Semantic Inspection of Early UML Designs

Tim Heyer

Abstract— **In this paper we present an approach to tool-based inspection focusing on the functional correctness of early designs expressed in a subset of UML. The approach is based on traditional inspection but extended with elements of formal verification. The idea is to relax the requirements on formal rigor to yield a novel method that makes it easier to express and reason about designs while still allowing the automatic generation of questions that help in finding defects.**

I. Introduction

WE present an approach for automatically generating relevant, focused questions for the inspection of early designs expressed in the Unified Modeling Language (UML). In contrast to *testing* inspection facilitates early detection of defects. Other methods that are applicable at earlier stages are *formal verification* and *inspection*. Formal verification is based on mathematical principles to demonstrate the correctness of a formal artifact (e.g., the design or the implementation) with respect to a formal specification. The required knowledge of the formal notation and its proof system is often perceived a significant barrier against industrial use. Hence formal verification is typically used only for especially critical components. We believe greater acceptance of formal principles may be achieved by relaxing the amount of formality.

Inspection, first introduced by Fagan in 1976 (see [1]), is the process of finding defects in an artifact by human examination. Inspection is reported to be very effective both with respects to defect-detection and cost (a detailed description of software inspection can be found e.g. in [2]). However, generally the focus is on style rather than on functionality. Since functionality changes from artifact to artifact it is difficult to give general guidelines for how to systematically check the artifact for defects with respect to its intended functionality. The effectiveness of inspection is very much depending on the experience and discipline of the personnel involved. Several tools to support the inspection process are available. For instance, Macdonald and Miller (see [3]) compared 16 tools in 1999. However, the focus of these tools is almost entirely on administrative tasks like scheduling meetings and collecting defect reports. What is missing are guidelines that focus on the functional correctness of the particular artifact, i.e. guidelines that precisely state which questions to address to find defects. Ideally, these questions should be generated automatically from a given artifact. We call this *semantic inspection*, since the questions address the semantics of the artifact.

In this paper we present such an approach to inspection of designs expressed in UML (see [4]). Our approach combines today's inspection process with elements of formal verification to yield a systematic design inspection process. The aim is to develop tools to support the systematic inspection of software designs expressed in UML. The focus is on providing *practical* means for the inspection of software designs. In Sect. II we introduce technical details of how the questions to address during the inspection are automatically generated. A simple example to further explain our approach can be found in Sect. III. In Sect. IV we outline a semantic design inspection process based on the automatic generation of questions described ealier. Finally, after an overview of related work in Sect. V, we summarize our work in Sect. VI.

II. Our Approach to Inspection

Our approach to design inspection is based on traditional inspection but extended with elements of formal verification. The design is expressed using *annotated sequence and class diagrams*. The (system) specification consists of *annotated use-case diagrams*. The annotations are also called *assertions*. Assertions in this context are conditions on the state of the system or parts hereof which are supposed to be satisfied at certain reference points during the execution of the system. The conditions may be specified partly in an informal manner (see Sect. II-A). Given a specification and a design (both typically incomplete) it is possible to automatically generate a set of questions which address the critical issues to establish the functional correctness of the software.

Our main goal is to demonstrate a new method to the inspection of software artifacts given e.g. in UML; the goal is not primarily to provide a precise semantic description of UML. The principles for generating questions out of formal notations accompanied with semi-formal annotations are applicable to other types of artifacts and notations. In fact, they have been applied earlier to code (see [5]).

A. A Specification Notation

The *system specification* notation is based on UML and consists of annotated use-case diagrams. A use-case diagram describes the relationships between various uses of the system and its environment. To simplify the initial development and implementation we have chosen a subset of UML. For use-case diagrams we support *use-cases*, *actors*, and *association relationships* between actors and use-cases. The annotations consist of two types of assertions (explained later). Assertions are expressed in a notation similar to quantification-free first order predicate logic. An

T. Heyer is with the Department of Computer and Information Science, Linköping University, S-581 83 Linköping, Sweden. E-mail: timhe@ida.liu.se .

The project is funded by NUTEK and VINNOVA, and it is carried out in cooperation between Ericsson SoftLab AB and the Department of Computer and Information Science at Linköping University. Further information can be found at **http://www.ida.liu.se/~ulfni/nutek**.

assertion is a *well-formed formula* which may contain (well-formed) expressions (or terms). We assume the existence of a first order alphabet consisting of function symbols (denoted by f in our abstract syntax) and predicate symbols (denoted by p) both with an associated arity. Constants are considered function symbols of arity 0, except numerals denoted by n. Variables are denoted by x.

Definition 1 (Well-formed expression)
Numerals and variables are expressions. Complex expressions can be built by combining expressions with predefined operators or user-defined functions. The abstract syntax of expressions is as follows:

$$
\begin{aligned}
E ::= {} & n \\
& \mid x \mid \textbf{\textit{this}} \mid \textbf{\textit{result}} \\
& \mid -E \mid E-E \mid E+E \mid E/E \mid E*E \\
& \mid f(E,\ldots,E)
\end{aligned}
$$

The meaning and precedence of the predefined operations is like in conventional arithmetic. We assume that all well-formed expressions are type-correct.

All variables used in the design can be used in assertions. These variables are called *design variables*. However, often it is necessary to refer to initial values of design variables. For example, if we want to specify what an operation accomplishes we may need to refer to the initial values of its parameters, e.g. to express that a car has half its initial speed after breaking shortly. Therefore, e.g. initial values of variables are captured in logical variables which do not appear in the design. Unlike design variables, logical variables do not change their values. To distinguish logical variables from design variables they have a special appendage "@pre" or "#n" (where n is a natural number denoting a reference point in the sequence diagram, see Sect. II-B). For each design variable a corresponding logical variable with the same name and appendage "@pre" is implicitly defined. These logical variables refer to the initial values of the design variables with respect to the context of the assertion. For example, an assertion that expresses that a car has half its initial speed after the "brake-shortly" operation could be written as $2 * \textbf{speed} = \textbf{speed@pre}$. The variables **this** and **result** are special design variables. The variable **this** generically refers to each single instance of a class, i.e., it is a place-holder for the name of that instance (often when assertions are specified the name of the instance is not known). The variable **result** refers to the return value of an operation in case it has one.

Definition 2 (Well-formed formula)
The boolean constants true *and* false *are formulae. Predefined and user-defined predicates are atomic formulae. Complex formulae can be built by combining formulae with predefined operators:*

$$
\begin{aligned}
F ::= {} & \textbf{\textit{true}} \mid \textbf{\textit{false}} \\
& \mid \neg F \mid F \wedge F \mid F \vee F \mid F \Rightarrow F \\
& \mid E<E \mid E \leq E \mid E>E \mid E \geq E \mid E=E \mid E \neq E \\
& \mid p(E,\ldots,E)
\end{aligned}
$$

The meaning and precedence of the predefined predicates and operations is like in conventional predicate logic except when an actual parameter is undefined. The result of a conjunction is false if the first argument is false and the result of a disjunction is true if the first argument is true. We assume that all well-formed formulae are type-correct.

A user-defined function or predicate may be defined formally (using an expression respectively a formula) or informally (using e.g. natural language). The point is that any means is possible as long as it provides a unique interpretation of the function or predicate to the human reader. For example, a use-case postcondition that expresses that a phone **phoneA** is off-hook and connected to another phone **phoneB** over a network **net** could look like this:

offHook(phoneA) ∧ connected(phoneA, net, phoneB)

The definition of the predicates **offHook** and **connected** are provided together with the postcondition. For example, an informal definition of **connected** could look like this:

define connected(phoneA, net, phoneB)
The phone "phoneA" is connected with the phone "phoneB" over a full-duplex, dedicated line provided by the network "net".
end define

A limited form of quantification may be expressed with the help of natural language predicate definitions. For example, we may use a predicate **hasUser**. The definition of the predicate may be:

define hasUser(cell)
There is at least one active user (e.g., cellular phone) in the network cell "cell".
end define

An assertion using the predicate **hasUser(cell)** would express "∃" (there exists) without explicitly using the quantifier in the assertion. Instead, the quantifier is hidden in the definition of the predicate.

The two types of assertions which are attached to use-cases are *use-case pre-* and *postconditions*:

Use-case pre- and postcondition. The precondition is assumed to be satisfied prior to the execution of the use-case. The precondition specifies the states in which the use-case can be invoked. Hence the preconditions may only refer to the initial state of the objects involved in the use-case (using the "@pre" notation). The postcondition has to be satisfied after the execution of the use-case. It specifies the service provided by the use-case and it may refer to the initial state of the objects involved in the use-case as well as their final state (using the objects name). If U is a use-case then P_U denotes the condition that is satisfied when the use-case starts and $Q_U(r)$ denotes the condition that is satisfied when the use-case ends at reference point r (according to a related sequence diagram – see Sect. II-B and Sect. II-C). The $Q_U(r)$ is basically the postcondition with all design variables replaced with a corresponding logical variable with appendage "#r".

B. A Design Notation

The design notation consists of annotated class and sequence diagrams. Class diagrams describe the static structure of the system. They describe what to implement. Our UML subset comprises *classes*, and *association* and *generalization relationships* between classes. Sequence diagrams (see e.g. Fig. 1), on the other hand, describe dynamic behavior of the system. In sequence diagrams the focus is on

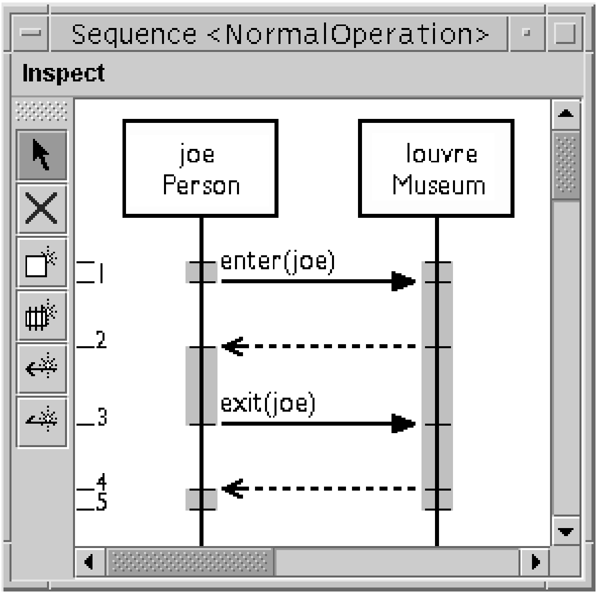
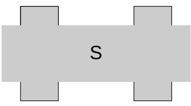


Fig. 1. Sequence diagram **LV** (lawful visit)

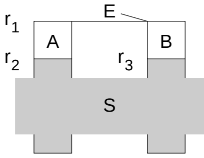
the temporal order of messages exchanged between concrete instances (i.e., objects) of classes. However, messages are only ordered within each thread. Messages of different threads may be sent in parallel or in an arbitrary order. Moreover, the actual time is not proportional to the distance between messages etc. Thus it is not possible to use distance relations in the sequence diagram to draw conclusions about time relations. In our approach each object has a single thread of execution. The period during which an object is active is called an *activation*. An *activation segment* is the period between two subsequent receptions of messages, dispatches of messages, start of execution, and/or end of execution. A *reference point* is a point on the time axis of a sequence where an activation segment starts or ends. A sequence diagram describes a *well-formed sequence* as described in the next three definitions.

Definition 3 (Partial sequence)
Below we will use the following symbol to represent a partial sequence containing an arbitrary (but fixed) number n of objects:

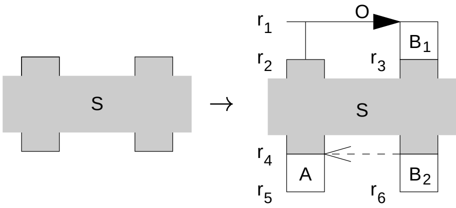


1. An empty sequence is a partial sequence. An empty sequence consisting of n objects is denoted ϵ_n.

2. If S is a partial sequence then so is S' obtained by prepending a signal and two activation segments to S:

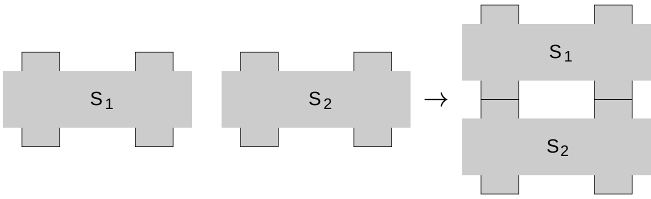


3. If S is a partial sequence then so is S' obtained by prepending an operation invocation and two activation segments, and by appending an operation return and two activation segments to S:

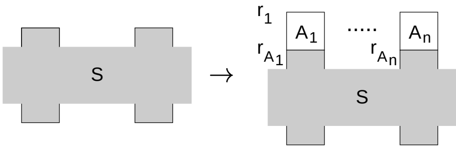


The above figure indicates that both objects get an additional activation segment attached to their top segment in S. However, since operation invocations are synchronous, the dispatching object is blocked after the invocation. This is denoted by drawing the corresponding activation segment as a line instead of a block.

4. If S_1 and S_2 are partial sequences over the same set of objects then so is S' obtained by prepending S_1 to S_2 (written $S_1 \circ S_2$):



Definition 4 (Complete sequence)
If S is a partial sequence then S', obtained by prepending an activation segment to each object, is a complete sequence:



Definition 5 (Well-formed sequence)
A complete sequence is well-formed if and only if every object is active when it dispatches a message.

When writing assertions the current state of a variable is denoted by the variable name whereas the previous state (if applicable) is denoted by the variable name followed by "@pre". For instance, an assertion that is attached to an activation segment A and that expresses that the object phone it belongs to has not changed during the activation segment could look like this:

phone=phone@pre

However, the questions which are eventually presented to the inspector typically consider the reference points to which various conditions apply by appending $\#r$ to variable names (where r is the reference point, e.g. 1, 2, 3, 4, and 5 in Fig. 1). For instance, if the activation segment A mentioned above starts at reference point 2 and ends at reference point 4 then questions may contain $A(2,4)$ rendered as:

phone#4=phone#2

The following types of assertions can be attached to class respectively sequence diagrams:

Sequence pre- and postconditions. A sequence is usually only applicable under certain circumstances. For example, it is common to have sequences describing execution under normal conditions and under exceptional conditions. A sequence precondition is supposed to be satisfied prior to the execution of the sequence. Like a use-case precondition, a sequence preconditions may only refer to the initial state of the objects contained in the sequence (using the "@pre" notation). A postcondition has to be satisfied after the execution of the sequence and it may refer to the initial state of the objects contained in the sequence as well as their final state (using the objects name). If S is a sequence then P_S denotes the condition that is satisfied when the sequence starts and $Q_S(r)$ denotes the condition that is satisfied when the sequence ends at reference point r. For instance, the postcondition of the sequence of Fig. 1 is referred to as $Q_{LV}(5)$. A pair of sequence pre- and postconditions is also called a *scenario specification.*

Sequence intermediate assertions. An intermediate assertion is a condition that specifies the state changes within an activation segment (i.e., the condition is satisfied when the end of the activation segment is reached). An intermediate assertions express what functionality later refinements (e.g. a state machine or code) provides. Whether the e.g. code complies to the assertion has to be verified at a later stage in the development. An intermediate assertion may refer to the object it belongs to and to parameters of messages the object has received from other objects. If A is an activation segment then $A(r_1, r_2)$ is the condition that is satisfied when the activation segment starts at reference point r_1 and ends at reference point r_2. For example, the assertion of the second activation segment of **louvre** in Fig. 1 is referred to as $\mathbf{louvre}_2(1,2)$.

Class invariants. A class invariant is a condition that is maintained by each instance of the class. For example, it is assumed that the condition is satisfied prior to each invocation of the object and that it is reestablished afterwards. If a is an object of the class A then $C_a(r)$ is the invariant condition of the class A at reference point r.

Operation pre- and postconditions. A precondition is supposed to be satisfied prior to the execution of the operation. The precondition specifies the states in which the operation can be invoked. An operation preconditions may only refer to the initial state of the object it belongs to and to the initial state of the parameters to the operation (using the "@pre" notation). A postcondition has to be satisfied after the execution of the operation. The postcondition specifies what the operation is doing and it may thus refer to the initial state of the object it belongs to and to the initial state of the parameters to the operation as well as their final state. If O is an operation then $I_O(r)$ is the condition that has to be satisfied when the operation is invoked at reference point r and $F_O(r_1, r_2)$ is the condition that is satisfied when the operation was invoked at reference point r_1 and returned at reference point r_2.

We require that all class attributes are private (i.e., not visible outside the class definition). To refer to the state of a particular attribute of an object we use user-defined functions instead. For example, an intermediate assertion saying a received record **r** is the same as the record contained in an entity **e** could be expressed by defining a function **record**. The assertion could look like this: **r=record(e)**. The (informal) definition of the function would simple state that the function returns the record contained in the argument (of the entity type) to the function.

C. Generation of Questions

Common to approaches to formal verification is the aim to prove that the artifact is correct with respect to the specification. Usually, the intention is to prove that the final state of the system satisfies a certain postcondition provided that the initial state satisfies a certain precondition.

In our approach the overall behavior of the system is specified by a set of use-cases and corresponding pre- and postconditions. The system is correct if the execution of each use-case in a state that satisfies the precondition ends in a state that satisfies the postcondition. That is, what we would like to verify corresponds to Arrow 1 in Fig. 2. However, to verify a use-case, its effect on the state has

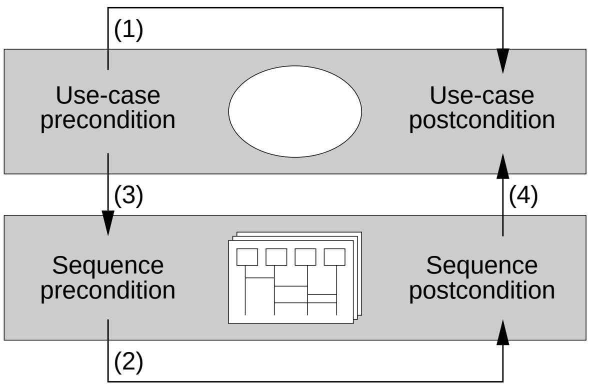


Fig. 2. Verification model

to be completely defined. In our approach each use-case is specified in more detail by a set of sequence diagrams. The sequence diagrams together with the class diagrams constitute our design. Each sequence diagram comes with corresponding pre- and postconditions. A sequence diagram is correct if its execution starting in a state that satisfies the precondition ends in a state that satisfies the postcondition (Arrow 2 in Fig. 2). We can generate questions to verify this since a sufficiently annotated sequence diagram contains enough detail. If we finally verify that each use-case precondition implies the disjunction of the corresponding sequence preconditions (Arrow 3 in Fig. 2) and that each sequence postcondition implies the corresponding use-case postcondition (Arrow 4 in Fig. 2) we have achieved our original goal. To generate questions to verify the last proposition is straightforward. However, the preceding proposition implies that the sequence diagrams belonging to a use-case cover all possible states when the use-case may be executed. We consider it very hard if at all possible, and seldom desirable to describe the behavior of a system completely with sequence diagrams. We are aiming at the development of *practical* means for the inspection of designs and our belief is that the verification of critical properties of particularly important uses of the system is sufficient with respect to cost and benefit. It is here where our approach differs in an important way from formal verification. We verify that, if the execution of each sequence starts in a state that satisfies the sequence precondition then it ends in a state that satisfies the corresponding use-case postcondition. However, for executions that are not captured in a sequence diagram we do not know if the use-case postcondition will be satisfied.

For every sequence diagram and every use-case, questions need to be generated. In particular, questions are generated to verify that all use-case postconditions, sequence postconditions, and operation preconditions are satisfied, and that all class invariants are maintained. In principle, one question is generated for each of the conditions that has to be verified. The resulting questions are the only questions needed to be answered during the design inspection (for functional correctness). A single question consists of a premise and a conclusion, i.e. it is of the form: assuming that x is satisfied, is also y satisfied? Possible answers are "yes", "no", and "don't know". From a logical point of view, "no" and "don't know" are the same. However, we suggest that "no" is used if there is a contradiction between the premise and the conclusion and that "don't know" is used if there is not. If all questions can be answered positively then the design is assumed to be correct.

The questions are generated from a given annotated UML diagrams using an *inference system.* The formulae of the inference system have either the form $\{P\}\, D\, \{Q\}$ or simply R, where D is a use-case or sequence and where P, Q, and R are formulae as described earlier. The triples are very similar to *Hoare triples* (see [6]) and are to be read "if D starts executing in a state where P holds, and if the execution of D terminates, then Q holds upon termination" (i.e., we are concerned with partial correctness only).

The following axioms and rules thus define the semantics of well-formed sequences as described in Def. 5.

Rule 1 (Use-case)
Let S_U be a complete sequence associated with the use-case U and let r be the reference point at which the sequence (and hence also the use-case) ends. If $P_{S_U} \wedge Q_{S_U}(r) \Rightarrow Q_U(r)$ and if executing S_U in a state where P_{S_U} is satisfied results in a state where $Q_{S_U}(r)$ is satisfied, then executing U in a state where P_{S_U} is satisfied must result in a state where $Q_U(r)$ is satisfied:

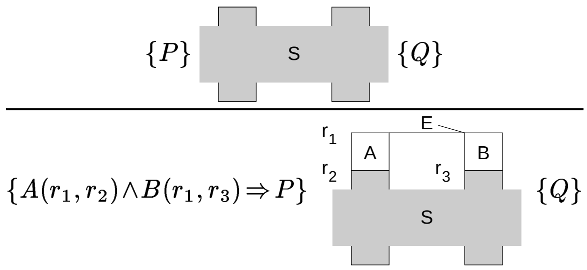
$$\frac{P_{S_U} \wedge Q_{S_U}(r) \Rightarrow Q_U(r) \qquad \{P_{S_U}\}\; S_U\; \{Q_{S_U}(r)\}}{\{P_{S_U}\}\; U\; \{Q_U(r)\}}$$

Rule 1 is used to generate questions to verify that each sequence precondition and postcondition imply the corresponding use-case postcondition (Arrow 4 in Fig. 2). The formula $P_{S_U} \wedge Q_{S_U}(r) \Rightarrow Q_U(r)$ usually contains predicates without formal definition and it is therefore not possible to formally prove that it is satisfied. Instead, this formula is presented as question to the human inspector.

Axiom 2 (Empty sequence)
Executing ϵ_n in a state where Q is satisfied must result in a state where Q is satisfied:

$$\{Q\}\; \epsilon_n\; \{Q\}$$

Rule 3 (Signal)
Let S_E be the partial sequence obtained by prepending a signal and two activation segments to the partial sequence S (see below). If executing S in a state where P is satisfied results in a state where Q is satisfied, then executing S_E in a state where $A(r_1, r_2) \wedge B(r_1, r_3) \Rightarrow P$ is satisfied must result in a state where Q is satisfied:

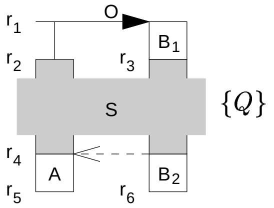


The effect of the signal E is not obvious from the above inference rule. However, the effect is that the arguments to the signal become visible to the receiving object and thus can be used in the intermediate assertion of activation segment B.

Axiom 4 (Operation invocation and return)
Let S_O be the partial sequence obtained by prepending an operation invocation and one activation segments, and by appending an operation return and two activation segments to the partial sequence S (see below). Executing S_O in a state where $I_O(r_1) \wedge (A(r_4, r_5) \wedge B_2(r_4, r_6) \wedge F_O(r_1, r_4) \wedge$

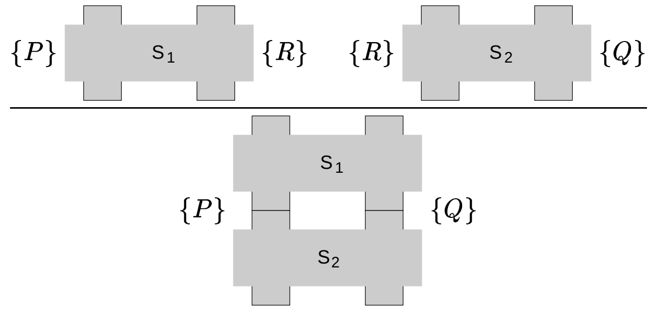
$C_b(r_4) \Rightarrow Q$ *is satisfied must result in a state where Q is satisfied:*

$$\left\{ \begin{array}{r} I_O(r_1)\wedge \\ (A(r_4,r_5)\wedge B_2(r_4,r_6)\wedge \\ F_O(r_1,r_4)\wedge C_b(r_4)\Rightarrow Q) \end{array} \right\}$$



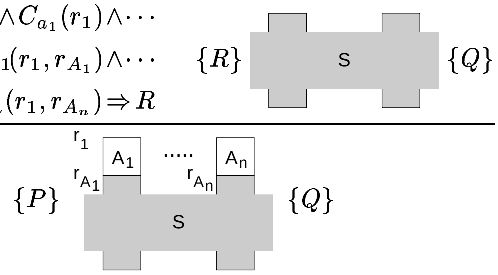
It should be noted that we do not verify operation postconditions. We simply assume that later implementations of operations satisfy their postconditions and leave the verification to later stages (see [5]). However, it is possible to replace the above axiom with a rule that verifies the postconditions already at the design stage.

Rule 5 (Composition)
Let S_C be the partial sequence obtained by appending the partial sequence S_2 to the partial sequence S_1 (see below). If executing S_1 in a state where P is satisfied results in a state where R is satisfied and if executing S_2 in a state where R is satisfied results in a state where Q is satisfied , then executing S_C in a state where P is satisfied must result in a state where Q is satisfied:



Rule 6 (Completion)
Let S_S be the complete sequence obtained by prepending an activation segment to each object in the partial sequence S (see below). If $P\wedge C_{a_1}(r_1)\wedge\cdots\wedge C_{a_n}(r_1)\wedge A_1(r_1,r_{A_1})\wedge\cdots\wedge A_n(r_1,r_{A_n})\Rightarrow R$ is valid and if executing S in a state where R is satisfied results in a state where Q is satisfied, then executing S_S in a state where P is satisfied must result in a state where Q is satisfied:

$$\begin{array}{r} P\wedge C_{a_1}(r_1)\wedge\cdots \\ \wedge C_{a_n}(r_1)\wedge A_1(r_1,r_{A_1})\wedge\cdots \\ \wedge A_n(r_1,r_{A_n})\Rightarrow R \end{array}$$



Rules and Axioms 2 to 6 are used to generate question to help the inspector to detect defects in a sequence diagram (Arrow 2 in Fig. 2). The formula $P \wedge C_{a_1}(r_1) \wedge \cdots \wedge C_{a_n}(r_1) \wedge A_1(r_1, r_{A_1}) \wedge \cdots \wedge A_n(r_1, r_{A_n}) \Rightarrow R$ in Rule 6 usually contains predicates without formal definition and it is therefore not possible to formally prove that it is satisfied. Hence the formula is presented as questions to the human inspector after certain automatic simplifications. For example, usually some atomic formulae in the premise are of the form $x\#r_1 = y\#r_2$. Then all occurrences of $y\#r_2$ in the question are replaced with $x\#r_1$ and the predicate $x\#r_1 = y\#r_2$ is removed. Another simplification is e.g. that predicates which appear both in the premise and conclusion can be removed from the conclusion.

The axioms and rules as they are presented above are not complete. Two simple types of variable substitutions have been omitted for better readability. Class invariants and operation pre- and postconditions usually refer to the object they belong to by using the special variable **this**. Whenever the variable **this** occurs while applying the axioms and rules of the inference system it has to be substituted with the actual name of the object it represents. In addition the formal parameters occurring in operation pre- and postconditions need to be substituted with the actual parameters appearing in the sequence diagram.

III. An Example

The example is intended to illustrate how the questions used in the semantic design inspection are generated. It does not describe a useful system.

A. System Specification

We consider a single use-case **visit** that represents the person **Joe** visiting a museum **Louvre**. One aspect of such a use-case is the location of Joe. When Joe starts visiting the Louvre he should be outside of the museum. Hence the use-case precondition is:

outside(Joe@pre, Louvre@pre)

When the use-case ends Joe should again be outside of the Louvre. Thus the use-case postcondition is:

outside(Joe, Louvre)

B. Scenario Specification

Figure 1 shows a very simple sequence diagram that describes the order of messages that occur during Joe's visit of the museum. The sequence precondition is the same as the use-case precondition:

outside(Joe@pre, Louvre@pre)

However, in our scenario we want to describe Joe's lawful visit of the museum. That is, after Joe's visit we want the museum to be the same as before his visit (in particular we want no painting to be missing or damaged). Therefore, the sequence postcondition is:

lawfulOutside(Joe, Louvre, Louvre@pre)

C. Design

A short visit of Joe consists of entering and leaving the museum. This is modeled by invoking the operations **enter** and **exit** of the museum. Both operations have one parameter **person** representing the person visiting the museum.

The precondition of the operation **enter** is that the person should be outside first:

outside(person@pre, this@pre)

The postcondition is that the person is inside the museum and that the museum otherwise remains unchanged:

entered(person, this, this@pre)

The precondition of the operation **exit** is that the person is inside the museum:

inside(person@pre, this@pre)

The postcondition is that the person is outside the museum and that the museum otherwise remains unchanged:

exited(person, this, this@pre)

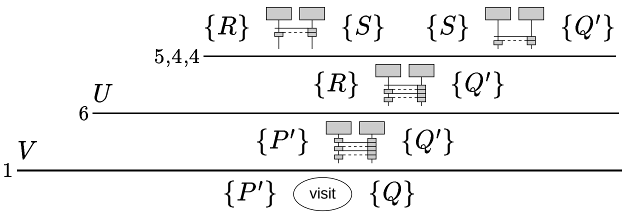
Operation pre- and postconditions (and class invariants which in this example are all *true*) are attached to class diagrams (which are not shown in this example).

Since the lawful visit requires the museum to remain unchanged in the end, we have to attach a couple of intermediate assertions to the sequence. The first, third, and fifth activation segment of the **Louvre** as well as the first, second, and third activation segment of the **Joe** have the following intermediate assertion:

this=this@pre

D. Generation of Questions

The generation of questions is based on the axioms and rules presented in Sect. II-C. The following notation is used. The person **Joe** is abbreviated with j and the museum **Louvre** with l. The activation segments of each object are denoted with the object letter and the segment number as index (the segments are sequentially numbered starting with 1 for the top segment). Since all class invariants are true, we omit them from the formulae below. The pre- and postcondition of **enter** are abbreviated i_i and f_i whereas the pre- and postcondition of **exit** are abbreviated i_o and f_o. The use-case postcondition is abbreviated Q, and the sequence pre- and postconditions are abbreviated P' and Q'. The following derivation tree shows how the questions are produced (the numbers in front of the lines indicate which axiom or rule has been used):



Where:

$$
\begin{aligned}
S &: \quad i_o(3)\wedge(j_3(4,5)\wedge l_5(4,5)\wedge f_o(3,4)\Rightarrow Q'(5))\\
R &: \quad i_i(1)\wedge(j_2(2,3)\wedge l_3(2,3)\wedge f_i(1,2)\Rightarrow S)\\
U &: \quad P'\wedge j_1(0,1)\wedge l_1(0,1)\Rightarrow R\\
V &: \quad P'\wedge Q'(5)\Rightarrow Q(5)
\end{aligned}
$$

In general, the formulae U and V cannot be formally verified because they contain predicates without formal definition. The formula U is presented as a single question to the inspector:

Assume:
1. outside(Joe@pre, Louvre@pre)
2. lawfulOutside(Joe, Louvre, Louvre@pre)

Then:
1. outside(Joe, Louvre)

The answer to the above question is obviously "yes" (if Joe is lawfully outside the Louvre then he is outside the Louvre).

The remaining formula V is presented to the inspector as well. Using conventional predicate logic the formula can easily be rewritten as a conjunction of three implications:

$$
\begin{gathered}
(P'\wedge j_1(0,1)\wedge l_1(0,1)\Rightarrow i_i(1))\\
\wedge\\
(P'\wedge j_1(0,1)\wedge l_1(0,1)\wedge j_2(2,3)\wedge l_3(2,3)\wedge\\
f_i(1,2)\Rightarrow i_o(3))\\
\wedge\\
(P'\wedge j_1(0,1)\wedge l_1(0,1)\wedge j_2(2,3)\wedge l_3(2,3)\wedge\\
f_i(1,2)\wedge j_3(4,5)\wedge l_5(4,5)\wedge f_o(3,4)\Rightarrow Q'(5))
\end{gathered}
$$

The second question checks if the precondition of **enter** is valid:

Assume:
1. outside(Joe@pre, Louvre@pre)

Then:
1. outside(Joe@pre, Louvre@pre)

The answer to the above question is obviously "yes". In fact, it could (and should) be answered automatically.

The third question checks if the precondition of **exit** is valid:

Assume:
1. outside(Joe@pre, Louvre@pre)
2. entered(Joe#2, Louvre#2, Louvre@pre)

Then:
1. inside(Joe#2, Louvre#2)

The answer to the above question is obviously "yes" (if Joe entered the Louvre then he is inside).

The final question checks if the sequence postconditions is valid:

Assume:
1. outside(Joe@pre, Louvre@pre)
2. entered(Joe#2, Louvre#2, Louvre@pre)
3. exited(Joe#4, Louvre#4, Louvre#2)

Then:
1. lawfulOutside(Joe#4, Louvre#4, Louvre@pre)

The answer to the question is "yes". Joe was outside the Louvre, he entered the Louvre without changing it (besides entering it), and then he left the Louvre again without changing it (besides leaving it). The premise indicates that we do not know if Joe changed while entering or leaving the museum.

IV. The Inspection Process

Our approach defines precisely what to do in the defect detection phase (i.e., answer the automatically generated questions). Hence, it may be adapted to different existing inspection processes. The resulting process has the advantage of being more precisely defined, e.g. the inspection phase is easy to document (storing the answers to the questions), and it is repeatable and systematic. Moreover, re-inspections after modifications of the design are facilitated since only the questions that are influenced by the modifications have to be considered. Questions that have not changed from the original inspection need not to be answered again (provided no definitions of involved predicates are altered).

Due to the automatic generation of questions our approach allows for a novel type of inspection based on voting. It was not our initial focus to define a complete inspection process. However, to fully exploit the advantages we suggest the following (design) inspection process:

1. During the *planning phase* the inspection team is selected, the design to inspect is chosen, and a deadline for the individual inspections is defined.
2. The *inspection phase* consists of one or two steps. In the first step each member of the inspection team answers individually the questions generated for the inspected design. After the deadline the answers of all members are compared. If a (qualified) majority answered "yes" to each question then the inspection is over. If there is no majority for some questions the inspection team meets to discuss these particular questions. If there still is no majority for "yes" for some questions then a potential defect has been found.
3. Finally, in the *rework phase* the discovered defects have to be removed. A defect may be a fault/omission in the specification or in the design. Thus the specification or the design has to be corrected or extended and an additional inspection has to be scheduled. However, as mentioned earlier, only the new questions that arise from the modification have to be investigated. The original question allows to locate the defect in the design.

The voting approach facilitates a distributed, asynchronous inspection. In conventional inspection the inspection phase includes a group meeting for group inspection and logging. Group meetings are typically limited to a maximum duration of two hours. Larger artifacts thus require several inspection meetings. Each meeting is associated with a large overhead due to problems finding a mutually agreeable time, a room for the meeting etc. This overhead is to a large extent avoided in the voting approach.

Tool-support is essential for our type of inspection. The tool should allow to enter specification and design, that is annotated use-case, class, and sequence diagrams. Moreover, it should be able to generate the questions and to present them to the user. The presentation should include both the question itself and the involved diagrams with the relevant parts highlighted. Finally, the tool has to collect the result and to establish the outcome of the voting. All the services above have to be provided in a distributed environment with some kind of central repository.

It is not actually necessary to build such a tool from scratch. For example, the functionality outlined above may be realized as a "plug-in" to an existing tool as e.g. Rational Rose. To evaluate our approach and to show its feasibility a prototype tool has been implemented in Java. The prototype allows to enter specification and design, and it generates and presents resulting questions in a single user environment but it does not support voting and re-inspections. The screenshot found in Fig. 1 is taken from that prototype.

V. Related Work

The basis of our approach to systematic functionality-oriented design inspection is traditional inspection and formal verification. A great number of books, articles, and conferences are concerned with these topics and we can only mention a fraction of them. However, existing approaches are either completely formal or completely informal. We are not aware of other approaches that combine traditional inspection and formal verification to systematically consider semantic issues in the design inspection process.

A. *Software Inspections*

Inspection techniques have been introduced for various artifacts that are created in the software development process (i.e. requirements, design, code, and tests). Several inspection processes have been presented, e.g. Two-Person Inspections by Bisant and Lyle (see [7]), N-Fold Inspections by Schneider and Martin and Tsai (see [8]), Phased Inspections by Knight and Myers (see [9]), and Active Design Reviews by Parnas and Weiss (see [10]). In general, software inspections are reported to be very effective both with respects to defect-detection and costs (see e.g. [2]). Various tools are available to facilitate inspections. For example, Macdonald and Miller (see [3]) presented a comparison of 16 inspection tools and there own tool ASSIST. However, the focus of these tools is almost entirely on administrative tasks. Thus, there is still a lack of guidelines which describe exactly how to find defects with respect to the intended functionality of the inspected artifact.

Porter et al. (see [11]) have investigated the effects of structural changes of the (code) inspection process (i.e., team size, number of sessions, repair occasions) on the inspection performance (i.e., inspection effectiveness and interval). However, they discovered that the performance varied widely independent of the treatment used. Also their data indicated that only 15 % of the issues reported during individual preparation concern true defects. This suggests that it may be much more important to develop better defect detection techniques than other inspection processes. Further studies (see [12]) consider the influence of process inputs, (e.g., code units and reviewers) on the defect detection. Porter et al. found that even when the process inputs are accounted for, then structural changes of the inspection process had little effect on the defect detection. Therefore, according to Porter et al., research on better techniques for the actual defect detection steps should not be neglected. The semantic software inspection approach presented by us is a suggestion of a more strict defect detection technique. Our approach defines precisely the questions that need to be addressed to check whether the software artifact contains defects.

B. Formalizations of UML

The lack of formal semantics of the UML has been recognized as a major problem. Ambiguities hinder the exchange of UML models between different parties and also impede the development of computer support for analysis and verification. Thus, several attempts for formalizing UML have been published. An overview can be found e.g. in [13]. In the next two paragraphs we discuss two specific approaches.

Back, Petre, and Paltor (see [14]) presented an approach that allows formalizing UML use-cases in the refinement calculus. That is, use-cases are annotated with a kind of pre- and postconditions in a formal *contract language*. The contract language allows to verify the use-cases against a kind of annotated class descriptions. Our focus is on the verification of early, usually incomplete designs expressed in UML.

In the Syntropy approach (see [15]) class and statechart diagrams are annotated with formulas in predicate logic to allow e.g. automatic consistency and completeness checks. Detailed knowledge of the formal notation is required and large portions of the system have to be annotated. Our focus is on semi-formal annotations and on sequence diagrams which only describe particular executions of the system. We can argue for the correctness of the software in certain scenarios which is, as we believe, an easier and more practical approach.

C. Sequence Charts

The key type of diagram for the generation of questions used in the our design inspection is the sequence diagram. UML sequence diagrams are very similar to Message Sequence Charts (MSC) and Live Sequence Charts (LSC).

Several formalizations of MSCs have been suggested. Among them are approaches based on automata theory (see [16]), Petri net theory (see [17]), and process algebra (see [18]). Common to these approaches is their focus on the communication, i.e., on the order, timing, and number of received and dispatched messages (similar approaches exist for UML as well, e.g. in [19] an algorithm to check compositions of UML sequence diagrams for timing inconsistency is presented). However, the actual calculations that are performed are not considered. To verify that the system provides the intended functionality we need a semantics that takes these calculations into account.

LSCs introduced by Damm and Harel (see [20]) are an extension to standard MSCs. The precise semantics of LSCs is the basis to relate an *inter*-object specification (e.g. LSCs) to an *intra*-object specification (e.g. statecharts) of a system. Klose and Wittke (see [21]) describe how a subset of LSCs can be transformed into a Timed Büchi Automaton from which a temporal logic formula can be generated. In principle this allows to verify (using a model checker) a statechart model of the system against LSCs (temporal logic formulae). However, we are interested in the verification of early designs and state charts are typically used later in the software development .

D. Assertions

The use of assertions to formally specify and verify software has been introduced by Hoare [6] in 1969. Since then the use of different types of assertions for various notations and constructs has been investigated in detail (in academia). Assertions are used for *formal verification*, *runtime checking*, and *documentation*. Our approach differs from others in that we use assertions containing predicates without formal definitions together with a formal design notation. The type of assertions we are using makes formal verification impossible but we are able to automatically generate questions which form the basis for a systematic design inspection.

VI. Conclusion

We have presented a novel approach to design inspection. In contrast to existing approaches we combine conventional inspection with elements of formal verification. Conventional inspection lacks guidelines that precisely state which questions to address to find defects. Formal verification is often perceived as being very difficult due to the required knowledge of the formal notation and its proof system.

In our approach, the design is expressed in annotated UML class and sequence diagrams whereas the specification is expressed in annotated UML use-case diagrams. The annotations (called assertions) are similar to quantification-free predicate logic. Our key idea, is that we allow predicates without associated formal definition to occur in the assertions. Instead the definitions are provided informally, e.g. natural language. These informal predicates make it easier to express the required assertions and enable reasoning about assertions at a high level. Since our assertions have a formal syntax it is still possible to automatically generate questions (verification conditions) to be answered during the design inspection. If all questions can be answered positively then the design is assumed to be correct with respect to the specification. The semantic design inspection we suggest allows a distributed, asynchronous, repeatable, and systematic inspection. Moreover, re-inspections after modifications of the design are facilitated.

From a technical point of view our approach is working and has successfully been applied to small designs (around five objects and ten messages). In-depth evaluations involving large real-world applications are still to come. The key issues are the amount and complexity of the assertions and the generated question. It is possible to decrease the complexity of assertions and questions by increasing their number and vice versa. For example, the introduction of sequence diagram invariants (i.e., conditions that remain satisfied during the whole execution of a sequence) decrease the complexity of assertions but increase their number. Global intermediate assertions in sequence diagrams (i.e., conditions that specify the state of the whole system at a certain reference point) increase the number of assertions and questions but decrease their complexity. The future challenge is to find the right mix of complexity and amount of assertions and questions.

References

[1] Michael E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Systems Journal*, vol. 15, no. 1, pp. 182–211, 1976.

[2] Tom Gilb and Dorothy Graham, *Software inspection*, Addison Wesley, 1993.

[3] Fraser Macdonald and James Miller, "A comparison of computer support systems for software inspection," *Automated Software Engineering*, vol. 6, no. 3, pp. 291–313, 1999.

[4] "OMG Unified Modeling Language specification 1.3," June 1999.

[5] Staffan Bonnier and Tim Heyer, "COMPASS: A comprehensible assertion method," in *TAPSOFT '97: Theory and Practice of Software Development*. 1997, pp. 803–817, Springer-Verlag.

[6] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communication of the ACM*, vol. 12, no. 10, pp. 576–80, 583, Oct. 1969.

[7] David B. Bisant and James R. Lyle, "A two-person inspection method to improve prog ramming productivity," *IEEE Transactions on Software Engineering*, vol. 15, no. 10, pp. 1294–1304, Oct. 1989.

[8] G. Michael Schneider, Johnny Martin, and W. T. Tsai, "An experimental study of fault detection in user requirements documents," *ACM Transactions on Software Engineering and Methodology*, vol. 1, no. 2, pp. 188–204, Apr. 1992.

[9] John C. Knight and E. Ann Myers, "An improved inspection technique," *Communication of the ACM*, vol. 36, no. 11, pp. 51–61, Nov. 1993.

[10] David L. Parnas and D. M. Weiss, "Active design reviews: principles and practices," *Journal of Systems and Software*, vol. 7, no. 4, pp. 259–265, Dec. 1987.

[11] A. Porter, H. Siy, C. A. Toman, and L. G. Votta, "An experiment to assess the cost-benefits of code inspections in large scale software development," in *Proceedings Symposium on the Foundations of Software Engineering (SIGSOFT '95)*, G. E. Kaiser, Ed. 1995, pp. 92–103, ACM Press.

[12] Adam A. Porter, Harvey Siy, Audris Mockus, and Lawrence G. Votta, "Understanding the sources of variation in software inspections," *ACM Transactions on Software Engineering and Methodology*, vol. 7, no. 1, pp. 41–79, 1998.

[13] Andy Evans, Jean-Michel Bruel, Robert B. France, and Kevin Lano, "Making UML precise," in *Proceedings Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'98)*, Oct. 1998.

[14] Ralph-Johan Back, Luigia Petre, and Iván Porres Paltor, "Formalising UML use cases in the refinment calculus," Technical report 279, Turku Centre for Computer Science (TUCS), May 1999.

[15] Steve Cook and John Daniels, "Let's get formal," *Journal of Object-Oriented Programming*, pp. 22–24, 64–66, 1994.

[16] Peter B. Ladkin and Stefan Leue, "What do message sequence charts mean?," in *Formal Description Techniques VI, IFIP Transactions C, Proceedings Conference on Formal Description Techniques (FORTE '93)*, 1994, vol. C-22, pp. 301–316.

[17] Peter Graubmann, Ekkart Rudolph, and Jens Grabowski, "Towards a Petri net based semantics definition for message sequence charts," in *Using Objects, Proceedings SDL Forum (SDL '93)*, 1993, pp. 179–190.

[18] S. Mauw and M. A. Reniers, "An algebraic semantics of basic message sequence charts," *Computer Journal*, vol. 37, no. 4, pp. 269–277, 1994.

[19] Xuandong Li and Johan Lilius, "Checking compositions of uml sequence diagrams for timing inconsistency," Technical report 363, Turku Centre for Computer Science (TUCS), Aug. 2000.

[20] Werner Damm and David Harel, "LSC's: breathing life into message sequence charts," in *Proceedings conference on formal methods for open object-based distributed systems (FMOODS '99)*. 1999, pp. 293–311, Kluwer Academic Publishers.

[21] Jochen Klose and Hartmut Wittke, "An automata based interpretation of live sequence charts," in *Proceedings Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '01)*. 2001, Lecture Notes in Computer Science (LNCS), pp. 512–527, Springer-Verlag.