URL: http://www.elsevier.nl/locate/entcs/volume80.html 23 pages

Integration of Informal and Formal Development of Object-Oriented Safety-Critical Software: A Case Study with the KeY System

Richard Bubel [1,2,3,4]

University of Karlsruhe
Department of Computer Science
D-76128 Karlsruhe, Germany

Reiner Hähnle [1,3,5]

Chalmers University of Technology
Department of Computing Science
S-41296 Gothenburg, Sweden

Abstract

The KeY system allows integrated informal and formal development of object-oriented JAVA software. In this paper we report on a major industrial case study involving safety-critical software for computation of a particular kind of railway time table used by train drivers. Our case study includes formal specification of requirements on the analysis and the implementation level. Particular emphasis in our research is put on the challenge of how authoring and maintenance of formal specifications can be made easier. We demonstrate that the technique of *specification patterns* implemented in KeY for the language OCL yields significant improvements.

1 Introduction

In this paper we report on one of the first case studies done with the KeY tool. This is also one of the first full-fledged industrial case studies, where formal methods are applied to object-oriented software development.

[1] We would would like to express our gratitude to all members of the KeY team for their continuing committment. Sincere thanks go to *DBSystems* for the kind permission to use their SbF product in this case study and for the permission to publish the results.
[2] Support by Deutsche Forschungsgemeinschaft is kindly acknowledged.
[3] Supported by a STINT Institutional Travel Grant.
[4] Email: `bubel@ira.uka.de`
[5] Email: `reiner@cs.chalmers.se`

The KeY system allows integrated informal and formal development of object-oriented Java software. The structure and philosophy of the KeY tool is briefly described in Section 2, and in detail in [1]. The case study is derived from a safety-critical industrial software package, and is concerned with computing a specialized railway time table for train drivers from certain properties of the engine and the track infrastructure. This application is described in Section 3.

Our starting point was an implementation in Smalltalk plus an informal requirements specification in German language. Using the KeY tool we began by extracting a UML-based analysis model from the informal specification. Requirements that could not be captured in UML, were expressed in the Object Constraint Language (OCL), a textual and formal constraint language, which is a substandard of the UML. We discuss our OCL formalization in Section 4.

The KeY tool supports users in writing and maintaining specifications written in OCL in various ways: of course, there is a parser and semantic analyser for OCL expressions. More importantly, a number of *specification idioms* and *patterns* help users to get started and produce specifications for standard situations in a fast and reliable way. Standard design patterns (such as GoF patterns) are enriched by formal specifications of their characteristic properties. We illustrate this feature and discuss its advantages in Section 5.

In a second step, we reimplemented the target application in Java, which is the target language of KeY. At this point, additional OCL constraints pertaining to the implementation level were added. Work on automatic refinement of constraints in KeY, based on user-defined refinements of the model or implementation, is in progress [11], but could not yet be employed. Accordingly, the analysis level constraints had to be remodelled on the implementation level by hand. This is work in progress, and it is described in Section 6.

Before we conclude the paper, we talk about the lessons that we learned from this case study. These relate to formal methods in object-oriented software development in general, to using UML/OCL, and to consequences for the further development and usability of KeY.

2 The KeY Project

The aim of the KeY project (`i12www.ira.uka.de/~key`) is the integration of informal, but practically used software development processes with formal methods. It has been shown time and again that formal methods work: used in suitable domains by expert users, they can drastically improve the quality of specifications and software, exhibit bugs that are not found by other means, and increase the trustworthyness of software by mechanical verification of critical properties. The number of security- and safety-critical software applications is growing significantly. The ubiquity of powerful, embedded computers only sharpens this trend.

Why then, are formal methods not used much more often, and often denigrated by practitioners as useless academic toys? One could try to blame it to short-sightedness and ignorance in the software industry (and perhaps point to the hardware sector, where formal methods became indispensable already a while ago). But the academic formal methods community should realize that many complaints of the industry directed to them are justified:

- Very few formal methods are integrated into standard industrial software development processes and tools.
- Very few formal methods support actually used programming languages.
- Many proponents of formal methods envisaged that they would replace traditional development tools. This is wrong: such ingredients of software engineering as object-oriented modeling, patterns, refactoring, testing, code reviews (to name just a few) are efficient and work well. The point of formal methods is to extend the reach of traditional methods in those areas, where quality matters a lot.

In the KeY project, we address the concerns expressed above: on the surface (that is, after starting up), the KeY tool looks like a standard UML-based CASE tool for object-oriented SW development. In fact, KeY is (currently) based on the commercial CASE tool Borland Together Control Center[6]. The user sees a few additional menus pertaining to formal method support. These can be used (or ignored) in as much as one wants to work formally. To use the "formal" part of KeY, one attaches system requirements formally expressed in OCL to UML model elements. The user is supported in authoring such OCL constraints. Next, from a predefined set of standard properties of specifications (e.g., consistency or adherence to structural subtyping), one can automatically generate proof obligations. This entails (fully automatic) translation from OCL to first-order logic [5].

For implemented classes and operations, it is also possible to formally verify the implementation against the attached OCL constraints. The target language of KeY is Java Card, a sublanguage of Java intended to run on small embedded devices such as smart cards. In practice, any Java program that adheres to the restrictions expressed in the Java Card standard [12] can be handled. The main restrictions are: no multithreading, no dynamic class loading, no floating point types. Many realistic Java applications, including the present case study, lie in this class. Although not all Java (or even Java Card) features can be handled yet in KeY, it has a larger coverage of the Java language than any other Java source code verifier. It includes all object-oriented features, exceptions, Java integer data types, loops, aliasing, expressions with side effects, and abrupt termination.

Again, there is a set of standard proof obligations (e.g., total correctness wrt given pre- and postconditions) that can be generated automatically. Proof

[6] `www.togethersoft.com/products/controlcenter/index.jsp`

obligations in logic are obtained from a given UML model, OCL constraints, and the currently selected class or method. We use an extension of dynamic logic called Java DL [4] to express and reason about properties of Java programs. Most aspects (currently about 90%) of the Java Card language are axiomatized in Java DL. The KeY tool features a powerful interactive theorem prover for mechanical derivation of Java DL formulas. The proofs follow the *symbolic execution* paradigm and are relatively intuitive.

3 The Case Study

Several hundred trains run at any given time on the network of the German railway company *Deutsche Bahn AG (*DB*)*. Strict compliance by the train driver to numerous restrictions such as speed limits, signals and brake distances, is an absolute safety requirement. [7]

Fixed rules (such as maximum speeds for a given stretch) as in use for individual road traffic are not enough: applicable restrictions for trains depend in complex ways on several track infrastructure and train parameters. These include acclivity/declivity, engine type, number and type of cars, tilting capabilities, etc. In order to realize full flexibility, also with regard to future technological developments, an individual schedule for the train driver is computed for each train/route combination, which contains the restrictions to obey on each segment of the journey. This schedule is published either in printed or in electronic form. Fig. 1 shows the initial part of the schedule for train 775 from Hamburg-Altona to Basel. The header contains general information such as:

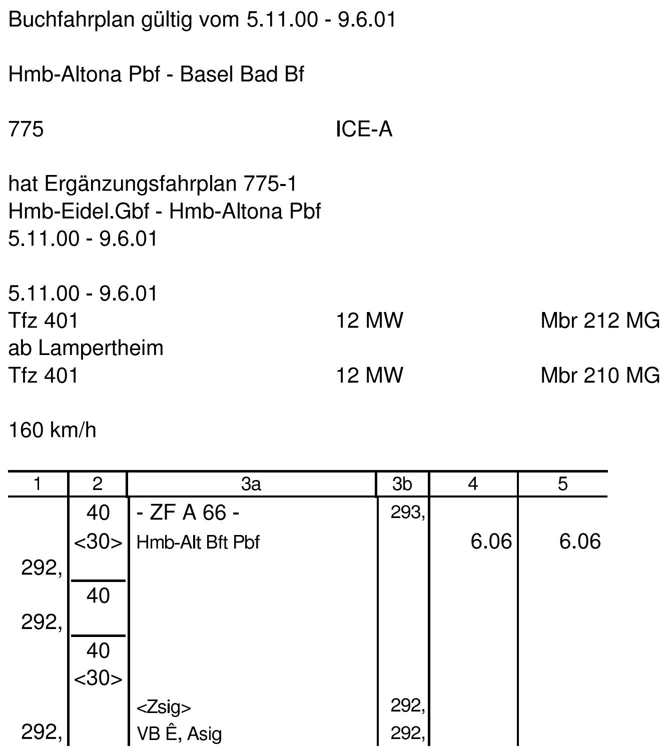
- period of validity (here: 5 Nov 2000 to 9 June 2001);
- route (Hamburg-Altona Pbf [8] to Basel Bad Bf), train type (ICE-A), and train number (775);
- used train engine (Tfz 401) and minimal required brake power (up to Lampertheim: Mbr 212 MG , ex Lampertheim: Mbr 210 MG), as well as the maximum of the computed train speed during any part of the journey.

The columns below the header give detailed information for each route segment: the first column is the kilometer distance that uniquely identifies a segment of a railway line. The value of the second column is the allowed maximum speed at the current position. This column can be divided up into two separate columns (2a and 2b) when, for example, train and infrastructure permit the use of tilting technology. If two rail tracks (left and right) are available, then the speed for the left rail is given in angle brackets. [9] Additional

[7] The case study uses traditional technical terms in German language. We tried as far as possible to give intuitive translations for them, but we do not claim adherence to railway specific vocabulary in English, which uses somewhat different terms anyway.

[8] Pbf: train station for passengers (in contrast to Gbf: station for freight).

[9] In contrast to most other countries, trains drive by default on the right in Germany.



Buchfahrplan gültig vom 5.11.00 - 9.6.01

Hmb-Altona Pbf - Basel Bad Bf

775 ICE-A

hat Ergänzungsfahrplan 775-1
Hmb-Eidel.Gbf - Hmb-Altona Pbf
5.11.00 - 9.6.01

5.11.00 - 9.6.01

Tfz 401	12 MW	Mbr 212 MG
ab Lampertheim		
Tfz 401	12 MW	Mbr 210 MG

160 km/h

1	2	3a	3b	4	5
	40	- ZF A 66 -	293,		
	<30>	Hmb-Alt Bft Pbf		6.06	6.06
292,					
	40				
292,					
	40				
	<30>				
		<Zsig>	292,		
292,		VB Ê, Asig	292,		

Fig. 1. Schedule for an ICE from Hamburg to Basel (from: [13])

information about signals, shortened brake distances, tunnels, etc., is given in column 3a; The remaining columns are not relevant for this paper.

These schedules are computed by the software system *Satzerstellung betrieblicher Fahrplanunterlagen (SbF)* developed by *DBSystems*, the information technology company of DB. SbF is written in Smalltalk. It was necessary to reimplement parts of the system, because KeY supports Java as target language for verification.

SbF is clearly a safety critical application, as for example a too high speed may easily result in a derailment of the train, in particular in curves or the train may be unable to stop in time.

DBSystems kindly let us have an informal requirements specification in (German) natural language and the implementation itself as a starting point. One of the first steps was to extract an analysis model out of the written specification with only marginal use of implementation details. The resulting class diagram is shown in Fig. 2. We now describe briefly its main classes.

`TrackPoint`**:** a train route is modelled by a directed acyclic graph with track points as nodes. A node can have up to two ingoing and up to two outgoing edges depending on whether the route allows to switch between rails at this point. The usage of an association (instead of an aggregation) when modelling the edges, enforces the explicit specification of the graph property "acyclic" in OCL. A train object (`train`) and a non-empty set of infrastructure properties (`properties`) is owned by each track point. A track

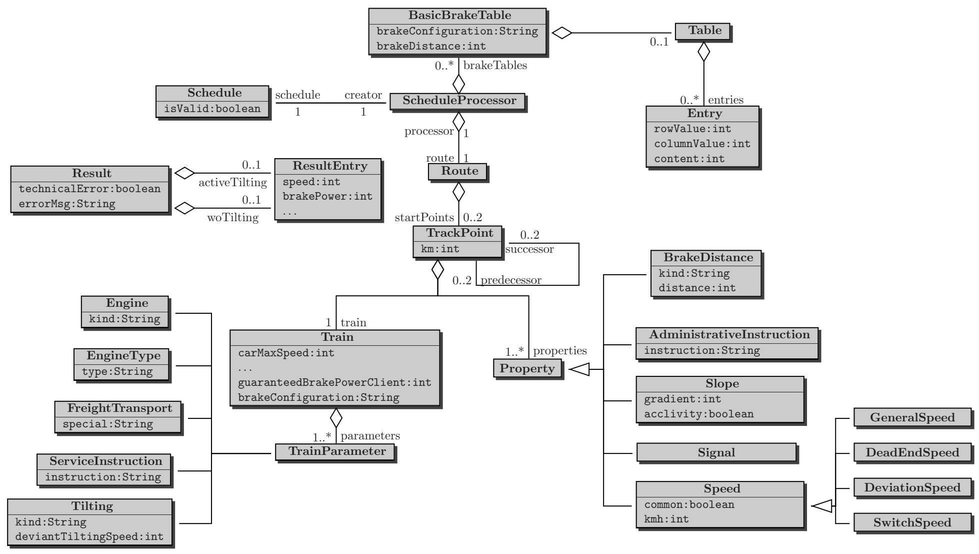


Fig. 2. Class diagram at analysis level

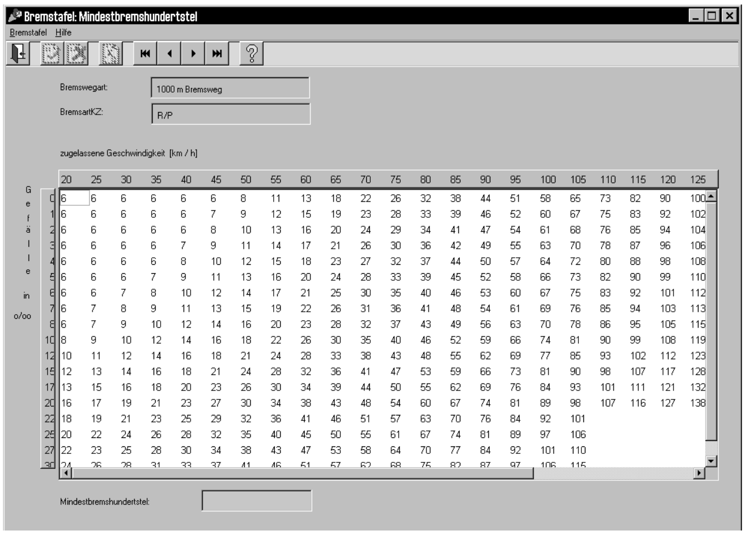
point is modelled each time the train or one of the infrastructure properties changes.

(Infrastructure) `Property` (and subclasses) model the characteristics of the rail infrastructure, for example, speed limits (`Speed`). Note that there can be several kinds of speed limits active at the same time: in addition to the general speed (`GeneralSpeed`) derived from physical track properties, this can be, for example, a dead-end track speed (`DeadEndSpeed`). Other properties include slope (`Slope`) or the available brake distance (`Brake-Distance`).

`Train`, `Trainparameter` (and subclasses) model a train including the used engine (`Engine`, `EngineType`), as well as service instructions or available technologies like tilting. The guaranteed brake power from the client (ie, train) side (`guaranteedBrakePowerClient`) and the configuration of brakes (`brakeConfiguration`) are important train attributes.

A `ScheduleProcessor` computes the schedule's speed column based on the modelled infrastructure. This computation makes crucial use of so-called brake tables:

`BasicBrakeTable` brake tables describe a relation between slope (rows), speed (columns), and the required brake power (entries). If the current position of a train has a certain declivity, then the train must guarantee at least the brake power specified by the brake table for the current declivity and speed limit. For the acclivity case, the speed limit is a default roll back speed value. The relation depends on the brake configuration (attribute `brakeConfiguration`) and the available brake distance (attribute `brakeDistance`). There is a unique brake table for each combination of these two attributes.[10] An example brake table (for brake configuration "R/P" and brake distance 1000m) is in Fig. 3.



Bremswegart: 1000 m Bremsweg

BremsartKZ: R/P

zugelassene Geschwindigkeit [km / h]

Gefälle in o/oo

	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95	100	105	110	115	120	125
0	6	6	6	6	6	6	8	11	13	18	22	26	32	38	44	51	58	65	73	82	90	100
1	6	6	6	6	6	7	9	12	15	19	23	28	33	39	46	52	60	67	75	83	92	102
2	6	6	6	6	6	8	10	13	16	20	24	29	34	41	47	54	61	68	76	85	94	104
3	6	6	6	6	7	9	11	14	17	21	26	30	36	42	49	55	63	70	78	87	96	106
4	6	6	6	6	8	10	12	15	18	23	27	32	37	44	50	57	64	72	80	88	98	108
5	6	6	6	7	9	11	13	16	20	24	28	33	39	45	52	58	66	73	82	90	99	110
6	6	6	7	8	10	12	14	17	21	25	30	35	40	46	53	60	67	75	83	92	101	112
7	6	7	8	9	11	13	15	19	22	26	31	36	41	48	54	61	69	76	85	94	103	113
8	6	7	9	10	12	14	16	20	23	28	32	37	43	49	56	63	70	78	86	95	105	115
10	8	9	10	12	14	16	18	22	26	30	35	40	46	52	59	66	74	81	90	99	108	119
12	10	11	12	14	16	18	21	24	28	33	38	43	48	55	62	69	77	85	93	102	112	123
15	12	13	14	16	18	21	24	28	32	36	41	47	53	59	66	73	81	90	98	107	117	128
17	13	15	16	18	20	23	26	30	34	39	44	50	55	62	69	76	84	93	101	111	121	132
20	16	17	19	21	23	27	30	34	38	43	48	54	60	67	74	81	89	98	107	116	127	138
22	18	19	21	23	25	29	32	36	41	46	51	57	63	70	76	84	92	101				
25	20	22	24	26	28	32	35	40	45	50	55	61	67	74	81	89	97	106				
27	22	23	25	28	30	34	38	43	47	53	58	64	70	77	84	92	101	110				
30	24	26	28	31	33	37	41	46	51	57	62	68	75	82	87	97	106	115				

Mindestbremshundertstel:

Fig. 3. Brake table for distance 1000m and configuration "R/P" (from: [13])

We illustrate the UML model with the (fictitious) route in Fig. 4. Fig. 5 shows an object diagram of the route which is an instance of Fig 2.

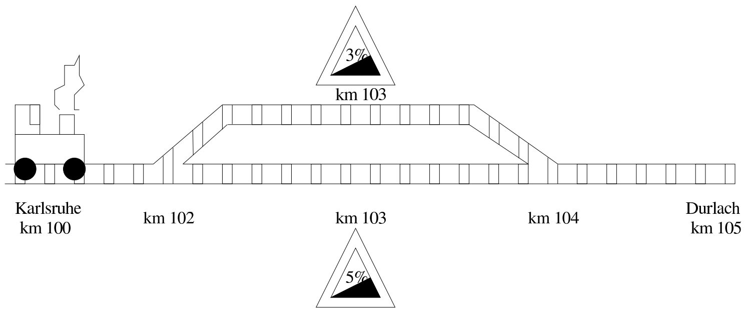


Fig. 4. Route from Karlsruhe to Durlach (fictitious route)

The computation of the speed entry[11] in a row of the schedule can be divided into two parts: first, the maximal permitted train speed ($v_{\max}$) for a given track point is computed, disregarding brake power and slope. This computation involves train parameters/attributes such as the train's top speed, the car covering, and speed properties of the infrastructure. The minimum of these speeds is computed obeying a prioritisation of infrastructure speed limits, in case there are several active speed limits. In a second step, the slope

[10] This account is somewhat simplified for this presentation. See [6] for a full discussion.
[11] Again, we omit some complications for purposes of presentation.

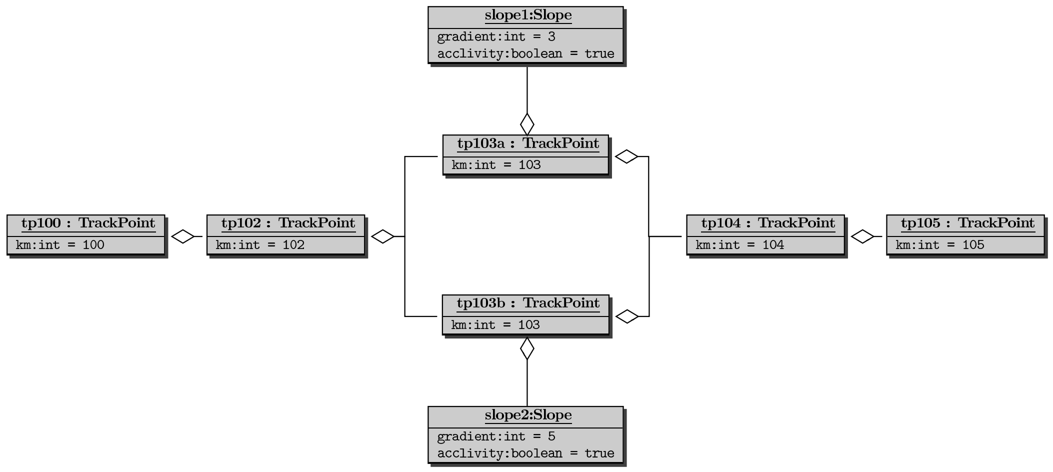


Fig. 5. Object diagram for route in Fig. 4

and available brake distance are taken into account. Further computation is broken down into several steps:

(i) compute the guaranteed brake power of the train

(ii) identify the applicable brake table by the key attributes `brakeConfiguration` and `brakeDistance`

(iii) obtain the brake table entry for the given declivity and speed v (declivity: $v = v_{\text{max}}$, acclivity: v is default roll back speed). If the given slope does not occur in the brake table, one has to linearly interpolate between the two nearest rows.

(iv) acclivity case: the value of the determined brake table entry is taken for the minimal required brake power

(v) declivity case: if the value of the determined brake table entry exceeds the guaranteed brake power as computed in step (i), one has to reduce v_{max} until the resulting brake table entry is less than or equal to the guaranteed brake power. The value of this brake table entry is then taken for the minimal required brake power and the resulting speed as the maximal speed in the current row of the schedule.

4 Formal Specification of Analysis Level

We illustrate our OCL formalization of requirements with the computation of minimal required brake power in the acclivity case (see previous section). The acclivity case is described in the SbF technical description [13] as follows (translated from German by us and slightly simplified):

If acclivity of the relevant slope is zero per mill, then the train requires 0 *Mbr.*[12] Otherwise, the required brake power has to be derived from the

[12] For German “Mindestbremshundertstel”, a measuring unit for brake power.

basic brake tables. Each basic brake table is identified by the pair of brake distance and brake configuration. For the brake distance of the current track point there must be a basic brake table (for steep slopes take 400m table). For trains with brake configuration "G", use the accordingly labeled brake table. For other brake configurations use brake tables labelled "R/P". In brake tables determined this way, the relevant *Mbr* value is the entry for the given acclivity and the default roll back speed. If the given acclivity lies between two acclivities of the basic brake table, then the value has to be interpolated between the smaller and the greater acclivity. Non-integer *Mbr* values must be rounded up to the safe side.

Note that, although quite precise, this natural language specification leaves several things unspecified. Partly, these are terms with an "obvious" meaning, such as "rounded up to the safe side" where, clearly, the next *larger* integer number is intended. Less obvious is the requested kind of interpolation. The implementation realizes linear interpolation, which is also what we take.

The specification above is implemented by method `requiredBrakePower()` in class `ScheduleProcessor`. Before the pre- and postconditions of this method are presented, we give some convenience definitions of operators in OCL. These are attached to class `ScheduleProcessor`, but this is a bit misleading, because the operators are fairly generally applicable, and should be available as part of an OCL library. We come back to this issue.

— OCL-Helpers —

```
context ScheduleProcessor def:

 let ceiling ( r : Real ) : Integer =
   if ( r.floor () < r ) then
     r.floor () + 1
   else
     r.floor ()
   endif

 let maximum ( entries : Collection(Entry) ) : Entry =
 -- return an element of 'entries' with maximal value
 -- of attribute 'rowValue'
   entries -> select ( max : Entry |
     entries -> forAll ( e : Entry |
       e.rowValue <= max.rowValue ) -> any ()

 let minimum ( entries : Collection(Entry) ) : Entry =
 -- analogous to 'maximum'
```

The following definitions are domain specific:

— OCL-Definitions —

```
context ScheduleProcessor def:

 let rollSpeed : Integer = 20

 let getBrakeTable ( dist : Integer,
                     conf : String ) : BasicBrakeTable =
   self.brakeTables ->
     select ( tbl:BasicBrakeTable |
       tbl.brakeConfiguration = conf and
       tbl.brakeDistance      = dist ) -> any ()

 let interpolate ( x1 : Real, y1 : Real,
                   x2 : Real, y2 : Real,
                   intermediate :  Real ) : Real =
   ( ( y2-y1 ) / ( x2-x1 ) ) * ( intermediate-x1 ) + y1

 let interpolate ( e1 : Entry, e2 : Entry,
                   intermediateSlope : Real ) : Real =
   interpolate ( e1.rowValue, e1.content,
                 e2.rowValue, e2.content,
                 intermediateSlope )
```

The first operation `getBrakeTable ( Integer, String )` selects the applicable brake table. The non-deterministic choice '`any ()`' is only apparently so, because the arguments are a key for the brake tables. For linear interpolation of brake table entries the next two definitions are used. On the analysis level interpolation operates on `Real` types, while in the implementation `Integer` with fixed precision is used.

We can now focus on the brake power computation. Method `required-BrakePower` takes three parameters. The first one indicates the mode in which the computation is done, that is, whether tilting technology is assumed. The only difference this makes here is the place where the computation result is stored in the result object that is the third parameter. The second parameter `tp` specifies the track point for which the speed and, hence, the minimal required brake power is computed.

— OCL-Specification: Acclivity Case —

```
context ScheduleProcessor::
  requiredBrakePower ( tilting : Boolean,
                       tp      : TrackPoint,
                       cRes    : Result )

pre  : -- make sure we deal with acclivity case
  ( tp.properties -> select (Slope)->any () ).acclivity
post: -- definitions :
```

```
let brakeDistance : Integer =
  ( tp.properties ->
      select (BrakeDistance) -> any () ).distance in

let brakeTable : Table =
  getBrakeTable ( brakeDistance,
                  tp.train.brakeConfiguration ).table
                  in

let slope : Slope =
  ( tp.properties -> select ( Slope ) -> any () ) in

let nextSmaller : Entry =
  maximum ( brakeTable.entries@pre ->
    select( e | e.rowValue <= slope.gradient and
                e.columnValue = rollSpeed ) ) in

let nextGreater : Entry =
  minimum ( brakeTable.entries@pre ->
    select ( e | e.rowValue >= slope.gradient and
                 e.columnValue = rollSpeed ) ) in

let computedBrakePower : Integer =
  if ( nextSmaller.rowValue = slope.gradient ) then
     nextSmaller.content
  else
     ceiling ( interpolate ( nextSmaller,
                             nextGreater,
                             slope.gradient ) )
  endif in
-- postcondition :
if ( computedBrakePower > guaranteedBrakePower ) then
  self.schedule.isValid = false and
  cRes.technicalError = true
else
  cRes.technicalError@pre = cRes.technicalError and
  if ( tilting ) then
    cRes.activeTilting.brakePower = computedBrakePower
    and
    cRes.activeTilting.speed =
    cRes.activeTilting.speed@pre
  else
    cRes.woTilting.brakePower = computedBrakePower and
    cRes.woTilting.speed = cRes.woTilting.speed@pre
  endif
endif
```

In the precondition we ensure that the method is called for the acclivity case. Then a series of **let**-expressions collects auxiliary values. The first three of these are self-explaining, so consider the `nextSmaller`-expression: first, all brake table entries for the default roll speed are collected[13] and from them only the ones, whose gradient is less than or equal to the gradient at the current track point. Then, `nextSmaller` is set to the table entry with maximal slope gradient (analogous for `nextGreater`). Here and in the following, expressions of the form `select(Class)` are used as a shorthand for the type selection expression `select( o | o.oclIsTypeOf ( Class ))`.

The result of the method is computed in `computedBrakePower` where, if possible, the exact entry is obtained (**then** case) and otherwise interpolation between the next smaller and next greater entry is performed (**else** case). The final **if**-cascade simply ensures that the result is stored in the correct attribute (tilting or non-tilting). It also ensures that, if the train's brake power is insufficient to handle the roll back speed, then the schedule is marked as invalid. Note that this error handling requirement does not explicitly occur in the specification of the acclivity case (although it occurs in other parts of the specification).

5 Specification Idioms and Patterns

Working with the OCL formalization in this case study shows some serious shortcomings: for a start, some (elements of) constraints are very repetitive. This in turn causes OCL constraints to be difficult to maintain, because it is time-consuming to track changes. The need to express closely related requirements often, seduces to resort to a cut-and-paste technique, which is likely to introduce errors. In addition, the constraints become hard to read. One could argue that OCL should offer a richer set of pre-defined operators (such as, for example, in RSL or Z), but this does not suffice in general: first, there will always be domain-specific stuff (here, for example, interpolation); second, a very large language is unlikely to be made proper use of; third, it is often desirable to have application-specific identifiers for operators instead of abstract mathematical nomenclature.

In Section 4 we alleviated the redundancy problem by starting with some helper definitions, but this doesn't solve all problems either:

- The `maximum(Collection(Entry))` operator, for example, is still application-specific and has to be rewritten for each new context. This moves the redundancy problem only from the constraints to helper definitions.
- Newcomers to OCL are unlikely to write elegant and correct constraints. In fact, formal specification languages are as difficult to master as programming languages, but the learning culture and tool support for them is much less

[13] The `@pre` qualification is necessary, because in the interpolation case, the interpolated entries could be (and in fact are) cached.

developed.

The solution that KeY provides is inspired by the success story of design patterns [7]. Design patterns capture the accumulated knowledge and best practices of designers and developers, they allow the reuse of good solutions for related design problems. They are presented on an abstraction level that is high enough to cover a large class of problems, but at the same time can be instantiated specifically for each application.

KeY extends the pattern mechansim to formal specifications: it offers an extensible library of specification patterns. More precisely, we distinguish between *KeY-Idioms*, which are OCL-specific solutions to self-contained, relatively small specification problems (for example, `maximum()`), and *KeY-Patterns*, which are attached to object-oriented design patterns.

Basically, these specification patterns are normal OCL constraints containing placeholders that have to be instantiated (example follows). We call them *OCL templates*. The instantiation mechanism makes use of the CASE tool's (here: TogetherCC) pattern support:

First the context model element (class or operation) is selected; then activation of the pattern library lists applicable patterns, including KeY-Idioms and -Patterns. After selection, the user adapts the pattern interactively to the model context. Finally, the pattern is instantiated by the system, and the resulting OCL constraints are attached to the classes and operations they belong to.

We illustrate how this mechanism can help to improve the specification process. We demonstrate the usage of a simple idiom first, followed by a more complex example related to relational databases.

```
/**
 * @invariants
 *  BasicBrakeTable.allInstances -> forAll
 *    ( b1, b2 : BasicBrakeTable |
 *     ( b1.brakeDistance = b2.brakeDistance and
 *        b1.brakeConfiguration = b2.brakeConfiguration )
 *       implies b1 = b2 )
 */
public class BasicBrakeTable { ... }
```

Fig. 6. Automatically generated OCL constraint for key property

A typical example for a KeY-idiom is the *key* property of a set of attributes, which occurred in the example of Section 4 alone twice already: the members of class `BasicBrakeTable` are uniquely identified by the pair (`brakeDistance`, `brakeConfiguration`), which is, therefore, a (derived) key attribute of the class. A similar situation occurs for table entries, which are uniquely characterized by (`rowValue`, `columnValue`). Several of the constraints that select certain entries rely on this property to work correctly (witnessed by the use of `any ()`).

The KeY user simply selects the "key attribute" idiom and interactively specifies the set of attributes that form a derived key. The correct OCL constraint is generated and attached to class `BasicBrakeTable` as shown in Fig. 6 by the push of a button.

An important point is that OCL templates can be written in a *generic* way, independently of the number and kind of features in the underlying UML model. For example, the OCL template from which the constraint in Fig. 6 is derived, is written independently of the number and type of key attributes. This is possible if one permits access to the UML meta model in OCL expressions, see [2] for details.

Selection of certain entries in a brake table is central for the computation of minimal required brake power. In addition to the acclivity case shown in the previous section, this must be done as well for declivity and in combination with several other parameters. A moment's reflection shows that this kind of operation can be seen as a relational database query. This motivates the inclusion of specification patterns that characterize the result of frequently needed SQL queries.

For example, in SQL a query for the next greater entry would be written like this:

```
select * from tbl
  where
    columnValue = rollSpeed and
    rowValue    =
      ( select min ( rowValue ) from tbl
          where
            rowValue   >= gradient and
            columnValue = rollSpeed );
```

In KeY we provide a pattern that models SQL queries as shown in Fig. 7 (underlined identifiers represent the parameters of the pattern). All predefined SQL queries such as `selectAll(), min`, etc., come already formally specified via OCL templates as shown in Fig. 8.

Instantiating such templates allows one to formulate OCL specifications of custom SQL expressions ("`mySelect`") in SQL style, which is familiar to many users. It is merely necessary to specify the mapping from OCL template parameters to the concrete model as done in Fig. 9.

The only non-trivial expression is the `where` argument of custom query `mySelect`. Its value in Fig. 9, and that of the **where** clause in the SQL expression above are identical up to some syntactic sugar. The formal specification of `selectAll, select_rowValue, select_columnValue, min, max, all<=` is provided together with the pattern and needs not be supplied by the user.

In summary, from the mapping in Fig. 9 and the OCL templates in Fig. 9 KeY creates automatically the following OCL constraints:

— Instantiated OCL Pattern —

```
context Table def:
 let selectAll
   ( coll : Collection ( Entry ),
     expr : OCLExpression ) : Collection ( Entry ) =
   coll -> select ( expr )

 let select_rowValue
   ( coll : Collection ( Entry ) ) : Bag ( Integer ) =
   coll -> collect ( rowValue )

 ...

 let min ( coll : Collection ( Integer ) ) : Integer =
   coll ->
     iterate ( it  : Integer;
               res : Integer = coll -> any () |
                       if ( res > it ) then it
                                       else res endif )
 ...

 let nextGreater : Collection ( Entry ) =
   selectAll ( entries@pre,
     e | e.columnValue = rollSpeed and
         e.rowValue = min ( select_rowValue
         ( selectAll ( entries@pre,
             e | e.rowValue   >= gradient and
                 e.columnValue = rollSpeed ) ) ) )
```

Using these automatically created constraints, the formal specification of method `requiredBrakePower` becomes considerably simpler and more regular:

— OCL Specification with SQL Query Pattern —

```
context ScheduleProcessor :: requiredBrakePower
  ( tilting : Boolean,
    tp      : TrackPoint,
    cRes    : Result )

pre  :
  ( tp.properties -> select ( Slope ) -> any () ).
  acclivity
post :
 let brakeDistance: Integer =  ... as before ...

 ...

 let nextSmaller : Integer =
   brakeTable.nextSmaller -> any () in

 let nextGreater : Integer =
   brakeTable.nextGreater -> any () in

 let computedBrakePower : Integer  ... as before ...
```

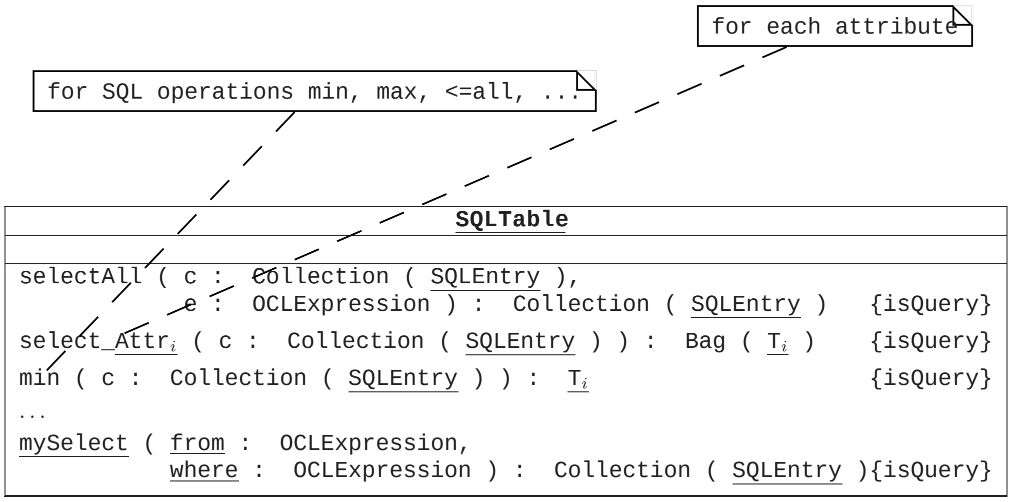

for each attribute
for SQL operations min, max, <=all, ...
SQLTable
selectAll (c : Collection (SQLEntry),
e : OCLExpression) : Collection (SQLEntry) {isQuery}
select_Attr$_i$ (c : Collection (SQLEntry)) : Bag (T$_i$) {isQuery}
min (c : Collection (SQLEntry)) : T$_i$ {isQuery}
...
mySelect (from : OCLExpression,
where : OCLExpression) : Collection (SQLEntry){isQuery}

Fig. 7. SQL Pattern

It is possible to extend this example with more complex SQL expressions. We really appreciated the advantages of the specification pattern technique in the case of determining next nearest entries (and closely related expressions), which was necessary eight times on the implementation level specification. In addition, users who are familiar with SQL need not know much about OCL at all.

Other specification patterns are available in KeY (for example, relating to the GoF [7] design patterns) [3].

In the present case study, which we consider as representative, judicious use of these patterns can reduce the amount (number of lines) of the specification that must be hand-coded by 25%. As a side effect, this reduced the potential of introducing errors considerably.

6 Formal Specification of Implementation Level

On the implementation level the system is split into several modules (in Java: packages). The modules consist of 10 to 30 classes. Altogether the specified system consists of around 80 classes with an average of 10 methods per class. At least 75 % of the methods have been formally specified. The number of

```
context SQLTable def:

 let selectAll
   ( coll : Collection ( SQLEntry ),
     expr : OCLExpression ) : Collection ( SQLEntry ) =
   coll -> select ( expr )

 -- created for all attributes of SQLEntry with type T_i
 let select_Attr_i
   ( coll : Collection ( SQLEntry  ) ) : Bag ( T_i ) =
   coll -> collect ( Attr_i )

 -- SQL operations like min, max, avg, >=all, etc.
 -- created for all attributes of SQLEntry with type T_i
 -- for which '<', '>' ... are defined
 let min ( coll : Collection ( T_i  ) : T_i =
   coll ->
     iterate ( it  : T_i ;
                res : T_i = coll -> any () |
                       if ( res > it ) then it
                                       else res endif )
 ...
 let all<=  ( below : T,
              coll  : Collection( T ) ) : Boolean =
   coll -> forAll ( e : T | below <= e )

 let mySelect : Collection ( SQLEntry ) =
   selectAll ( from, where )
```

Fig. 8. SQL query pattern

lines needed to specify a method vary between a few lines (for example, get/set methods or simple queries) and over 80 lines for methods performing complex computations with a number of side effects.

To get a rough impression of the complete system, the most important modules are described below:

TrainDataAnalysis initialises schedule computation. The route is partitioned into sections of maximal length, where the train object is invariant.

TableComputation controls computation of the schedule table. Also serves as interface to infrastructure and train data. Actual computation of speed entries in schedule is done in:

SpeedComputation contains speed computation logic and classes representing brake tables.

InfrastructureView models track infrastructure at a suitable abstraction level.

$$\text{map}_{\text{general}} = \begin{cases} \underline{\texttt{SQLTable}} \rightsquigarrow \texttt{Table} \\ \underline{\texttt{SQLEntry}} \rightsquigarrow \texttt{Entry} \end{cases}$$

$$\text{map}_{\text{specific}} = \begin{cases} \underline{\texttt{T}} & \rightsquigarrow \texttt{Integer} \\ \underline{\texttt{mySelect}} & \rightsquigarrow \texttt{nextGreater} \\ \underline{\texttt{from}} & \rightsquigarrow \texttt{entries@pre} \\ \underline{\texttt{where}} & \rightsquigarrow \begin{array}{l} \texttt{e} \mid \texttt{e.columnValue} \; = \; \texttt{rollSpeed and} \\ \quad \texttt{e.rowValue} \; = \; \texttt{min} \; (\; \texttt{select\_rowValue} \; (\\ \qquad \texttt{selectAll} \; (\; \texttt{entries@pre}, \\ \qquad\quad \texttt{e} \mid \texttt{e.rowValue} \; >= \; \texttt{gradient and} \\ \qquad\qquad \texttt{e.columnValue} \; = \; \texttt{rollSpeed} \;) \;) \;) \end{array} \end{cases}$$

. . .

Fig. 9. SQL Pattern Instantiation Mapping

ScheduleInterface models the resulting schedule.

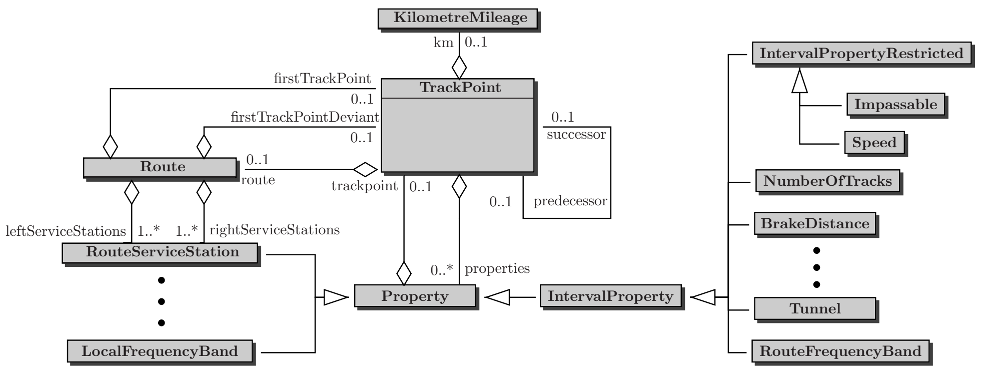


Fig. 10. Implementation Level: Partial View of Package **InfrastructureView**

The specification of requirements on the implementational level differs from the analysis level in several important aspects: for example, one must take into account caching of previously computed results (interpolated brake table entries), and also the refined modelling of train and infrastructure properties with accordingly more complex navigation expressions.

To illustrate the difference between the analysis and implementation level we focus on the infrastructure properties for a given track point. Fig. 10 shows part of the class diagram of the **InfrastructureView** package. In total the package consists of over 30 classes.

On the analysis level, an infrastructure property holds at a track point if and only if the property occurs in the value of the `properties` attribute of this track point. At the implementation level this is not true any longer, because one distinguishes between two different kinds of infrastructure properties:

(i) *Local properties* are like properties on the analysis level: they hold at a track point if and only if the property occurs in the value of the `properties` attribute of this track point. A typical example of a local property is a member of the class `RouteServiceStation`, which models service position along the route such as train stations.

(ii) *Interval properties* hold for a flexible sequence of track points. An interval property inherits from class `IntervalProperty`. Properties that apply only to certain kinds of trains inherit from its subclass `Interval-PropertyRestricted`. A typical interval property is a tunnel (member of `Tunnel`).

An interval property starts to hold at a track point containing this property with attribute `isEnd` set to `false`. It holds for this track point and all successor track points until the occurrence of a track point containing *the same* interval property whose `isEnd` attribute has value `true`. This marks the first track point at which the property does not hold any longer. While an interval property holds, it can be overwritten anytime by interval property objects of the same subclass with possibly updated information. Different interval properties may overlap arbitrarily.

Fig. 11 shows a chain of track points: a tunnel starts at kilometre 22.5 and ends just before kilometre 24.0. The route radio frequency is 200Khz from kilometre 22.6, and this value is kept until kilometre 25.1, where it is overwritten and set to 220Khz. The local radio frequency is a different, local property. It does not override the route radio frequency and it only holds only at kilometre 23.5.

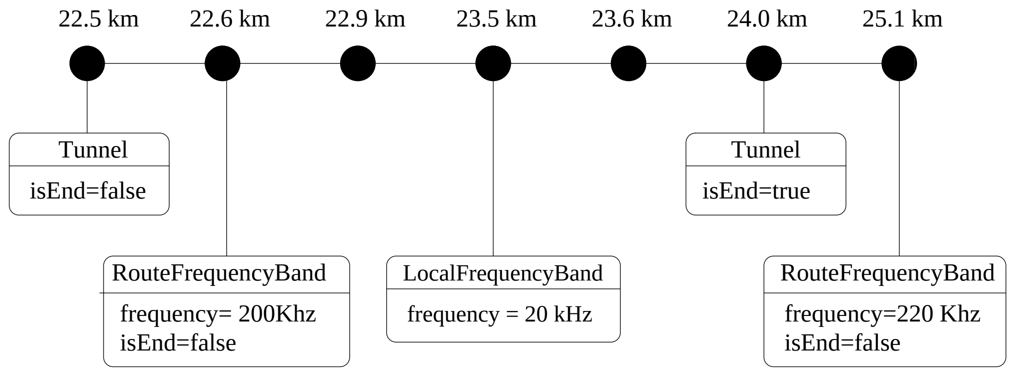


Fig. 11. Interval Properties

To keep track of interval properties, that hold at a given point, there is a status object of type `CreationStatus`, which contains a set with the currently holding properties. It has a method `newProperty` that inserts a new interval property into that set. It must first remove all properties in the same category.

Only in case the new property is not marked as an end point (attribute `isEnd` is `false`) it is added. The formal specification of `newProperty` is as follows:

— OCL: Tracking Interval Properties —

```
context CreationStatus def:

 let updateResult (
   oldPropertySeq : Sequence( IntervalProperty ),
   newProperty    : IntervalProperty )
    : Sequence ( IntervalProperty ) =

   if ( not newProperty.isEnd ) then
     oldPropertySeq -> reject ( a : IntervalProperty |
       newProperty.equal_category ( a ) ) ->
         including( newProperty )
   else
     oldPropertySeq -> reject ( a : IntervalProperty |
       newProperty.equal_category ( a ) )
   endif

context CreationStatus::newProperty (
   newProperty:IntervalProperty )

pre  : newProperty <> null
post : self.intervalProperties = self.updateResult
         ( self.intervalProperties@pre -> asSequence (),
           newProperty )
```

7 Discussion

Although the work on this case study is not yet completed, we are able to draw a number of conclusions from our experience so far.

First of all, we note the usual finding of most industrial projects using formal methods: the requirements specifications in natural language contain a considerable number of ambiguities, even though the SbF software is very well designed and carefully implemented. We did so far not encounter inconsistencies or major errors in the specification, but nevertheless we found some interesting ambiguities: at some point, the rules for choosing the kind of speed limit with the highest precedence are laid down (recall that there can be several speed limits in place at a track point). These rules allow two different interpretations of which one is potentially fatal. As it turns out, the correct interpretation is implemented, but it would be better not to leave this open. Some of the other ambiguities are mentioned in Section 4.

More surprising is perhaps that our formal analysis exhibited an inefficiency in the coding that led to a suggestion for considerable performance improvement of the product: infrastructure speeds are ordered wrt a given prioritization and the one with highest priority is taken for further computation. Specifying the method to return the speed of highest priority from a given list of speeds revealed an inefficient way of determining the maximum: first the list is sorted and then its head is returned. The architecture of Smalltalk's collection framework seduces the developer to such solutions.

Let us now turn to some problems. To begin with, there are shortcomings of OCL as a specification language: some important concepts, such as built-in support for tuples or transitive relations are missing. What is perhaps surprising is that OCL is ignorant of certain object-oriented concepts: there is no notion of constraint inheritance or visibility. Both is very annoying in the presence of overloaded or overwritten methods. Some issues will be addressed in OCL 2.0. Notwithstanding these problems, OCL worked reasonably well: important object-oriented concepts, such as navigation and object types are integrated well. We were able to work around the missing features without too much overhead.[14] We see currently no serious contender for OCL. For example, in JML [9] even our simpler requirements would be very cumbersome to express, because set theoretic and collection-related operators are missing.

All major software projects go through various levels of refinement. The tendency of modern development processes to have incremental, short cycles strengthens this even. Unfortunately, neither the UML, nor the OCL, nor the Java community has developed any suggestions for notions of refinement. Relatively few formal approaches include refinement aspects at all. We are not aware of any suitable formal refinement approach to object-oriented software development (see [11] for a literature overview). An object-oriented theory of refinement tailored to UML and Java is currently developed within the KeY project [11]. We plan to graft our case study onto this refinement framework once it is implemented. An important advantage of a formal refinement relation is that part of the target specification are automatically generated from the source specification, thus reducing effort and eliminating errors.

A major challenge in large formal specification projects is the phenomenon that very similar requirements occur in many places. This is, of course, aggravated by the current shortcomings of OCL discussed above, but it is not specific to OCL. The problem is to keep specifications consistent and readable, and to avoid the cut-and-paste bad practice. It turns out that the usual abbreviation mechanisms (such as the **let** construct) of specification languages don't suffice. Since we are working in an object-oriented setting, we are able to make use of a mechanism akin to design patterns. Our *specification patterns* discussed in Section 5 turned out to be a powerful tool when authoring specifications. It is also an important pedagogical help for novices who learn about good solutions to specification problems, and need not start with a blank page. In a larger setting, specification patterns lead to a natural division of labour

[14] We don't discuss these solutions here, because we find them less interesting and they are, hopefully, of a temporary nature. Full details are in [6].

methods specialists write domain- and application specific patterns, while the others only use them.

8 Conclusion, Future Work

In summary, we think our work shows that industrial object-oriented software of non-trivial size can be formally specified. We think that the resulting OCL specification is reasonably easy to understand and to maintain. We expect that a consequent usage of specification patterns will diminish the effort for similar projects in the future considerably. The design of an extensive specification pattern library is ongoing work.

Some future work has already been mentioned: an interesting task will be the formal verification of the Java reference implementation against the implementation level specification. Once a refinement concept is implemented we intend to connect analysis and implementation level formally.

Although OCL has been designed to be more easily readable than most traditional formal specification languages, it is still much too "formal", for example, with managers or customers. In KeY we explore the use of technology from computational linguistics to provide a systematic link between OCL and less formal descriptions: the Grammatical Framework [10] links formal and informal languages with a common abstract grammar. We have a prototypic instance of this framework that allows to render OCL automatically in English [8]. We plan to use the present case study to improve the legibility of this rendering by domain-specific rules, which can be extracted from the underlying UML model.

References

[1] Ahrendt, W., T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager and P. H. Schmitt, *The KeY tool*, Technical Report 2003-05, Department of Computing Science, Chalmers University of Technology and Göteborg University (2003).

[2] Baar, T. and R. Hähnle, *An integrated metamodel for OCL types*, in: R. France, B. Rumpe and J. Whittle, editors, *Proc. OOPSLA 2000 Workshop Refactoring the UML: In Search of the Core, Minneapolis/MI, USA*, 2000.

[3] Baar, T., R. Hähnle, T. Sattler and P. H. Schmitt, *Entwurfsmustergesteuerte Erzeugung von OCL-Constraints*, in: K. Mehlhorn and G. Snelting, editors, *Softwaretechnik-Trends*, Informatik Aktuell (2000), pp. 389–404.

[4] Beckert, B., *A dynamic logic for the formal verification of Java Card programs*, in: I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, LNCS **2041** (2001), pp. 6–24.

[5] Beckert, B., U. Keller and P. H. Schmitt, *Translating the Object Constraint Language into first-order predicate logic*, in: *Proceedings, VERIFY, Workshop at Federated Logic Conferences (FLoC), Copenhagen, Denmark*, 2002, `http://i12www.ira.uka.de/~key/doc/2002/BeckertKellerSchmitt02.ps.gz`.

[6] Bubel, R., "Formale Spezifikation und Verifikation sicherheitskritischer Software mit dem KeY-System," Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe (2002), (English, with German abstract).

[7] Gamma, E., R. Helm, R. Johnson and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software," Addison-Wesley, Reading/MA, 1995.

[8] Hähnle, R., K. Johannisson and A. Ranta, *An authoring tool for informal and formal requirements specifications*, in: R.-D. Kutsche and H. Weber, editors, *Fundamental Approaches to Software Engineering (FASE), Part of Joint European Conferences on Theory and Practice of Software, ETAPS, Grenoble*, LNCS **2306** (2002), pp. 233–248.

[9] Leavens, G. T., E. Poll, C. Clifton, Y. Cheon and C. Ruby, "JML Reference Manual," (2002).
URL `ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmlrefman.pdf`

[10] Ranta, A., *Grammatical framework: A type-theoretical grammar formalism*, Journal of Functional Programming (to appear, 2003).
URL `http://www.cs.chalmers.se/~aarne/articles/gf-jfp.ps.gz`

[11] Roth, A., "Deduktiver Softwareentwurf am Beispiel des Java Collections Frameworks," Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe (2002),
`http://i12www.ira.uka.de/~aroth/publications/diplomarbeit.pdf`.

[12] Sun Microsystems, Inc., Palo Alto/CA, "Java Card 2.0 Language Subset and Virtual Machine Specification," (1997),
`ftp://ftp.javasoft.com/docs/javacard/JC20-Language.pdf`.

[13] Transport-, Informatik- und Logistik-Consulting GmbH, "DELTA — Gemeinsame Fahrplandatenhaltung, Produktbeschreibung Redesign SbF (SbF-R)," (2001).