Chapter 8

Designing and documenting the behavior of software

Authors: Gürcan Güleşir, Lodewijk Bergmans, Mehmet Akşit

Abstract The development and maintenance of today's software systems is an increasingly effort-consuming and error-prone task. A major cause of this problem is the lack of formal *and* human-readable documentation of software design. In practice, software design is often informally documented (e.g. texts in a natural language, 'boxes-and-arrows' diagrams without well-defined syntax and semantics, et cetera), or not documented at all. Therefore, the design cannot be properly communicated between software engineers, it cannot be formally analyzed, and the conformance of an implementation to the design cannot be formally verified.

In this chapter, we address this problem for the design and documentation of the behavior implemented in procedural programs. We introduce a solution that consists of three components: The first component is a graphical language called VisuaL, which enables engineers to specify constraints on the possible sequences of function calls from a given program. Since the specifications may be inconsistent with each other, the second component of our solution is a tool called CheckDesign, which automatically verifies the consistency between multiple specifications written in VisuaL. The third component is a tool called CheckSource, which automatically verifies that a given implementation conforms to the corresponding specifications written in VisuaL.

This solution has been evaluated empirically through controlled experiments with 71 participants: 23 professional developers of ASML, and 49 Computer Science M.Sc. students. These experiments showed that, with statistical significance of 0.01, the solution reduced the effort of typical maintenance tasks by 75% and prevented one error per 140 lines of source code. Further details about these results can be found in Section 10.5.

8.1 Introduction

VisuaL and the associated tools are outcome of our close collaboration with ASML. Our collaboration with ASML was divided in four phases: In the first phase, we identified a number of effort-consuming and error-prone tasks in software development and maintenance processes. In the second phase, we developed VisuaL and the tools to automate these tasks. In the third phase, we conducted formal experiments to evaluate CheckSource. In the fourth and the final phase, ASML initiated a half-year project to embed VisuaL and the tools into their software development and maintenance processes. This project was ongoing at the time of writing this chapter.

The remainder of this chapter is structured as follows: In Section 8.2, we present some common problems of today's software engineering practice. In Section 8.3, we explain our approach for improving the software engineering practice. In Section 8.4, we introduce VisuaL, and illustrate it with seven use cases. In Sections 8.5, and 8.6, we respectively introduce CheckDesign and CheckSource. We conclude with Section 8.7.

8.2 Obstacles in the development of embedded software

In the first phase of the Ideals project, we investigated the software engineering process of ASML, and identified a number of general problems. In this section, we explain these problems.

8.2.1 Informal documentation of software design

Natural languages are frequently used in the industrial practice, for documenting the design of software. For instance, we have seen several design documents containing substantial text in English, written in a 'story-telling' style. Although the unlimited expressive power is an advantage of using a natural language, this freedom unfortunately allows for ambiguities and imprecision in the design documents.

In addition to the texts in a natural language, design documents frequently contain figures that illustrate various facets of software design, such as the structure of data, flow of control, decomposition into (sub)modules, et cetera. These figures provide valuable intuition about the structure of software. However, typically such figures cannot be used as precise specifications of the actual software, since they are abstractions with no well-defined mapping to the final implementation in source code.

As we discuss in Sections 8.2.2 and 8.2.3, ambiguous and informal software design documents are a major cause of excessive manual effort and human errors during software development and maintenance.

8.2.2 Obstacles in the software development process

In Figure 8.1, we illustrate a part of the software development process of ASML, showing four steps:

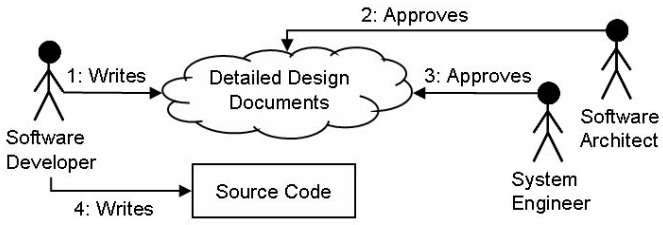

Figure 8.1: This figure shows part of the software development process at ASML.

In the first step, a software developer writes detailed design documents about the new feature that she will implement. The detailed design documents are depicted as a cloud to indicate that they are informal and potentially ambiguous.

In the second step, a software architect reviews the documents. If the architect concludes that the design of the new feature 'fits' the architecture of software, then she approves the design documents.

In the third step, a system engineer reviews the design documents. If the system engineer concludes that the new feature 'fits' the electro-mechanical parts of the system, and fulfills the requirements, then she approves the design documents.

In the fourth step, the developer implements the feature by writing source code. The source code is depicted as a regular geometric shape (i.e. rectangle in this case) to indicate that the source code is written in a formal language.

After the feature is implemented, it is not possible to conclude with a large certainty that the source code is consistent with the design documents, because the design documents are informal and potentially ambiguous. Therefore, the following problems may arise:

- The structure of the source code may be inconsistent with the structure approved by the software architect, because the architect may have interpreted the design differently than the software developer.
- The implemented feature may not 'fit' the electro-mechanical parts of the system, because the system engineer may have interpreted the design differently than the software developer. In such a case, the source code is defective.

8.2.3 Obstacles in the software maintenance process

In Figure 8.2, we illustrate a part of the software maintenance process of ASML, showing five steps: In the first step, a developer receives a change request (or a problem report) related to the implementation of an existing feature. If the developer concludes that the change request has an impact on the detailed design, then she accordingly modifies the detailed design documents, in the second step. If the design documents are modified, then a software architect and a system engineer review and approve the modified design documents, in the third and the fourth steps. In the fifth step, the developer implements the change by modifying the existing source code.

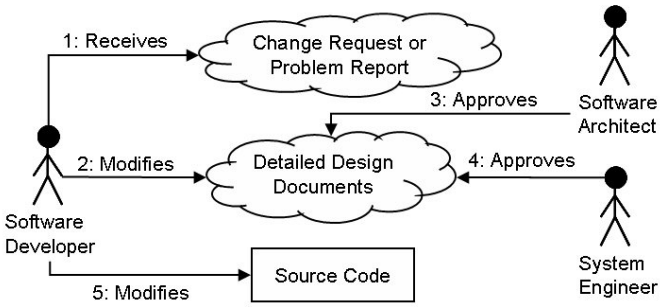

Figure 8.2: This figure shows part of the software maintenance process at ASML.

In practice, engineers can follow shortcuts in the maintenance process explained above, because they are often urged to decrease the time-to-market of a product. They can skip the second, third, or fourth steps, because the design documents are not a part of the product that is shipped to customers. This shortcut leads to the following problems:

- The source code 'drifts away' from the design documents. More precisely, the design that is implemented in the source code becomes substantially different than the design that is written in the documents. In such a case, the design documents become useless, because the source code is the only artifact that 'works', and the design documents do no longer provide any useful information about the source code.
- Since the design documents become useless, a developer has to directly read and understand the source code, whenever she needs to modify software. Consequently, maintenance becomes more effort-consuming and error-prone, because the developer is constantly exposed to the whole complexity and the lowest level details of software.
- Since the design documents become useless, the software architect and the system engineer cannot effectively control the quality of software during evolution, which results in the same problems listed in Section 8.2.2.
- Since the design documents become useless, the initial effort spent by the developer to write the design documents, and the effort spent by the software architect and the system engineer to review them, are no longer utilized.

The problems explained so far in Section 8.2 are more broadly explained in Section 1.3.2.

8.2.4 The scope in this chapter

The scope of the problems that we explained so far is too broad to be effectively addressed by a single solution. Therefore, we communicated with the engineers of ASML to determine a sub-scope that is narrow enough to be effectively addressed, general

enough to be academically interesting, and important enough to have industrial relevance. As a result, we chose to restrict our scope to the design and documentation of the control flow within C functions. In the remainder of this section, we explain the reason for this choice.

Abstractly speaking, the manufacturing machines produced by ASML perform certain operations on some input material. These operations must be performed in a sequence that satisfies certain temporal constraints, otherwise the machines cannot fulfill one or more of their requirements. For example, a machine must clean the input material *before* processing it, otherwise the required level of mechanical precision cannot be achieved during processing; loss of precision results in defective output material.

In software, the input material is modeled as a data structure, and each operation is typically implemented as a function that can read or write instances of the data structure. The possible sequences of operations are determined by the control flow structure of a separate function that calls the functions corresponding to the operations.

During software maintenance, the engineers of ASML frequently change the control flow structure of functions, and unintentionally violate the temporal constraints. These violations result in software defects. Finding and repairing these defects is effort-consuming and error-prone, because (a) the constraints are either not documented at all, or poorly documented, as explained in Section 8.2.1, and (b) there are no explicit means for the engineers to tell them if and where the constraints are violated.

Based on these observations, we decided to find a better way to document the temporal constraints, and to develop tools that can help engineers in finding and repairing the defects. As a result, we developed a solution that consists of a graphical language *VisuaL*, a tool for verifying internal consistency of the design *CheckDesign*, and a tool for verifying the consistency between the design and the source code, *CheckSource*.

VisuaL is a graphical language for expressing temporal constraints on operations in a system, in particular on the operations within a specified function body. It aims at being both intuitive (through a UML-style visual notation), precise (a VisuaL diagram can be mapped to a formal representation of automata), and evolution-proof (through the use of wildcards, one can specify only necessary ordering constraints).

8.3 Solution approach

In this section, we explain how our solution (i.e. VisuaL, CheckDesign, and CheckSource) can be used during software development and maintenance. The details of the solution are presented throughout Sections 8.4-8.7.

8.3.1 Adapting the software development process

We present the software development process in which our solution is used, in two steps: (1) the software design process, and (2) the software implementation process.

The Software Design Process

In Figure 8.3, we illustrate the software design process, in which VisuaL and CheckDesign are used. This process consists of four steps:

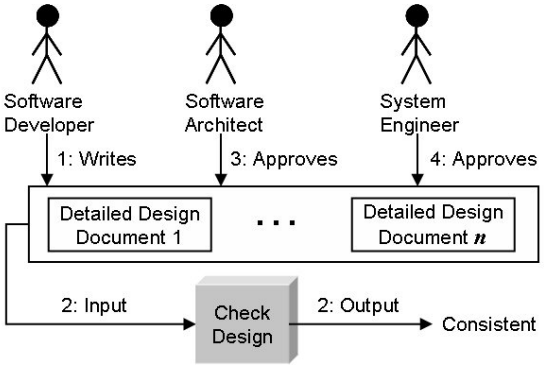

Figure 8.3: This figure shows the design process with VisuaL and CheckDesign.

In the first step, a software developer writes detailed design documents about the feature that she will implement. She uses VisuaL for writing the documents. Therefore, the resulting documents are formal and unambiguous.

In the second step, CheckDesign automatically verifies the consistency between those documents that apply to the same function. If the documents are not consistent, CheckDesign outputs an error message that contains information for locating and resolving the inconsistency. Note that in the original development process (see Section 8.2.2), design level verification was not possible due to the informal and potentially ambiguous documentation.

If CheckDesign outputs a success message, a software architect and a system engineer review and approve the design documents, in the third and fourth steps. Thus, an important requirement is that 'The design documents should be easily read and understood by humans'.

The software implementation process

Figure 8.4 shows the software implementation process in which the formal design documents and CheckSource are used. This process consists of two steps:

In the first step, a software developer implements the feature by writing source code.

In the second step, CheckSource verifies the consistency between the source code and the design documents. If the source code is inconsistent with the documents, CheckSource outputs an error message that contains information for locating and resolving the inconsistency.

An inconsistency can be resolved through one of the following scenarios:

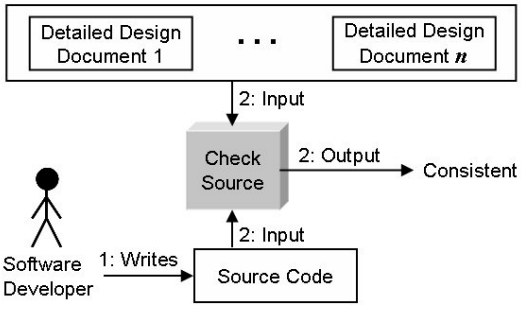

Figure 8.4: This figure shows the implementation process with formal design documents and CheckSource.

- The developer decides that the inconsistency is due to a defect in the source code, so she repairs (i.e. modifies) the source code, and then reruns CheckSource.
- The developer decides that the inconsistency is due to a defect in design documents, so she repairs the design documents and then performs the second, third, and the fourth steps of the design process (see Figure 8.3). After these steps, she reruns CheckSource.
- The developer decides that the inconsistency is due to the defects in both the design documents and the source code. So she repairs the design documents and then performs the second, third, and the fourth steps of the design process (see Figure 8.3). After these steps, she repairs the source code and reruns CheckSource.

The design and implementation processes presented above address the problems listed in Section 8.2.2.

8.3.2 Improving the software maintenance process

Whenever a developer receives a change request (or a problem report) about the implementation of an existing feature, she decides whether the change request has an impact on the detailed design. If the developer decides that there is no such impact, then she directly implements the request by following the implementation process depicted in Figure 8.4. If the developer decides that the change request has an impact on the detailed design, then she realizes the change request by following the design process depicted in Figure 8.3. Subsequently, she implements the change by following the implementation process depicted in Figure 8.4. The maintenance process explained in this section addresses the problems listed in Section 8.2.3. In Sections 8.4, 8.5, and 8.6, we respectively present VisuaL, CheckDesign, and CheckSource.

8.4 VisuaL

VisuaL is a graphical language for specifying constraints on the possible sequences of function calls from a given C function. In this section, we explain VisuaL by presenting the specification of seven example constraints, each demonstrating a distinct 'primitive' usage of the language.

8.4.1 Example 1: 'At least one'

Figure 8.5 shows a specification of the following constraint:

C1: In each possible sequence of function calls from the function `f`, there must be *at least one* call to the function `g`.

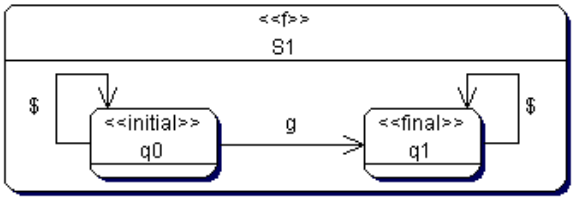

Figure 8.5: An example specification demonstrating the usage of 'At least one'.

The outer rectangle (i.e. the rectangle with the stereotype <<f>>) defines –a view on– the control-flow behavior as implemented by (the body of) function `f`. The label `S1` is the name (i.e. identifier) of the specification. The arrows represent the function calls from `f`, and the inner rectangles (e.g., the rectangle that is labeled with `q0`) represent locations within the control flow of `f`.

Inside the outer rectangle, there is a structure consisting of the arrows and the inner rectangles. We call such a structure a **pattern**.

The stereotype <<f>> means 'each possible sequence of function calls from the function `f` must be *matched by the pattern*[1], otherwise the constraint that is represented by the specification is not satisfied'.

The rectangle `q0` represents the beginning of a given sequence of function calls, because it has the stereotype <<initial>>. We call this the **initial rectangle**. There must be exactly one initial rectangle in each VisuaL specification.

The `$`-labelled arrow originating from `q0` matches each function call from the beginning of a sequence, until a call to `g` is reached. This 'until' condition is due to the existence of the `g`-labelled arrow originating from the same rectangle (i.e. `q0`).

In general, a `$`-labelled arrow matches a function call, if and only if this call cannot be matched by the other arrows *originating from the same rectangle*. In VisuaL, no two arrows originating from the same rectangle can have the same label.

[1] We precisely define this later in this section.

Note the difference between the $-labelled arrow pointing to q0 and the $-labeled arrow pointing to q1: the former arrow can match a call to any function except g, whereas the latter arrow can match a call to any function (i.e. including g), since q1 has no other outgoing arrow.

During the matching of a given sequence of function calls, if the first call to g is reached, then this call is matched by the arrow labeled with g. If there are no more calls in the sequence, then the sequence **terminates** at q1, because the last call of the sequence is matched by an arrow that points to q1.

If there are additional calls after the first call to g, then each of these calls is matched by the $-labelled arrow pointing to q1, hence the sequence eventually terminates[2] at q1.

A given sequence of function calls is **matched by a pattern**, if and only if the sequence terminates at a rectangle with the stereotype <<final>>. We call such a rectangle **final rectangle**. There can be zero or more final rectangles in a VisuaL specification.

8.4.2 Example 2: 'Immediately followed by'

Figure 8.6 shows a specification of the following constraint:

C2: In each possible sequence of function calls from f, the first call to g, if it exists, must be *immediately followed by* a call to h.

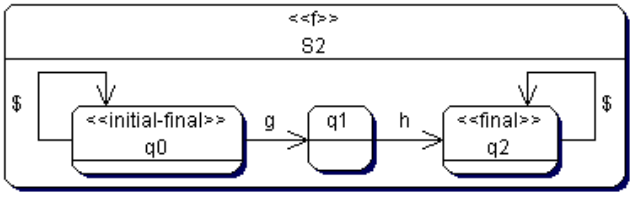

Figure 8.6: An example specification demonstrating the usage of 'Immediately followed by'.

In Figure 8.6, the stereotype <<initial-final>> means that q0 has both <<initial>> and <<final>> stereotypes.

8.4.3 Example 3: 'Each'

Figure 8.7 shows a specification of the following constraint:

C3: In each possible sequence of function calls from f, *each* call to g must be immediately followed by a call to h.

[2] Infinite sequences of function calls are out of the scope of this chapter, because VisuaL is *not* a language for specifying constraints on the execution of possibly non-terminating programs. This topic is already studied in [22].

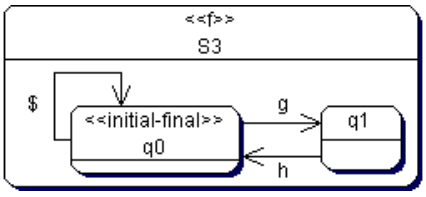

Figure 8.7: An example specification demonstrating the usage of 'each'.

8.4.4 Example 4: 'Until'

Figure 8.8 shows a specification of the following constraint:

C4: In each possible sequence of function calls from `f`, each function call must be a call to `g`, *until* a call to `h` is reached.

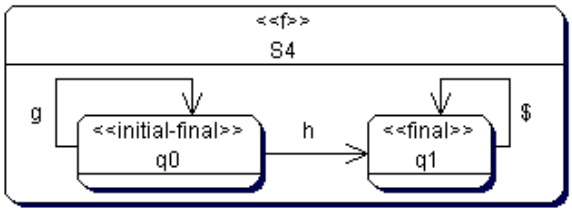

Figure 8.8: An example specification demonstrating the usage of 'until'.

8.4.5 Example 5: 'Not'

Figure 8.9 shows a specification of the following constraint:

C5: In each possible sequence of function calls from `f`, a call to `g` must *not* exist.

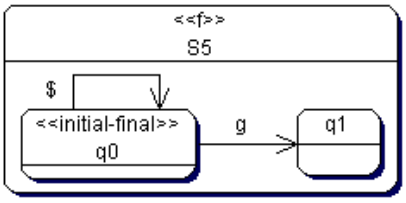

Figure 8.9: An example specification demonstrating the usage of 'not'.

Note that `q1` does not have the stereotype <<final>>, and no arrow originates from `q1`. We call such a rectangle **trap rectangle**. For a given sequence *seq* of function calls, if a call *c* in *seq* is matched by an arrow pointing to a trap rectangle *tr*, then either of the following scenarios occurs:

- *c* is the last call in *seq*. Since *tr* does not have the stereotype <<final>>, *seq* is not matched by the pattern.

- *c* is not the last call in *seq*. In this case, for each remaining call in *seq*, there is no matching arrow in the pattern. Therefore, *seq* is not matched by the pattern.

To sum up, if a sequence 'visits' a trap rectangle, then the sequence cannot be matched by the pattern.

8.4.6 Example 6: 'Or'

Figure 8.10 shows the specification of the following constraint:

C6: In each possible sequence of function calls from `f`, the first function call, if exists, must be a call to `g` *or* `h`.

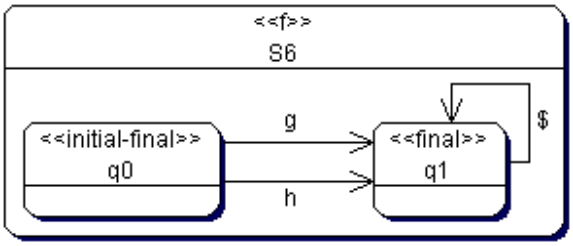

Figure 8.10: An example specification demonstrating the usage of 'or'.

8.4.7 Example 7: 'And'

Figure 8.11 shows the specification of the following constraint:

C7: In each possible sequence of function calls from `f`, there must be at least one call to `g`, *and* the first call to `g` must be immediately followed by a call to `h`.

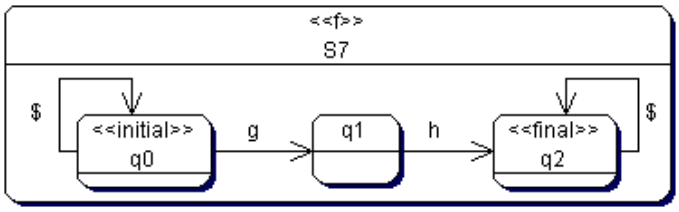

Figure 8.11: An example specification demonstrating the usage of 'and'.

8.5 CheckDesign

Using VisuaL, one can create multiple specifications each representing a different constraint on the same function. For example, each of the seven specifications in Section 8.4 represents a different constraint on the same function: `f`.

When creating multiple VisuaL specifications to express different constraints on the same function, it must be ensured that the specifications are **consistent**: There is at least one possible implementation of the function, such that the implementation satisfies each of the constraints. If there is no possible implementation of the function that satisfies all specified constraints, then the VisuaL specifications are **inconsistent**.

For example, the specifications S1 (Figure 8.5) and S5 (Figure 8.9) are *in*consistent: If an implementation of the function `f` satisfies the constraint C1 (Section 8.4.1), then this implementation cannot satisfy the constraint C5 (Section 8.4.5). Conversely, if an implementation of the function `f` satisfies C5, then this implementation cannot satisfy C1. Hence, it is impossible to implement `f`, such that the implementation satisfies both C1 and C5.

Manually finding and resolving an inconsistency among multiple VisuaL specifications is an effort-consuming and error-prone task. CheckDesign can reduce the effort and prevent the errors. CheckDesign takes a finite set of VisuaL specifications, and automatically finds out whether the specifications are consistent or not. If the specifications are not consistent, CheckDesign outputs an error message that can help in understanding and resolving the inconsistency.

8.6 CheckSource

After creating consistent VisuaL specifications, a developer typically writes source code to implement the specifications. For example, after creating the specification S2 (Figure 8.6), a developer may implement the function `f` as shown in Listing 8.1.

```
1  void f(int i)
2  {
3     g();
4     if(i)
5     {
6        h();
7     }
8  }
```

Listing 8.1: An example implementation of the function f in C.

A function and a corresponding specification may be inconsistent with each other. For example, the function shown in Listing 8.1 is inconsistent with the specification S2 (Figure 8.6): There are two possible sequences of function calls from `f`, and these sequences are $seq_1 =$<`g`, `h`> and $seq_2 =$<`g`>. Although seq_1 is matched by the pattern of S2, seq_2 cannot be matched by this pattern. Therefore, this implementation (Listing 8.1) is inconsistent with S2, which indicates that the implementation does not satisfy the constraint C2.

Manually finding and resolving inconsistencies between a function and its specification is an effort-consuming and error-prone task. CheckSource can reduce the effort and prevent errors. CheckSource takes a function and a corresponding VisuaL specification as the input, and finds out whether they are consistent or not. If they are consistent, then CheckSource outputs a success message, else an error message that is useful for understanding and resolving the inconsistency.

8.7 Conclusions

To conclude, we summarize the possible use cases of VisuaL, CheckDesign, and CheckSource:

- A software engineer can use VisuaL for designing the control flow of a new function to be implemented. After the engineer creates the VisuaL specifications, she can use CheckDesign for verifying that the specifications are consistent with each other.
- A software engineer can use CheckSource to automatically verify that a given function is consistent with the corresponding specifications. Whenever a specification or the function evolves, the verification can be automatically repeated.
- A software engineer can also use VisuaL for designing an additional feature of an existing function. The additional feature can be designed either within the existing specifications, or as a separate specification besides the existing ones. In either case, CheckDesign can be used for ensuring the consistency between the specifications, and CheckSource can be used for ensuring the consistency between the specifications and the implementation.
- Whenever a function evolves, a software engineer can use CheckSource for automatically detecting inconsistencies between the function and the specifications. Such an inconsistency indicates either a bug in the source code, or an outdated specification.
- A software engineer can use VisuaL to distinguish anticipated bugs from features. She can specify the 'illegal' sequences of function calls together with the 'legal' sequences of function calls. For example, any sequence that visits `q3` of S7 (Figure 8.11) is illegal, and any sequence that terminates at `q2` of S7 is legal.
- If VisuaL specifications are kept consistent both with each other and with the source code, then these specifications can be used during code inspections. If engineers suspect a bug in the function call sequences, then they can abstract away from details such as data flow, and focus on the function call sequence, by inspecting only the VisuaL specifications.