Tool Support for Design Inspection: A Specification Notation

Tim Heyer

Department of Computer and Information Science, Linköping University,
S-581 83 Linköping, Email: Tim.Heyer@ida.liu.se

Contents

1 Introduction

The project "Tool Support for Design Inspection" was commenced in April 1999. The project is supported by the Swedish National Board for Industrial and Technical Development (NUTEK). The project is carried out in cooperation between Ericsson Softlab AB and the Department of Computer and Information Science at Linköpings Universitet. The objective of the project is to develop design principles and tools to support the systematic inspection of software designs expressed in the Unified Modeling Language (UML). The primary goals are:

1. To identify *verifiable properties* of software developed within the telecommunications industry.
2. To develop and to define a *design notation* based on UML 1.3 [1]. The design notation has to be formally definable.
3. To develop and to define a *specification notation* based on UML. The specification notation has to be formally definable with the exception of natural language predicate definitions as introduced in [2].
4. To develop *methods to automatically generate a set of questions* that is appropriate to ensure the specified properties of the software system. The questions are to be answered during design inspection.

We are aiming at providing practical means for the inspection of software designs. We assume that software is developed in a way similar to that given by the Unified Software Development Process (USDP) that is described e.g. in [6].

In other reports [4, 5] we presented the design notation (item 2 above) and the question generation (item 4 above). This report is concerned only with item 3 above and to understand it knowledge of the other reports is necessary, i.e. this report is not self-contained.

The specification notation that is presented in the report at hand, is based on the UML, the Object Constraint Language (OCL), and the COMPASS assertion language (see [3]). We start with a brief discussion of the relevant parts of the USDP in Sect. 1.1 and the UML in Sect. 1.2. Then in Sect. 2 and 3 the specification and assertion notation is presented. Section 4 gives an overview of the relationships between the specification and the design. Finally, we summarize in Sect. 5. An example of a specification can be found in [5].

1.1 The Unified Software Development Process (USDP)

The USDP is an *use-case driven*, *architecture-centered*, *iterative*, and *incremental* software development process. It describes *how* to develop software. A detailed description can be found e.g. in [6].

The process results in a set of models which describe the software from different angles and at different stages of development. Only a subset of these models are relevant for our approach. The relevant models are:

Use-Case Model: The use case model describes the requirements as seen from the outside of the system. It is used by customers and developers.

Analysis Model: The analysis model describes the requirements more precisely as seen from the inside of the system. It is used by developers to increase their understanding of the requirements. The analysis model is often incomplete and may sometimes be omitted completely.

Design Model: The design model describes the details of the realization of the system. It is used by programmers and forms the basis of the implementation.

Each model may make use of different kinds of diagrams. These diagrams are expressed in the UML. Our choice for the specification notation is to adopt the notation of the use-case model and to back it up with a set of so called *assertions* which are attached to certain parts of the use-case model and the design model. These assertions are logical formulae which express conditions on the system state. These conditions are supposed to hold at certain points during the lifetime of the system. The assertion notation is based on the OCL, that is part of the UML, and the COMPASS assertion language. It is possible to adopt the notation of the analysis model as specification notation, but as the analysis model is often incomplete or omitted, it is not considered an appropriate choice. Moreover, the analysis model is (if it exists) quite similar to the design model and may in fact be seen as a special form of design model and it may thus be verified in the same manner.

1.2 The Unified Modeling Language (UML)

A detailed description of the UML can be found e.g. in [1]. The diagram of the UML that is used in the USDP for the use-case model is:

Use-Case Diagram: A single use-case describes one specific use of the system. A use-case diagram describes the relationships between a set of use-cases performed by the system and the systems environment.

In addition to the use-case model the specification comprises other elements which are integrated into the use-case and design model. These elements are the different assertions, i.e. intermediate assertions, class invariants, use-case pre- and postconditions, loop invariants, and operation pre- and postconditions.

As mentioned this assertion notation is partly based on the OCL that is part of the UML. The OCL forms essentially a first order predicate logic. In the UML it is used to specify constraints which can be attached to UML models. A detailed description of the OCL is omitted here, it can be found e.g. in [1].

2 The Specification

In the USDP the specification consists of use-case diagrams with attached documentation. As in the USDP use-case diagrams are part of the specification in our approach.

We support the following elements of a use-case diagram: *use-cases*, *actors*, and relationships between actors/use-cases and use-cases/use-cases (how these elements are represented in an use-case diagram is shown in Fig. 1):

Actor: An actor represents a role of an entity that is outside the system and that interacts with the system. Actors can be persons, things, or other systems. An actor has a name and a type. The details of the role that an actor represents are described in the attached documentation.

Use-Case: A use-case represents one specific use of the system. Each use-case has a name. The details are described in the attached documentation.

Relationship: The only relationship that can occur in our use-case diagrams is a *communicates relationship*. It means that the two elements are communicating with each other.

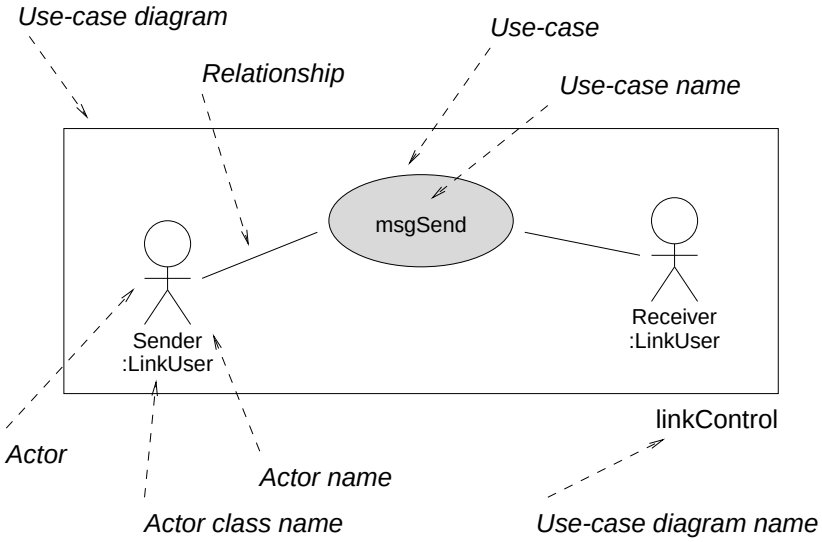

Figure 1: Representation of a actors, use-cases, and their relationships

In the USDP the attached documentation consists of a set of, in natural language written, descriptions of when it is safe to apply a certain use-case of the system, what the outcome of the use-case is, and roughly how the system establishes that outcome. Instead of this completely informal description, in our approach each use-case is described in a semi-formal way by using use-case pre- and postconditions (the notation of the conditions is described in Sect. 3):

Use-case pre- and postcondition. A use-case precondition specifies a condition on the state that is supposed to be satisfied prior to the execution of the use-case. The

precondition specifies the states in which the use-case can be invoked. A postcondition on the other hand specifies a condition on the state that has to be satisfied after the execution of the use-case. The postcondition specifies the service that is provided by the use-case (given that the precondition was satisfied when it was invoked).

The pre- and postcondition can refer to related actors, the system in general, and possible input and output to the use-case.

In the approach that we are developing and investigating, the specification consists, in addition to annotated use-cases, of:

Sequence diagram intermediate assertions. Sequence diagrams describe the interaction of objects. The intermediate assertions specify the state changes of an object between two subsequent receptions of messages, dispatches of messages, beginning of loops, ends of loops, and/or termination. The assertions are assigned to the corresponding segement of the activation of a particular object.

The assertions can refer to the state of the object itself. Moreover, the "input", i.e. the data that is conveyed by messages, can be used (if applicable). If the intermediate assertions appear inside an loop they can also refer to the iterator variable.

Loop invariants. Sequence diagrams describe the interaction of objects. Loop invariants specify a condition that is maintained by an loop, i.e. a condition that has to be satisfied prior to the first iteration and that is reestablished after each iteration over a continuous part of a sequence diagram.[1]

The assertion can refer to the state of all objects which are included in the box. Moreover, the iterator variable that is defined for the loop can be used.

Class invariants. Class diagrams contain descriptions of objects that share the same properties (attributes and operations) and their relationships to other classes. Class invariants specify a condition that is maintained by each instance of the class. That is, it is assumed that the assertion is satisfied prior to each invocation of the object and it has to be satisfied after each invocation of the object. The condition may be temporarily violated during the computations within a single operation.

The class invariant may refer to the state of the instance of the class.

Operation pre- and postconditions. Class diagrams contain descriptions of objects that share the same properties (attributes and operations) and their relationships to other classes. A operation precondition specifies a condition that is supposed to be satisfied prior to the execution of the operation. The precondition specifies the states in which the operation can be invoked. A postcondition specifies a condition

[1]The scope of the loop is represented graphically by a box over all lifelines in the sequence diagram.

that has to be satisfied after the execution of the operation. The postcondition specifies what the operation is doing (given that the precondition was satisfied when it was invoked).

Both assertions can refer to the state of the object and arguments to the operation (if available).

Sequence diagram intermediate assertions, class invariants, operation preconditions, and operations postconditions are expressed in the assertion notation described in Sect. 3. The visibility rules are reflected by the nesting of context environments as presented later in in Sect. 3.

3 The Assertion Notation

As mentioned previously, the assertion notation is based on a subset of the OCL. The OCL has been chosen as basis because it is part of the UML which is the basis for the design model and the use-case model. An additional corner stone of the assertion notation is the assertion language described in [3]. The assertion language provided in the licentiate thesis comprises a notation for operation pre- and postcondition and class invariants. Moreover it introduces the notion of semi-formal assertions that we apply in the current work too.

In the following, keywords that appear in the assertion notation itself are written in typewriter, words that are place-holders for keywords of the assertion notation are written in typewriter enclosed by "<" and ">", and words that represent phrases to be invented by the user are written in typewriter enclosed by single quotes "' '".

In all parts of the written annotations of the specification it is possible to insert comments. Comments are preceded by two dashes, e.g.:

```
-- 'This is a comment.'
```

The two dashes have to appear in front of each line that is supposed to be part of the comment (i.e. there is no kind of *start* and *end* marking of comments).

3.1 The Context of an Assertion

Before writing an assertion the context in which that formula belongs and applies has to be defined. This can be done in two ways:

1. The assertions may be directly attached to a diagram element. This requires a graphical tool that allows to do this. The context of an assertion can be determined from the diagram itself. This is the normal, preferred way and a prototype tool to enter the various diagrams and assertions and to generate and present the questions is currently developed.

2. The assertions may be provided in a separate document. The context then has to be explicitly defined. This approach is useful to present the specification in a sequential form as e.g. in this report.

For the first case no explicit definition of the context of an assertion is required. For the second case, the context of an assertion is explicitly defined in the following way:

```
context <element-type> <element-name>
   'assertion descriptions etc.'
end context <element-type> <element-name>;
```

The `<element-type>` specifies the type of the element for which the body applies. Possible element types are: use-case diagram, sequence diagram, class, diagram, use-case, loop, object, and class. The `<element-name>` is the name of the element as it appears in a diagram. Context environments are usually nested. The nesting has to follow a certain pattern that is illustrated in Fig. 2.

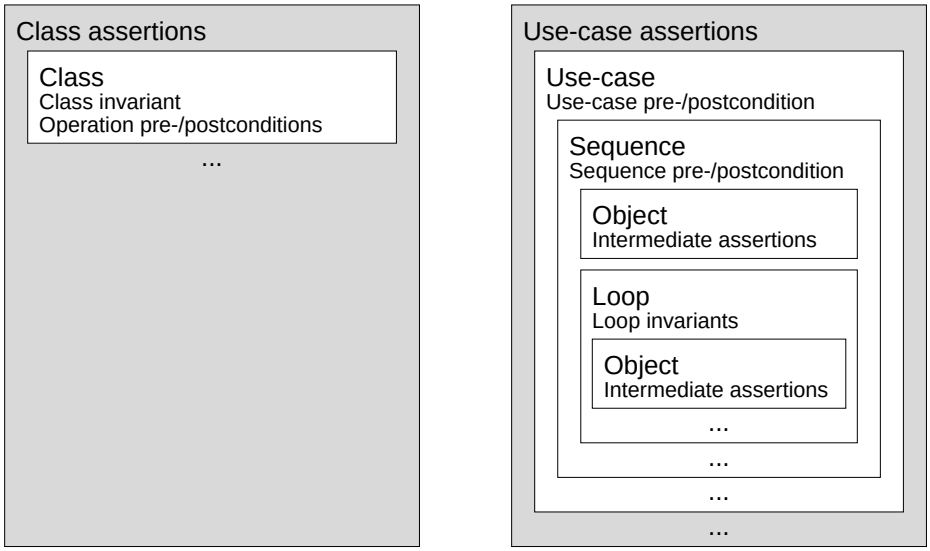

Figure 2: The scope of specification elements

If we e.g. want to specify the use-case precondition of a use-case `msgSend` in the use-case diagram `linkControl` we would describe the context in the following way:

```
context Use-Case-Diagram linkControl
   context Use-Case msgSend
      'definition of precondition etc.'
   end context Use-Case msgSend;
end context Use-Case-Diagram linkControl;
```

3.2 Specifying the Type of Assertion

The different types of assertions are use-case pre- and postcondition, sequence intermediate assertions, loop invariants, class invariants, and operation pre- and postconditions. All assertions are defined in a corresponding context environment.

3.2.1 Use-Case Pre- and Postcondition

An use-case precondition is denoted by the key word `requires` (as the safe application of the use-case respectively operation *requires* the subsequent formula to hold):

```
requires
   'specification of use-case precondition'
end requires;
```

Similarly, an use-case postcondition is denoted by the key word `ensures` (as the application of the use-case *ensures* that the subsequent formula holds if the precondition held previously):

```
ensures
   'specification of use-case postcondition'
end ensures;
```

3.2.2 Operation Pre- and Postcondition

An operation precondition is also denoted by the key word `requires` but in addition it is preceded by the name of the operation (since there may be more than one operation in a class):

```
<operation-name> requires
   'specification of operation precondition'
end requires;
```

Similarly, an operation postcondition is denoted by the key word `ensures` preceded by the name of the operation:

```
<operation-name> ensures
   'specification of use-case postcondition'
end ensures;
```

3.2.3 Class Invariant

A class invariant is denoted by the key word `maintains` (as each operation of an instance of the class can assume that the subsequent formula was satisfied prior to its invocation and as each operation has to ensure that the subsequent formula is satisfied after it finished):

```
maintains
   'specification of class invariant'
end maintains;
```

3.2.4 Loop Invariant

A loop invariant is denoted by the key word `maintains` (as the condition is supposed to be satisfied before and after each iteration of the loop):

```
maintains
   'specification of loop invariant'
end maintains;
```

3.2.5 Intermediate Assertion

The key word for an intermediate assertion is `establishes` (as the computation of the concerned object *establishes* the subsequent formula).

```
<activation-segment> establishes
   'specification of intermediate assertion'
end establishes;
```

An intermediate assertion belongs to a particular activation segment. The specification of the activation segment is of the form `<start>-<end>`. The segment is delimited by receptions of messages, dispatches of messages, beginning of loops, ends of loops, and/or termination. Messages are distinguished by their number, loops by their name and the termination is denoted by the key-word `end`.

3.3 The Formula of an Assertion

In our work we initially assume that the logical formulae are expressed as described in the following sections. The logic we introduce forms essentially a first order predicate logic. It is also possible to specify assertions completely in natural language by writing the key-word `informally` immediately behind the head for any particular assertion, e.g.:

```
requires informally
   'natural language definition of a use-case precondition'
end requires;
```

3.3.1 Types

A set of predefined types are available to the user of the logic. Values of these types may be used anywhere in assertions. The basic types are:

Boolean: This type comprises the two values *true* and *false*.

Integers: This type represents the set of all whole numbers both negative and positive ones, e.g. `2`, `-42`, `29`, and `1967`.

Besides these basic types, all classes that appear in the use-case, sequence, and class diagrams are considered types.

3.3.2 Variables

Basically, all variables that occur in the models can be used in expressions. Besides these variables, some special variables are predefined:

self: As each assertion is written in the context of an instance of a specific type (see Sect. 3.1) it is necessary to be able to refer to that instance. That is done by the special variable `self`.

result: The return value of an operation (in case it has one) is denoted by the variable `result`.

In use-case and operation postconditions it is usually necessary to refer to initial values of variables. The reason is that a postcondition describes the intended behavior of the use-case or operation in terms of changes to the state of the system. To describe the difference of the final state compared to the initial state, it must be possible to refer to that initial state. This is done by appending `@pre` to appropriate variables, e.g. `i > i@pre` specifies that the final value (i.e. the value of the variable *after* the use-case or operation has been executed) of the variable `i` is greater than its initial value (i.e. the value of the variable *immediately before* the use-case or operation is invoked).

3.3.3 Operations

Corresponding to the types presented in the previous section, we define a set of operations that can be applied on these types. The predefined operations are:

`*`, `/`, `+`, and `-`: Multiplication, division, addition, and subtraction can be applied on integers only. The result of the division is truncated (i.e. it is not rounded). The meaning is as in common arithmetic.

`>`, `>=`, `<`, `<=`, `==`, and `!=`: Greater than, greater than or equal to, less than, less than or equal to, equal to, and not equal to can be applied on integers only. The result however, is of type boolean. The meaning is as usual.

`and`: Logical conjunction can be applied on booleans. The meaning is as in common logic, except that any of the two subexpressions may be undefined (e.g. through division by zero). In that case the result of the conjunction is undefined, except when the first subexpression is evaluated to false. False and-ed with anything is always false no matter whether the second subexpression is false, true, or undefined.

`or`: Logical disjunction can be applied on booleans only. The meaning is as in common logic, except that any of the two subexpressions may be undefined. In that case the result of the disjunction is undefined, except when the first subexpression is evaluated to true. True or-ed with anything is always true no matter whether the second subexpression is false, true, or undefined.

`not`: Even logical negation can be applied on booleans only. Again, the meaning is as in common logic, except that the negated expression may be undefined in which case the whole expression is undefined.

`.` **(dot-operation):** The dot-operation is used do refer to attributes and/or operations of objects. The syntax is as follows:

```
<objectName>.<propertyName>
```

The `<objectName>` is simply a string that specifies the name of the object. The `<propertyName>` is a string that denotes the name of an attribute or operation as it is defined in the objects class description.

In addition to the above operations, parentheses (i.e. `(` and `)`) can be used to explicitly specify the order in which the operations in composed expressions apply. If parentheses are omitted the following precedence ordering applies:

`@pre`	Reference to initial value of a variable
`.` (dot-operation)	Reference to object attributes or operations
`not` and unary `-`	Logical negation and negative integers
`*` and `/`	Multiplication and division
`+` and binary `-`	Addition and subtraction
`>`, `>=`, `<`, and `<=`	Greater than, greater than or equal to, less than, and less than or equal to
`<>` and `=`	Not equal to and equal to
`and` and `or`	Logical conjunction and logical disjunction

The `@pre` has the highest priority, i.e. it is evaluated prior to all other operations. The operations `and` respectively `or` have the lowest priority, i.e. they are evaluated after all other operations (with the possible exceptions mentioned in the description of each operation).

3.3.4 Predicates

So far logical formulae can be constructed from values of the basic types (i.e. boolean and integer), variables, and the predefined operations. In addition, it is possible to use *predicates*. A predicate is a logical condition that is given a name. A predicate is written the following way: first comes the `<predicateName>` and then in parentheses `()` follows a (possibly empty) list of comma separated `<arguments>`, i.e. :

```
<predicateName>(<arguments>)
```

The meaning of a predicate may be defined by a logical formula that in turn is constructed from values of the basic types, variables, predefined operations, and predicates, i.e. :

```
define <predicateName>(<arguments>) formally
  'logical formula definition'
end define;
```

Another possibility is, to define the predicate with natural language, i.e. :

```
define <predicateName>(<arguments>) informally
  'natural language definition'
end define;
```

The predicate definitions are given in the context where the predicates are used. It is also possible to use predicates whose definition appear in a higher context.

4 Relationships Between Specification and Design

The USDP establishes relationships between the relevant models and diagrams. Figure 3 shows these relationships including the new parts of the specification as presented in this report.

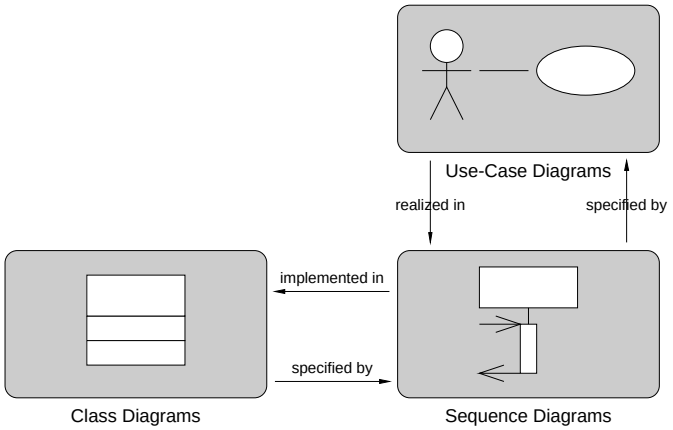

Figure 3: Relationships between specification, design, and involved diagrams

5 Summary

This report introduces a description of a specification notation for designs expressed in the design notation presented in [4] and the question generation presented in [5]. The corner stones of the specification notation are the UML, the OCL, and the assertion language presented in [3].

The specification notation is intended as a basis for continued discussions and evaluations of our approach to design development and inspection. The report at hand will

be revised in the forthcoming reports. Among open questions that have to be further discussed are:

- The form of operation pre- and postcondition does not allow overloading because the parameters are not included.
- Do we need some kind of modifies-clause for operations or use-cases (frame problem)?

References

[1] OMG Unified Modeling Language Specification 1.3, June 1999.

[2] S. Bonnier and T. Heyer. COMPASS: A comprehensible assertion method. In *TAPSOFT '97: Theory and Practice of Software Development*, pages 803–817. Springer-Verlag, 1997.

[3] T. Heyer. *COMPASS: Introduction of Formal Methods in Code Development and Inspection.* Licentiate thesis, Department of Information and Computer Science, Linköping University, Apr. 1998. LiU-Tek-Lic 1998:30.

[4] T. Heyer. Tool support for design inspection: A design notation. Project report, Department of Information and Computer Science, Linköping University, 1999.

[5] T. Heyer. Tool support for design inspection: Automatic generation of questions. Project report, Department of Information and Computer Science, Linköping University, 2000.

[6] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process.* Addison Wesley, 1999.

Revisions

1.1 Small changes in the notation for assertion contexts and assertions types. Addition of loop invariants. (Time-stamp: "2000-07-07 11:30")

1.0 Initial revision