formally specify the evaluation semantics described in the previous paragraph, in order to support the full formal verification of a run of the calculation method. For this purpose we specify the semantics in HASCASL and keep it parametric over the background theory of the term language. This allows us to test different prover back-ends in parallel with different instantiations of the background theory.

Numerical Expressions and Uncertainty Propagation

An important issue for the CAS-Interface are numbers with a bounded precision. Consider, e.g., that we want to accept a certain tolerance in the input values and we provide as input to the computation instead of a value an interval, representing the bounds for the value. On the other hand the assignment store could also be limited concerning the precision of the result and give us only a numerical approximation instead of an exact representation of the result. In order to treat such imprecise values correctly we need the assignment store to support uncertainty propagation. Current computer algebra systems provide here only limited support. Mathematica uses significance arithmetic as built-in support for uncertainty propagation [17] whereas the *intpakX* package [9] provides support for interval arithmetic in Maple which unfortunately does not apply to computations from other packages such as *Optimization*.

In the presence of uncertain values we have to review the generation of verification conditions. An uncertain value v in the assignment store is an interval, i.e., lower and upper bound ($v = [\underline{v}, \overline{v}]$) for the actual value, and we have to generate instead of an equation two inequalities with a common precondition expressing the bounds for the input values. Replacing the inequalities by interval-membership, the verification condition from the previous paragraph becomes

$$\forall x_1, ..., x_n. (\bigwedge_{i=1}^{n} x_i \in v_i) \Rightarrow t(x_1, ..., x_n) \in w$$

4.3   EnCL as an Institution

As indicated in Sec. 3, we need to define an institution for EnCL in order to integrate EnCL into the HETS network. This is not an entirely straightforward enterprise as EnCL mixes logical features with traits that are more typical of a programming language. A definition which leads to a relatively smooth integration with the existing logic graph is the following.

- *Signatures* in EnCL are just collections of function symbols with associated arities, including nullary functions, i.e. constants. These are understood as functions on the space of real numbers.
- A *model* of a signature is just an environment, i.e. an assignment of an n-ary function on the reals to every n-ary function symbol in the signature. Intuitively (although not formally), there is the slight twist here that environments are variable: executing a program will typically modify the environment.
- EnCL has two types of *sentences*:
  - The syntax described in Sec. 4.1 yields *programs*, which form one type of EnCL-sentences.
  - A second type of sentences called *answers* captures the results of actually running a EnCL program. These sentences are not typically expected to be input by the user (although this is technically possible, e.g., in order to simplify the answers manually), but are instead generated as lemmas after running a EnCL program using the back-end CAS. The syntactic format for answers are pairs (p, η) consisting of a program p and an environment η, to be understood intuitively as the statement 'correct evaluation of p yields the environment η'.

– Corresponding to the two types of sentences, there are two cases in the definition of satisfaction:
- A model *satisfies* a *program* p if p terminates successfully when run in the corresponding environment (e.g., when the desired factorizations or minima exist).
- A model η *satisfies* an *answer* (ϕ, η') if correct execution of p in the environment η terminates successfully and yields η'.

This definition is designed in such a way that the translation of calculation goals in EnCL programs (such as minimization of a function) into existential formulas is sound. Formally, we have the following comorphism from a sublogic of HASCASL to EnCL, which serves the purpose of making EnCL available as a calculational tool in specification development. To begin, the relevant sublogic of HASCASL is the one given by all theories that

- extend the standard HASCASL theory of the real numbers;
- introduce no additional signature except constants of types that occur as input or result types in EnCL statements, i.e. n-ary real functions for $n \geq 0$;
- introduce formulas only in a limited syntax mirroring the abilities of EnCL. Specifically, sentences can be of the form (1) $\exists x.\, \phi(x)$, or (2) $\phi(c)$, where ϕ corresponds to the semantics of a CAS calculation and, e.g., states that x is the set of zeros of a given polynomial, the minimum of a given function, the factorization of a given number etc., and c is a constant. Moreover, all formulas are marked as implied, i.e. no new axioms are introduced. Formulas of type (1) represent goals, and as such are meant to be input by the user, while formulas of type (2) represent answers from the CAS, typically generated automatically as exported lemmas as described in Sec. 4.2.

Theories in this sublogic are translated into EnCL along a comorphism which suppresses the explicit theory of the reals (which is implicit in EnCL) and otherwise behaves as follows.

- Signatures remain unchanged, except that the explicit type information present in the original HASCASL signature is erased.
- Existential formulas of the form (1) are replaced by assignments $x := e$ where e is the CAS statement corresponding to ϕ, e.g., a factorization or minimization statement. Formulas of the form (2) are trivially reinterpreted as answers in the sense of the definition of EnCL sentences.
- Model reduction trivially reinterprets EnCL Models as HASCASL models.

It is easy to check that this does indeed constitute a comorphism, i.e. fulfills the satisfaction condition. Note that this is independent from the fact that the CAS itself may be buggy – in a manner of speaking, EnCL is an abstraction of the workings of the CAS which presupposes correctness.

Remark 1. One could extend the definition of the relevant sublogic of HASCASL so as to exploit the full programming power of EnCL. As this does not really yield additional insights conceptually, details are omitted.

Conversely, we have a translation of EnCL into HASCASL which makes the semantics of EnCL programs explicit and hence enables full formal reasoning over entire calculations. It takes the shape of a so-called *theoroidal comorphism* where we associate to each EnCL signature not just a HASCASL signature (as in a plain comorphism) but a HASCASL theory, which imports a HASCASL specification of the EnCL semantics. The former is just a straightforward encoding of the transformations on environments effected by the various EnCL constructs. A EnCL program then induces a partial function p representing its semantics, and as a EnCL sentence is translated into a definedness assertion amounting to the statement that p terminates when run from the specified state. A EnCL answer (p, η) is just reinterpreted as the obvious formula stating that running p yields η. As indicated above, this translation enables fine-grained reasoning over EnCL calculations, including, besides the verification of individual results of the CAS, the verification of entire EnCL programs.

```
spec POLYFACTOR =
    . z0 := z4 + z3 + 20 %(coef0)%              . z1 := z2 + 3 * z4 + 4 %(coef1)%
    . z2 := 3 * z3 − 30 %(coef2)%               . z3 := 5 * z4 + 10 %(coef3)%
    . z4 := 15 %(coef4)%
    sequence
      . z := factor(x ˆ 5 − z4 * x ˆ 4 + z3 * x ˆ 3 − z2 * x ˆ 2 + z1 * x − z0)
    end                                                      %(program)%
end
```

Fig. 10. EnCL specification for polynomial factorization

4.4   Example

To illustrate the work-flow of a calculation in the EnCL framework, we process the specification shown in Fig. 10 step-by-step. The static analysis splits the specification into an assignment store with five assignments, coef0, ...,coef4, and a program consisting of the single assignment in the *s*equence block. The evaluator moves the assignment store to a computer algebra system via the CAS-Interface and puts the instruction pointer at the single assignment of the program. At the beginning all assignments in the assignment store are marked. The assignment for z depends on the constants x and $z0$ to $z4$, but there are only assignments for $z0$ to $z4$ in the assignment store from which we can generate verification conditions. Hence we order the corresponding assignments w.r.t. the dependency graph and generate the verification conditions for them. This produces only trivial verification conditions such as $15 + 85 + 20 = 120$. The constants $z0$ to $z4$ are now unmarked. In the next step, the evaluator stores the current assignment in the assignment store and generates the verification condition for this assignment,

$$factor(x\,\hat{}\,5 − 15 * x\,\hat{}\,4 + 85 * x\,\hat{}\,3 − 225 * x\,\hat{}\,2 + 274 * x − 120)$$
$$= (x − 5) * (x − 4) * (x − 3) * (x − 2) * (x − 1)$$

This verification condition is translated to Isabelle and proved by a short Isar-proof in three steps Fig. 11. The background theory is based on the Isabelle formalization of the reals where we added a definition for the (nearly dummy) factor operator requiring only that factorizing a term does not change its value.

```
theory factor imports Real
begin constdefs factor :: "real => real" "factor(x) == x"
theorem factor1 : "!! x. factor(x^5-15*x^4+85*x^3-225*x^2+
274*x-120) = (x-5)*(x-4)*(x-3)*(x-2)*(x-1)"
(is "!! x. factor (?a x) = (?b x)")
proof -
  fix x::real
  have "(x-5)*(x-4)*(x-3)*(x-2)*(x-1) =
  x*x*x*x*x-(x*x*x*x)*15+(x*x*x)*85-(x*x)*225+274*x-120"
  by (simp add: ring_simps)
  also have "... = x^5-15*x^4+85*x^3-225*x^2+274*x-120"
  by (simp add: Groebner_Basis.class_semiring.semiring_rules)
  also have "... = factor (?a x)" by (simp add: factor_def)
  finally show "factor (?a x) = (?b x)" by simp
qed
```

Fig. 11. Isabelle proof for the validity of the polynomial factorization

5   Conclusion

We have developed a methodology to formalize industrial standards and a method to execute such formalizations based on the HETS framework. Specifically, we have designed a domain specific language EnCL for engineering calculations which allows for the formulation of a given calculation method that stays close to the original formulation in the standard. The integration of this language into the heterogeneous logic framework HETS enables us to relate these specifications to theories available in HAS-CASL, such as ontological summaries of CAD designs [8] or abstract geometric representations of CAD objects [12], in order to automate the parameter extraction for the concrete computation. We have also integrated a computer algebra system (CAS) interface into HETS and instantiated it with several state-of-the-art CAS. This allows us to outsource the calculational part of EnCL specifications, which is a rather natural choice to handle the computations in the presence of implicit definitions, such as references to the argument value which minimizes a function in a given range. A key point here was to cast a mainly procedural input language for a CAS as an institution.

Potential benefits of the formal approach beyond the applications presented here include

– statement and proof of formal consequences of the prescriptions of the standard, e.g., explicit formulas for maximum calculations;
– partial instantiations of the standard to particular situations and ensuing simplification of the calculation procedures (e.g., when some parameters become 0, a fairly typical situation);
– full formal verification of designs.

Acknowledgements

References

1. Abdul-Ghafour, S., Ghodous, P., Shariat, B., Perna, E.: A common design-features ontology for product data semantics interoperability. In: IEEE/WIC/ACM International Conference on Web Intelligence, WI 2007, pp. 443–446. IEEE Computer Society, Los Alamitos (2007)
2. Akbarpour, B., Paulson, L.C.: Metitarski: An automatic theorem prover for real-valued special functions. J. Autom. Reasoning 44(3), 175–205 (2010)
3. Barendregt, H., Wiedijk, F.: The challenge of computer mathematics. Phil. Trans. R. Soc. A 363(1835), 2351–2375 (2005)
4. Bullack, H.-J.: Flanschberechnungen nach EN 1591. Kamprath interaktiv, 1st edn. (2006)
5. Camossi, E., Giannini, F., Monti, M., Brogotto, P., Pittiglio, P., Ansaldi, S.: Ontology Driven Certification of Pressure Equipments. Process safety progress 27(4), 313–322 (2008)
6. Colombo, G., Mosca, A., Sartori, F.: Towards the design of intelligent cad systems: An ontological approach. Advanced Engineering Informatics 21(2), 153–168 (2007)
7. Farmer, W.M.: Biform theories in chiron. In: Kauers, M., Kerber, M., Miner, R., Windsteiger, W. (eds.) MKM/CALCULEMUS 2007. LNCS (LNAI), vol. 4573, pp. 66–79. Springer, Heidelberg (2007)
8. Franke, M., Klein, P., Schröder, L.: Ontological semantics of standards and plm repositories in the product development phase. In: Proc. 20th CIRP Design Conference 2010. Springer, Heidelberg (to appear, 2011)
9. Grimmer, M., Petras, K., Revol, N.: Multiple precision interval packages: Comparing different approaches. In: Alt, R., Frommer, A., Kearfott, R.B., Luther, W. (eds.) Dagstuhl Seminar 2003. LNCS, vol. 2991, pp. 64–90. Springer, Heidelberg (2004)
10. Hearn, A.C.: REDUCE User's Manual, Version 3.8. RAND (2005)
11. Hölzl, J.: Proving real-valued inequalities by computation in Isabelle/HOL. Diploma thesis, Institut für Informatik, Technische Universität München (April 2009)
12. Kohlhase, M., Lemburg, J., Schröder, L., Schulz, E.: Formal management of cad/cam processes. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 223–238. Springer, Heidelberg (2009)
13. Maplesoft. Maple 10 User Manual (2005)
14. Mossakowski, T., Maeder, C., Lüttich, K.: The Heterogeneous Tool Set. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 519–522. Springer, Heidelberg (2007)
15. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)
16. Schröder, L., Mossakowski, T.: HasCASL: Integrated higher-order specification and program development. Theoret. Comput. Sci. 410, 1217–1260 (2009)
17. Sofroniou, M., Spaletta, G.: Precise numerical computation. J. Logic Algebraic Programming 64(1), 113–134 (2005)
18. Technical Committee CEN/TC 74. EN 1591 – Flanges and their joints – Design rules for gasketed circular flange connections (2001)