Evolvable Behavior Specifications Using Context-Sensitive Wildcards

Gürcan Güleşir

Evolvable Behavior Specifications Using Context-Sensitive Wildcards

Gürcan Güleşir

Ph.D. dissertation committee:

Chairman and secretary:
Prof. Dr. Ir. A.J. Mouthaan, University of Twente, The Netherlands

Promoter:
Prof. Dr. Ir. M. Akşit, University of Twente, The Netherlands

Assistant promoter:
Dr. Ir. L.M.J. Bergmans, University of Twente, The Netherlands

Members:
Prof. Dr. A. van Deursen, Delft University of Technology, The Netherlands
Prof. Dr. J.C. van de Pol, University of Twente, The Netherlands
Prof. Dr. D.S. Rosenblum, University Collage London, United Kingdom
Prof. Dr. P. Runeson, Lund University, Sweden
Prof. Dr. R.J. Wieringa, University of Twente, The Netherlands

CTIT Ph.D. thesis series no. 08-114. Centre for Telematics and Information Technology (CTIT), P.O. Box 217 - 7500 AE Enschede, The Netherlands.

IPA Dissertation Series 2008-13. The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

ISBN 978-90-365-2633-3
ISSN 1831-36-17(CTIT Ph.D. thesis series no. 08-114)

(The lack of) Cover design by Gürcan Güleşir
Printed by PrintPartners Ipskamp, Enschede, The Netherlands
Copyright © 2008, Gürcan Güleşir, Enschede, The Netherlands

Evolvable Behavior Specifications Using Context-Sensitive Wildcards

DISSERTATION

to obtain
the degree of doctor at the University of Twente,
on the authority of the rector magnificus,
prof. dr. W.H.M. Zijm,
on account of the decision of the graduation committee,
to be publicly defended
on Thursday the 13th of March 2008 at 13.15

by

Gürcan Güleşir

born on the 12th of June 1979
in Izmir, Turkey

This dissertation is approved by

Prof. Dr. Ir. M. Akşit (promoter)
Dr. Ir. L.M.J. Bergmans (assistant promoter)

"The hardest thing to see is what is in front of your eyes."

Johann Wolfgang von Goethe

Acknowledgements

The past four years of my life, hence this thesis, has been shaped in the caring and talented hands of Lodewijk Bergmans, my daily supervisor. I benefited from every single intellectual exchange with him. The key ideas presented in this thesis emerged from such exchanges. I have also been deeply influenced by Lodewijk's passion to simplify a scientific argument down to its essence, where the argument can effortlessly be followed by novice readers. If you think that this thesis is easy-to-read, then you should thank Lodewijk. I can never thank him enough for his contributions to my intellectual growth, and I am extremely excited to know that we will continue working together in the upcoming years. Outside work, I have also enjoyed the exceptional hospitality of Ingrid and Lodewijk Bergmans.

During my Ph.D., I worked in the software engineering group led by Mehmet Akşit, my promoter. Every second week, Mehmet and I were meeting to discuss my progress. Especially in the first 18 months, we developed the following communication pattern: Gürcan: "I found the great idea X to save the world. This may be a small step for me, but it will be a giant leap for the mankind." Mehmet: "Hmm. This is a very interesting idea. Look at these slides/papers from 1980s... Note that the idea X is the essence of the phenomena Y, which was studied two thousand years ago by philosopher Z". (Un)fortunately, he has never been wrong. I would like to thank him for making sure that I worked on non-trivial and relevant problems.

I benefited extensively from the experience of Klaas van den Berg while designing and conducting the controlled experiments presented in this thesis. Klaas has also gave me the opportunity to have his students as the participants of my experiments. I am very pleased to know that we are going to be working together more intensively, following my doctoral studies.

I am grateful for the valuable feedback provided by the members of my Ph.D. committee: Arie van Deursen, Jaco van de Pol, David Rosenblum, Per Runeson, and Roel Wieringa. Their feedback enabled me to dramatically improve this work. In addition, David Parnas welcomed me in Ireland, so that we could exchange ideas during the five days I enjoyed his and Lilian Parnas' hospitality.

While Bedir Tekinerdoğan was a visiting assistant professor at Bilkent University, I was a M.Sc. student there. Without Bedir's recommendation, I probably would not get the opportunity to work in the software engineering group at the University of Twente. Hereby, I express my gratitude for the faith Bedir had in me.

Pascal Dürr is my fellow Ph.D. colleague, who has been my 'life-saver' in many technical and practical issues related to my work and my life as a foreigner in The Netherlands. He helped me with literally any problem I faced during my doctoral studies. Pascal was also my close friend with whom I had long conversations about some aspects of life.

Remco van Engelen has created a stable environment for me to work at ASML, despite the rapidly changing priorities of this dynamic company. Niels van den Broek assisted me in several ways to develop the language and the algorithms explained in this thesis. Marco de Boer is the braveheart, who insisted to experiment with my prototypes (and eventually did it), although his managers resisted him due to other urgencies.

Magiel Bruntink and Tom Tourwé have been very generous to provide feedback on my work. Similarly, the members of the software engineering group have always been eager to reflect on my ongoing work, whenever I gave a presentation or asked them to read the draft versions of my papers. István Nagy, Ivan Kurtev, Hasan Sözer, Selim Çıracı, Arda Göknil, and Christian Hofmann have spent their precious time for reading my text. From the formal methods group, Harmen Kastenberg, Mariëlle Stoelinga, and Arend Rensik have also provided feedback on my work. Without the administrative support of Ellen Roberts-Tieke and Joke Lammerink, my Ph.D. life would have been a lot harder.

Aylin, my dearest! Your love gives me the strength to survive the most difficult times. Your presence defines the true meaning of my achievements...

Abstract

The development and maintenance of today's software systems is an increasingly effort-consuming and error-prone task. A major cause of the effort and errors is the lack of human-readable *and* formal documentation of software design. In practice, software design is often informally documented, or not documented at all. Therefore, (a) the design cannot be properly communicated between software engineers, (b) it cannot be automatically analyzed for finding and removing faults, (c) the conformance of an implementation to the design cannot be automatically verified, and (d) source code maintenance tasks have to be manually performed, although some of these tasks can be automated using formal documentation.

In this thesis, we address these problems for the design and documentation of the behavior implemented in procedural programs. We present the following solutions each addressing the respective problem stated above: (a) A graphical language called VisuaL, which enables engineers to specify constraints on the possible sequences of function calls from a given procedural program, (b) an algorithm called CheckDesign, which automatically verifies the consistency between multiple specifications written in VisuaL, (c) an algorithm called CheckSource, which automatically verifies the consistency between a given implementation and a corresponding specification written in VisuaL, and (d) an algorithm called TransformSource, which uses VisuaL specifications for automatically inserting additional source code at well-defined locations in existing source code.

Empirical evidence indicates that CheckSource is beneficial during some of the typical control-flow maintenance tasks: 60% effort reduction, and prevention of one error per 250 lines of source code. These results are statistically significant at the level 0,05. Moreover, the combination of CheckSource and TransformSource is beneficial during some of the typical control-flow maintenance tasks: 75% effort reduction, and prevention of one error per 140 lines of source code. These results are statistically significant at the level 0,01.

The main contribution of this thesis is the graphical language VisuaL with its formal underpinning *Deterministic Abstract Recognizers (DARs)*, which defines a new

family of formal languages called *Open Regular Languages (ORLs)*. The key feature of VisuaL is the *context-sensitive wildcard*, which makes VisuaL specifications more evolvable (i.e. less susceptible to changes), and more concise.

Contents

Chapter 1

Introduction

1.1 Problem Summary

The development and maintenance of today's software systems is an increasingly effort-consuming and error-prone task. A major cause of the effort and errors is the lack of precise, unambiguous, and human-readable documentation of software design. In today's industrial practice, software design is often imprecisely documented as texts in a natural language, or as diagrams without a well-defined structure and meaning. Consequently;

- **Problem 1**: The design cannot be properly communicated between software engineers.
- **Problem 2**: The design cannot be automatically analyzed for finding and removing faults.
- **Problem 3**: The conformance of an implementation to the design cannot be verified.
- **Problem 4**: Source code maintenance tasks have to be manually performed, although some of these tasks can be automated using formal documentation.

In this thesis, we address these problems for the design, documentation, and maintenance of algorithms [63] that are implemented in procedural programs such as C [58] functions. We present a solution that consists of four parts, each addressing one of the problems listed above. In addition, we report on the controlled experiments that we conducted for evaluating the solution. 71 subjects (23 professional software developers and 48 M.Sc. computer science students) participated in these experiments. The results of these experiments indicate that the solution can reduce the effort spent for some of the typical control-flow maintenance tasks by 75%, and pre-

vent one error per 140 lines of source code. These results are statistically significant at level 0,01.

The solution presented in this thesis is the outcome of our close collaboration with industry. In this collaboration, we conducted joint research with ASML [4], which is a company that produces semiconductor manufacturing machines. These machines are large-scale embedded systems, each having approximately 400 sensors, 300 actuators, 50 processors, and embedded software consisting of approximately 15 million lines of source code mostly written in C. More than 500 software engineers maintain and expand this software on a daily basis.

Our collaboration with ASML consisted of four phases: In the first phase, we surveyed the long-standing challenges faced by the software and system engineers of ASML. We interviewed the senior engineers, and collected nearly 30 challenges. Based on these challenges we formulated the four problems listed above, and identified a number of effort-consuming and error-prone tasks in ASML's software development and maintenance processes. In the second phase, we developed the solution to automate these tasks. In the third phase, we conducted controlled experiments to evaluate the solution. In the fourth and the final phase, ASML committed to conduct a transfer project to embed the solution into their software development and maintenance processes. In Section 1.2, we report on the first phase of our collaboration. The four problems listed above are generalized from the results of this phase.

1.2 Motivation

In the industrial practice, natural languages are frequently used for documenting the design of software. For instance, at ASML 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 diagrams that illustrate various facets of software design, such as the structure of data, flow of control, decomposition into (sub)modules, etc. These diagrams provide valuable intuition about the structure of software. However, typically such diagrams cannot be used as precise specifications of the actual software, since they are abstractions without a well-defined mapping to the final implementation in source code. Many of such diagrams do not have well-defined and precise semantics, either.

As we discuss in Sections 1.2.1 and 1.2.2, ambiguous and informal design documents

are a major cause of excessive manual effort and human errors during software development and maintenance.

1.2.1 Some Obstacles in Software Development

In Fig. 1.1, we illustrate a part of the software development process of ASML, showing four steps:

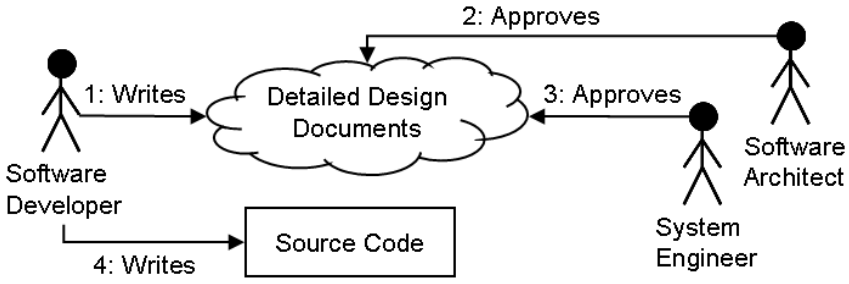

Figure 1.1: This figure shows part of the software development process at ASML.

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 usually 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); this indicates 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.

1.2.2 Some Obstacles in Software Maintenance

In Fig. 1.2, 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

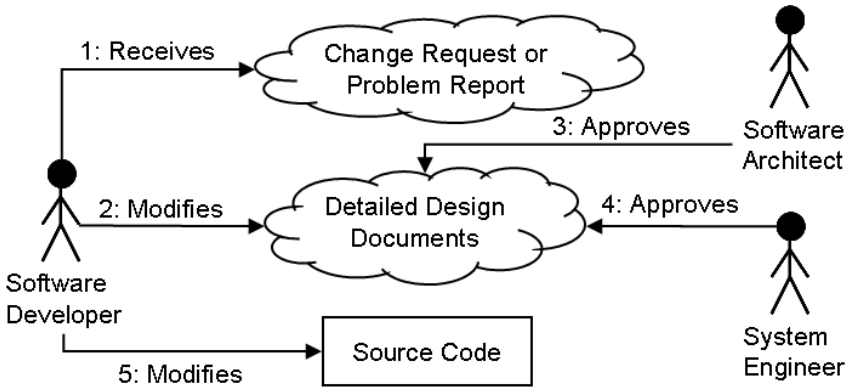

Figure 1.2: This figure shows part of the software maintenance process at ASML.

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.

In practice, developers may apply 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:

- While modifying the existing source code, developers typically take new decisions that has an impact on the design. These decisions remain undocumented.
- Since the new decisions remain undocumented, the source code eventually 'drifts away' from the design documents. More precisely, the design that is implemented in the source code becomes substantially different from 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 contain incorrect, incomplete, or misleading 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 1.2.1.
- 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. This suboptimal utilization also has a negative impact on the motivation for investing the time and energy for producing high-quality design documents.

1.3 Scope of this Thesis

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 motivation for this choice.

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 do not 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 possible sequences of operations are determined by the control flow structure of a function that calls the functions corresponding to the operations. Thus, the flow of control implemented in a function must satisfy the relevant temporal constraints.

During software maintenance, the engineers of ASML frequently change the control flow structure of functions, thereby unintentionally violating the temporal con-

straints. 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 inadequately documented, as explained in Section 1.2, and (b) there is no systematic way for engineers to tell whether the constraints are violated and where the constraints are violated. We have also observed that some of the control flow maintenance tasks could be automated if the temporal constraints were formally documented. In Chapter 6, we discuss these tasks in detail.

Based on these observations, we decided to find a better way to document the temporal constraints, and to develop algorithms that can help engineers in finding and repairing the defects. As a result, we developed a solution that consists of VisuaL, CheckDesign, CheckSource, and TransformSource.

1.4 Solution Approach

In this section, we explain how VisuaL, CheckDesign, and CheckSource can be used during software development and maintenance. The approach for using TransformSource is explained in Chapter 6.

1.4.1 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 Fig. 1.3, we illustrate the software design process, in which VisuaL and CheckDesign are used. This process consists of four steps: In the first step, a software developer specifies the temporal constraints, using VisuaL. Therefore, the resulting specifications are formal and unambiguous. A VisuaL specification is intended to be a part of a detailed design document, and such a document may contain multiple VisuaL specifications, as depicted in Fig. 1.3.

In the second step, CheckDesign automatically verifies the consistency between the specifications that apply to the same function. If the specifications 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

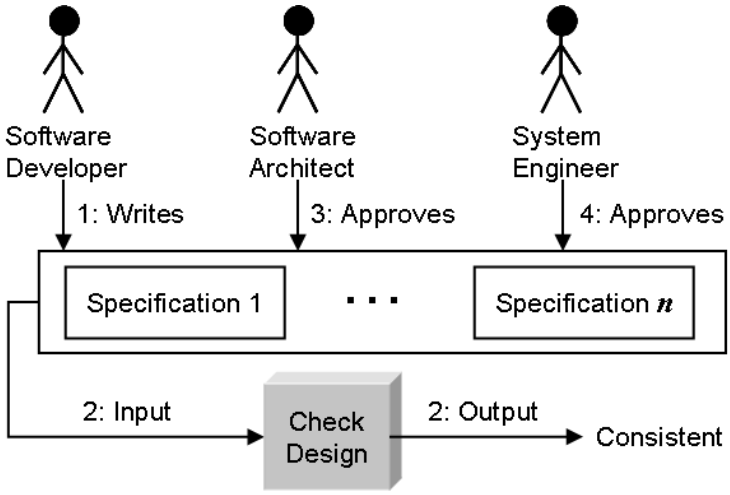

Figure 1.3: This figure shows the design process with VisuaL and CheckDesign.

Section 1.2.1), 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 VisuaL specifications, in the third and fourth steps. Thus, an important requirement is "The specifications written in VisuaL must be easily read and understood by people".

The Software Implementation Process

Fig. 1.4 shows the software implementation process in which the VisuaL specifications 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 specifications. If the source code is inconsistent with the specifications, 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:

- 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 the specifications, so she repairs the specifications and then performs the second, third,

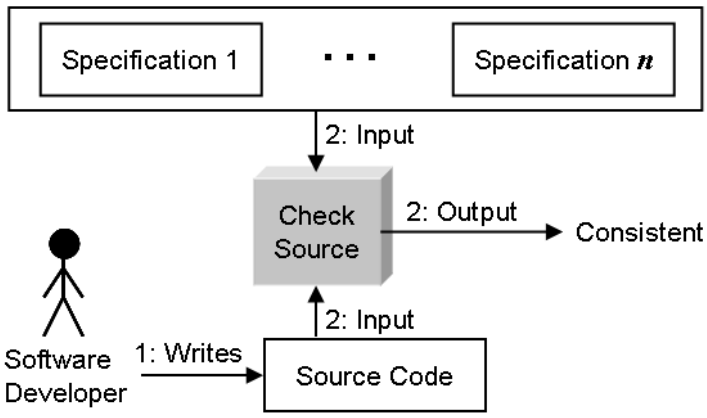

Figure 1.4: The implementation process with formal design documents and CheckSource.

and the fourth steps of the design process (see Fig. 1.3). After these steps, she reruns CheckSource.

- The developer decides that the inconsistency is due to the defects in both the specifications and the source code. So she repairs the specifications and then performs the second, third, and the fourth steps of the design process (see Fig. 1.3). After these steps, she repairs the source code and reruns CheckSource.

The design and implementation processes presented above address the problems listed in Section 1.2.1.

1.4.2 Adapting 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 specifications (i.e. 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 Fig. 1.4. If the developer decides that the change request has an impact on the specifications, then she realizes the change request by following the design process depicted in Fig. 1.3. Subsequently, she implements the change by following the implementation process depicted in Fig. 1.4. The maintenance process explained in this section addresses the problems listed in Section 1.2.2. In Section 1.5, we present a summary of the solution presented in this thesis. This summary is organized according to the structure of the thesis.

1.5 Summary of the Proposed Solution

The solution presented in this thesis consists of VisuaL, CheckDesign, CheckSource, and TransformSource. In this section, we summarize them one-by-one.

1.5.1 VisuaL

VisuaL is a graphical language that is intended for specifying design constraints on the behavior of algorithms. Such a constraint is a logical or temporal property that must be satisfied by each possible execution of the corresponding algorithm. Below, we present some examples of such constraints, expressed in English. Each of these constraints restrict the possible executions of an algorithm that is expressed as a C function:

- In each possible sequence of function calls from any given C function, the first function call must be a call to `traceIn`.
- In each possible sequence of function calls from `f`, there must eventually be a call to `g`.
- In each possible sequence of function calls from `f`, there must not be any call to `h` until a call to `g` is reached.
- In each possible sequence of function calls from any given function, the last function call must be call to `traceOut`.

A VisuaL specification consists of labelled *rectangles* and labelled *arrows* that visualize a *pattern*. To see some examples of VisuaL specifications, the readers can browse the figures in Section 2.2. Each VisuaL specification may contain *context-sensitive wildcards* (denoted by the `$` symbol). Context-sensitive wildcards are used for making VisuaL specifications more evolvable (i.e. less susceptible to changes) and more concise, as we explain in Section 2.8.

A VisuaL specification represents a *deterministic abstract recognizer (DAR)*, which is a variant of a deterministic finite-state automaton (DFA) [63]. The only difference between a DAR and a DFA is as follows: A DFA accepts or rejects finite sequences of symbols from a *predefined* and *finite* set of symbols, whereas a DAR accepts or rejects finite sequences of symbols from *the set of all symbols*, which is obviously an *infinite* and *'open-ended'* set. Since DARs are not specific to a predefined and finite set of symbols, DARs are '*abstract*'. Due to this fundamental difference between DARs and DFA, DARs define a new family of formal languages, which we call *open regular languages (ORLs)*:

- The set of regular languages [63] is a proper subset of the set of ORLs.

- There are ORLs that are not in the set of context-free languages (CFL) [63].
- There are CFLs that are not in the set of ORLs.
- ORLs are closed under the basic set operations complement, union, and intersection.
- ORLs are closed under the computation-theoretic operations string concatenation and Kleene closure [63].

Using VisuaL, one can express any ORL, and nothing else:

- Each VisuaL specification represents a specific DAR, and the DAR can systematically be constructed in polynomial time, based on the VisuaL specification.
- Each DAR is represented by a particular VisuaL specification, and the VisuaL specification can systematically be constructed in polynomial time, based on the DAR.

New VisuaL specifications can be systematically constructed in polynomial time, by composing existing specifications using boolean operators *not*, *or*, and *and*; and temporal operators *next*, *repeatedly*, *eventually*, *until*, and *release*.

A VisuaL specification is more concise than the DAR represented by the specification. Furthermore, a VisuaL specification can systematically be transformed in polynomial time to a particular VisuaL specification that (a) has minimal number of graphical elements, and (b) represents the same ORL as the original specification represents.

Each of the example constraints presented at the beginning of this section contains a temporal property that has to be satisfied by each possible path in the control-flow [38] of a given C function. Since linear-time temporal logic (LTL) [23] is also a language for expressing similar temporal properties, one may think that VisuaL is indifferent than LTL. However, despite the similarity, VisuaL is fundamentally different than both LTL and any other model checking formalism [23]. Using LTL or any other model checking formalism, one specifies constraints (i.e. properties, requirements) that are either satisfied or dissatisfied by *infinite* sequences (of function calls, execution states, etc.). Therefore, LTL is not intended for specifying "In each possible sequence of function calls from any function, the last function call must be call to `traceOut`". In contrast, using VisuaL, one specifies constraints (i.e. properties, requirements) that are either satisfied or dissatisfied by *finite* sequences of function calls.

Graphical languages such as UML activity diagrams [8] or flowcharts [71] are frequently used for designing the flow of control within procedural programs such as C functions. Although VisuaL specifications are also graphical artifacts of behavioral design, they are fundamentally different than activity diagrams. An activity

diagram is a control-flow *model* [74] of a function (or procedure, method, subroutine); different functions that implement the same activity diagram have the same control-flow. Whereas, a VisuaL specification is a *constraint* (i.e. formally specified requirement) on the control-flow of a function; different implementations that conform to a VisuaL specification may have different control-flow. Thus, VisuaL specifications are typically more abstract than activity diagrams: a VisuaL specification is a constraint on not only the implementation of a procedure but also the activity diagram that is the control-flow model of the procedure.

VisuaL addresses the first problem stated in Section 1.1, and enables us to address the remaining three problems, as we explain in the upcoming sections. In Chapter 2, VisuaL is presented in detail. The composition operators over VisuaL specifications are presented in Chapter 4.

1.5.2 CheckDesign

CheckDesign is an algorithm for checking the consistency of VisuaL specifications, as we briefly explain below.

Using VisuaL, one can create multiple specifications each representing a different constraint on the same function. When such specifications are created, it must be ensured that the specifications are *consistent*: There is at least one possible control-flow of the function, such that the control-flow satisfies each of the constraints. If there is no possible control-flow of the function that satisfies each of the constraints, then the VisuaL specifications are *inconsistent*.

Whenever VisuaL specifications are created or modified in the software life cycle, the consistency between the specifications must be verified. Manually verifying the consistency is an effort-consuming and error-prone task. If the specifications are inconsistent, then manually finding and resolving the inconsistency is an effort-consuming and error-prone task, too. CheckDesign can reduce the effort and automatically detect the errors: CheckDesign takes a set of VisuaL specifications as input, and automatically finds out, in polynomial time, whether the specifications are consistent or not. If the specifications are inconsistent, then CheckDesign outputs an error message that can help in understanding and resolving the inconsistency. Hence, CheckDesign addresses the second problem stated in Section 1.1. In Chapter 3, CheckDesign is presented in detail.

1.5.3 CheckSource

CheckSource is an algorithm for checking the consistency between VisuaL specifications and source code, as we briefly explain below.

After creating consistent VisuaL specifications, a developer typically writes source code to implement the function corresponding to the specifications. A function and a corresponding specification may be inconsistent with each other. Manually finding and resolving an inconsistency between a function and a specification is an effort-consuming and error-prone task. CheckSource can reduce effort and detect errors. CheckSource takes a function and a corresponding VisuaL specification as the input, and automatically finds out, in polynomial time, whether the function and specification are consistent or not. To determine if a specification and a function are consistent, CheckSource first parses the function and creates an abstract syntax tree. Second, CheckSource derives the control-flow graph of the function by traversing the abstract syntax tree. Finally, CheckSource finds out whether each possible path in the control flow graph satisfies the constraint expressed in the VisuaL specification. If there is at least one possible path that does not satisfy the constraint, then the function and the specification are inconsistent. If there is an inconsistency, CheckSource outputs an error message containing an example path that does not satisfy the constraint. This error message helps in understanding and resolving the inconsistency. In this way, CheckSource addresses the third problem stated in Section 1.1. In Chapter 5, we first present CheckSource, and then we report on the controlled experiment that we conducted for evaluating CheckSource.

1.5.4 TransformSource

TransformSource is an algorithm for inserting additional source code at well-defined locations in given source code. In this section, we briefly explain TransformSource.

Let us consider the following constraint: "In each possible sequence of function calls from `f`, each call to `g` must be immediately followed by a call to `h`, and there must be no call to `h` that is not preceded by a call to `g`". According to this constraint, whenever a new call to `g` is added to the body of `f`, it is necessary to insert a new call to `h` as the next function call. If this constraint is formally specified, the insertion of the calls to `h` can be automated.

To enable the automation, we extended VisuaL such that each VisuaL specification represents a *deterministic abstract transducer (DAT)*, which is a variant of a Moore machine [54]. As a result of this extension, a VisuaL specification (e.g. the specification of the example constraint above) is capable of translating an input sequence

(e.g. <a, g, b, g>) into an output sequence (e.g. <a, g, h, b, g, h>) that satisfies the constraint represented by the specification. Based on such a VisuaL specification, TransformSource automatically inserts the additional calls (e.g. the calls to h) into the body of a function (e.g. f), so that the function satisfies the constraint.

Since the additional calls are automatically inserted by TransformSource, developers can work with the functions that do not contain the additional calls. Whenever such a function is modified due to maintenance, TransformSource can automatically reinsert the additional calls at the necessary places in the source code of the function, in which case the consistency between the function and the specification is always automatically ensured. In this way, TransformSource addresses the fourth and the final problem stated in Section 1.1. In Chapter 6, we first present a real-life problem in an industrial context, and show how this problem is solved using VisuaL, CheckSource, and TransformSource, in combination. In Chapter 7, we report on the controlled experiment we conducted for evaluating the combination of CheckSource and TransformSource. This combination exhibits some of the fundamental characteristics of a weaver [39] in aspect-oriented programming.

1.6 An Overview of this Thesis

In Fig. 1.5, we present an overview of this thesis.

In Chapter 2, we first present an overview of VisuaL by examples, and then define the notation, syntax, and semantics of VisuaL. In addition, we define both the underlying formalism DARs, and the new family of formal languages ORLs that DARs express. Finally, we discuss the expressive power of VisuaL, and provide an algorithm for reducing the size of VisuaL specifications without changing their semantics.

In Chapter 3, we explain how to detect possible inconsistencies among multiple VisuaL specifications (i.e. CheckDesign). In addition, we explain how to locate and report such inconsistencies.

In Chapter 4, we investigate some of the closure properties of ORLs, and based on these properties we define operators for composing new VisuaL specifications from existing ones.

In Chapter 5, we present CheckSource, and then we report on the controlled experiments we conducted for evaluating CheckSource.

In Chapter 6, we first present a real-life problem in an industrial context, and show how this problem is solved using VisuaL, CheckSource, and TransformSource, in

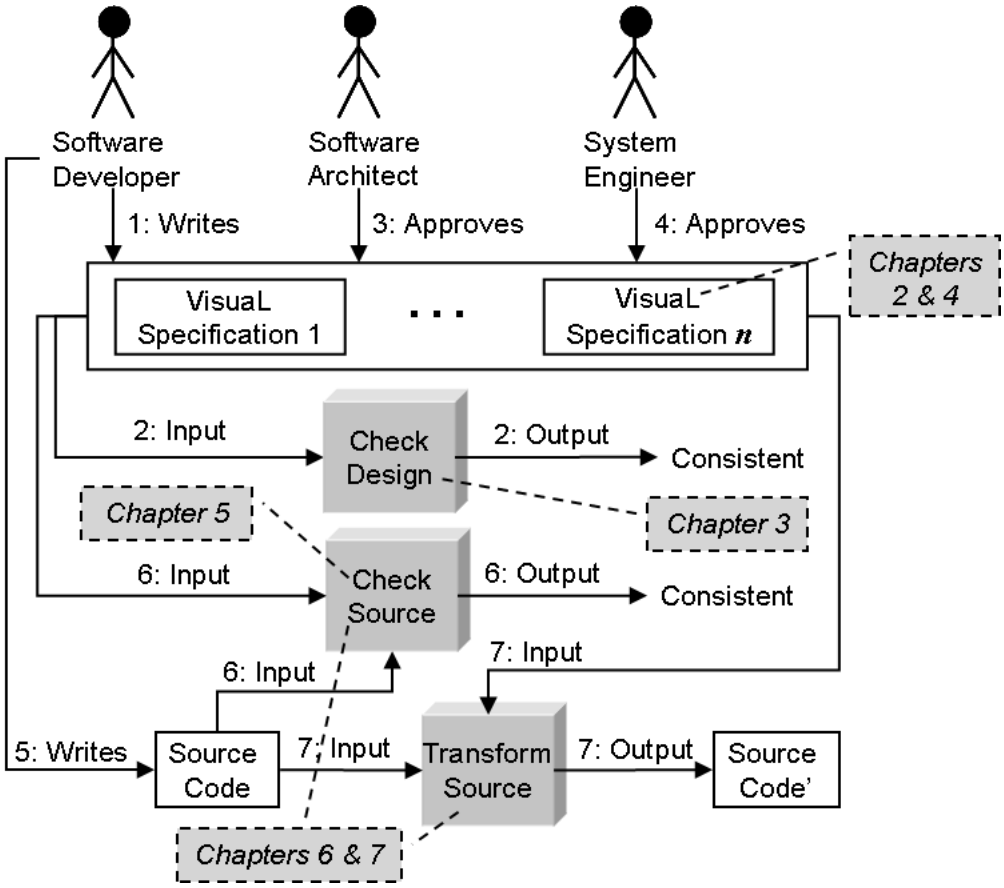

Figure 1.5: An Overview of this thesis.

combination. In this chapter we present TransformSource, and an extended version of VisuaL and CheckSource.

In Chapter 7, we report on the controlled experiments we conducted both with professional developers and with M.Sc. students for evaluating the combination of CheckSource and TransformSource.

Chapter 8 contains the related work, discussion, and conclusions.

1.7 Contributions of this Thesis

VisuaL, whose key feature is *context-sensitive wildcard*, is the key contribution of this thesis. The purpose of context-sensitive wildcards is to make VisuaL specifications more evolvable (i.e. less susceptible to changes), and more concise. VisuaL addresses

the requirements specification problem stated by Hatcliff and Dwyer [51]:

- "**The requirement specification problem**: the difficulty of expressing software requirements in the temporal specification languages of the existing model-checking tools. Although model-checker property specification languages are built on the theoretically elegant temporal logics, practitioners and even researchers find it difficult to use them to accurately express complex event-sequencing properties. Once written, the specifications are often hard to read and debug." [51]

The algorithms of CheckSource, CheckDesign, and TransformSource are additional contributions of this thesis. CheckSource addresses the model construction problem and the output interpretation problem stated by Hatcliff and Dwyer [51]:

- "**The model construction problem:** bridging the semantic gap between the artifacts produced by current verification tools. Most development is done using general-purpose programming languages (e.g. C, C++, Java, Ada), but most verification tools accept specification languages designed for the simplicity of their semantics (e.g. process algebras, state machines). In order to use a verification tool on a real program, a developer must extract an abstract mathematical model of the program's salient behavior and specify this model in the input language of the verification tool. This process is both effort-consuming and error-prone." [51]
- "**The output interpretation problem:** When a property fails when checking large models (and software systems typically produce very large models), the counter example traces produced by the checker can be hundreds even thousands of steps long. Manually matching up these counter examples is extremely tedious for several reasons. First, the length is quite long and it may require hours to walk through the trace. Second, the error trace is expressed in terms of the low-level, possibly highly optimized model representations ... Typically, one step in the source program may correspond to as many as ten steps in the low-level model representation." [51]

In this thesis, we provide empirical evidence indicating that VisuaL can be used by professional developers and M.Sc. students to debug source code, and CheckSource and TransformSource can save effort and reduce errors during the debugging. These empirical results are contributions of this thesis, too.

Chapter 2

VisuaL

2.1 Introduction

New generations of large-scale and complex embedded systems such as wafer scanners [4], medical MRI[1] scanners, and electron microscopes are rarely developed from scratch [81]. Instead, engineers continuously modify older generations to develop new ones. Therefore, evolvability is one of the key quality factors that determine the commercial success (or failure) of large-scale and complex embedded systems.

In the Ideals project [81], we investigated the evolvability of the wafer scanner software, and discovered that engineers spend excessive effort to keep the behavior specifications consistent with the evolving source code. We have seen that engineers cannot express the behavioral design as abstractly as they intend to, because the abstraction mechanisms offered by the commonly used graphical languages (e.g statecharts [47]) are not always sufficient to achieve the intended level of abstraction. Consequently, the specifications contain excessive details about the implementation, and these details increase (a) the coupling between the specifications and source code, and (b) the size and complexity of the specifications. Due to the high coupling, the specifications need to be frequently updated during the evolution of the source code; and due to large and complex specifications, excessive effort has to be spent for each update.

According to a survey [84] of software specification methods and techniques, the existing graphical languages support hierarchies (i.e. nested structures), so that one can define different levels of abstraction. Using statecharts [47] for instance, one can abstract from a set of states, by defining a super state that stands for this set.

[1] Magnetic Resonance Imaging

In this chapter, we present an additional mechanism for abstraction, which we call *Context-Sensitive Wildcard* (*CSW*). Intuitively, a CSW is a transition that stands for an infinite set of transitions, such that the elements of this set is determined by the 'context' of the CSW. In this chapter, we define CSW as the key feature of a simple graphical language, which we call *VisuaL*. We provide a detailed analysis of VisuaL, such that this analysis reveals the theoretical and practical implications of using CSWs, in the graphical specifications of software behavior.

VisuaL is intended for expressing constraints on the behavior of algorithms. Such a constraint is a logical or temporal property that must be satisfied by each possible execution of the corresponding algorithm. A VisuaL specification represents a *deterministic abstract recognizer* (*DAR*), which is a variant of a Deterministic Finite Accepter (DFA) [63]. The key difference between a DFA and a DAR is as follows: A DFA with an alphabet Σ either accepts or rejects a finite sequence of symbols, provided that each symbol of the sequence is an element of Σ; whereas a DAR either accepts or rejects *any* finite sequence of symbols. The difference between DFAs and DARs is formally explained in Section 2.3.4.

Although VisuaL is a language for expressing the properties of algorithms, it is possible to extend VisuaL for expressing the properties of reactive systems [46], too. In Section 8.1.3, we discuss how VisuaL could be extended, such that a VisuaL specification represents a variant of a Büchi automaton [23]. In Section 8.1.3, we also explain why we think that a recent implementation of the LTSA model checker [42] already has a suitable foundation for supporting an extended version of VisuaL.

Hatcliff and Dwyer [51] indicate that one of the major problems that are currently preventing the successful application of model checking technology to software is "the requirement specification problem: the difficulty of expressing software requirements in the temporal specification languages of the existing model-checking tools. Although model-checker property specification languages are built on the theoretically elegant temporal logics, practitioners and even researchers find it difficult to use them to accurately express complex event-sequencing properties. Once written, the specifications are often hard to read and debug" [51]. Empirical evidence (Chapters 5 and 7) indicates that VisuaL has the potential to solve "the requirement specification problem". We conducted controlled experiments where 24 professional software engineers and 49 M.Sc. computer science students used industrial VisuaL specifications for finding and repairing realistic defects in industrial C code. Since the participants did not have any previous experience with VisuaL, they were given a 15-minute tutorial of the VisuaL language. After this tutorial, the participants could efficiently use the VisuaL specifications and a model checker tool (i.e. CheckSource) for finding and successfully repairing the defects in the source code.

The idea of using wildcards in state-transition diagrams is not new, as we discuss in

Section 8.1.4. To our best knowledge however, CSW has not been offered as a feature of a graphical language, and the theoretical and practical implications of using CSWs were not investigated. Therefore, the investigation we provide throughout Sections 2.3-2.8, and the conclusions drawn from this investigation can be seen as the contribution of this chapter.

In Section 2.2, we provide an intuitive overview of VisuaL. Next, we formally define VisuaL, in Section 2.3. Throughout Sections 2.4-2.6, we analyze VisuaL from a theoretical perspective; and in Sections 2.7 and 2.8, we analyze VisuaL from an engineering perspective. The remaining sections contain the related work, future work, and conclusions.

2.2 An Overview of VisuaL by Examples

In this section, we intuitively explain VisuaL, by presenting the specifications of three example constraints. Each of these constraints restrict the possible executions of an algorithm that is expressed as a C function. These constraints are simple examples that demonstrate the basic features of VisuaL. The notation, syntax, and semantics of VisuaL are provided in Section 2.3.

2.2.1 Example 1: "At Least One"

The VisuaL specification shown in Fig. 2.1 is a formal 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`.

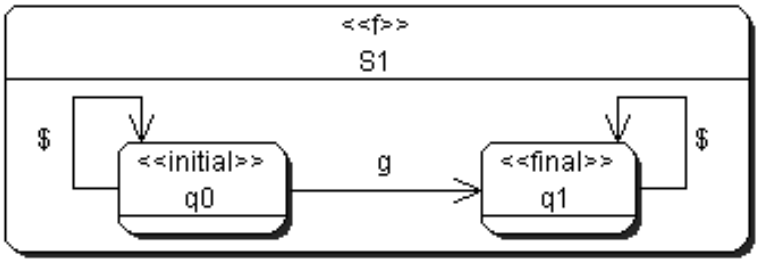

Figure 2.1: An example VisuaL specification demonstrating the usage of "at least one".

In this section, we first explain the syntactic elements of the VisuaL specification

shown in Fig. 2.1, and the semantics of these elements. Subsequently, we discuss why Fig. 2.1 is a specification of the constraint C1 stated above.

Syntactic Elements and Their Semantics

The rounded rectangle with the stereotype <<f>> is called **container node**, which defines a view on the flow of control (to be) implemented within the body of the function `f`. In the stereotype of a container node one can also write a regular expression that matches the identifiers of multiple functions. In such a case, the container node defines a common view on multiple functions.

The label `S1` is the name (i.e. identifier) of both the container node and the specification. Inside the container node, there is a structure consisting of (a) arrows called **edges**, and (b) rounded rectangles called **nodes**. Such a structure is called **pattern**. The edges represent the function calls from `f`, and the nodes (e.g. the rounded rectangle with the label `q0`) represent locations on the control flow of `f`.

The stereotype <<f>> means "each possible sequence of function calls from the function `f` must be *matched by the pattern*[2], otherwise `f` does not satisfy the constraint represented by the specification".

The node `q0` represents the beginning of a given sequence of function calls, because it has the stereotype <<initial>>. Such a node is called **initial node**. There is exactly one initial node in each VisuaL specification.

The `$`-labelled edge 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 edge originating from the same node (i.e. `q0`). In VisuaL, no two edges originating from the same node have the same label; therefore VisuaL specifications are deterministic.

In general, a `$`-labelled edge matches a function call, if and only if this call cannot be matched by the other edges *originating from the same node*. That is, the matching of a `$`-labelled edge is 'sensitive' to the other edges originating from the same node. Therefore, a `$`-labelled edge e is a **Context-Sensitive Wildcard (CSW)**, where the context is the set of labels of the other edges whose source node is the same as the source node of e.

Note the difference between the CSW pointing to `q0` and the CSW pointing to `q1`: the former CSW can match a call to any function except `g`, whereas the latter CSW can match a call to any function (i.e. including `g`), since `q1` does not have any other

[2]We precisely define the matching later in this section.

outgoing edge.

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 edge labelled 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 edge that points to q1.

If there are additional calls after the first call to g, then each of these calls is matched by the CSW pointing to q1, hence the sequence eventually terminates[3] at q1.

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

S1 is a specification of C1

We can assert that S1 (Fig. 2.1) is a specification of C1 (see the beginning of Section 2.2.1), if and only if the following two requirements are fulfilled: (1) If a given sequence of function calls contains no call to g, then this sequence must not be matched by the pattern shown in Fig. 2.1. (2) If a given sequence of function calls contains *at least one* call to g, then this sequence must be matched by the pattern shown in Fig. 2.1. Below, we show that these requirements are indeed fulfilled.

Let *seq* be a finite sequence of function calls, such that *seq* contains no call to g. In this case, each call in *seq* is matched by the CSW originating from q0. Thus, *seq* eventually terminates at q0. Since q0 is not a final node, *seq* is not matched by the pattern shown in Fig. 2.1.

Let *seq* be a finite sequence of function calls, such that *seq* contains *at least one* call to g. In this case, each function call from the beginning of *seq* until the first call to g is matched by the CSW originating from q0. The first call to g is matched by the g-labelled edge, upon which *seq* reaches q1. Now, there are two cases to consider: (1) If *seq* does not contain any other call after the first call to g, then *seq* terminates at the final node q1. Thus, *seq* is matched by the pattern shown in Fig. 2.1. (2) If *seq* contains additional calls after the first call to g, then each of these calls is matched by the CSW originating from q1. Consequently, *seq* eventually terminates at the final node q1, which means *seq* is matched by the pattern shown in Fig. 2.1.

[3]Infinite sequences of function calls are out of the scope of this thesis, because VisuaL is *not* a language for specifying constraints on the execution of possibly non-terminating programs.

2.2.2 Example 2: "Immediately Followed By"

Fig. 2.2 shows a specification of the following constraint:

C2: In each possible sequence of function calls from f, if there is at least one call to g, then the first call to g must be *immediately followed by* a call to h.

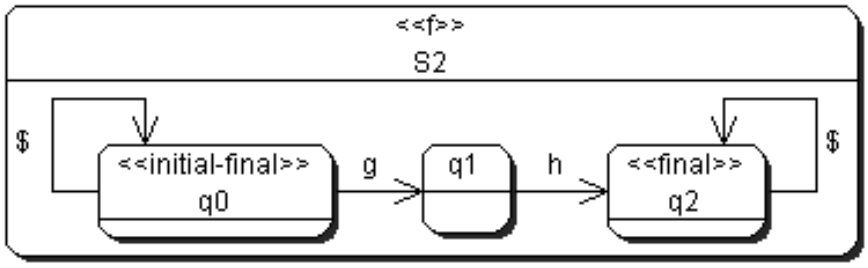

Figure 2.2: An example specification demonstrating the usage of "immediately followed by".

Syntactic Elements and Their Semantics

In Fig. 2.2, the stereotype <<initial-final>> means that q0 has both the <<initial>> and <<final>> stereotypes. Therefore, q0 is both an initial and a final node. We call such a node **initial-final node**. In Fig. 2.2, the node q1 does not have any stereotype. Therefore, we call such a node **plain node**. A plain node is neither the initial nor a final node. The other syntactic elements and their semantics are already explained in Section 2.2.1.

S2 is a specification of C2

We can assert that S2 (Fig. 2.2) is a specification of C2 (see the beginning of Section 2.2.2), if and only if the following three requirements are fulfilled: (1) If a given sequence of function calls contains no call to g, then this sequence must be matched by the pattern shown in Fig. 2.2. (2) If a given sequence of function calls contains at least one call to g, and the first call to g is not *immediately followed by* a call to h, then this sequence must not be matched by the pattern shown in Fig. 2.2. (3) If a given sequence of function calls contains at least one call to g, and the first call to g is *immediately followed by* a call to h, then this sequence must be matched by the pattern shown in Fig. 2.2. Below, we show that these requirements are indeed fulfilled.

Let *seq* be a finite sequence of function calls, such that there is no call to g in *seq*. In this case, each call in *seq* is matched by the CSW originating from q0 (see Fig. 2.2); hence *seq* terminates at q0, and is matched by the pattern, because q0 is the initial-final node, which is a final node.

Let *seq* be a finite sequence of function calls, such that there is at least one call to g in *seq*, and the first call to g is not *immediately followed by* a call to h. In this case, the first call to g is matched by the g-labelled edge (see Fig. 2.2), upon which *seq* reaches q1. Now, *seq* cannot be matched by the pattern, because (a) q1 is a non-final node, (b) the only outgoing edge from q1 is the h-labelled edge, and (c) the first call to g is not *immediately followed by* a call to h.

Let *seq* be a finite sequence of function calls, such that there is at least one call to g in *seq*, and the first call to g is *immediately followed by* a call to h. In this case, *seq* reaches q2 upon encountering the call to h that *immediately follows* the first call to g. Now, there are two cases to consider: (1) If this call to h is the last call in *seq*, then *seq* is matched by the pattern, because q2 is a final node. (2) If this call to h is not the last call in *seq*, then each of the subsequent calls is matched by the CSW originating from q2. Hence, *seq* eventually terminates at the final node q2, in which case *seq* is matched by the pattern.

2.2.3 Example 3: "Not"

Fig. 2.3 shows a specification of the following constraint:

C3: In each possible sequence of function calls from f, a call to g must *not* exist.

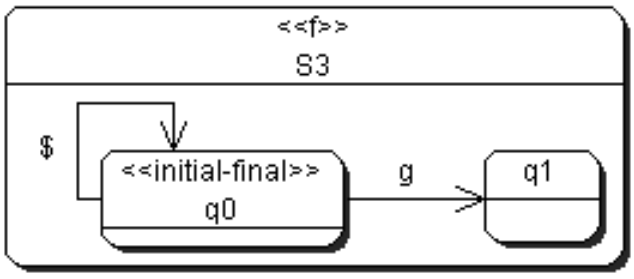

Figure 2.3: An example specification demonstrating the usage of "not".

Syntactic Elements and Their Semantics

In Fig. 2.3, q1 does not have the stereotype <<final>>, and no edge originates from q1. We call such a node **trap node**. For a given sequence *seq* of function calls, if a call *c* in *seq* is matched by an edge pointing to a trap node *tr*, then either of the following scenarios occur:

- *c* is the last call in *seq* (i.e. *seq* terminates at *tr*). 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, there is no edge that can match the remaining calls in *seq*. Therefore, *seq* is not matched by the pattern.

To sum up, if a sequence 'visits' a trap node, then the sequence is not matched by the pattern. The other syntactic elements and their semantics are already explained in Sections 2.2.1 and 2.2.2.

S3 is a specification of C3

We can assert that S3 (Fig. 2.3) is a specification of C3 (see the beginning of Section 2.2.3), if and only if the following two requirements are fulfilled: (1) If a given sequence of function calls does *not* contain any call to g, then this sequence must be matched by the pattern shown in Fig. 2.3. (2) If a given sequence of function calls contains at least one call to g, then this sequence must not be matched by the pattern shown in Fig. 2.3. Below, we show that these requirements are indeed fulfilled.

Let *seq* be a finite sequence of function calls, such that *seq* does *not* contain any call to g. In this case, each call in *seq* is matched by the CSW originating from q0. Thus, *seq* eventually terminates at q0. Since q0 is a final node, *seq* is matched by the pattern shown in Fig. 2.3.

Let *seq* be a finite sequence of function calls, such that *seq* contains *at least one* call to g. In this case, each function call from the beginning of *seq* until the first call to g is matched by the CSW originating from q0. The first call to g is matched by the g-labelled edge, upon which *seq* reaches q1. Since q1 is a trap node, *seq* is not matched by the pattern shown in Fig. 2.3.

2.2.4 Example 4: "And"

Fig. 2.4 shows a specification of the following constraint:

C4: 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`.

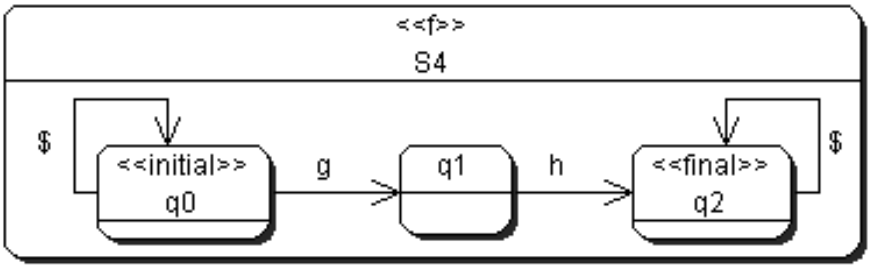

Figure 2.4: An example specification demonstrating the usage of "and".

Syntactic Elements and Their Semantics

The syntactic elements shown in Fig. 2.4, and their semantics are already explained in Section 2.2.1.

S4 is a specification of C4

Note that C4 (see the beginning of Section 2.2.4) is "C1 *and* C2", i.e. an implementation of `f` satisfies C4, if and only if the implementation satisfies both C1 and C2. We can assert that S4 (Fig. 2.4) is a specification of C4, if and only if the pattern shown in Fig. 2.4 fulfills the following four requirements: the first and the second requirements stated in Section 2.2.1, and the second and the third requirements stated in Section 2.2.2. Due to the "*and*" in C4, the first requirement stated in Section 2.2.2 is overridden by the first requirement stated in Section 2.2.1.

In Section 2.2.1, we explained that S1 fulfils the first and the second requirements stated in that section, and in Section 2.2.2 we explained that S2 fulfils the second and the third requirement stated in that section. These explanations can be reused for showing that S4 indeed fulfils the four requirements.

C4 hints on the conjunction (i.e. "*and*") operator over VisuaL specifications. There are other operators, as well. The operators can be used for deriving new specifications from existing ones (e.g. deriving S4 from S1 and S2). In Section 4.3, we precisely explain these operators, and how each operator can be applied to compose VisuaL specifications.

2.3 Notation, Syntax, and Semantics of VisuaL

In this section, we precisely define VisuaL, by presenting its notation, syntax, and semantics.

2.3.1 Notation of VisuaL

In Fig. 2.5, the notational elements of VisuaL are depicted as numbered images. The first five elements are **node**s, and the last two elements are **edge**s. To explain

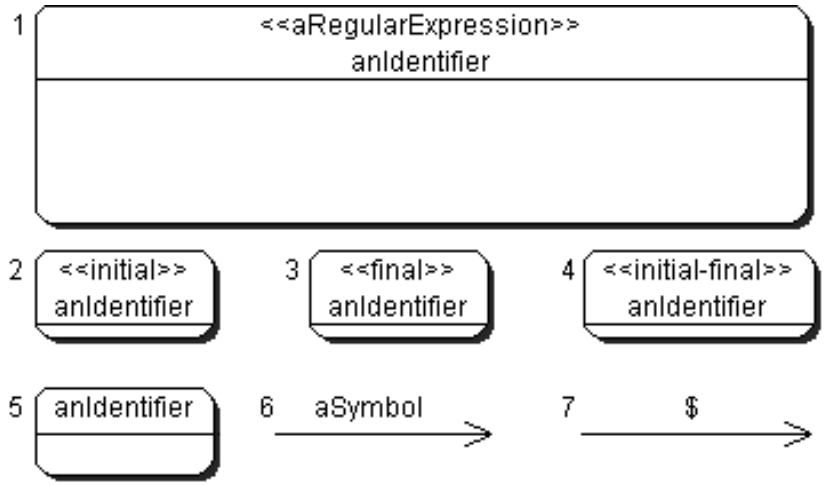

Figure 2.5: The elements of the notation of VisuaL.

these elements, we use the terms "alphabet" and "string" defined in [63], as follows: A finite and non-empty set of symbols is called **alphabet**. A finite sequence of symbols from an alphabet is called **string**.

A **VisuaL identifier** is a string consisting of symbols from $\{c|c$ is an uppercase or lowercase letter in the English alphabet$\} \cup \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

In Fig. 2.5, the first element is a rounded rectangle with the stereotype <<aRegular-Expression>>. This element is called **container node**. aRegularExpression is the place holder of a regular expression [63] that matches the identifiers of a set of C functions. anIdentifier is the placeholder of a VisuaL identifier that is the name of the container node. An example of a container node is S4 in Fig. 2.4.

The second element, which is a rounded rectangle with the stereotype <<initial>>, is called **initial node**. anIdentifier is the placeholder of a VisuaL identifier that is the name of the initial node. An example of an initial node is q0 in Fig. 2.4.

The third element, which is a rounded rectangle with the stereotype <<final>>, is called **final node**. anIdentifier is the placeholder of a VisuaL identifier that is the name of the final node. An example of a final node is q2 in Fig. 2.4.

The fourth element, which is a rounded rectangle with the stereotype <<initial-final>>, is called **initial-final node**. anIdentifier is the placeholder of a VisuaL identifier that is the name of the initial-final node. An example of an initial-final node is q0 in Fig. 2.3.

The fifth element, which is a rounded rectangle without any stereotype, is called **plain node**. anIdentifier is the placeholder of a VisuaL identifier that is the name of the plain node. An example of a plain node is q1 in Fig. 2.4.

If a given node n is an initial node, initial-final node, final node, or plain node, then n is generally called **inner node** (i.e. a node that is inside a container node).

The sixth element, which is an arrow with the label aSymbol, is called **edge**. aSymbol is the placeholder of a symbol. In Fig. 2.4, an example of an edge is the arrow with the label g.

The seventh element is called **Context-Sensitive Wildcard (CSW)**: A CSW is an edge whose label is the $ symbol. In Fig. 2.4, there are two CSWs.

initial, initial-final, final, and $ are the reserved words [79] of VisuaL. Each of these reserved words has a mathematical meaning defined in Section 2.3.5.

2.3.2 Syntax of VisuaL

A VisuaL specification has one container node. Inside the container node, (a) there is either one initial node or one initial-final node, (b) there are zero or more final nodes, and (c) there are zero or more plain nodes.

Inside a container node, there are zero or more edges. Each edge has a source and target, which are inner nodes. Each edge has a label, and no two edges have both the same source node and the same label.

2.3.3 Deterministic Finite Accepter (DFA)

To precisely define the semantics of VisuaL, we introduce a new formalism called *deterministic abstract recognizer (DAR)*, in Section 2.3.4. A *DAR* is a variant of a *deterministic finite accepter (DFA)* defined in [63]. In this section, we provide this definition of *DFA*, which we use in this thesis.

A **DFA** M is a quintuple $\langle Q, \Sigma, \delta, q_0, F \rangle$, where

- Q is a finite set of **states**.
- Σ is a finite and non-empty set of symbols called **input alphabet**.
- $\delta : Q \times \Sigma \to Q$ is a total function called the **transition function**.
- $q_0 \in Q$ is the **initial state**.
- $F \subseteq Q$ is a set of **final states**.

To explain how M accepts or rejects a given string, we use the following terms: λ denotes the **empty string** (i.e. the string that contains no symbol). If w and x are strings, then wx denotes the string obtained by concatenating w and x. Σ^* denotes the set of strings obtained by concatenating zero or more symbols from Σ.

Let $q \in Q$, $a \in \Sigma$, and $w \in \Sigma^*$. The function $\delta^* : Q \times \Sigma^* \to Q$ is called **extended transition function**, which is recursively defined as follows: $\delta^*(q, \lambda) = q$, and $\delta^*(q, wa) = \delta(\delta^*(q, w), a)$. M **accepts** w if and only if $\delta^*(q_0, w) \in F$. M **rejects** w if and only if $\delta^*(q_0, w) \notin F$.

$L(M) = \{w \in \Sigma^* | \delta^*(q_0, w) \in F\}$ is the **language** of M. A set L of strings is a **Regular Language (RL)**, if and only if there is a DFA M such that $L(M) = L$.

2.3.4 Deterministic Abstract Recognizer (DAR)

In this section, we introduce a new formalism called *deterministic abstract recognizer* (*DAR*), which we use for defining the semantics of VisuaL, in Section 2.3.5. A *DAR* is a variant of a DFA. The key difference between a DFA and a *DAR* is the following: A DFA with an alphabet Σ either accepts or rejects a finite sequence of symbols, provided that the sequence is an element of Σ^*; whereas a *DAR* either accepts or rejects *any* finite sequence of symbols. To precisely explain this difference, we first need to formally define *DAR*:

A **DAR** M is a septuple $\langle Q, \Sigma_a, \delta, q_0, F, \Xi, \eta \rangle$, where

- $Q = \Omega \cup \{q_t\}$ is a finite set of **states**, where Ω is the set of **user-defined states**, q_t is the **default trap state**, and $q_t \notin \Omega$.
- $\Sigma_a = \Sigma_b \cup \{\#\}$ is the **abstract input alphabet**, where Σ_b is a finite set of symbols such that $\# \notin \Sigma_b$. Σ_b is called the **base input alphabet**. $\#$ is a reserved symbol that will be explained in this section.
- $\delta : Q \times \Sigma_a \to Q$ is a total function called **transition function**. $\forall a \in \Sigma_a (\delta(q_t, a) = q_t)$.
- $q_0 \in \Omega$ is the **initial state**.
- $F \subseteq \Omega$ is a set of **final states**.

- Ξ is a regular expression called **scope expression** that matches a set of strings. This set is the **scope** of M.
- η is the **name** of M. The name of M is a VisuaL identifier.

To explain how M accepts or rejects a given sequence of symbols, we use the following terms: The set of all possible symbols is called the **universal set of symbols**, and this set is denoted by Υ. A finite sequence of symbols from Υ is called **strand**[4]. ϵ denotes the **empty strand** (i.e. the strand that contains no symbol). If w and x are strands, then wx denotes the strand obtained by concatenating w and x. Υ^* denotes the set of strands obtained by concatenating zero or more symbols from Υ. Note that any alphabet is a proper subset of Υ, and Υ^* is the set of all possible finite sequence of symbols.

Let $q \in Q$, $a \in \Upsilon$, and $w \in \Upsilon^*$. The function $\delta^* : Q \times \Upsilon^* \to Q$ is called **extended transition function**, which is recursively defined as follows: $\delta^*(q, \epsilon) = q$, and

$$\delta^*(q, wa) = \begin{cases} \delta(\delta^*(q, w), a) & \text{if } a \in \Sigma_b \\ \delta(\delta^*(q, w), \#) & \text{if } a \notin \Sigma_b \end{cases}$$

M **accepts** w if and only if $\delta^*(q_0, w) \in F$. M **rejects** w if and only if $\delta^*(q_0, w) \notin F$. The asymptotic time complexity of the extended transition function is $O(|w|)$, where $|w|$ denotes the number of symbols in w. Note that the definition of the extended transition function provides the semantics of the # symbol. We call the # symbol **wildcard**, because it matches any symbol in $\Upsilon \setminus \Sigma_b$.

$L(M) = \{w \in \Upsilon^* | \delta^*(q_0, w) \in F\}$ is the **language** of M. A set L of strands is an **Open Regular Language (ORL)**, if and only if there is a DAR M such that $L(M) = L$.

If L is an ORL, then the strands in L consist of symbols from Υ; i.e. not from an alphabet, which is a non-empty and *finite* set of symbols. Since Υ is an 'open-ended' set, we chose the name "open regular language". In Section 2.4, we compare ORLs with regular languages and context-free languages [63].

"Strand" and "string" are different but related terms: First of all, both a string and a strand are finite sequences of symbols. A string consists of symbols from an alphabet. Since any alphabet is a subset of Υ, a string is a strand. Since a strand w consists of finite number of symbols, the set Σ of symbols in w is also finite. Therefore, Σ is an alphabet, and w can be interpreted as a string that consists of symbols from Σ. The empty string can be interpreted as the empty strand, and vice

[4] "Strand" and "string" are different but related terms. We will explain the relation, in this section.

versa.

We conclude this section by revisiting the key difference between a DFA and a DAR: A DFA with an alphabet Σ either accepts or rejects a finite sequence of symbols, provided that the sequence is an element of Σ^*; whereas a DAR either accepts or rejects a finite sequence of symbols, provided that the sequence is an element of Υ^*. Since Υ^* is the set of all possible finite sequences of symbols, a DAR either accepts or rejects *any* finite sequence of symbols.

2.3.5 Semantics of VisuaL

The semantics of VisuaL is defined by the total function $getDARof : V \rightarrow D$, where V is the set of VisuaL specifications, and D is the set of DARs. Let S be a VisuaL specification, and M be a DAR, such that $getDARof(S) = M$. In this section, we step-by-step explain how $getDARof$ constructs M, based on S.

Step 1: Initialization of M

At this step, $getDARof$ initializes $M = \langle Q, \Sigma_a, \delta, q_0, F, \Xi, \eta \rangle$, such that

- $Q = \Omega \cup \{q_t\}$ and $\Omega = \{q_0\}$,
- $\Sigma_a = \Sigma_b \cup \{\#\}$ and $\Sigma_b = \emptyset$,
- δ is not defined yet,
- $F = \emptyset$,
- $\Xi = exp$, where exp is the regular expression on the container node of S, and
- $\eta = name$, where $name$ is the name of the container node of S.

Now, let us see an example. Fig. 2.6 shows an example VisuaL specification that we denote using S^e. We use the superscript ^{e} for distinguishing the example specification

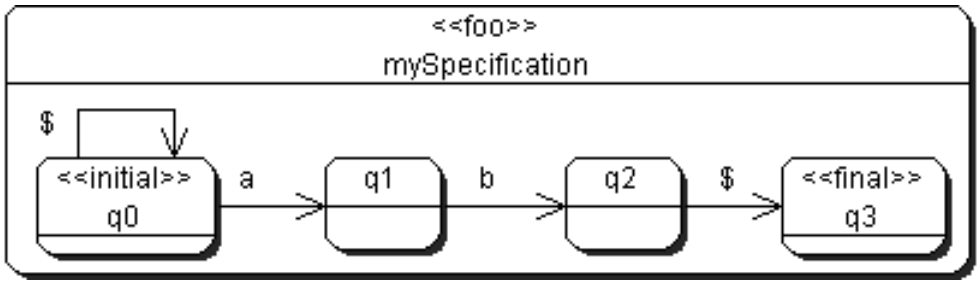

Figure 2.6: An example specification in VisuaL.

from the general specification S (see above). In the remainder of Section 2.3.5, we

will consistently use the superscript ^{e} for distinguishing an example entity from its general counterpart.

If S^e is given to $getDARof$ as the input, then $getDARof$ initializes a DAR $M^e = \langle Q^e, {\Sigma_a}^e, \delta^e, q_0, F^e, \Xi^e, \eta^e \rangle$, such that

- $Q^e = \Omega^e \cup \{q_t\}$ and $\Omega^e = \{q_0\}$,
- ${\Sigma_a}^e = {\Sigma_b}^e \cup \{\#\}$ and ${\Sigma_b}^e = \emptyset$,
- δ^e is not defined yet,
- $F^e = \emptyset$,
- $\Xi^e = foo$, and
- $\eta^e = mySpecification$.

Step 2: Adding the User-Defined States

At this step, $getDARof$ adds new states to Ω and F, as follows: Let $n_0, n_1, ..., n_m$ be the inner nodes of S, such that n_0 is either the initial or the initial-final node. Given $n_0, n_1, ..., n_m$, $getDARof$ performs the following steps:

1. Define new states $q_1, q_2, ..., q_m$, and add them to Ω.
2. For each n_i where $0 \leq i \leq m$, map n_i to q_i. We denote this mapping with the total, one-to-one, and onto function $getStateOf : IN \rightarrow \Omega$, where IN is the set of inner nodes of S.
3. If n_0 is the initial-final node, then add q_0 to F.
4. For each final node n_f of S, add $getStateOf(n_f)$ to F.

Now, let us revisit the example. The inner nodes of S^e (see Fig. 2.6) are `q0`, `q1`, `q2`, and `q3`. Accordingly, $getDARof$

1. Defines new states q_1, q_2, and q_3; and adds them to Ω^e.
2. Maps `q0` to q_0, `q1` to q_1, `q2` to q_2, and `q3` to q_3.
3. Does not add q_0 to F^e.
4. Adds q_3 to F^e.

Consequently, Ω^e becomes $\{q_0, q_1, q_2, q_3\}$, F^e becomes $\{q_3\}$, and Q^e becomes $\{q_0, q_1, q_2, q_3, q_t\}$. In Fig. 2.7, the states of M^e are depicted. The initial state is depicted as the circle that is the target of the only arrow without any source, each non-final state is depicted as a single circle, and each final state is depicted as a double circle.

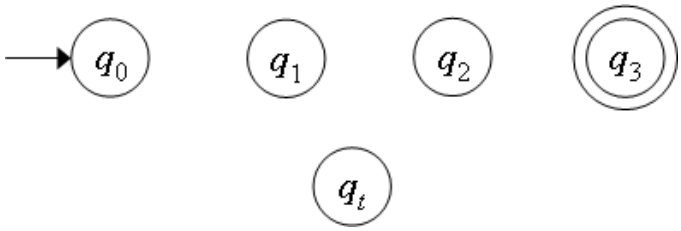

Figure 2.7: The states of M^e, after step 2.

Step 3: Adding the Symbols

At this step, $getDARof$ adds new symbols to Σ_b, as follows: Let LBL be the set of the labels of the edges of S. For each $lbl \in (LBL \setminus \{\$\})$, $getDARof$ defines a new symbol a, maps lbl to a, and adds a to Σ_b. The mapping between the labels and the symbols is denoted by the total, one-to-one, and onto function $getSymbolOf$: $(LBL \setminus \{\$\}) \rightarrow \Sigma_b$.

Now, let us revisit the example. The set of the labels of the edges of S^e (see Fig. 2.6) is { a, b, $ }. Accordingly, $getDARof$ defines new symbols, say, a and b. Subsequently, $getDARof$ maps a to a, and b to b. Finally, $getDARof$ adds a and b to ${\Sigma_b}^e$. Thus, ${\Sigma_b}^e$ becomes $\{a, b\}$, and ${\Sigma_a}^e$ becomes $\{a, b, \#\}$.

Step 4: Partially Defining the Transition Function

At this step, $getDARof$ partially defines δ, as follows: For each edge e (of S) with the source node sn, target node tn, and label lbl, $getDARof$ does the following: If lbl = $, then $getDARof$ defines that $\delta(getStateOf(sr), \#) = getStateOf(tr)$. If $lbl \neq$ $ then $getDARof$ defines that $\delta(getStateOf(sr), getSymbolOf(lbl)) = getStateOf(tr)$.

Now, let us revisit the example. In S^e (see Fig. 2.6), there is a $-labelled edge from q0 to q0; thus, $getDARof$ defines that $\delta^e(q_0, \#) = q_0$. There is a $-labelled edge from q2 to q3, thus $getDARof$ defines that $\delta^e(q_2, \#) = q_3$. There is an a-labelled edge from q0 to q1; thus $getDARof$ defines that $\delta^e(q_0, a) = q_1$. There is a b-labelled edge from q1 to q2; thus, $getDARof$ defines that $\delta^e(q_1, b) = q_2$. These transitions are depicted as labelled arrows in Fig. 2.8.

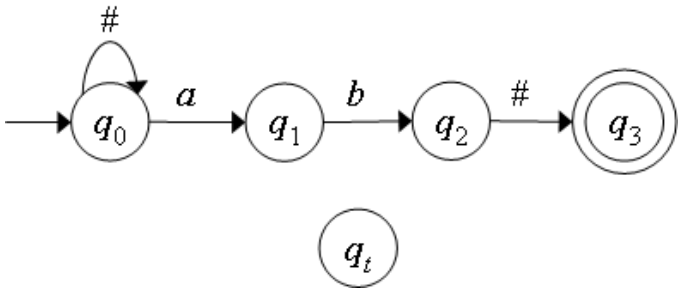

Figure 2.8: The states and some of the transitions of M^e, after step 4.

Step 5: Defining the Remaining Transitions with #

At this step, *getDARof* defines the remaining transitions performed by M upon encountering the symbol #, as follows: For each state $q \in Q$, if $\delta(q, \#)$ is not defined yet, then *getDARof* defines that $\delta(q, \#) = q_t$.

Now let us revisit the example. The transitions $\delta^e(q_1, \#)$, $\delta^e(q_3, \#)$, and $\delta^e(q_t, \#)$ are not defined yet (see Fig. 2.8). Therefore, *getDARof* defines that $\delta^e(q_1, \#) = q_t$, $\delta^e(q_3, \#) = q_t$, and $\delta^e(q_t, \#) = q_t$, as depicted in Fig. 2.9.

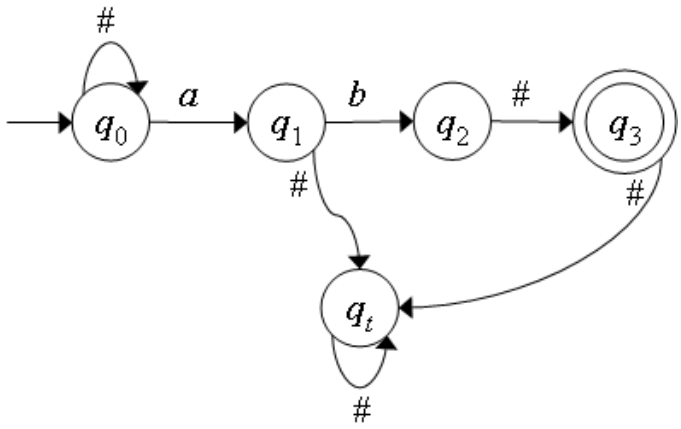

Figure 2.9: The states and some of the transitions of M^e, after step 5.

Step 6: Defining the Remaining Transitions

At this step, *getDARof* defines the remaining transitions of M, as follows: For each state $q \in Q$, and for each symbol $a \in \Sigma_b$, if $\delta(q, a)$ is not defined yet, then

$getDARof$ defines that $\delta(q, a) = \delta(q, \#)$.

Now, let us revisit the example. The transitions $\delta^e(q_0, b)$, $\delta^e(q_1, a)$, $\delta^e(q_2, a)$, $\delta^e(q_2, b)$, $\delta^e(q_3, a)$, $\delta^e(q_3, b)$, $\delta^e(q_t, a)$, and $\delta^e(q_t, b)$ are not defined yet (see Fig. 2.9). Therefore, $getDARof$ defines that $\delta^e(q_0, b) = \delta^e(q_0, \#) = q_0$, $\delta^e(q_1, a) = \delta^e(q_1, \#) = q_t$, $\delta^e(q_2, a) = \delta^e(q_2, \#) = q_3$, $\delta^e(q_2, b) = \delta^e(q_2, \#) = q_3$, $\delta^e(q_3, a) = \delta^e(q_3, \#) = q_t$, $\delta^e(q_3, b) = \delta^e(q_3, \#) = q_t$, $\delta^e(q_t, a) = \delta^e(q_t, \#) = q_t$, and $\delta^e(q_t, b) = \delta^e(q_t, \#) = q_t$, as visible in Fig. 2.10, which is the *transition graph* of M^e. The **transition graph** of a

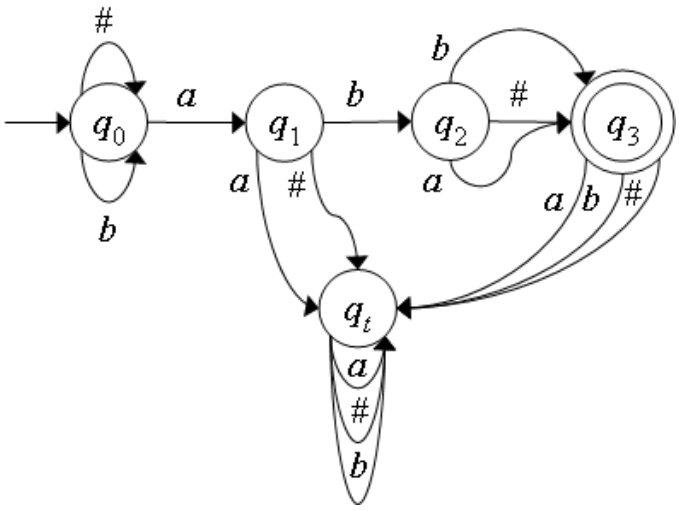

Figure 2.10: The transition graph of M^e, after step 6.

DAR is a graph where nodes represent the states, and edges represent the transitions of the DAR.

Upon the completion of this step, the construction of M and M^e are also completed. This six-step construction (i.e. $getDARof : V \rightarrow D$) defines the semantics of VisuaL. The asymptotic time complexity of $getDARof$ is $O(|Q| \times |\Sigma_b|)$, which is determined by the sixth step.

In Figures 2.11, 2.12, 2.13, and 2.14, we provide the transition graphs of the DARs corresponding to the example VisuaL specifications shown in Figures 2.1, 2.2, 2.3, and 2.4, respectively.

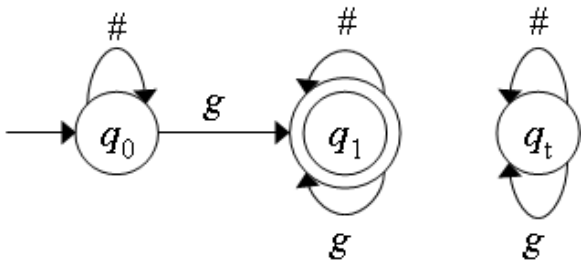

Figure 2.11: The transition graph of the DAR corresponding to the VisuaL specification shown in Fig. 2.1.

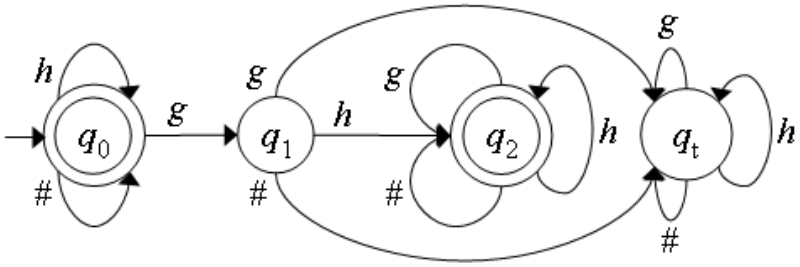

Figure 2.12: The transition graph of the DAR corresponding to the VisuaL specification shown in Fig. 2.2.

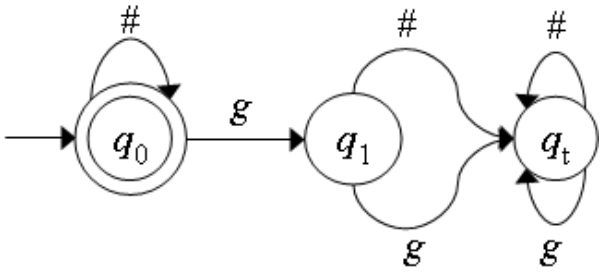

Figure 2.13: The transition graph of the DAR corresponding to the VisuaL specification shown in Fig. 2.3.

2.4 Open Regular Languages versus other Language Families

In Section 2.3.4, we have already defined the Open Regular Languages (ORLs). In this section, we compare ORLs with the Regular Languages (RLs) [63] and Context-Free Languages (CFLs) [63].

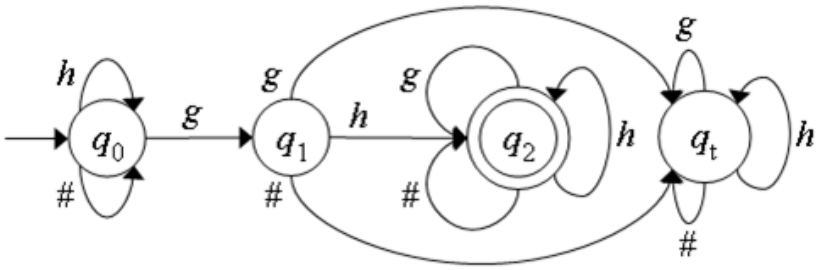

Figure 2.14: The transition graph of the DAR corresponding to the VisuaL specification shown in Fig. 2.4.

2.4.1 Open Regular Languages versus Regular Languages

In this section, we compare ORLs with RLs. In particular, we answer the following two questions: Is any RL also an ORL? Is any ORL also an RL?. To answer the first question, we use the following theorem:

Theorem 2.4.1 *For any given RL L, there is a DAR M^{dar}, such that $L(M^{dar}) = L$.*

Proof: Let L be an arbitrary RL. By the definition of RL (Section 2.3.3), there is a DFA $M^{dfa} = \langle Q^{dfa}, \Sigma^{dfa}, \delta^{dfa}, {q_0}^{dfa}, F^{dfa} \rangle$, such that $L(M^{dfa}) = L$. Based on M^{dfa}, we can step-by-step construct a DAR $M^{dar} = \langle Q^{dar} = Q^{dfa} \cup \{q_t^{dar}\}, \Sigma_a = \Sigma^{dfa} \cup \{\#^{dar}\}, \delta^{dar}, {q_0}^{dfa}, F^{dfa}, \Xi, \eta \rangle$, such that $L(M^{dar}) = L(M^{dfa}) = L$. The steps of this construction are as follows:

1. For each state $q \in Q^{dfa}$ and for each symbol $a \in \Sigma^{dfa}$, define $\delta^{dar}(q, a) = \delta^{dfa}(q, a)$.
2. For each symbol $a \in \Sigma^{dfa}$, define $\delta^{dar}({q_t}^{dar}, a) = {q_t}^{dar}$.
3. For each $q \in Q^{dar}$, define $\delta^{dar}(q, \#^{dar}) = {q_t}^{dar}$.
4. Define Ξ and η arbitrarily. ■

Based on Theorem 2.4.1 and the definition of ORL (Section 2.3.4), we conclude that any RL is also an ORL. Consequently, the answer to the first question stated at the beginning of this section is "yes". This also entails that DFAs are not more expressive than DARs. To answer the second question, we use the following theorem:

Theorem 2.4.2 *For any given ORL L, it is not guaranteed that there is a DFA M^{dfa}, such that $L(M^{dfa}) = L$.*

Proof: For any given ORL L, assume that there is a DFA M^{dfa}, such that $L(M^{dfa}) = L$ (i.e. assume that Theorem 2.4.2 is false). Let M^{dar} denote a DAR whose transition

graph is shown in Fig. 2.13. Observe that $L(M^{dar}) = (\Upsilon \setminus \{g\})^*$, which is the set of all possible strands that do not contain the symbol g. By the definition of ORL (Section 2.3.4), $(\Upsilon \setminus \{g\})^*$ is an ORL. Based on the assumption above, there is a DFA M^{dfa} such that $L(M^{dfa}) = (\Upsilon \setminus \{g\})^*$. Hence, the input alphabet of M^{dfa} is $\Upsilon \setminus \{g\}$. Since the set $\Upsilon \setminus \{g\}$ is infinite, this set is not an alphabet (Section 2.3.1). Note that the previous two sentences contradict with each other.■

Based on Theorem 2.4.2 and the definition of RL (Section 2.3.4), we can conclude that a given ORL is not necessarily an RL. Consequently, the answer to the second question stated at the beginning of this section is "no". This also entails that DARs have different expressive power than DFAs.

Since (a) any RL is also an ORL, and (b) a given ORL is not necessarily an RL, we conclude that the set *RLs* of regular languages is a proper subset of the set *ORLs* of open regular languages, as depicted in Fig. 2.15.

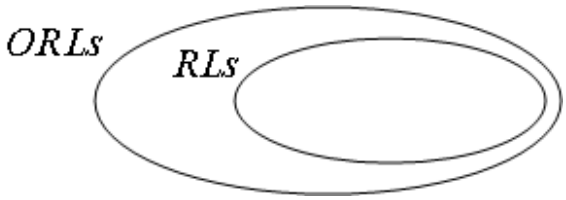

Figure 2.15: The set *RLs* of regular languages is a proper subset of the set *ORLs* of open regular languages.

Since (a) DFAs are not more expressive than DARs, and (b) DARs have different expressive power than DFAs, we conclude that DARs are more expressive than DFAs.

2.4.2 Open Regular Languages versus Context-Free Languages

A given set L of strings is a Context-Free Language (CFL), if and only if there is a Non-deterministic Pushdown Automaton (NPDA) that exclusively accepts the strings in L [63]. The set of RLs is a proper subset of the set of CFLs [63]. Since the set of RLs is a proper subset of both the set of CFLs and the set of ORLs, the relation between ORLs and CFLs is an interesting topic to investigate. In this section, we study this relation. In particular, we answer the following two questions: Is any ORL also a CFL? Is any CFL also an ORL?.

In the Venn diagram depicted in Fig. 2.16, $CFLs$ denotes the set of context-free

languages, and each number denotes the distinct set represented by the closed region where the number is placed. By discovering whether each of these four sets is empty

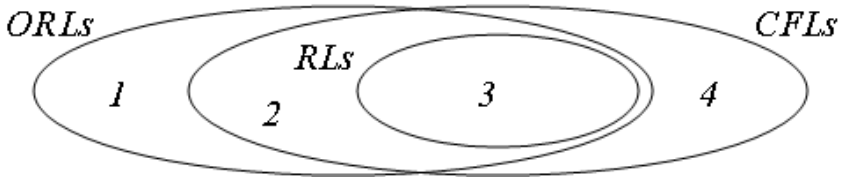

Figure 2.16: Each number in this Venn diagram denotes a distinct set represented by the region where the number is placed.

or not, we can understand the relation between CFLs and ORLs.

Set 3 contains RLs, so it is non-empty. Since an RL is both a CFL and an ORL, $CFLs \cap ORLs \neq \emptyset$.

Let $a^n b^n$ denote the set of strings, where a string consists of n number of a's followed by n number of b's, such that $n \geq 0$. For example, *aaabbb* is in $a^n b^n$, but *abb* not. $a^n b^n$ is known to be a CFL that is not an RL [63]. Now, the question is, whether $a^n b^n$ is in Set 2 or 4 (see Fig. 2.16).

$a^n b^n$ is not an RL, due to the following facts: Any DFA has a finite memory (i.e. a finite set of states), thus for a sufficiently large value of n, a DFA cannot 'remember' how many a's it encountered. Therefore, the DFA cannot 'know' how many b's the string should have. This means that it is impossible to construct a DFA that exclusively accepts the strings in $a^n b^n$. Since a set of strings is an RL if an only if there is a DFA that exclusively accepts these strings, $a^n b^n$ is not an RL.

Similar to a DFA, a DAR also has a finite memory (i.e. a finite set of states). Hence, it is not possible to construct a DAR that exclusively accepts the strings in $a^n b^n$. As a result, $a^n b^n$ is in Set 4 (Fig. 2.16), so Set 4 is non-empty.

In Section 2.4.1, we have shown that $(\Upsilon \setminus \{g\})^*$ is an ORL but not an RL. $(\Upsilon \setminus \{g\})^*$ is not an RL, because the sequences in an RL (i.e. the sequences accepted by a DFA) consist of symbols from a *finite* set of symbols, whereas the sequences in $(\Upsilon \setminus \{g\})^*$ consist of symbols from $\Upsilon \setminus \{g\}$, which is *infinite*. Since, the sequences in a CFL (i.e. the sequences accepted by an NPDA) also consist of symbols from a *finite* set of symbols, we can conclude that $(\Upsilon \setminus \{g\})^*$ is not a CFL, either. Consequently, $(\Upsilon \setminus \{g\})^*$ is an element of Set 1, that is, Set 1 is non-empty.

Based on what we discussed in this section, we conclude the following:

- Since Set 1 is non-empty, a given ORL is not necessarily a CFL. Hence, the answer to the first question stated at the beginning of this section is "no".

- Since Set 4 is non-empty, a given CFL is not necessarily an ORL. Hence, the answer to the second question stated at the beginning of this section is "no".
- Based on (a) the definition of ORL (Section 2.3.4), (b) the definition of CFL (see the beginning of this section), and (c) the fact that Set 1 and Set 4 are non-empty, we conclude that DARs and NPDAs have different expressive power; i.e. NPDAs are not more expressive than DARs, and vice versa.

Currently, we do not know whether Set 2 is empty or not. We are going to investigate this in the future.

2.5 Added Value of DARs

In Section 2.4.1, we concluded that DARs are more expressive than DFAs, and in Section 2.4.2 we concluded that NPDAs and DARs have different expressive power. Hence, we showed that the DAR formalism has an added value from a theoretical point of view. In this section, we explain the added value of DARs from an engineering point of view.

Let the C function `f`, which is shown in Listing 2.1, be an implementation of an algorithm.

```
1 void f()
2 {
3     g(); h(); x();
4 }
```

Listing 2.1: An implementation of an algorithm in C.

In Section 2.2.2, we have stated the constraint C2 as follows: "In each possible sequence of function calls from `f`, if there is at least one call to `g`, then the first call to `g` must be *immediately followed by* a call to `h`". Note that `f` (i.e. Listing 2.1) satisfies this constraint.

In Fig. 2.12, we have shown the transition graph of the DAR that is a formal specification of C2. In this section, we will use M^{dar} to denote this DAR.

Based on the implementation of `f` (i.e. Listing 2.1), we can formally specify C2 using the DFA formalism, too. The transition graph of such a DFA is shown in Fig. 2.17. In this section, we will use M^{dfa} to denote this DFA.

Now, let us compare the evolvability of M^{dar} with the evolvability of M^{dfa}, based on the following evolution scenario: Assume that the source code in Listing 2.1 evolves into the source code in Listing 2.2, which also satisfies C2.

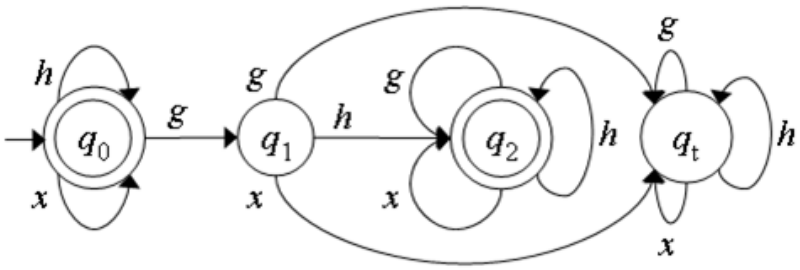

Figure 2.17: The transition graph of the DFA representing the constraint C2, based on the implementation in Listing 2.1.

```
1 void f()
2 {
3     g(); h(); x(); y();
4 }
```

Listing 2.2: The next version of the source code in 5.1.

Considering the definition of C2 (i.e. the English text), one would expect that a formal specification of C2 must not need to be modified due to this evolution, because the definition of C2 is oblivious to the existence or non-existence of a call to `y` in the implementation of `f`. This expectation is fulfilled by M^{dar}, because M^{dar} (see Fig. 2.12) has a #-labelled transition from q_2 to q_2, and this transition matches the newly added call to `y`. However, M^{dfa} does not fulfill the expectation mentioned above, because one has to add the following four transitions to M^{dfa} (Fig. 2.17), so that M^{dfa} remains in sync with the implementation of `f`: a y-labelled transition from (a) q_0 to q_0, (b) q_1 to q_t, (c) q_2 to q_2, and (d) q_t to q_t.

As we have exemplified above, the wildcard symbol # enables us to abstract away from the unnecessary details, such as the call to `y` in Listing 2.2. Due to such abstractions, a DAR is always in sync with the subsequent versions of the corresponding algorithm, as long as the constraint represented by the DAR remains correct and unchanged. To sum up, DARs are more evolvable than DFAs. This is the added value of DARs, from an engineering point of view.

2.6 Expressive Power of VisuaL

In Section 2.3.5, we have seen that each VisuaL specification represents a DAR, hence expresses an ORL. However, we do not yet know the answer to the following question: For any ORL L, is there a VisuaL specification that can express L? To

answer this question, we state the following theorem:

Theorem 2.6.1 *For any DAR M, there is a VisuaL specification S such that $L(getDARof(S)) = L(M)$.*

Proof: For any DAR $M = \langle Q = \Omega \cup \{q_t\}, \Sigma_a = \Sigma_b \cup \{\#\}, \delta, q_0, F, \Xi, \eta \rangle$, it is possible to construct a VisuaL specification S such that $L(getDARof(S)) = L(M)$. This construction is denoted by the total function $getVisuaLof : D \rightarrow V$, where D is the set of DARs, and V is the set of VisuaL specifications. If M is given to $getVisuaLof$ as the input, then $getVisuaLof$ performs the following steps to construct S:

Step1: Initialization of S

At this step, $getVisuaLof$ initializes S, such that

- The name of the container node of S is equal to η, which is the name of M.
- The regular expression on the container node of S is equal to Ξ, which is the scope expression of M.

Now, let us see an example. In Section 2.3.5, we constructed the DAR M^e, whose transition graph is shown in Fig. 2.10. If M^e is given to $getVisuaLof$ as the input, then $getVisuaLof$ initializes a new VisuaL specification shown in Fig. 2.18. We denote this new specification using S^e.

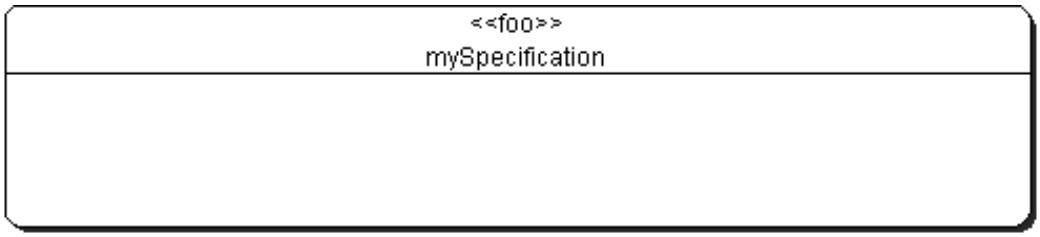

Figure 2.18: The VisuaL specification S^e, after step 1.

The name of the container node of S^e is mySpecification (see Fig. 2.18), because the name η^e of M^e is $mySpecification$ (see Section 2.3.5). The regular expression on the container node of S^e is foo, because the scope expression Ξ^e of M^e is foo (see Section 2.3.5).

Step 2: Adding the Inner Nodes

At this step, $getVisuaLof$ adds inner nodes to S, as follows: For each state $q \in Q$, $getVisuaLof$ creates a distinct inner node n, such that

- q is mapped to n.
- If $q = q_0$ and $q \in F$, then n is the initial-final node.
- If $q = q_0$ and $q \notin F$, then n is the initial node.
- If $q \neq q_0$ and $q \in F$, then n is a final node.
- If $q \neq q_0$ and $q \notin F$, then n is a plain node.

We denote the mapping from the states of M to the inner nodes of S using the total, one-to-one, and onto function $getVisuaLnodeOf : Q \rightarrow IN$, where IN is the set of inner nodes of S.

Now, let us revisit the example. As visible in Fig. 2.10, the states of M^e are q_0, q_1, q_2, q_3, and q_t. q_0 is the initial state, and q_3 is the only final state. Accordingly, $getVisuaLof$ creates the inner nodes, say, `q0`, `q1`, `q2`, `q3`, and `qt` of S^e, such that

- $getVisuaLnodeOf(q_0) =$ `q0`, and `q0` is the initial node.
- $getVisuaLnodeOf(q_1) =$ `q1`, and `q1` is a plain node.
- $getVisuaLnodeOf(q_2) =$ `q2`, and `q2` is a plain node.
- $getVisuaLnodeOf(q_3) =$ `q3`, and `q3` is the final node.
- $getVisuaLnodeOf(q_t) =$ `qt`, and `qt` is a plain node.

Fig. 2.19 shows the VisuaL specification S^e, upon the creation of the inner nodes.

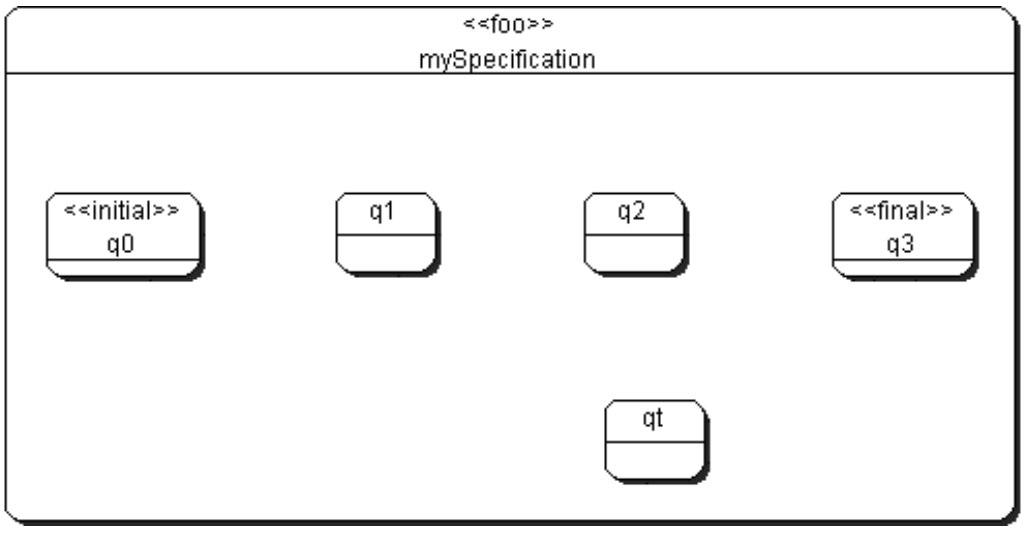

Figure 2.19: The VisuaL specification S^e, after step 2.

Step 3: Adding the Edges

At this step, for each transition $\delta(q_i, a_j) = q_k$ of M, $getVisuaLof$ adds an edge e to S, such that the source of e is $getVisuaLnodeOf(q_i)$, and the target of e is $getVisuaLnodeOf(q_k)$. If $a_j = \#$, then the label of e is \$, else the label of e is the symbol denoted by a_j.

Now, let us revisit the example. Fig. 2.20 shows the specification S^e, upon the addition of the edges according to the transitions of M^e (see Fig. 2.10).

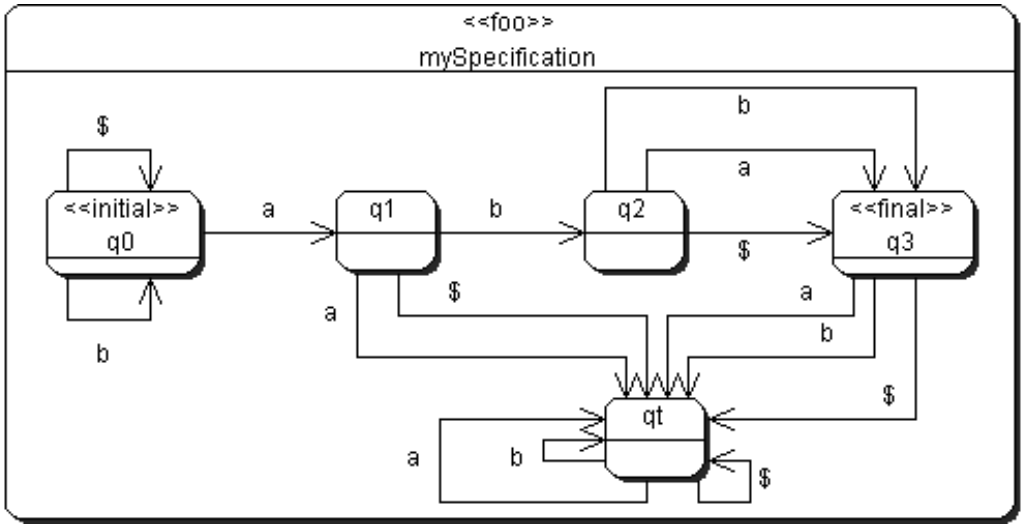

Figure 2.20: The VisuaL specification S^e, after step 3.

Upon the completion of this step, the construction of S and S^e are also completed. $L(getDARof(S)) = L(M)$, because the only difference between $getDARof(S)$ and M is as follows: $getDARof(S)$ has one extra state ${q_t}^{new}$, which is the *unreachable* default trap state of $getDARof(S)$. Since this is the only difference between $getDARof(S)$ and M, we conclude that $L(getDARof(S)) = L(M)$.

Now, let us revisit the example. Fig. 2.21 shows the transition graph of $getDARof(S^e)$, where S^e is shown in Fig. 2.20. By comparing the Figures 2.21 and 2.10, the readers can notice that $L(getDARof(S^e)) = L(M^e)$. ■

Based on (a) Theorem 2.6.1, and (b) the semantics of VisuaL (Section 2.3.5), we conclude the following: Using VisuaL, one can express any ORL and nothing else. Hence, VisuaL and DARs have the same expressive power. Upon this conclusion, it is natural to ask the following question: Since VisuaL and DARs have the same expressive power, then what is the added value of using VisuaL? In Section 2.8, we

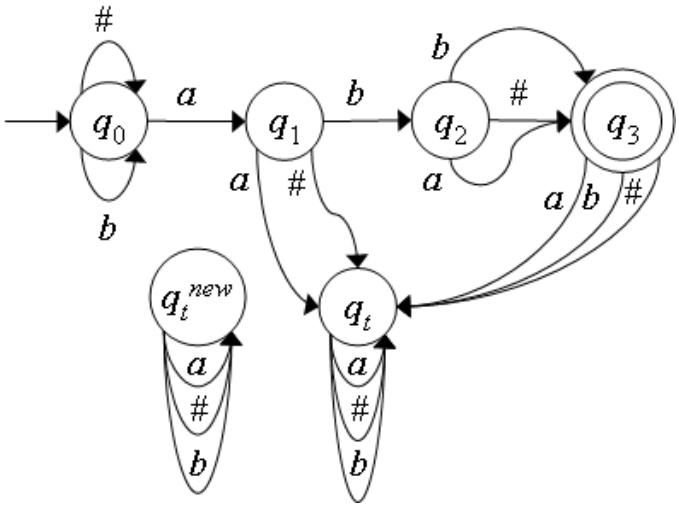

Figure 2.21: The transition graph of $getDARof(S^e)$, after step 3

answer this question. To precisely answer this question, we first need to explain how to construct minimal VisuaL specifications, in Section 2.7.

2.7 Constructing Minimal VisuaL Specifications

A given VisuaL specification expresses a unique ORL, whereas a given ORL can be expressed by more than one VisuaL specification. If there are multiple VisuaL specifications each expressing the same ORL, then the number of edges and inner nodes of these specifications may be different. For example, the specifications shown in Figures 2.6 and 2.20 express the same ORL, but the former specification has 4 edges and 4 inner nodes, whereas the latter specification has 15 edges and 5 inner nodes. Thus, the latter specification is unnecessarily large.

In this section, we explain how to construct VisuaL specifications that are not unnecessarily large. To precisely explain this construction, we use the following terms: Let S_1 and S_2 be VisuaL specifications. S_1 and S_2 are **equivalent**, if and only if $L(getDARof(S_1)) = L(getDARof(S_2))$. A VisuaL specification S_1 is **minimal**, if and only if there is no other VisuaL specification S_2, such that (a) S_1 and S_2 are equivalent, and (b) S_2 has less number of edges or inner nodes.

For a given VisuaL specification S, it is possible to construct the minimal VisuaL specification S_{min} that is equivalent to S. We denote this construction using the total and many-to-one function $minimizeVisuaL : V \rightarrow V$, where V is the set of

VisuaL specifications. In the remainder of this section, we step-by-step explain how $minimizeVisuaL$ constructs S_{min} based on S.

2.7.1 Step 1: Constructing the DAR

At this step, $minimizeVisuaL$ constructs the DAR M, where $M = getDARof(S) = \langle Q, \Sigma_a, \delta, q_0, F, \Xi, \eta \rangle$. This construction is already explained in Section 2.3.5.

Now, let us see an example. In Fig. 2.22, a specification of the following constraint is shown:

C5: In each possible sequence of function calls from `f`, there must be at least two function calls, and the second function call must be a call to `g`.

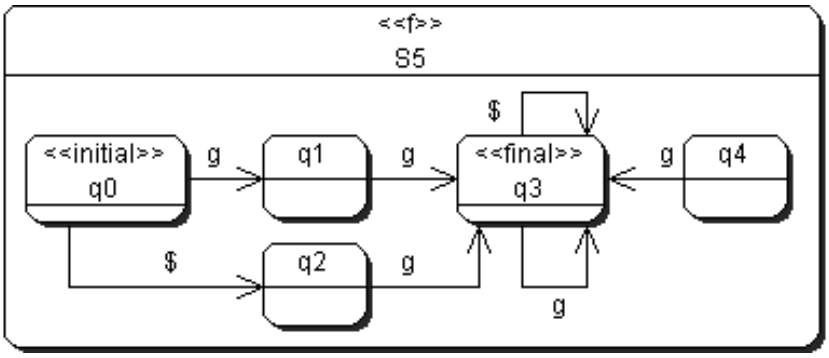

Figure 2.22: A VisuaL specification with unnecessary edges and inner nodes.

Let S^e denote the specification shown in Fig. 2.22. At this step, $minimizeVisuaL$ constructs the DAR $getDARof(S^e)$, whose transition graph is shown in Fig. 2.23. In the remainder of Section 2.7, we use M^e for denoting the DAR whose transition graph is shown in Fig. 2.23.

2.7.2 Step 2: Constructing the Minimal DAR

To precisely explain this step, we first need to define the following terms: Let M_1 and M_2 be DARs. M_1 and M_2 are **equivalent**, if and only if $L(M_1) = L(M_2)$. A DAR M_1 is **minimal**, if and only if there is no other DAR M_2 such that (a) M_1 and M_2 are equivalent, and (b) M_2 has less number of states. These definitions of the equivalence and minimality apply to DFAs, too.

At this step, $minimizeVisuaL$ constructs the minimal DAR M_{min} that is equivalent to M. This construction is denoted by the total and many-to-one function

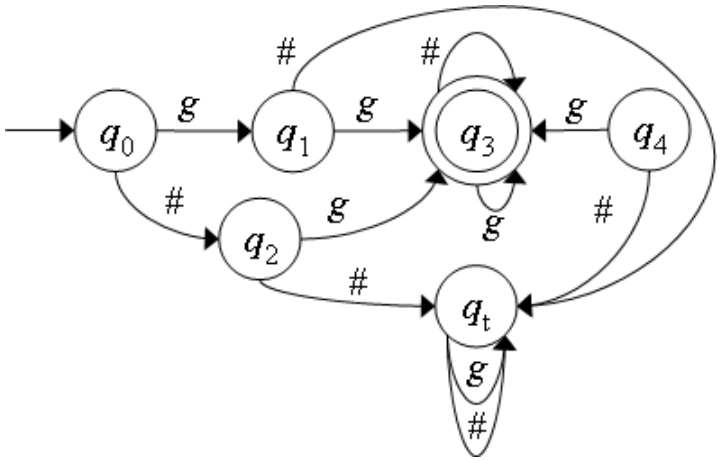

Figure 2.23: The transition graph of the DAR corresponding to the VisuaL specification shown in Fig. 2.22.

$minimizeDAR : D \rightarrow D$, where D is the set of DARs. In the remainder of this section, we step-by-step explain how $minimizeDAR$ constructs M_{min} based on M.

Step 2.1: Finding the Pairs of Distinguishable States

Let q_1 and q_2 be two different states of M. These states are **distinguishable**, if there is at least one strand $w \in \Upsilon^*$ such that (a) $\delta^*(q_1, w) \in F$ and $\delta^*(q_2, w) \notin F$, or (b) $\delta^*(q_1, w) \notin F$ and $\delta^*(q_2, w) \in F$. We say q_1 and q_2 are **indistinguishable**, if and only if they are not distinguishable.

At this step, $minimzeDAR$ finds the distinguishable states of M, by performing the following three-step procedure called $mark$:

1. Remove all unreachable states. This can be done by enumerating all simple paths starting at the initial state in the transition graph of M. If a state does not appear in such a path, then this state is unreachable.
2. For each pair (q_1, q_2) of states, if (a) $q_1 \in F$ and $q_2 \notin F$, or (b) $q_1 \notin F$ and $q_2 \in F$, then mark the pair (q_1, q_2) as distinguishable.
3. Repeat the following step until none of the unmarked pairs can be marked: For each pair (q_1, q_2), if there is an $a \in \Sigma_a$ such that the pair $(\delta(q_1, a), \delta(q_2, a))$ is marked as distinguishable, then mark (q_1, q_2) as distinguishable.

The procedure $mark$ is originally presented in [63], for finding the distinguishable states of a DFA. In [63], it is proven that $mark$ terminates and determines all pairs of distinguishable states.

mark can be implemented by partitioning the states into equivalence classes. Whenever two states are found to be distinguishable, they are immediately put into separate equivalence classes.

Now, let us revisit the example. If M^e (see Fig. 2.23) is given to *mark* as input, then *mark* first removes the state q_4, since this state is unreachable. Next, *mark* partitions Q into two equivalence classes $\{q_0, q_1, q_2, q_t\}$ and $\{q_3\}$. Finally, the execution of the last step of *mark* results in four equivalence classes: $\{q_0\}$, $\{q_1, q_2\}$, $\{q_3\}$, and $\{q_t\}$.

Step 2.2: Reducing the States of M

After the equivalence classes are found, the construction of M_{min} consists of the following six-step procedure called *reduce*: Given $M = \langle Q, \Sigma_a, \delta, q_0, F, \Xi, \eta \rangle$, *reduce* constructs the minimal DAR $M_{min} = \langle Q_{min}, \Sigma_a, \delta_{min}, q_{0_{min}}, F_{min}, \Xi, \eta \rangle$, as follows:

1. Use the procedure *mark* to generate the equivalence classes of the states in Q.
2. For each equivalence class $\{q_i, q_j, ..., q_k\}$ of indistinguishable states, add a state labelled $ij...k$ to Q_{min}.
3. For each transition $\delta(q_r, a) = q_p$ of M, find the equivalence classes to which q_r and q_p belong. If $q_r \in \{q_i, q_j, ..., q_k\}$ and $q_p \in \{q_l, q_m, ..., q_n\}$, then define that $\delta_{min}(ij...k, a) = lm...n$.
4. The initial state $q_{0_{min}}$ is the state whose label includes 0.
5. F_{min} is the set of all states whose label contains i such that $q_i \in F$.
6. Since M has at least one trap state, M_{min} has exactly one trap state. Designate the trap state of M_{min} as the default trap state.

The first five steps of the procedure *reduce* are originally presented in [63], for constructing a minimal DFA. In [63], it is proven that *reduce* terminates, and the resulting DFA is both equivalent to the original DFA and minimal. We added the sixth step, because a DAR must have a default trap state.

Now, let us revisit the example. If M^e (Fig. 2.23) is given to *reduce* as the input, then the output is the DAR ${M^e}_{min}$, whose transition graph is shown in Fig. 2.24.

2.7.3 Step 3: Constructing the Intermediate Specification

At this step, $minimizeVisuaL$ constructs the VisuaL specification $getVisuaLof(M_{min})$. The function $getVisuaLof$ is already explained in Section 2.6.

Now, let us revisit the example. The VisuaL specification $getVisuaLof({M^e}_{min})$ is

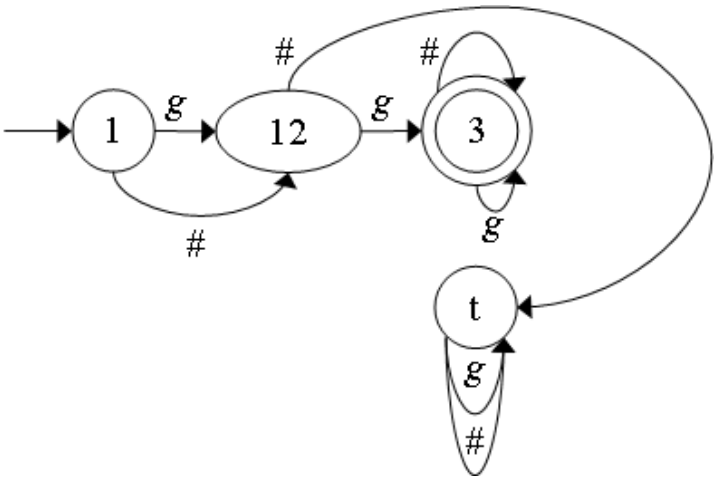

Figure 2.24: The transition graph of a minimal DAR that is equivalent to the DAR whose transition graph is shown in Fig. 2.23.

shown in Fig. 2.25. Note that this specification is equivalent to S^e (Fig. 2.22).

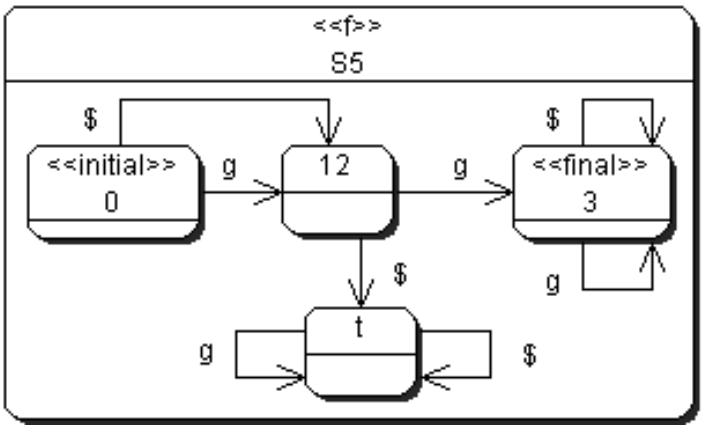

Figure 2.25: The non-minimal VisuaL specification $getVisuaLof(M^e{}_{min})$.

2.7.4 Step 4: Removing the Trap Node and Unnecessary Edges

The VisuaL specification $getVisuaLof(M_{min})$ has a trap node and some edges that are unnecessary. At this step, $minimizeVisuaL$ removes these unnecessary elements, which results in the minimal specification S_{min}. The steps of the removal are as follows:

1. Let n denote the trap node of $getVisuaLof(M_{min})$ (i.e. an inner node n, such that (a) n is not a final or the initial-final node, and (b) the target of each outgoing edge from n is n). Since $minimizeDAR$ (Section 2.7.2) guarantees the existence of exactly one trap state in M_{min}, there is exactly one trap node in $getVisuaLof(M_{min})$. At this step, $minimizeVisuaL$ removes the edges whose source or target is n. If n is not the initial node of $getVisuaLof(M_{min})$, then $minimizeVisuaL$ removes n, too.
2. For each edge e whose label is different than `$`, $minimizeVisuaL$ does the following: If there is a `$`-labelled edge (i.e. a context-sensitive wildcard) with the same source and target as e has, then $minimizeVisuaL$ removes e.

Upon removal of the trap node and the unnecessary edges as explained above, the resulting VisuaL specification is the minimal specification S_{min} that is equivalent to S. S_{min} is minimal, because (a) M_{min} is a minimal DAR, (b) $getDARof(S_{min}) = M_{min}$, and (c) there is no node or edge that can be removed without breaking this equality.

Now, let us revisit the example. If the unnecessary edges and inner nodes of $getVisuaLof(M^e{}_{min})$ (Fig. 2.25) are removed, then the resulting specification is the specification $S^e{}_{min}$, which is shown in Fig. 2.26. Note that (a) $getDARof(S^e{}_{min}) =$

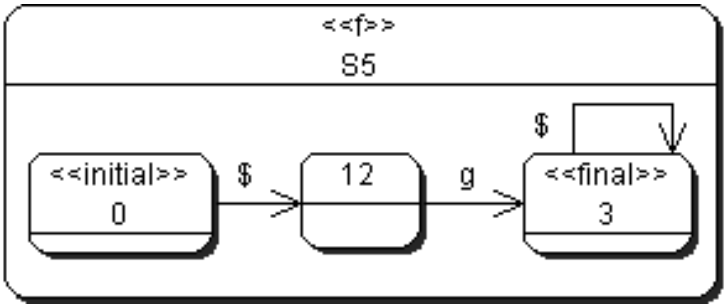

Figure 2.26: A minimal VisuaL specification that is equivalent to the specification shown in Fig 2.22.

$M^e{}_{min}$ (Fig. 2.24), (b) $S^e{}_{min}$ is equivalent to both $getVisuaLof(M^e{}_{min})$ (Fig. 2.25) and S^e (Fig. 2.22), and (c) $S^e{}_{min}$ is minimal.

2.8 Added Value of VisuaL

In Section 2.6, we concluded that the VisuaL language and the DAR formalism have the same expressive power. Thus, from a theoretical point of view, the added value of the VisuaL language is the same as the added value of the DAR formalism. From

an engineering point of view however, VisuaL specifications can be preferred over DARs, due to the reasons explained in the remainder of this section.

A VisuaL specification S has fewer edges and inner nodes than the states and transitions of the corresponding DAR M (i.e. $M = getDARof(S)$); because (a) there is no inner node in S, such that this node corresponds to the default trap state of M, and (b) there are no edges in S, such that these edges correspond to the transitions from- or to the default trap state of M. Hence, S is by default more concise than M.

Context-Sensitive Wildcards (CSWs) (see Section 2.2.1) enable us to reduce the number of edges in VisuaL specifications: In a given VisuaL specification, if there is a CSW with the source node sn and the target node tn, then all the other edges whose source and target nodes are respectively sn and tn can be removed; in which case the resulting specification is equivalent to the original one.

Since VisuaL specifications are more concise than the corresponding DARs, VisuaL specifications are more evolvable than the corresponding DARs. For example, let us imagine that the constraint C5, which is stated at the beginning of Section 2.7.1, evolves into the following constraint: "In each possible sequence of function calls from f, there must be at least two function calls, such that the first function call must be a call to h, and the second function call must be a call to g". To implement this change in the VisuaL specification ${S^e}_{min}$ (Fig. 2.26), it is necessary and sufficient to replace the CSW originating from q0 with an h-labelled edge. Whereas, to implement this change in the DAR ${M^e}_{min}$ (Fig. 2.24), one has to (a) change the target state of the two transitions from q_0, and (b) add four new transitions with the symbol h, such that the transition graph of ${M^e}_{min}$ (Fig. 2.24) is transformed into the transition graph shown in Fig. 2.27.

Currently, we know two alternative ways to formally express a given open regular language L[5]: (1) To create a DAR M, such that $L(M) = L$; or (2) to create a VisuaL specification S, such that $L(getDARof(S)) = L$. Among these alternatives, the most concise (hence evolvable) artifact that expresses L is the minimal VisuaL specification S_{min}, such that $L(getDARof(S_{min})) = L$. This is the added value of VisuaL, from an engineering point of view.

[5]If L is finite, then the third alternative is to enumerate the strands in L.

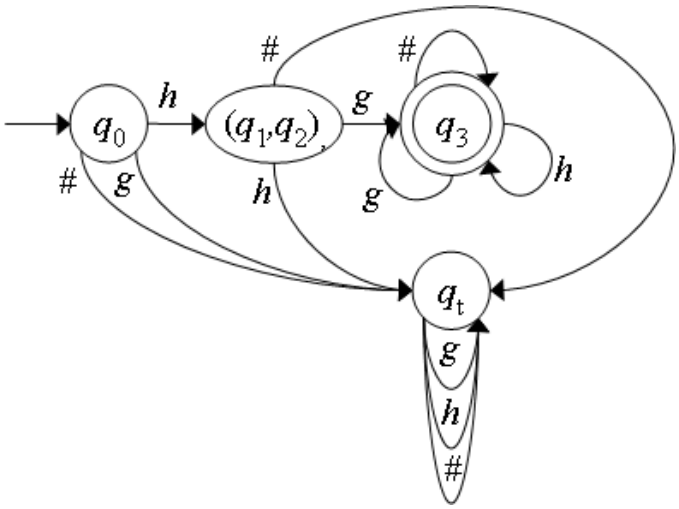

Figure 2.27: The transition graph of a minimal DAR that is the result of an evolution.

2.9 Conclusions

The commonly used graphical languages such as statecharts support hierarchies (i.e. nested structures), so that one can define different levels of abstraction in software specifications. In this chapter, we presented an additional mechanism for abstraction, which we call Context-Sensitive Wildcard (CSW). We defined CSW as the key feature of VisuaL, which is a simple graphical language for expressing the logical and temporal properties of the possible executions of an algorithm. We provided a detailed analysis of VisuaL, such that this analysis reveals the theoretical and practical implications of using CSWs, in the graphical specifications of software behavior.

The theoretical findings presented in this chapter are as follows: a VisuaL specification represents an automaton called Deterministic Abstract Recognizer (DAR) (Section 2.3.5), which is a variant of a Deterministic Finite Accepter (DFA) [63]. DARs express a new family of formal languages called Open Regular Languages (ORLs) (Section 2.3.4). The set of ORLs is a proper superset of the set of regular languages [63]; but the set of context-free languages [63] is neither a superset nor a subset of the set of ORLs (Section 2.4). Using VisuaL, one can express any ORL and nothing else; thus the VisuaL language and the DAR formalism have the same expressive power (Section 2.6). These theoretical findings can be reused for extending the existing graphical languages with CSW.

The practical findings presented in this chapter are as follows: DARs have a wildcard

symbol in their alphabet. Due to this symbol, DARs are more evolvable than DFAs (Section 2.5). The key feature of VisuaL is CSW. Due to this feature, it is possible to create VisuaL specifications that are more concise, hence more evolvable than the corresponding DARs (Section 2.8).

The state-of-the-art verification tools such as Bandera [25, 51, 34] and LTSA [65, 42] support temporal logics (e.g. LTL [23], FLTL [42, 61]) for expressing the properties of software systems. Temporal logics are textual formalisms. In today's industrial practice however, there is a large group of practitioners who prefer graphical formalisms (e.g. statecharts) for specifying their software. Hence, there is a gap between the preferences of these practitioners and the specification languages supported by the state-of-the art model checking tools. Empirical evidence (Chapters 5 and 7) indicates that VisuaL, whose key feature is CSW, has the potential to bridge this gap. Hence, VisuaL addresses the "requirements specification problem" stated by Hatcliff and Dwyer [51]. Last but not the least, VisuaL addresses the first problem stated in Section 1.1, and enables us to address the remaining three problems, as we explain in the remainder of this thesis.

Chapter 3

Checking the Consistency between VisuaL Specifications

3.1 Introduction

Using VisuaL, one can create multiple specifications each representing a different constraint on the same function. For example, each of the specifications presented in Section 2.2 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 each of the constraints, then the VisuaL specifications are **inconsistent**.

For example, the specifications S1 (Fig. 2.1) and S3 (Fig. 2.3) are *in*consistent: If an implementation of the function `f` satisfies the constraint C1: "In each possible sequence of function calls from the function `f`, there must be *at least one* call to the function `g`." (reprinted from Section 2.2.1), then this implementation cannot satisfy the constraint C3: "In each possible sequence of function calls from `f`, a call to `g` must *not* exist." (reprinted from Section 2.2.3). Conversely, if an implementation of the function `f` satisfies C3, then this implementation cannot satisfy C1. Hence, it is impossible to implement `f`, such that the implementation satisfies both C1 and C3.

Whenever VisuaL specifications are created or modified in the software life cycle, the consistency between the specifications must be verified. Manually verifying the consistency is an effort-consuming and error-prone task. If the specifications

are inconsistent, then manually finding and resolving the inconsistency is an effort-consuming and error-prone task, too. CheckDesign can reduce the effort and prevent the errors: CheckDesign takes a set of VisuaL specifications, and automatically finds out whether the specifications are consistent or not. If the specifications are inconsistent, then CheckDesign outputs an error message that can help in understanding and resolving the inconsistency. In this chapter, we explain how CheckDesign works, in four steps: In the first step, CheckDesign determines the clusters of specifications that may be inconsistent. In the second step, CheckDesign derives the DARs of the specifications using which it performs the consistency analysis. In the third step, CheckDesign *aligns* the DARs within a given cluster, so that these DARs can be composed for constructing the *cluster DAR*, in the fourth step. The *cluster DAR* contains the necessary and sufficient information for concluding whether the VisuaL specifications are consistent or not.

3.2 Step1: Clustering the Specifications

If CheckDesign is provided with multiple specifications, then it clusters the specifications according to the functions for which they are written. More precisely, if there are one or more specifications that are created for expressing constraints on the same function, then these specifications are put into the same cluster.

For example, let us assume that CheckDesign is provided with the specifications S1 (Fig. 2.1) and S2 (Fig.2.2). In this case, CheckDesign creates only one cluster, say CLS, and puts S1 and S2 into CLS, because both S1 and S2 are written for the same function: `f`.

As explained in Section 2.2.1, one can write a VisuaL specification for multiple functions, by writing a regular expression in the stereotype of the container node of the specification. Such a specification may appear in multiple clusters.

The reason for the clustering is to determine the group of specifications for which the consistency analysis is relevant. If each specification in a given group is written for the same function, then a consistency analysis of these specifications is relevant, since the specifications may be inconsistent. If specifications are written for different functions, then a consistency analysis of these specifications is irrelevant, since the specifications cannot be inconsistent.

Once the specifications are clustered, for each cluster that contains more than one specification, CheckDesign performs the steps explained in the remainder of Chapter 3.

3.3 Step 2: Deriving DARs from Specifications

For each specification within a given cluster, CheckDesign derives the corresponding DAR. For example, let us reconsider the cluster CLS (see Section 3.2) that contains the specifications S1 and S2. CheckDesign would perform this step by deriving

1. the DAR of S1. The transition graph of this DAR is depicted in Fig. 2.11. We will use M_{S1} to denote this DAR.
2. the DAR of S2. The transition graph of this DAR is depicted in Fig. 2.12. We will use M_{S2} to denote this DAR.

If a specification appears in multiple clusters, then one instance of the DAR of the specification is created for each cluster.

The DAR of a specification provides the semantics of the specification. Therefore, CheckDesign uses DARs for performing the remaining steps of the consistency analysis.

After the DAR of each specification is derived, the next step is the alignment of each DAR with respect to its cluster, which we explain in Section 3.4.

3.4 Step 3: Aligning the DARs of a Cluster

Before we start explaining this step, we need to define the following set: Υ denotes the set of all symbols.

The set of symbols represented by the # (i.e. wildcard symbol) of a given DAR of a given cluster may be different than the set of symbols represented by the # of another DAR of the same cluster. For example, the set of symbols represented by the # of M_{S1} (Fig. 2.11) is $\{sym|sym \in (\Upsilon \setminus \{g\})\}$, whereas the set of symbols represented by the # of M_{S2} (Fig. 2.12) is $\{sym|sym \in (\Upsilon \setminus \{g, h\})\}$. Elimination of such differences without altering the sets of strands (see Section 2.3.4) accepted by the DARs is called **alignment** of DARs with respect to their cluster. An alignment is necessary for the unification explained in Section 3.5.2, and consists of two steps explained below.

3.4.1 Step 3.1: Constructing the Cluster Alphabet

To precisely explain this step we first need to revisit the following term, which is already defined in Section 2.3.4:

Definition: If Σ is the abstract input alphabet of a DAR, then $\Sigma \setminus \{\#\}$ is the **base input alphabet** of the DAR.

At this step, CheckDesign constructs the **cluster alphabet**, which is the union of the base input alphabets of the DARs within the cluster. For example, the base input alphabet of M_{S1} is $\Sigma_{S1} = \{g\}$, and the base input alphabet of M_{S2} is $\Sigma_{S2} = \{g, h\}$. Hence, the cluster alphabet is $\Sigma_{CLS} = \Sigma_{S1} \cup \Sigma_{S2} = \{g, h\}$.

3.4.2 Step 3.2: Transforming the DARs

To precisely explain this step, we need to revisit the following term, which is already defined in Section 2.7.2:

Let M_1 and M_2 be DARs. $L(M_1)$ denotes the set of strands accepted by M_1.

Definition: M_1 and M_2 are **equivalent**, if and only if $L(M_1) = L(M_2)$.

At this step, CheckDesign transforms each DAR to an equivalent DAR whose base alphabet is equal to the cluster alphabet. For a given DAR, this transformation is performed as follows:

For each symbol *sym* that is in the cluster alphabet but not in the base alphabet of the DAR, CheckDesign does the following: For each state s of the DAR, CheckDesign identifies the target state ts of the #-labelled transition whose source state is s, and then adds a new *sym*-labelled transition whose source and target states are respectively s and ts.

Note that each newly added transition is an 'implicit' part of a #-labelled transition of the original DAR, and the only thing this transformation does is to make these 'implicit' transitions 'explicit'. Consequently, the transformation explained above preserves the set of strands accepted by the original DAR.

For example, M_{S1} (Fig. 2.11) and M_{S2} (Fig. 2.12) are transformed as follows: Since $\Sigma_{CLS} \setminus \Sigma_{S1} = \{h\}$, CheckDesign transforms M_{S1} to an equivalent DAR M'_{S1} (Fig. 3.1) by adding three transitions that are labelled with h: An h-labelled transition from q_0 to q_0, from q_1 to q_1, and from q_t to q_t. Since $\Sigma_{CLS} \setminus \Sigma_{S2} = \emptyset$, CheckDesign transforms M_{S2} to an equivalent DAR M'_{S2} without adding any transition to M_{S2}; i.e. M_{S2} is already aligned, and M_{S2} and M'_{S2} are identical.

The alignment presented above is correct, because (a) M'_{S1} is equivalent to M_{S1}, (b) M'_{S2} is equivalent to M_{S2}, and (c) the set of symbols represented by the # of M'_{S1} is equal to the set of symbols represented by the # of M'_{S2}.

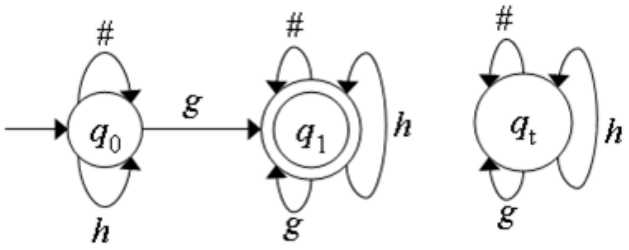

Figure 3.1: The transition graph of a DAR that is equivalent to the DAR whose transition graph is shown in Fig. 2.11.

3.5 Step 4: Constructing the Cluster DAR

For a given cluster *cluster* of aligned DARs $M'_1, ..., M'_n$, CheckDesign constructs the **cluster DAR** $M_{cluster}$, such that $L(M_{cluster}) = L(M'_1) \cap ... \cap L(M'_n)$. If $L(M_{cluster}) \neq \emptyset$, then there is at least one strand (i.e. sequence of function calls) that is matched by the patterns of the VisuaL specifications whose DARs are $M_1, ..., M_n$. Hence, there is at least one possible implementation of the function, such that the implementation satisfies the constraints represented by the specifications. Thus, if $L(M_{cluster}) \neq \emptyset$, then the specifications are **consistent**, else **inconsistent**.

Note that *cluster* and CLS are different clusters: *cluster* is a general cluster containing general DARs $M_1, ..., M_n$, whereas CLS is an example cluster introduced in Section 3.2. CLS contains the example DARs M_{S1} and M_{S2} introduced in Section 3.3.

The construction of a cluster DAR is based on the DeMorgan's law:

$$L(M'_1) \cap ... \cap L(M'_n) = \overline{\overline{L(M'_1)} \cup ... \cup \overline{L(M'_n)}}$$

Instead of constructing $M_{cluster}$ such that $L(M_{cluster}) = L(M'_1) \cap ... \cap L(M'_n)$, we construct $M_{cluster}$ such that $L(M_{cluster}) = \overline{\overline{L(M'_1)} \cup ... \cup \overline{L(M'_n)}}$, because this enables us to decompose the construction into five simple steps explained in the remainder of this section. To perform these steps, we benefit from the proven algorithms available in almost every introductory textbook on the theory of computation (e.g. [63]). If we would construct the cluster DAR without using the DeMorgan's law, then we would have to invent our own algorithm for the construction.

3.5.1 Step 4.1: Complementing the Aligned DARs

At this step, CheckDesign performs the following operation for each of the aligned DARs $M'_1, ..., M'_n$:

Let M' be an aligned DAR, Q' denote the set of states of M', and F' denote the set of final states such that $F' \subseteq Q'$. Given M', CheckDesign creates another DAR M'' whose alphabet, transitions, and the set of states are identical to those of M', but the set of final states is $Q' \setminus F'$.

We call M'' the **complement** of M' (and vice versa), because $L(M'') = \overline{L(M')}$. In [63], this claim is already proven for DFA. The same proof applies to DARs as well, because the complement operation preserves the set of symbols represented by the # symbol of a given DAR.

For example, the complements of the aligned DARs M'_{S1} (Fig. 3.1) and M'_{S2} (Fig. 2.12) are respectively M''_{S1} (Fig. 3.2), and M''_{S2} (Fig. 3.3).

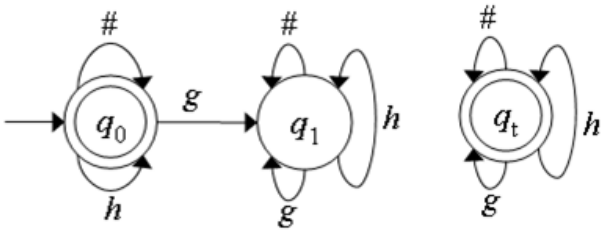

Figure 3.2: The transition graph of the DAR that is the complement of the DAR whose transition graph is shown in Fig. 3.1.

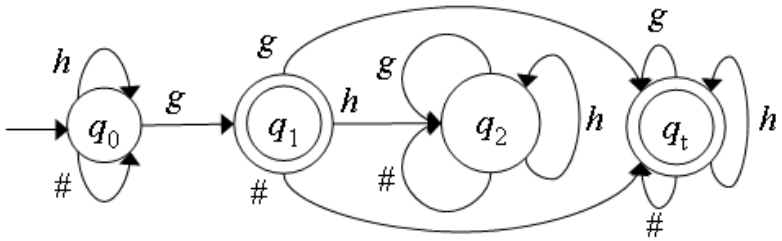

Figure 3.3: The transition graph of the DAR that is the complement of the DAR whose transition graph is shown in Fig. 2.12.

The automaton resulting from a complement operation may not have any trap state. In such a case, we assume that the automaton has a default trap state that is not reachable. This assumption enables us to treat the automaton as an official DAR, according to the definition of DARs in Section 2.3.4.

3.5.2 Step 4.2: Unifying the Complemented DARs

To explain this step, we use the following well-known concepts: ϵ denotes the empty strand (Section 2.3.4), which does not contain any symbol. A key difference of non-deterministic finite-state automata with λ-transitions (NFA-λ) [63] from deterministic finite state automata (DFA) [63] is as follows: an NFA-λ may contain λ-labelled transitions that are performed without consuming the 'current' symbol of the input string, whereas a DFA consumes exactly one symbol from the input string to perform a transition.

To perform the unification explained in this section, we use *non-deterministic abstract recognizers with ϵ transitions (NAR-ϵ)*, which is defined as follows:

Definition: Let $P(Q)$ denote the power set of a given set Q. **Non-Deterministic Abstract Recognizers with ϵ Transitions (NAR-ϵ)** are defined exactly the same as DARs (Section 2.3.4), except the transition function. The transition function of a NAR-ϵ is defined as $\delta : Q \times (\Sigma_a \cup \{\epsilon\}) \rightarrow P(Q)$.

The difference between a NAR-ϵ and a DAR is the same as the difference between a NFA-λ and a DFA. To understand the remainder of this section, knowing the difference explained at the beginning of this section is necessary and sufficient.

CheckDesign unifies the complemented DARs $M''_1, ..., M''_n$ by constructing a NAR-ϵ $M'''_{cluster}$[1], such that $L(M'''_{cluster}) = L(M''_1) \cup ... \cup L(M''_n)$. To construct $M'''_{cluster}$, CheckDesign first defines a new initial state ${q_0}^{nar}$, and then for each M''_i where $1 \leq i \leq n$, CheckDesign adds an ϵ-labelled transition whose source and target states are respectively ${q_0}^{nar}$ and the initial state of M''_i. As a result, $L(M'''_{cluster}) = L(M''_1) \cup ... \cup L(M''_n)$. In [63], the same construction is used for unifying DFAs. Although the upper bound of the asymptotic time complexity of this construction is exponential, the complexity remains polynomial while unifying the complemented DARs. This is due to the fact that DARs are deterministic. In fact, the unification is similar to taking the product of deterministic automata, and this operation has a polynomial time complexity.

For example, the complemented DARs M''_{S1} (Fig. 3.2) and M''_{S2} (Fig. 3.3) are unified by constructing the NAR-ϵ M'''_{CLS} whose transition graph is shown in Fig. 3.4. In this figure, we superscripted the name of each non-initial state with the name of the related specification, so that the states can be uniquely identified. In addition, we have drawn dashed rectangles to indicate which part of M'''_{CLS} represents which DAR.

[1]The construction of $M_{cluster}$ starts with $M'''_{cluster}$. Each step of the construction removes one prime (i.e. $'$), hence after three steps $M'''_{cluster}$ is transformed into $M_{cluster}$.

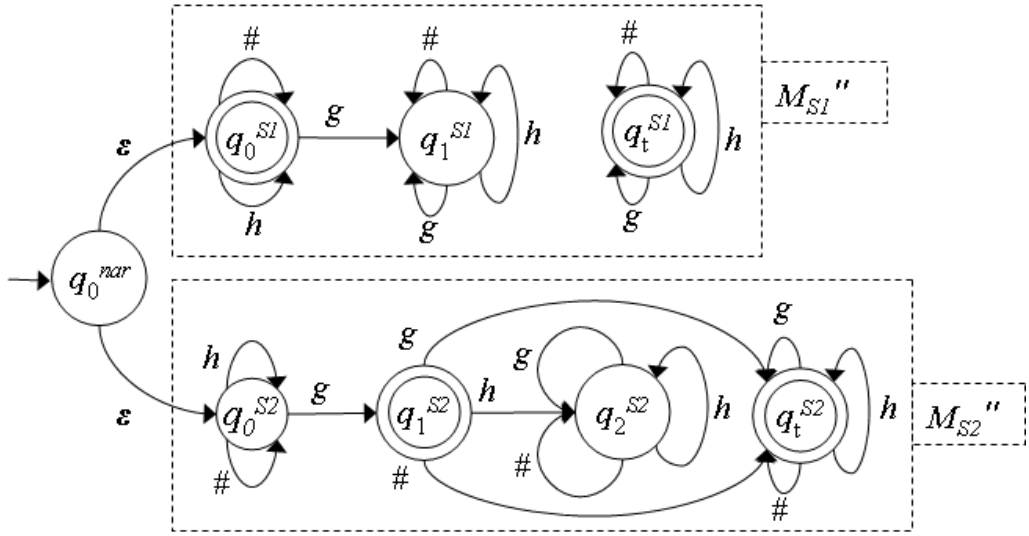

Figure 3.4: The transition graph of the NAR-ϵ resulting from the unification of the DARs whose transition graphs are shown in Fig. 3.2 and 3.3.

Thanks to the alignment (Section 3.4), each # in Fig. 3.4 represents the same set of symbols: $\{sym|sym \in (\Upsilon \setminus \{g, h\})\}$.

The NAR-ϵ resulting from a unification contains multiple trap states. One of the trap states is considered to be the default trap state of the NAR-ϵ. For example, M'''_{CLS} (Fig. 3.4) has two trap states: (${q_t}^{S1}$ and ${q_t}^{S2}$). One of these trap states, say ${q_t}^{S1}$, is considered to be the default trap state.

3.5.3 Step 4.3: Constructing a DAR equivalent to NAR-ϵ

At this step, CheckDesign constructs a DAR $M''_{cluster}$ that is equivalent to the NAR-ϵ $M'''_{cluster}$ constructed in Section 3.5.2. For the construction of $M''_{cluster}$, CheckDesign uses the existing algorithm [63] that constructs a DFA that is equivalent to a given NFA-λ. Since this algorithm preserves the set of symbols represented by the # symbol of a given NAR-ϵ, the resulting DAR is equivalent to NAR-ϵ.

For example, CheckDesign constructs the DAR M''_{CLS} (Fig. 3.5), which is equivalent to the NAR-ϵ M'''_{CLS} (Fig. 3.4), as follows: In M'''_{CLS} (Fig. 3.4), ${q_0}^{S1}$ and ${q_0}^{S2}$ are reachable from the initial state ${q_0}^{nar}$, without consuming any symbol. Therefore, the initial state of M''_{CLS} (Fig. 3.5) is defined as $\{{q_0}^{nar}, {q_0}^{S1}, {q_0}^{S2}\}$. Since this initial state contains a final state (i.e. ${q_0}^{S1}$), the initial state is designated as a final state, too.

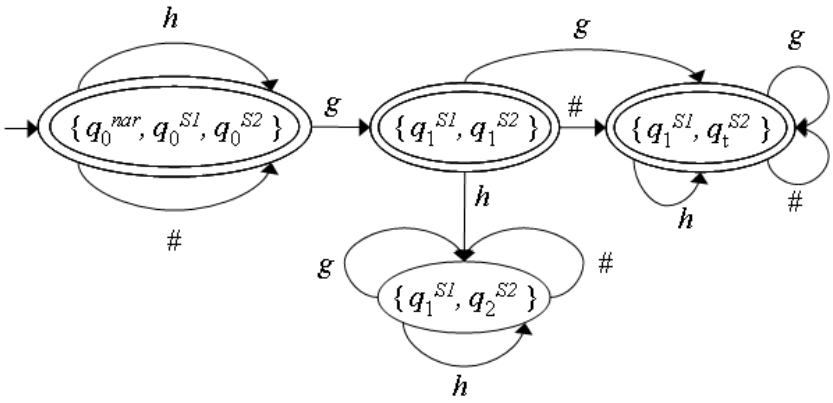

Figure 3.5: The transition graph of the DAR that is equivalent to the NAR-ϵ whose transition graph is shown in Fig. 3.4.

When the current state of M'''_{CLS} is the initial state (i.e. ${q_0}^{nar}$), if the incoming symbol is anything else than g, then M'''_{CLS} performs a transition to ${q_0}^{S1}$ and ${q_0}^{S2}$. This defines the two transitions whose source and target is $\{{q_0}^{nar}, {q_0}^{S1}, {q_0}^{S2}\}$ in M''_{CLS}, because $\{{q_0}^{nar}, {q_0}^{S1}, {q_0}^{S2}\}$ contains ${q_0}^{S1}$ and ${q_0}^{S2}$.

When the current state of M'''_{CLS} is the initial state, if the incoming symbol is g, then M'''_{CLS} performs a transition to ${q_1}^{S1}$ and ${q_1}^{S2}$. This defines (a) the state $\{{q_1}^{S1}, {q_1}^{S2}\}$ of M''_{CLS}, and (b) the g-labelled transition from $\{{q_0}^{nar}, {q_0}^{S1}, {q_0}^{S2}\}$ to $\{{q_1}^{S1}, {q_1}^{S2}\}$. Since ${q_1}^{S2}$ is a final state, $\{{q_1}^{S1}, {q_1}^{S2}\}$ is also designated as a final state.

We do not explain the construction of M''_{CLS} any further, because the explanation above should be sufficient to provide the intuition about the construction. Interested readers can continue the construction to verify that M''_{CLS} and M'''_{CLS} are indeed equivalent. The algorithm of the construction is available in [63].

The deterministic automaton that is derived from a NAR-ϵ may not have any trap state. In such a case, we assume that the automaton has a default trap state that is not reachable. This assumption enables us to treat the automaton as an official DAR in the next step.

3.5.4 Step 4.4: Minimizing the Number of States and Transitions of DAR

At this step, CheckDesign constructs the DAR $M'_{cluster}$ that (a) is equivalent to the DAR $M''_{cluster}$, and (b) has the minimum number of states and transitions. This construction is already explained in Section 2.7.2.

For example, the DAR M'_{CLS} resulting from the minimization of M''_{CLS} (Fig. 3.5) is identical to M''_{CLS}, because the number of states of M''_{CLS} is already minimal. Interested readers can verify that there is no other DAR that is both equivalent to M''_{CLS} and has less number of states, by using the minimization algorithm explained in Section 2.7.2.

3.5.5 Step 4.5: Complementing the Minimized DAR

At this step, CheckDesign derives the cluster DAR $M_{cluster}$ by complementing the DAR $M'_{cluster}$. The complement operation is already explained in Section 3.5.1.

Thanks to the DeMorgan's law (see the beginning of Section 3.5), the set of strands accepted by the cluster DAR $M_{cluster}$ is equal to the intersection of the sets of strands accepted by the aligned DARs (i.e. $L(M_{cluster}) = L(M'_1) \cap ... \cap L(M'_n)$).

In the transition graph of $M_{cluster}$, if there is at least one final state that is reachable, then this indicates that $L(M_{cluster}) \neq \emptyset$, i.e. there is at least one possible implementation of the function that satisfies each constraint in the cluster; the specifications of the cluster are **consistent**. If there is not any final state that is reachable, then the specifications are **inconsistent**.

For example, M'_{CLS} (Fig. 3.5) is complemented to obtain the cluster DAR M_{CLS} (Fig. 3.6). Note that there is a final state in Fig. 3.6, and this state is reachable.

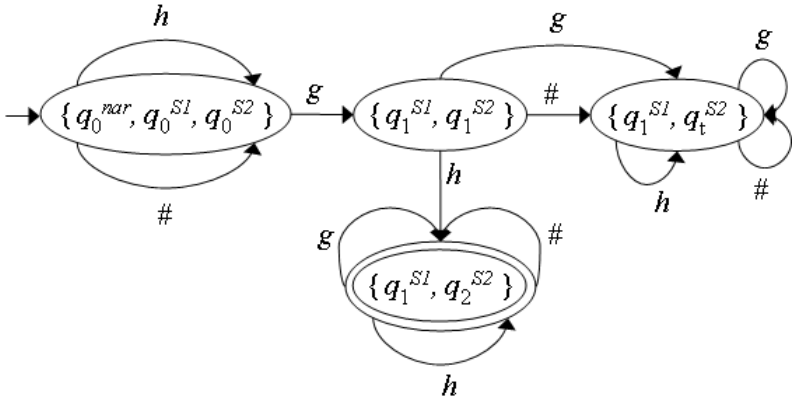

Figure 3.6: The transition graph of the DAR that is the complement of the DAR whose transition graph is shown in Fig. 3.5.

Therefore, S1 (Fig. 2.1) and S2 (Fig. 2.2) are consistent.

Note that M_{CLS} (Fig. 3.6) is equivalent to the DAR whose transition graph shown

in Fig. 2.14. If this equivalence did not exist, then the steps performed so far in Section 3 would be incorrect.

3.6 Deriving the Cluster Specification

For a given cluster DAR $M_{cluster}$, CheckDesign can derive and output the corresponding VisuaL specification $S_{cluster}$. Such a specification can be useful for documentation and debugging purposes. For example, if the cluster DAR M_{CLS} (Fig. 3.6) is given to CheckDesign, then CheckDesign can output the cluster specification that is identical to the specification shown in Fig. 2.4, except the name would be SCLS. To create $S_{cluster}$ based on $M_{cluster}$, CheckDesign first performs the steps explained in Section 2.6, and then the steps explained in Section 2.7.

3.7 Analysis Report

After the construction of a cluster DAR (e.g. Fig. 3.6), CheckDesign outputs an analysis report, which contains the result of the consistency analysis. If the specifications are consistent, then the analysis report can be used for generating source code, as we discuss in Section 8.2.2. If the specifications are inconsistent, then the report can be used for understanding and resolving the inconsistency.

The analysis report is derived from the not-minimized version of a cluster DAR, which is constructed by skipping the minimization step explained in Section 3.5.4. In this section, we explain how CheckDesign constructs an example analysis report from an example not-minimized cluster DAR, and discuss how an analysis report is constructed in general.

Imagine that we provide CheckDesign with the specifications shown in Fig. 2.1, 2.2, and 2.3, whose corresponding DARs M_{S1}, M_{S2}, and M_{S3} are respectively shown as transition graphs in Fig. 2.11, 2.12, and 2.13. In this case, CheckDesign would construct the not-minimized cluster DAR whose transition graph is shown in Fig 3.7. Note that there is no reachable final state in Fig. 3.7, because the specifications are inconsistent.

CheckDesign outputs an analysis report, which consists of multiple parts; each part is formulated based on the states of the not-minimized cluster DAR, and the elements of one of the eight sets visualized in the Venn diagram in Fig. 3.8. Any number shown in the diagram denotes a distinct set represented by the region where the number is placed; i.e. the numbers are not elements. Elements are not visible in the

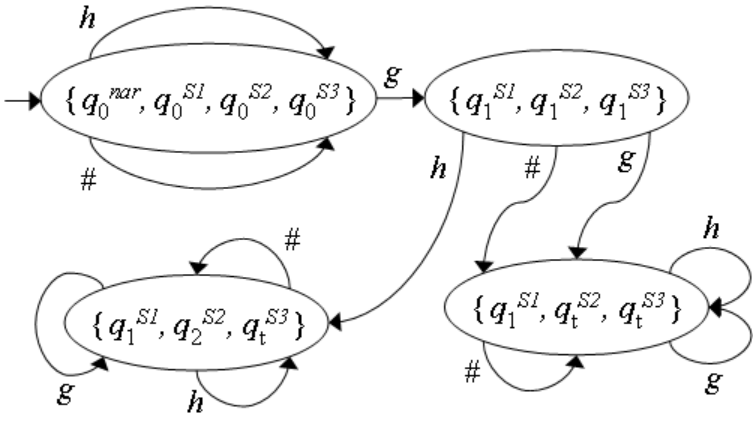

Figure 3.7: The transition graph of a not-minimized cluster DAR that does not have any reachable final state.

diagram, we provide them later in this section. U denotes the universal set.

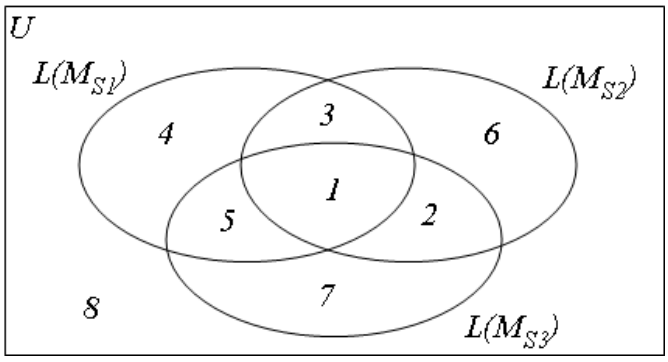

Figure 3.8: Any number in this Venn diagram denotes a distinct set represented by the region where the number is placed. CheckDesign uses this Venn diagram while constructing an analysis report.

Since there is no reachable final state in Fig. 3.7, $L(M_{S1}) \cap L(M_{S2}) \cap L(M_{S3}) = \emptyset$. That is, Set 1 (see Fig. 3.8) is empty. Consequently, the first part of the report is "There is no possible implementation of `f`, such that the implementation satisfies the constraints represented by the specifications S1, S2, and S3."

Before we present the remaining parts of the report, we need to provide the following preliminary information: As visible in Fig. 3.7, each state of the cluster DAR is a set of states. For example, the initial state of the cluster DAR is the set $\{{q_0}^{nar}, {q_0}^{S1}, {q_0}^{S2}, {q_0}^{S3}\}$. In such a set, each state other than ${q_0}^{nar}$ represents either a non-final or a final state of the DAR that is derived from a VisuaL specification.

For example, ${q_0}^{S1}$ represents the *non-final* state q_0 of M_{S1} (Fig. 2.11) that is derived from the VisuaL specification shown in Fig. 2.1, whereas ${q_0}^{S2}$ represents the *final* state q_0 of M_{S2} (Fig. 2.12) that is derived from the VisuaL specification shown in Fig. 2.2.

The state $\{{q_0}^{nar}, {q_0}^{S1}, {q_0}^{S2}, {q_0}^{S3}\}$ (see Fig. 3.7) contains two states each representing a final state: ${q_0}^{S2}$ and ${q_0}^{S3}$. These final states respectively belong to M_{S2} and M_{S3}, and there is no state in $\{{q_0}^{nar}, {q_0}^{S1}, {q_0}^{S2}, {q_0}^{S3}\}$ that represents a final state belonging to M_{S1}. This indicates that there is at least one strand $str \in ((L(M_{S2}) \cap L(M_{S3})) \setminus L(M_{S1}))$. That is, Set 2 (see Fig. 3.8) has at least one element. This element can be identified by following the shortest path to $\{{q_0}^{nar}, {q_0}^{S1}, {q_0}^{S2}, {q_0}^{S3}\}$. Hence, ϵ (i.e. the empty strand) is an element of Set 2. Consequently, the second part of the report is "There is at least one possible implementation of `f`, such that the implementation satisfies the constraints represented by the specifications S2 and S3, but not S1. Example sequence: the empty sequence."

The state $\{{q_1}^{S1}, {q_2}^{S2}, {q_t}^{S3}\}$ (see Fig. 3.7) contains two states each representing a final state: ${q_1}^{S1}$ and ${q_2}^{S2}$. These final states respectively belong to M_{S1} and M_{S2}, and there is no state in $\{{q_1}^{S1}, {q_2}^{S2}, {q_t}^{S3}\}$ that represents a final state belonging to M_{S3}. This indicates that there is at least one strand $str \in ((L(M_{S1}) \cap L(M_{S2})) \setminus L(M_{S3}))$. That is, Set 3 (see Fig. 3.8) has at least one element. This element can be identified by following the shortest path to $\{{q_1}^{S1}, {q_2}^{S2}, {q_t}^{S3}\}$. Hence, the strand gh is an element of Set 3. Consequently, the third part of the report is "There is at least one possible implementation of `f`, such that the implementation satisfies the constraints represented by the specifications S1 and S2, but not S3. Example sequence: <`g`, `h`>."

The state $\{{q_1}^{S1}, {q_1}^{S2}, {q_1}^{S3}\}$ (see Fig. 3.7) contains one state representing a final state: ${q_1}^{S1}$. This final state belongs to M_{S1}, and there is no state in $\{{q_1}^{S1}, {q_1}^{S2}, {q_1}^{S3}\}$ that represents a final state belonging to M_{S2} or M_{S3}. This indicates that there is at least one strand $str \in (L(M_{S1}) \setminus (L(M_{S2}) \cup L(M_{S3})))$. That is, Set 4 (see Fig. 3.8) has at least one element. This element can be identified by following the shortest path to $\{{q_1}^{S1}, {q_1}^{S2}, {q_1}^{S3}\}$. Hence, the strand g is an element of Set 4. Consequently, the fourth part of the report is "There is at least one possible implementation of `f`, such that the implementation satisfies the constraint represented by the specification S1, but not S2 or S3. Example sequence: <`g`>."

The state $\{{q_1}^{S1}, {q_t}^{S2}, {q_t}^{S3}\}$ (see Fig. 3.7) contains one state representing a final state: ${q_1}^{S1}$. This final state belongs to M_{S1}, and there is no state in $\{{q_1}^{S1}, {q_t}^{S2}, {q_t}^{S3}\}$ that represents a final state belonging to M_{S2} or M_{S3}. This indicates that there is at least one strand $str \in (L(M_{S1}) \setminus (L(M_{S2}) \cup L(M_{S3})))$. That is, Set 4 (see Fig. 3.8) has at least one element. Since this fact is already discovered during the formulation of the fourth part of the report (see above), CheckDesign skips the state $\{{q_1}^{S1}, {q_t}^{S2}, {q_t}^{S3}\}$.

At this point, despite each state (in Fig. 3.7) has already been utilized for constructing the report, the contents of Sets 5, 6, 7, and 8 (see Fig. 3.8) are still unknown, i.e. the states of the cluster DAR do not contain sufficient information for discovering the contents of Sets 5, 6, 7, and 8. Therefore, CheckDesign performs additional analysis to discover the contents of these sets, as explained below.

To discover the contents of Set 5, CheckDesign constructs a new not-minimized cluster DAR M, such that

$$L(M) = L(M_{S1}) \cap \overline{L(M_{S2})} \cap L(M_{S3})$$

If M has at least one reachable final state (i.e. $L(M) \neq \emptyset$), then Set 5 has at least one element, else Set 5 is empty. Accordingly, CheckDesign constructs the fifth part of the report.

If the states of M provide enough information for discovering the contents of the remaining sets (i.e. Sets 6, 7, and 8), then CheckDesign does not construct any other not-minimized cluster DAR; it uses the states of M for constructing the remaining parts of the report. Otherwise, CheckDesign constructs additional not-minimized cluster DARs as necessary, and eventually outputs the report that consists of 8 parts.

So far in this section, we have seen how the analysis report corresponding to three specifications is constructed. For three specifications, the complete analysis report has eight parts. In general, if there are n number of VisuaL specifications written for the same function, then there are 2^n number of sets (i.e. regions) in the venn diagram; hence a complete report has 2^n parts. Thus, the construction of a complete report does not scale up: $\Theta(2^n)$ time needs to be spent for constructing a complete report. In practice however, a partial report that is constructed based on the initial cluster DAR should usually be sufficient for understanding and resolving an inconsistency. Furthermore, after the partial report is constructed, software engineers can explicitly indicate which unknown region(s) in the Venn diagram they are interested in, so that CheckDesign can construct only the relevant missing part(s) of the report. In this way, the scalability problem of the report construction can be avoided. If the VisuaL specifications are consistent, then the partial report is already sufficient for generating the skeleton code of the function for which the specifications are written. The code generation is discussed in Section 8.2.2.

In addition to the parts explained above, an analysis report also contains the following information in a tabular form: the VisuaL specifications, the DARs corresponding to the specifications, the cluster specification, and the not-minimized and minimized cluster DARs resulting from the analysis. Hence, the analysis report can be used as a design document, and it can be archived for keeping track of the

subsequent versions of the VisuaL specifications.

3.8 Conclusions

Using VisuaL, one can create multiple specifications each representing a different constraint on the same function. When such specifications are created, it must be ensured that the specifications are *consistent*: There is at least one possible control-flow of the function, such that the control-flow satisfies each of the constraints. If there is no possible control-flow of the function that satisfies each of the constraints, then the VisuaL specifications are *inconsistent*.

Whenever VisuaL specifications are created or modified in the software life cycle, the consistency between the specifications must be verified. Manually verifying the consistency is an effort-consuming and error-prone task. If the specifications are inconsistent, then manually finding and resolving the inconsistency is an effort-consuming and error-prone task, too. CheckDesign can reduce the effort and automatically detect the errors: CheckDesign takes a set of VisuaL specifications as input, and automatically finds out, in polynomial time, whether the specifications are consistent or not. If the specifications are inconsistent, then CheckDesign outputs an error message that can help in understanding and resolving the inconsistency. Hence, CheckDesign addresses the second problem stated in Section 1.1.

Chapter 4

Operators over VisuaL Specifications

4.1 Introduction

In this chapter, we define operators for composing new VisuaL specifications from existing ones. Some of these operators are unary, whereas the others are binary. In mathematical terms, a unary operator is a function that takes a VisuaL specification as the input and outputs a VisuaL specification; whereas a binary operator takes two VisuaL specifications as the input, and outputs a VisuaL specification.

To be able to precisely define the operators, we first have to investigate some of the closure properties of ORLs. In particular, we need to answer questions such as "Is the family of ORLs closed under union? (i.e. given two ORLs L_1 and L_2, is $L_1 \cup L_2$ also an ORL?)". The answers to such questions enable us to precisely define the operators over VisuaL specifications. Therefore, we first investigate the closure properties of ORLs, in Section 4.2. Subsequently, we define a collection of operators over VisuaL specifications, in Section 4.3.

4.2 Closure Properties of ORLs

In this section, we investigate some of the closure properties of ORLs. These properties are organized under two sections: In Section 4.2.1, we investigate whether ORLs are closed under the basic set-theoretic operations: complement, union, and intersection. In Section 4.2.2 we investigate whether ORLs are closed under the

basic computation-theoretic operations: concatenation and Kleene [63].

4.2.1 Closure Under Set-Theoretic Operations

Closure Under the Complement Operation

In this section, we answer the following question: "Is the family of ORLs closed under the complement operation?" (i.e. For any given ORL L, is $\overline{L}$ also an ORL?). To answer this question, we prove the following theorem:

Theorem 4.2.1 *If L is an ORL, then $\overline{L}$ is also an ORL.*

Proof: If L is an ORL, then there is a DAR M such that $L(M) = L$. Using the construction explained in Section 3.5.1, we can construct a DAR M' such that $L(M') = \overline{L(M)}$. Since, (a) $L(M') = \overline{L(M)}$, and (b)$\overline{L(M)} = \overline{L}$, we can conclude that $L(M') = \overline{L}$. Since (a) $L(M') = \overline{L}$, and (b) $L(M')$ is an ORL, we can conclude that $\overline{L}$ is an ORL. ■

The construction of M' based on M is denoted by the function $notDAR : D \rightarrow D$, where D is the set of DARs. This function is bijective. We will use this function later in this chapter.

Closure Under the Union Operation

In this section, we find the answer to the following question: "Is the family of ORLs closed under union?" (i.e. For any two ORLs L_1 and L_2, is $L_1 \cup L_2$ also an ORL?). To answer this question, we prove the following theorem:

Theorem 4.2.2 *If L_1 and L_2 are ORLs, then $L_1 \cup L_2$ is also an ORL.*

Proof: If L_1 and L_2 are ORLs, then there are DARs M_1 and M_2 such that $L(M_1) = L_1$ and $L(M_2) = L_2$. Using the construction explained in Sections 3.4 and 3.5, we can construct a DAR M such that $L(M) = L(M_1) \cup L(M_2)$. Since (a) $L(M) = L(M_1) \cup L(M_2)$, (b) $L(M_1) = L_1$, and (c) $L(M_2) = L_2$, we can conclude that $L(M) = L_1 \cup L_2$. Since (a)$L(M) = L_1 \cup L_2$, and (b) $L(M)$ is an ORL, we can conclude that $L_1 \cup L_2$ is an ORL. ■

The construction of M based on M_1 and M_2 is denoted by the function $orDAR : D \times D \rightarrow D$, where D is the set of DARs. This function is total. We will use this function later in this chapter.

Closure Under the Intersection Operation

In this section, we answer the following question: "Is the family of ORLs closed under intersection?" (i.e. For any two ORLs L_1 and L_2, is $L_1 \cap L_2$ also an ORL?). To answer this question, we prove the following theorem:

Theorem 4.2.3 *If L_1 and L_2 are ORLs, then $L_1 \cap L_2$ is also an ORL.*

Proof: To prove this theorem, we use the equation $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$, which is known as DeMorgan's law, and the functions $notDAR$ and $orDAR$ defined earlier in Section 4.2.1.

If L_1 and L_2 are ORLs, then there are DARs M_1 and M_2 such that $L(M_1) = L_1$ and $L(M_2) = L_2$. Based on M_1 and M_2, we can construct a DAR M such that $L(M) = L(M_1) \cap L(M_2) = \overline{\overline{L(M_1)} \cup \overline{L(M_2)}}$. This construction is denoted by the function $andDAR : D \times D \rightarrow D$, where D is the set of DARs. This function is total and one-to-one, and defined as follows:

$M = andDAR(M_1, M_2) =$
$notDAR($
  $orDAR($
    $notDAR(M_1),$
    $notDAR(M_2)))$

Since (a) $L(M) = L_1 \cap L_2$, and (b) $L(M)$ is an ORL, we can conclude that $L_1 \cap L_2$ is also an ORL. ■

4.2.2 Closure Under Computation-Theoretic Operations

Closure Under the Concatenation Operation

In this section, we answer the following question: "Is the family of ORLs closed under the strand concatenation operation?" (i.e. For any two ORLs L_1 and L_2, is the set of strands obtained by concatenating a strand from L_1 with a strand from L_2 also an ORL?). To answer this question we prove the Theorem 4.2.4, which uses the following definition:

Definition: Let L_1 and L_2 be two sets of strands. $L_1 \cdot L_2$ denotes the set of strands obtained by concatenating a strand from L_1 and a strand from L_2.

Theorem 4.2.4 *If L_1 and L_2 are ORLs, then $L_1 \cdot L_2$ is also an ORL.*

Before we start proving this theorem, we need to remember that in Section 3.5.3, we have seen the construction of a DAR that is equivalent to a given NAR-ϵ. We denote this construction with the function $getDARofNAR\epsilon : N \rightarrow D$, where N is the set of NAR-ϵs, and D is the set of DARs. This function is total.

Proof: If L_1 and L_2 are ORLs, then there are DARs M_1 and M_2 such that $L(M_1) = L_1$ and $L(M_2) = L_2$. Based on the transition graphs of M_1 and M_2, we can step-by-step construct the transition graph of a DAR M such that $L(M) = L(M_1) \cdot L(M_2)$:

1. Create a state ${q_0}^{nar}$.
2. Designate ${q_0}^{nar}$ as the initial state of a NAR-ϵ M'.
3. Designate ${q_0}^{nar}$ as a non-final state.
4. Create an ϵ-labelled transition from ${q_0}^{nar}$ to the initial state of M_1.
5. From each final state of M_1, create an ϵ-labelled transition to the initial state of M_2.
6. For each final state q of M_1, designate q as a non-final state of M'.
7. Create the final state q_f of M'.
8. From each final state of M_2, create an ϵ-labelled transition to q_f.
9. For each final state q of M_2, designate q as a non-final state of M'.
10. $M = getDARofNAR\epsilon(M')$.

Since (a) $L(M_1) = L_1$, (b) $L(M_2) = L_2$, and (c) $L(M_1) \cdot L(M_2) = L(M)$, we can conclude that $L_1 \cdot L_2$ is an ORL. ■

The construction of M based on M_1 and M_2 is denoted by the function $concatDAR : D \times D \rightarrow D$, where D is the set of DARs. This function is total. We will use this function later in this chapter.

Closure under the Kleene Operation

In this section, we answer the following question: "Is the family of ORLs closed under the Kleene operation [63]?" (i.e. For any ORL L, is the set of strands obtained by concatenating zero or more strands from L also an ORL?). To answer this question we prove the Theorem 4.2.5, which is based on the following definition:

Definition: Let L be a set of strands. L^* denotes the set of strands obtained by concatenating zero or more strands from L.

Theorem 4.2.5 *If L is an ORL, then L^* is also an ORL.*

Proof: If L is an ORL, then there is a DAR M such that $L(M) = L$. Based on the transition graph of M, we can step-by-step construct the transition graph of a DAR M'' such that $L(M'') = L(M)^*$:

1. Create a state ${q_0}^{nar}$.
2. Designate ${q_0}^{nar}$ as the initial state of a NAR-ϵ M'.
3. Designate ${q_0}^{nar}$ as a non-final state.
4. Create the final state q_f of M'.
5. Create an ϵ-labelled transition from ${q_0}^{nar}$ to the initial state of M.
6. Create an ϵ-labelled transition from ${q_0}^{nar}$ to q_f.
7. From each final state of M, create an ϵ-labelled transition to q_f.
8. For each final state q of M, designate q as a non-final state of M'.
9. Create an ϵ-labelled transition from q_f to ${q_0}^{nar}$.
10. $M'' = getDARofNAR\epsilon(M')$.

Since (a) $L(M) = L$, and (b) $L(M'') = L(M)^*$, we can conclude that L^* is an ORL. ■

The construction of M'' based on M, is denoted by the function $kleeneDAR : D \rightarrow D$, where D is the set of DARs. This function is total and one-to-one. We will use this function later in this chapter.

4.3 Composition Operators over VisuaL Specifications

In this section, we define operators for composing new VisuaL specifications from existing ones. We organized these operators under two sections: In Section 4.3.1, we define the boolean operators *not*, *or*, and *and*. In Section 4.3.2, we define the temporal operators *next*, *repeatedly*, *eventually*, *until*, and *release*. These temporal operators are analogous to the temporal operators used in temporal logic [23]. The key difference of our temporal operators stem from the fact that VisuaL specifications represent properties of *finite* sequences, whereas temporal logic formulas represent properties of *infinite* sequences.

To define the operators over Visual Specifications, we use the functions defined in Section 4.2, and the functions $getDARof : V \rightarrow D$, $getVisuaLof : D \rightarrow V$, and $minimizeVisuaL : V \rightarrow V$, which are respectively defined in Sections 2.3.5, 2.6, and 2.7. Let S_1 and S_2 be two VisuaL specifications, such that the regular expression written on the container node of S_1 is equal to the regular expression written on the container node of S_2. In the remainder of this section, we define the operators using

S_1 and S_2.

4.3.1 Boolean Operators

Not

The VisuaL specification that represents "not S_1" is defined as follows:

$$
\begin{array}{l}
not(S_1) = \\
minimizeVisuaL(\\
\quad getVisuaLof(\\
\quad\quad notDAR(\\
\quad\quad\quad getDARof(S_1))))
\end{array}
$$

Or

The VisuaL specification that represents "S_1 or S_2" is defined as follows:

$$
\begin{array}{l}
or(S_1, S_2) = \\
minimizeVisuaL(\\
\quad getVisuaLof(\\
\quad\quad orDAR(\\
\quad\quad\quad getDARof(S_1), \\
\quad\quad\quad getDARof(S_2))))
\end{array}
$$

And

The VisuaL specification that represents "S_1 and S_2" is defined as follows:

$$
\begin{array}{l}
and(S_1, S_2) = \\
minimizeVisuaL(\\
\quad getVisuaLof(\\
\quad\quad andDAR(\\
\quad\quad\quad getDARof(S_1), \\
\quad\quad\quad getDARof(S_2))))
\end{array}
$$

4.3.2 Temporal Operators

Next

The VisuaL specification that represents "S_1 next S_2" is defined as follows:

$$
\begin{array}{l}
next(S_1, S_2) = \\
minimizeVisuaL(\\
\quad getVisuaLof(\\
\quad\quad concatDAR(\\
\quad\quad\quad getDARof(S_1), \\
\quad\quad\quad getDARof(S_2))))
\end{array}
$$

Repeatedly

The VisuaL specification that represents "repeatedly S_1" is defined as follows:

$$
\begin{array}{l}
repeatedly(S_1) = \\
minimizeVisuaL(\\
\quad getVisuaLof(\\
\quad\quad kleeneDAR(\\
\quad\quad\quad getDARof(S_1))))
\end{array}
$$

Eventually

To define the *eventually* operator, we first define the function $addPrefix : D \rightarrow N$, where D is the set of DARs, and N is the set of NAR-ϵs. Let M be a DAR with the abstract input alphabet Σ_a and initial state q_0. If M is given to $addPrefix$ as the input, then $addPrefix$ does the following: For each symbol $a \in \Sigma_a$, if there is no transition from q_0 to q_0 with the symbol a, then $addPrefix$ adds a new transition from q_0 to q_0 with the symbol a. The resulting NAR-ϵ is the output of $addPrefix$.

The VisuaL specification that represents "eventually S_1" is defined as follows:

$$
\begin{array}{l}
eventually(S_1) = \\
\quad getVisuaLof(\\
\quad\quad getDARofNAR\epsilon(\\
\quad\quad\quad addPrefix(\\
\quad\quad\quad\quad getDARof(S_1))))
\end{array}
$$

Until

The VisuaL specification that represents "S_1 until S_2" is defined as follows:

$$
\begin{array}{l}
until(S_1, S_2) = \\
minimizeVisuaL(\\
\quad getVisuaLof(\\
\quad\quad concatDAR(\\
\quad\quad\quad kleeneDAR(\\
\quad\quad\quad\quad getDARof(S_1)), \\
\quad\quad\quad getDARof(S_2))))
\end{array}
$$

Release

The VisuaL specification that represents "S_1 release S_2" is defined as follows:

$$
\begin{array}{l}
release(S_1, S_2) = \\
not(\\
\quad until(\\
\quad\quad not(S_1), \\
\quad\quad not(S_2)))
\end{array}
$$

4.4 Conclusions

In this chapter, we have investigated some of the closure properties of ORLs, and defined operators for composing new VisuaL specifications from existing ones.

In Section 4.2.1, we have shown that ORLs are closed under the set-theoretic operations complement, union, and intersection; and in Section 4.2.2, we have shown that ORLs are closed under the computation-theoretic operations concatenation and the Kleene operation [63].

Based on the closure properties of ORLs, we defined operators over VisuaL specifications. The operators we defined in Section 4.3.1 are the boolean operators *not*, *or*, and *and*; and the operators we defined in Section 4.3.2 are the temporal operators *next*, *repeatedly*, *eventually*, *until*, and *release*. These temporal operators are analogous to the temporal operators used in temporal logic. The key difference of our temporal operators stem from the fact that VisuaL specifications represent properties of *finite* sequences, whereas temporal logic formulas represent properties of *infinite* sequences.

Chapter 5

Checking the Consistency between Source Code and Design

5.1 Introduction

After creating consistent VisuaL specifications during the software design process explained in Section 1.4.1, a developer typically writes source code to implement the specifications, during the software implementation process explained in Section 1.4.1. For example, after creating the specification S4 (Fig. 2.4), a developer may implement the function `f` as shown in Listing 5.1.

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

Listing 5.1: 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 5.1 is inconsistent with the specification S4 (Fig. 2.4): 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 S4, seq_2 cannot be matched by this pattern. Therefore, this implementation (Listing 5.1) is inconsistent with S4, which indicates that the implementation does

not satisfy the constraint C4.

Manually finding and resolving an inconsistency between a function and a specification is an effort-consuming and error-prone task. In this Chapter, we present a tool called CheckSource that can reduce the effort and prevent the 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 inconsistent, CheckSource outputs an error message that helps in understanding and resolving the inconsistency. In this chapter, we explain how CheckSource works, in three steps, and then present the experiment we conducted for evaluating CheckSource.

Hatcliff and Dwyer [51] indicate that two of the major problems that are currently preventing the successful application of model checking technology to software are

- "**The model construction problem:** bridging the semantic gap between the artifacts produced by current verification tools. Most development is done using general-purpose programming languages (e.g. C, C++, Java, Ada), but most verification tools accept specification languages designed for the simplicity of their semantics (e.g. process algebras, state machines). In order to use a verification tool on a real program, a developer must extract an abstract mathematical model of the program's salient behavior and specify this model in the input language of the verification tool. This process is both effort-consuming and error-prone." [51]
- "**The output interpretation problem:** When a property fails when checking large models (and software systems typically produce very large models), the counter example traces produced by the checker can be hundreds even thousands of steps long. Manually matching up these counter examples is extremely tedious for several reasons. First, the length is quite long and it may require hours to walk through the trace. Second, the error trace is expressed in terms of the low-level, possibly highly optimized model representations ... Typically, one step in the source program may correspond to as many as ten steps in the low-level model representation." [51]

The empirical evidence we provide in this chapter and Chapter 7 indicates that CheckSource prevented these problems for the maintenance tasks performed by the participants of the experiments.

5.2 Step 1: Creation of Abstract Syntax Tree (AST)

If the function `f` (Listing 5.1) is given to CheckSource as input, then CheckSource parses `f`, and constructs an abstract syntax tree [9] AST_f shown at the top of Fig. 5.1. The rectangles labelled with *FDef* or *FCall* are the abstract nodes denoting a

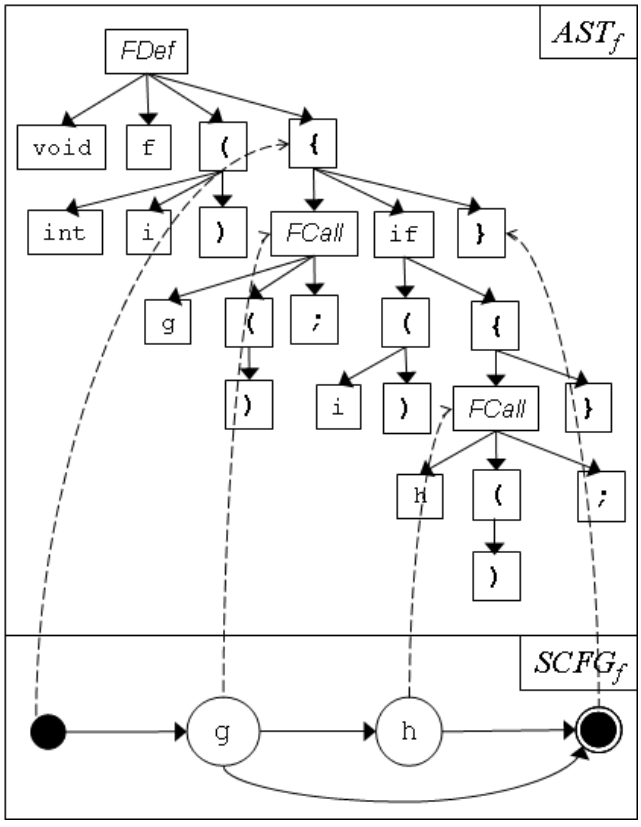

Figure 5.1: The abstract syntax tree (AST_f) and the simplified control flow graph ($SCFG_f$) of the function `f` in Listing 5.1.

function definition or a function call, respectively.

5.3 Step 2: Derivation of Simplified Control Flow Graph

Using VisuaL, one can specify constraints only on the possible sequences of function calls from C functions. Therefore, only a part of the information that is in the AST is needed during the consistency analysis. Thus, CheckSource constructs a model (of the AST) that contains only the function calls and the possible flow of control between them. We call this model **simplified control flow graph (SCFG)**, which is a 'lightweight' version of the traditional control flow graph. A formal definition of a SCFG and its relation to source code are provided in Section 5.3.1.

A SCFG decouples the analysis algorithm (explained in Section 6.7.3) from the implementation language of the programs, which is C in this case. Consequently, the implementation of the analysis algorithm does not need to be adapted if the implementation language of the programs is changed to another language in which the flow of control is explicit (i.e. imperative languages). Furthermore, the use of a SCFG enables a simpler implementation of the analysis algorithm, and a higher performance during analysis.

CheckSource traverses AST_f in the depth-first [26] manner to create $SCFG_f$ depicted at the bottom of Fig. 5.1. The black dot on the left represents the **initial node**, which denotes the beginning of `f`. The `g`-labelled circle represents an **internal node** that denotes the call to `g`. The `h`-labelled circle represents an internal node that denotes the call to `h`. The circled black dot represents the **final node**, which denotes the end of `f`, and the arrows between these shapes represent the possible flow of control between the entities denoted by the nodes. As visualized by the dashed arrows, CheckSource maintains a one-to-one mapping from the nodes of $SCFG_f$ to the related nodes of AST_f.

5.3.1 Simplified Control Flow Graph

Let f be a function definition in the C programming language.

$callsFrom(f)$ denotes the set of function calls whose static scope [79] is the definition of f. For example, let g and f respectively denote the calls to the functions `g` and `h` in Listing 5.1, then $callsFrom(\texttt{f}) = \{g, h\}$.

Let G be the control flow graph [38] derived from the definition of f. $firstCall(c, f)$ denotes that c is a first function call according to G. For example, $firstCall(g,\texttt{f})$ is true.

$nextCall(c_1, c_2, f)$ denotes that c_2 is a next function call after function call c_1, according to G. For example, $nextCall(g, h, \mathtt{f})$ is true.

$lastCall(c, f)$ denotes that c is a last function call according to G. For example, $lastCall(h, \mathtt{f})$ is true.

$noCall(f)$ denotes that there is at least one path [38] p in G such that $\nexists c(c \in callsFrom(f) \wedge c \in p)$.

$name(c)$ denotes the identifier [79] of the C function one of whose call is c. For example, $name(g) = \mathtt{g}$

$SCFG_f$ denotes the **simplified control flow graph** of f, which is a tuple $\langle V = \{\nu_0\} \cup V_I \cup \{\nu_F\}, E = E_0 \cup E_I \cup E_F \rangle$, where

ν_0 is the **initial node**.

V_I is a finite set of **internal nodes** such that $\nu_0 \notin V_I$, and a bijection $nodeOfCall : callsFrom(f) \rightarrow V_I$ exists. The label of any $\nu \in V_I$ is $name(nodeOfCall^{-1}(\nu))$.

ν_F is the **final node** such that $\nu_F \neq \nu_0$ and $\nu_F \notin V_I$.

$E_0 \subseteq \{\nu_0\} \times (V_I \cup \{\nu_F\})$ is the set of **initial edges** such that $\forall c(firstCall(c, f) \Rightarrow (\nu_0, nodeOfCall(c)) \in E_0)$, and $noCall(f) \Rightarrow (\nu_0, \nu_f) \in E_0$.

$E_I \subseteq V_I \times V_I$ is a set of **internal edges** such that $\forall \nu_1, \nu_2, c_1, c_2(nodeOfCall(c_1) = \nu_1 \wedge nodeOfCall(c_2) = \nu_2 \wedge nextCall(c_1, c_2, f) \Rightarrow (\nu_1, \nu_2) \in E_I)$.

$E_F \subseteq (V_I \cup \{\nu_0\}) \times \{\nu_F\}$ is the set of **final edges** such that $\forall c(lastCall(c) \Rightarrow (nodeOfCall(c), \nu_F) \in E_F)$, and $noCall(f) \Rightarrow (\nu_0, \nu_f) \in E_F$.

$SCFG_f$ can be constructed by traversing the abstract syntax tree (AST) [9] rooted at the signature of f.

A **simplified control flow path (SCFP)** p of f is a finite path in $SCFG_f$ such that ν_0 is the first node lying on p, and ν_f is the last node lying on p.

5.4 Step 3: Analysis of Simplified Control Flow Graph with respect to VisuaL Specification

To verify that the function `f` is consistent with the specification S4 (Fig. 2.4), CheckSource has to find out whether all possible sequences of function calls from `f` are matched by the pattern depicted in Fig. 2.4. To generate the function call sequences, CheckSource traverses $SCFG_f$ in a depth-first manner. As understand-

able from $SCFG_f$, there are two possible sequences of function calls: seq_1 and seq_2, both of which are already presented at the beginning of Section 5. The analysis of seq_1 reveals that seq_1 terminates at q2 (see Fig. 2.4). Since this node has the stereotype <<final>>, seq_1 is matched by the pattern. The analysis of seq_2 reveals that seq_2 terminates at q1 (see Fig. 2.4). Since this node does not have the stereotype <<final>>, seq_2 is not matched by the pattern, which means f is inconsistent with S4, and f does not satisfy the constraint C4. Consequently, CheckSource outputs seq_2, which is useful for understanding and resolving the inconsistency. If there were no inconsistency, CheckSource would output a success message.

In this chapter, we are assuming that functions terminate upon execution. Hence, the possible sequences of function calls from a given function must be finite. Given this fact, the verification algorithm can be explained as follows: Let *ptrn* denote a pattern that represents a constraint *cns*. *ptrn* can be interpreted as a deterministic finite state automaton [63] that accepts a set S_{ptrn} of finite sequences. Let S_f denote the set of possible sequences of function calls from f. Note that S_f can be computed by traversing $SCFG_f$. f **satisfies** *cns* if and only if $S_f \subseteq S_{ptrn}$. A mathematical explanation of the analysis algorithm and its asymptotic time complexity (polynomial) are provided in Section 5.4.1.

5.4.1 The Analysis Algorithm of CheckSource

Sections 2.3.4 and 5.3.1 are the prerequisites for this section.

The algorithm of CheckSource is denoted by the function $checkSource : SG \times D \rightarrow \{true, false\} \times \Upsilon^*$, where SG denotes the set of simplified control flow graphs, D denotes the set of DARs, and Υ^* denotes the set of strands.

Let $SCFG = \langle V = \{\nu_0\} \cup V_I \cup \{\nu_F\}, E\rangle$ be a simplified control flow graph, and $M = \langle Q, \Sigma_a = \Sigma_b \cup \{\#\}, \delta, q_0, F, \Xi, \eta\rangle$ be a DAR. If $SCFG$ and M are given to *checkSource* as the input, then *checkSource* performs the following steps:

1. Define a boolean variable *result* whose value is *true*.
2. Define a strand variable *counterExample* whose value is ϵ.
3. Evaluate $result \leftarrow analyze(q_0, \nu_0, \&counterExample)$. The function *analyze* is defined later in this section. $\&counterExample$ denotes the pointer of *counterExample*.
4. Return $(result, counterExample)$.

The analysis algorithm is denoted by the function $analyze : Q \times V \times SP \rightarrow \{true, false\}$, where SP denotes the set of strand pointers. If $q \in Q$, $\nu \in V$, and $sp \in SP$ are given as the input to *analyze*, then *analyze* performs the following

steps:

1. Define a boolean variable *result* whose value is *true*.
2. If q has not been already mapped to ν, then perform the following steps:
   (a) append $name(nodeOfCall^{-1}(\nu))$ to $*sp$, where $*sp$ denotes the strand variable whose pointer is sp.
   (b) map q to ν.
   (c) If $q = q_t$, then assign $false$ to $result$.
   (d) If $\nu = \nu_F$ and $q \notin F$, then assign $false$ to $result$.
   (e) While $result$ is $true$, iterate over the outgoing edges $(\nu, \nu_i) \in E$ of ν, and perform the following steps at each iteration:
      i. Create a clone c of $*sp$.
      ii. If $name(nodeOfCall^{-1}(\nu_i)) \in \Sigma_b$, then assign $analyze(\nu_i, \delta(q, name(nodeOfCall^{-1}(\nu_i))), \&c)$ to $result$, else assign $analyze(\nu_i, \delta(q, \#), \&c)$ to $result$.
      iii. If $result$ is $true$, then destroy c, else first destroy $*sp$, and next assign $\&c$ to sp.
3. Return $result$.

If the value of $checkSource(SCFG, M) = (false, str)$, then str is a counter-example showing that the $SCFG$ does not satisfy M. Otherwise, $SCFG$ satisfies M.

The asymptotic time complexity of *checkSource* is $O(|E| \times (|Q| \times |\Sigma_a|))$, which is determined by the Step 1.e of the *analyze* function.

5.5 Experiment Definition and Planning

In this section, we present the definition and planning of the experiment we conducted for evaluating CheckSource. For preparing, conducting, and documenting the experiment, we followed the guidelines proposed by Kitchenham et. al. [59] and Wohlin et. al. [86].

5.5.1 Background Information

In the controlled experiment, we decided to use real-life C functions and real-life VisuaL specifications. Therefore, we trained a software engineer of ASML, so that he can create some VisuaL specifications corresponding to some of the functions of the software component that he maintains. After the training, the developer selected three functions, and created one VisuaL specification per selected function.

He was motivated to create the specifications, because he needed to understand and document the possible sequences of function calls from these three functions, so that he can maintain these functions.

5.5.2 Motivation and Overview

The purpose of this experiment is to evaluate the effect of CheckSource on the cost of maintaining source code to eliminate inconsistencies between source code and VisuaL specifications. 27 M.Sc. computer science students from the University of Twente participated in this experiment. The participants worked with three C functions selected by the domain expert (see Section 5.5.1), and the corresponding specifications that were created by the expert using VisuaL. We injected an inconsistency defect into each of the selected functions, by removing a function call from each of the selected functions. We requested the participants to repair these defects by modifying the functions, such that each function would become consistent with the corresponding VisuaL specification.

5.5.3 Hypotheses

We formulated the following hypotheses to be tested in this experiment:

- $H_0{}^1$: The tool CheckSource does not have any effect on the amount of effort spent by M.Sc. students.
- $H_0{}^2$: The tool CheckSource does not have any effect on the number of errors made by M.Sc. students.

We chose 0,05 as the significance level for rejecting the hypothesis above.

5.5.4 The Variables of the Experiment

Factors

- **Tool support** (i.e. existence of CheckSource) is the only factor of this experiment. This factor is measured in the nominal scale, at two levels: exists, not exists.

Non-factor Independent Variables

There are two independent variables that we kept at fixed levels in this experiment. The first one is the function-specification pair, and the second one is the injected defect. Below, we explain these variables in detail.

- **Function-Specification Pair** is an independent variable kept at a fixed level:

Each participant was treated with the same set of three C functions and the corresponding VisuaL specifications. We measured the size and cyclomatic complexity [67] of both the functions and the specifications.

For a given function, the size is measured by counting the physical lines of code, and the complexity is measured by calculating the cyclomatic complexity number. In Table 5.1, the size and complexity of the three functions are listed. These functions

Table 5.1: The size and complexity of the C functions.

Functions	**# Lines of Code**	**Cyclomatic Complexity**
Function1	88	20
Function2	127	27
Function3	280	51

are originally located in a file that has 55 functions. This file is one of the several files in the software component mentioned in Section 5.5.1. For a better understanding of the characteristics of the functions in this file, the descriptive statistics about the 55 functions can be found in Table 5.2. Based on these statistics, one can realize the

Table 5.2: Descriptive statistics of the 55 functions in the file.

	Avg.	**Min.**	**Max.**	**Std. Dev.**
Lines of Code	133	24	390	89
Cyclomatic Complexity	28	4	114	20

following: Compared to the other functions in the file, Function1 is relatively small and simple; Function2 has average size and complexity; and Function3 is relatively large and complex.

For a given VisuaL specification, the size is measured by counting the nodes and the edges, and the complexity is measured by calculating the cyclomatic complexity number. In Table 5.3, the size and complexity of the three specifications are listed. These specifications were created by the domain expert at ASML. Specification1, Specification2, and Specification3 respectively corresponds to Function1, Function2, and Function3.

Table 5.3: The size and complexity of the VisuaL specifications.

Specifications	# Nodes	# Edges	Cyclomatic Complexity
Specification1	11	19	10
Specification2	11	23	14
Specification3	10	20	12

- **Injected defect** is an independent variable kept at a fixed level:

We injected the same kind of defect into each of the three functions: We removed the first possible function call to inject an inconsistency between the function and the corresponding VisuaL specification.

Dependent Variables

There are two dependent variables in this experiment:

- Amount of **effort** is a dependent variable measured in the ratio scale. We measure this variable in terms of minutes.
- Number of **errors** is a dependent variable measured in the absolute scale.

5.5.5 Selection of Participants

This experiment was an integral part of the 2007 spring semester Software Management course at the University of Twente. Hence, the students of this course participated in the experiment. These students were M.Sc. computer science students.

To collect some information about the software development experience of these students, we asked them the size of the largest computer program they have written using one of the imperative languages (e.g. C, Java). The students had to select one of the following answers:

1. Less than 100 lines of source code
2. More than 100, less than 1000 lines of source code
3. More than 1000, less than 5000 lines of source code
4. More than 5000, less than 10000 lines of source code
5. More than 10000 lines of source code

No student selected 1, no student selected 2, eight students selected 3, eight students selected 4, and eleven students selected 5. None of the students had any previous

experience about the instruments listed in Section 5.5.7.

5.5.6 Experiment Design

As visible in Fig. 5.2, we designed an experiment that has one factor and two levels. The factor and its levels are already explained in Section 5.5.4. Each level of the

Factor: Tool Support	
Level: Exists	Level: Not Exists
14 Students	13 Students

Figure 5.2: The experiment has one factor with two levels each of which is one of the two treatments. The number of participants per treatment in each of the experiments is also shown in this figure.

factor is a treatment in this experiment.

The participants were randomly assigned to one of the two treatments (i.e. there were two independent groups of participants). We balanced the design by assigning (almost) equal number of participants per treatment. In the remainder of this chapter, we will use **tool-supported participant** for referring to a participant treated with the tool support, and **manual participant** for referring to a participant treated without tool support.

5.5.7 Instrumentation

The instruments of this experiment are

- the C functions into which we injected defects,
- the VisuaL specifications,
- the tool using which the participants repaired the defects (i.e. CheckSource),
- the tutorial slides that we presented to the participants to train them for repairing the defects,
- the documents containing the stepwise instructions for the participants to repair the defects, and
- the facilitating software that we developed for automatic data collection.

Interested readers can request the instruments from us by providing personal details and affiliation. If ASML approves the request, then we can send a non-disclosure

agreement (NDA). After the NDA is signed and returned, we can provide the instruments.

5.6 Experiment Operation

The operation phase of the experiment consisted of three steps: preparation, execution, and data validation. In this section, we explain these steps in detail.

5.6.1 Preparation

We prepared a tutorial for teaching the participants how to (a) interpret the specifications, (b) relate the specifications to the source code, and (c) repair the defects in the source code using the specifications. For the tool-supported participants, the tutorial also included how to use the tools. We presented this tutorial before the experiment as a slide show, and we distributed hard copies of the slides to the participants, after the presentation.

We prepared step-wise instructions for the participants. By following these instructions, a participant could find the source code in the directory structure of the computer, run the tools, etc.

We implemented facilitating software that puts a time stamp on the source code modified by a participant, and logs the source code in a file. The manual participants ran this software twice: once at the beginning of the treatment, and once at the end of the treatment. The facilitating software was integrated with the tool support (i.e. CheckSource). Consequently, the tool-supported participants ran the facilitating software at least twice: once at the beginning of the treatment (i.e. when they initially used the tool to find and understand the defects), once at the end of the treatment (i.e. after they modified the source code), and zero or more times during the treatment (i.e. each additional time they used the tool to see whether they could successfully repair the defects).

We prepared an example treatment for the participants, so that they get used to the tasks they are required to perform. In this way, we aimed at improving the accuracy of our measurements, by decreasing the learning overhead in the actual treatments. The example treatment was the first treatment of each participant.

We conducted preliminary runs of the experiment to test the artifacts explained above. These runs enabled us to improve the instruments of the experiment. The four participants of these preliminary runs were different than the participants of

the actual experiment. During the analysis presented in Section 5.7, we excluded the data of the preliminary runs.

To motivate the students for performing the tasks as carefully and quickly as they can, we rewarded the first, second, and the third best performers in each of the tool-supported and manual groups with 50 EUR, 40 EUR, and 30 EUR, respectively. The ranking criteria was performing the tasks with least number of errors in least amount of time, where the number of errors had priority over the amount of time. Besides the top three prizes, each student received 10 EUR for his participation. Before the students started the experiment, we informed them about the prizes and the ranking criteria. The results of the students were kept anonymous, and these results did not have any impact on their course grade.

5.6.2 Execution

During the experiment, the students worked at the computer laboratories of the university, and they used the computers in the laboratories to modify source code, and to run the tools.

To ensure the independence of the observations, each student participated in the experiment at the same time. This required an instructor to give the tutorial for the tool-supported group in a laboratory, and another instructor to give the tutorial for the manual group in another laboratory. Moreover, the instructors and two additional assistants were present at the laboratories.

5.6.3 Data Validation

As explained in Section 5.6.1, each participant ran a facilitating software that logs the source code with a time stamp. The participants were not authorized to modify the clock of the operating system.

To validate the data contained in the files, we compared the latest time stamp in a file with the last modified time of the file. If they were different, this would indicate that the participant had manually modified the file, hence the data is invalid. In this way, we found four invalid log files, and we did not include their data in the analysis.

We informed each participant about his result, and asked whether the result is as he expected; each participant informed us that his result is as he expected. This supports the claim that the participants have understood the instructions, and followed

them properly (i.e. this is a positive indication about the validity of data).

5.7 Data Analysis

By investigating the log files created during the experiment, we realized that each tool-supported participant worked until the CheckSource gave no more error messages. Therefore, after a tool-supported participant finished a treatment, the resulting source code was consistent with the corresponding VisuaL specifications. On the other hand, the manual participants made errors while repairing the inconsistencies. To calculate the number of errors, we counted the minimum number of function calls that has to be added or removed for repairing the inconsistency.

The raw data of the experiment is provided in Appendix B.1. In the remainder of this section, we analyze the data in three steps: First, we discuss the screening and cleaning of the raw data, second we present the descriptive statistics of the clean data, and third we present the statistical tests we applied to the hypotheses stated in Section 5.5.3.

We used SPSS Version 12.0.1 for Windows [7] for analyzing our data, and testing the hypotheses.

5.7.1 Screening and Cleaning the Data

Our investigations on the log files revealed that the logged data of two tool-supported and two manual student were manually modified (i.e. corrupted). We understood this by comparing the time stamps in the files with the last modified time of the files. Therefore, we excluded their data from our calculations.

5.7.2 Descriptive Statistics

In Fig. 5.3, the descriptive statistics of the data collected from the experiment is presented. Since each tool-supported student worked until the CheckSource gave no more error messages, the descriptive statistics of the number of errors in the existence of tool support is omitted in Fig. 5.3.

The mean amount of effort spent by the tool-supported students is 14 minutes[1],

[1]Wherever it is appropriate, we present rounded numbers for increasing the readability of the text. More accurate numbers are presented in the figures. For example, this number (i.e. 14) is

	Tool Support			Statistic	Std. Error
Effort	Exists	Mean		14,17	2,377
		95% Confidence Interval for Mean	Lower Bound	8,94	
			Upper Bound	19,40	
		5% Trimmed Mean		13,69	
		Median		13,00	
		Variance		67,788	
		Std. Deviation		8,233	
		Minimum		4	
		Maximum		33	
		Range		29	
		Interquartile Range		9	
		Skewness		1,144	,637
		Kurtosis		1,433	1,232
	Not Exists	Mean		34,18	3,655
		95% Confidence Interval for Mean	Lower Bound	26,04	
			Upper Bound	42,33	
		5% Trimmed Mean		34,31	
		Median		33,00	
		Variance		146,964	
		Std. Deviation		12,123	
		Minimum		12	
		Maximum		54	
		Range		42	
		Interquartile Range		16	
		Skewness		-,230	,661
		Kurtosis		-,291	1,279
Errors	Not Exists	Mean		1,64	,544
		95% Confidence Interval for Mean	Lower Bound	,42	
			Upper Bound	2,85	
		5% Trimmed Mean		1,54	
		Median		2,00	
		Variance		3,255	
		Std. Deviation		1,804	
		Minimum		0	
		Maximum		5	
		Range		5	
		Interquartile Range		3	
		Skewness		,667	,661
		Kurtosis		-,730	1,279

Figure 5.3: The descriptive statistics of the data collected from the experiment. The data consists of effort measured in minutes, and the number of errors. Since the number of errors is constant when the tool support exists, the related statistics is omitted in this figure.

whereas the mean amount of effort spent by the manual students is 34 minutes.

presented as 14,17 in Fig. 5.3.

		Kolmogorov-Smirnov[a]			Shapiro-Wilk		
	Tool Support	Statistic	df	Sig.	Statistic	df	Sig.
Effort	Exists	,175	12	,200*	,914	12	,238
	Not Exists	,168	11	,200*	,976	11	,937
Errors	Not Exists	,272	11	,022	,841	11	,032

*. This is a lower bound of the true significance.

a. Lilliefors Significance Correction

Figure 5.4: The results of the normality tests for the data collected from the experiment. Since the number of errors is constant when the tool support exists, the related statistics is omitted in this figure.

Hence, we can conclude that the tools reduced the effort spent by an average student approximately by 60% in this experiment.

The mean number of errors made by the tool-supported students is 0, whereas the mean number of errors made by the manual students is 2. Since each participant worked with 500 lines of source code in total (see Table 5.1), we can conclude that the tools prevented approximately one error per $500 \div 2 = 250$ lines of source code in this experiment.

Note that the 5% trimmed means (i.e. the means calculated upon excluding 5% of the data at the extremes) are very close to the original means. For instance, the original mean of the amount of effort in the existence of tool support is 14 minutes, and the corresponding trimmed mean is 13 minutes. Due to the closeness of each trimmed mean to the corresponding original mean, we can conclude that the extreme values of the dependent variables do not have a strong influence on the original means.

The positive skewness of the effort in the existence of tool support (1,144) indicates that the majority of the tool-supported students spent less than 14 minutes during the experiment. The negative skewness of the effort in the lack of tool support (-0,230) indicates that the majority of the manual students spent more than 34 minutes during the experiment.

The positive value of Kurtosis of the effort in the existence of tool support indicates that the distributions of the values are relatively peaked (i.e. clustered in the center), with long thin tails. The negative values of Kurtosis of the errors indicate that the distributions of the values are relatively flat (i.e. too many values at the extremes).

In Fig. 5.4, the results of the normality tests for the data collected from the experiment is shown. The values of the effort in both the existence and non-existence of the tool support are very likely to be normally distributed, because the significance values are greater than 0,05. The values of the errors are less but still likely to be

normally distributed, because the significance values are less than 0,05 but greater than 0,01.

In Figures 5.5 and 5.6, the box plots of the amount of effort versus tool support, and the number of errors versus tool support are respectively shown. The grey

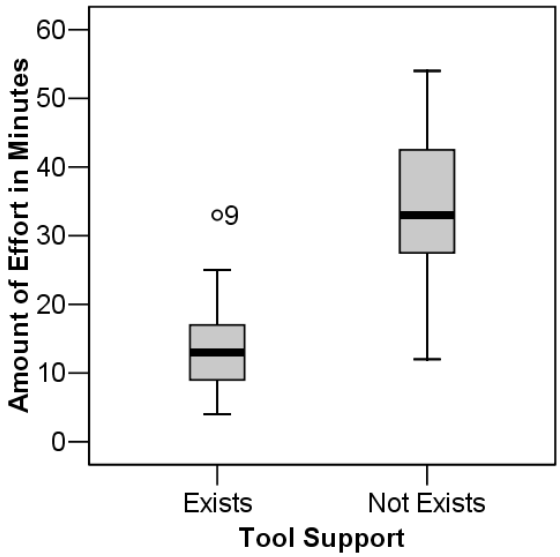

Figure 5.5: Box plot of effort vs. tool support in the experiment.

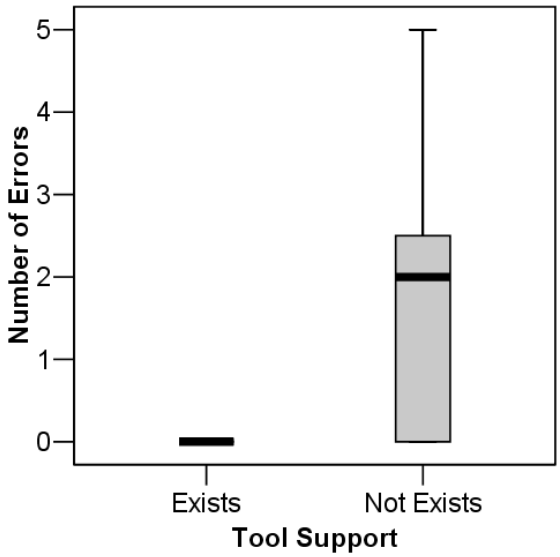

Figure 5.6: Box plot of errors vs. tool support in the experiment.

		Levene's Test for Equality of Variances		t-test for Equality of Means						
									95% Confidence Interval of the Difference	
		F	Sig.	t	df	Sig. (2-tailed)	Mean Difference	Std. Error Difference	Lower	Upper
Effort	Equal variances assumed	2,552	,125	-4,668	21	,000	-20,015	4,287	-28,931	-11,099
	Equal variances not assumed			-4,591	17,414	,000	-20,015	4,360	-29,197	-10,833
Errors	Equal variances assumed	32,514	,000	-3,149	21	,005	-1,636	,520	-2,717	-,556
	Equal variances not assumed			-3,008	10,000	,013	-1,636	,544	-2,848	-,424

Figure 5.7: The results of the independent samples t-test for assessing the differences between the tool-supported and manual students.

rectangles represent 50% of the values, with the whiskers (i.e. the lines below and above the rectangles) going to the minimum and the maximum values. As visible in Fig. 5.5, SPSS detected only one outlier. This outlier, which is 33 minutes, is the amount of effort spent by the student number nine (see Table B.3).

5.7.3 Hypothesis Testing

For testing the hypotheses stated in Section 5.5.3, we used the independent-samples t-test provided by SPSS. The assumptions for using the t-test hold in our experiment: each dependent variable is measured in the ratio scale (see Section 5.5.4); each participant is randomly assigned to either the tool-supported or the manual group (see Section 5.5.6); the observations made during the experiment are independent of each other (see Section 5.6.2); it is likely that the dependent variables (i.e. amount of effort and the number of errors) have a normal distribution (see Section 5.7.2).

Testing $H_0{}^1$

An independent-samples t-test was conducted to compare the amount of effort spent by the tool-supported students versus the manual students (see Fig. 5.7). Since the significance value of Levene's test (0,125) is greater than 0,05, the equality of variances is assumed (i.e. the first row in Fig. 5.7 is considered). There was a significant difference in the amount of effort spent by the tool-supported students (Mean = 14; Std. Dev. = 8) and the manual students (Mean = 34; Std. Dev = 12; t(21) = -4,67; p = 0,01). Therefore, we can reject $H_0{}^1$.

Testing $H_0{}^2$

An independent-samples t-test was conducted to compare the number of errors made by the tool-supported students versus the manual students (see Fig. 5.7). Since the significance value of Levene's test (0,000) is less than 0,05, the equality of variances is not assumed (i.e. the fourth row in Fig. 5.7 is considered). There was a significant difference in the number of errors made by the tool-supported students (Mean = 0; Std. Dev. = 0) and the manual students (Mean = 1,64; Std. Dev = 1,8; t(10) = -3; p = 0,05). Therefore, we can reject $H_0{}^2$.

5.8 Validity Evaluation

In this section, we discuss the threats to the validity of the experiment. We organized these threats using the categorization proposed in [24]; each title in this section is a category of validity threats. For each category, we first provide a short explanation, and then discuss how we addressed this category of threats in our experiment. Most of the short explanations are adopted from [86].

5.8.1 Conclusion Validity

This category of threats effect the ability to draw correct conclusion about the relation between the treatment and the outcome of the experiment.

In our experiment, we identified two categories of threats to the conclusion validity: low statistical power, and reliability of treatment implementation [24].

Low Statistical Power

The power of a statistical test is the ability of the test to reveal a true pattern in the data. If the power is low, then there is a high risk that an erroneous conclusion is drawn. Therefore, we performed a post-hoc analysis to find the actual power we achieved in our statistical tests. We used G*Power [37] for calculating the power. For each hypothesis, the input and output parameters of the power analysis are listed in Fig. 7.12.

As visible in the last row, second column of Fig. 7.12, the power we achieved for the first hypothesis is more than 0,80. Since 0,80 is the commonly accepted minimum level of power, we can conclude that the power level of our analysis is not a major

	Hypotheses	
	$H_0{}^1$	$H_0{}^2$
Input		
μ_1	14,17	1,64
μ_2	34,10	0,000001
σ_1	8,233	1,804
σ_2	12,129	1
d	1,961161	1
α	0,01	0,05
n_1	11	11
n_2	11	11
Output		
δ	4,599331	2,345208
Critical t	2,845340	2,085963
Df	20	20
$1-\beta$	0,948500	0,607098

Figure 5.8: The input and output parameters of the post-hoc power analysis for independent-samples and two-tailed t-tests (per hypothesis). In this analysis, the equality of the sample sizes is assumed. μ_1 and μ_2 denote the means of the first and second samples. σ_1 and σ_2 denote the standard deviations of the first and second samples. d denotes the effect size calculated by G*Power based on the means and standard deviations. α denotes the significance level we have chosen for rejecting the null hypotheses. n_1 and n_2 denote the sizes of the samples. δ denotes the non-centrality parameter calculated by G*Power. *Critical t* denotes the critical t-value calculated by G*Power. Df denotes the degree of freedom. $1-\beta$ denotes the power we achieved in our statistical analysis.

threat to the validity of our conclusions related to the first hypothesis. The power we achieved also indicates that the number of participants was quite sufficient for testing the first hypothesis.

As visible in the last row, third column of Fig. 7.12, the power we achieved for the second hypothesis is less than 0,80. Therefore, the lack of sufficient power is a threat to the validity of our conclusions related to the second hypothesis. In the power calculation, G*Power required us to provide non-zero and positive values for the means and standard deviations. We could not give the real values of (a) the

mean number of errors (i.e. μ_2 of ${H_0}^2$) and (b) the standard deviation of the errors (i.e. σ_2 of ${H_0}^2$) of the tool-supported participants; because these values are all 0. Nevertheless, we approximated these values by giving the lowest possible values for the mean number of errors (i.e. 0,000001) and the standard deviation of errors (i.e. 1), regarding the tool-supported participants. This approximation can be considered as a threat to the validity of the conclusions drawn from the hypotheses ${H_0}^2$.

Note that, the threat mentioned above does not arise from the preparation, design, or operation of the experiments; it arises from the outcome of the experiments. Therefore, this threat could not be predicted and avoided before the experiment was conducted. In principle, the manual participants could have repaired all the defects, in which case the number of errors of manual participants would have been 0; or the tool-supported participants could have made some errors, in which case the number of errors of tool-supported participants would have been non-zero. The fact that the tool-supported participants did not make any errors is the outcome of the experiment.

Reliability of Treatment Implementation

The implementation of a treatment means the application of the treatment to a subject. To improve the reliability of treatment implementation, the implementation must be as standard as possible over different participants and occasions.

In the experiment with the students, each student participated in the experiment at the same time. This was important to avoid information exchange between the students, hence to prevent the threat explained in Section 5.8.2. Consequently, this required an instructor to give the tutorial for the tool-supported group in a laboratory, and another instructor to give the tutorial for the manual group in another laboratory. Since different instructors gave the tutorial, there may be a threat to the reliability of treatment implementation.

5.8.2 Internal Validity

Internal validity threats are issues that can affect the measurements of the independent variable, without the researcher's knowledge. Therefore, these kinds of threats may influence the validity of conclusions about a possible causal relationship between a treatment and the corresponding outcome.

In our experiment, we identified and addressed three types of threats to the internal validity: maturation, instrumentation, and diffusion or imitation of treatments [24].

Maturation

The maturation threat arises when subjects are affected negatively (e.g. tired or bored), or positively (unintended learning) during the experiment.

To reduce the unintended learning effect in our experiment, we prepared an example (i.e. preliminary) treatment for the participants, so that they got used to the tasks they were required to perform. In this way, we aimed at improving the accuracy of our measurements. The example treatment was the first treatment of each participant, and the related data is excluded during the analysis presented in Section 5.7.

Instrumentation

This type of threat arises from an improper design of instruments such as data collection forms, document to be inspected in an inspection experiment, etc.

We conducted preliminary runs of the experiment to test the quality of the instruments listed in Section 5.5.7. These runs enabled us to improve the quality of these instruments. The four participants of these preliminary runs were different than the participants of the actual experiment. During the analysis presented in Section 5.7, we excluded the data of the preliminary runs.

Diffusion or Imitation of Treatments

This threat arises if participants are prematurely informed about the treatments, and behave differently due to this information. As explained in Section 5.6.2, we avoided this threat in the experiment.

5.8.3 Construct Validity

Threats to construct validity influence the ability to draw correct conclusions about the relation between the results of the experiment and the hypotheses that are being tested using these results. Some of such threats are related to the experimental design, and others are related to social factors.

In our experiment, we identified and addressed two types of threats to the construct validity: confounding constructs and levels of constructs, and experimenter expectancy [24].

Confounding Constructs and Levels of Constructs

These kinds of threats arise from the fact that there are confounding constructs (e.g. experience of subjects) that are not taken into account in an experiment.

As explained in Section 5.5.5, we measured the programming experience of the students to understand their background. However, we did not balance the tool supported v.s. manual groups according to the experience of the students, because we did not have any means to validate their programming experience. Instead, we divided them randomly. As a result, in the tool supported group there were six students with experience level 5 (see Section 5.5.5), five students with experience level 4, and three students with experience level 3. Whereas, in the manual group there were five students with experience level 5, three students with experience level 4, five students with experience level 3. The lack of balance in the experience may be a threat to the validity of the results related to the students. However, we do not think that this threat is severe, because the weighted average of the experience in the tool-supported and manual groups were not too different (i.e. respectively 4,2 and 4).

Experimenter Expectancy

The experimenters may bias the result of an experiment based on what they expect from the experiment. This is a threat to the construct validity.

The purpose of our experiment was to evaluate the tools developed by the author of this thesis. Hence, the experimenter expected that the tools are beneficial. To eliminate this threat, we planned, conducted, and analyzed this experiment together with Klaas van den Berg, who did not have any specific expectations from this experiment.

5.8.4 External Validity

The threats to external validity limit the ability to generalize the results of the experiment.

Interaction of Selection and Treatment

This threat arises if the selection of subjects do not adequately represent the population for which the results need to be generalized.

The participants of this experiment are not randomly selected from a large population of students. The students were the participants of a course at the university. Therefore, the results of this experiment cannot be generalized for a larger population of students or professional developers. However, this does not devaluate the results of this experiment, because our purpose was to evaluate the tools, and we have empirical evidence that the tools are beneficial for a homogenous set of students.

Interaction of Setting and Treatment

This threat arises if the experimental setting or the instruments are not representative of, for example, industrial practice.

In our experiments, we used real-life source code and real-life VisuaL specifications, but we injected relatively simple defects. Below, we explain why we could not use real-life evolution scenarios instead of injecting simple defects.

At ASML, developers maintain source code upon receiving a "change request / problem report (CRPR)". Implementing a CRPR typically involves several modifications to the existing source code. Hence, a real evolution scenario typically consists of several additions, deletions, and modifications of function calls, control statements, variables etc. Using a real CRPR in our experiments was infeasible, because

- A CRPR is informally written in English. Therefore, different participants might have (mis)interpreted the CRPR differently. Consequently, we would have lost the control in the experiment, and the results would have been inconclusive.
- No matter there is tool support or not, domain expertise is necessary for implementing a CRPR. Hence, the students could not have implemented the CRPRs.
- The implementation of a CRPR involves multiple changes to the source code. The changes that were not due to the inconsistencies between source code and VisuaL specifications would have been confounding factors in our experiment. In other words, we would have lost control in the experiment, and the results would have been inconclusive.
- Due to the domain expertise required for implementing a CRPR, we cannot estimate how much time is necessary for an average person to implement a given CRPR. Since we could not occupy the participants for more than 3 hours during the experiment, we could not have used a real CRPR.

Due to the reasons listed above, we had to inject relatively simple defects that can be repaired without having any domain expertise. This may be a threat to the

generalization of our results to the industrial practice. Frankl et. al. [40] discuss the challenges in creating defect models that enable systematic investigation of reliability in software systems.

The functions that were used during our experiments were chosen by the domain expert of the component in which the functions were placed. The expert chose these functions, because on that day he had to maintain them. Hence, the functions were not randomly chosen. This is a threat to the generality of our results to arbitrary functions.

5.9 Conclusions

A function and a corresponding specification may be inconsistent with each other. Manually finding and resolving such an inconsistency is an effort-consuming and error-prone task. CheckSource can reduce effort and detect errors. CheckSource takes a function and a corresponding VisuaL specification as the input, and automatically finds out, in polynomial time, whether the function and specification are consistent or not. To determine if a specification and a function are consistent, CheckSource first parses the function and creates an abstract syntax tree. Second, CheckSource derives the simplified control-flow graph of the function by traversing the abstract syntax tree. Finally, CheckSource finds out whether each possible path in the control flow graph satisfies the constraint expressed in the VisuaL specification. If there is at least one possible path that does not satisfy the constraint, then the function and the specification are inconsistent. If there is an inconsistency, CheckSource outputs an error message containing an example path that does not satisfy the constraint. This error message helps in understanding and resolving the inconsistency. In this way, CheckSource addresses the third problem stated in Section 1.1. The results of the controlled experiment we conducted for evaluating CheckSource indicate that CheckSource can reduce the effort spent for some of the typical control-flow maintenance tasks by 60%, and prevent one error per 250 lines of source code. These results are statistically significant at level 0.05. The experiment show that the use of CheckSource makes it easier to detect and fix certain types of defects in certain types of functions. Compared to a manual approach, the experiments show less effort, and a reduced likelihood of introducing new errors while fixing others.

Chapter 6

Improving the Evolvability of Event-Driven Software

6.1 Introduction

In event-driven systems, separating the reactive part of software (i.e. event-driven control) from the non-reactive part is a common design practice; the reactive part is typically structured according to the states and transitions of a system [46, 50, 48, 47, 49, 60], whereas the non-reactive part is typically structured according to the concepts of the application domain (e.g. system services, hardware components, etc.).

The reactive part of software responds to occurrences of events [47]; it regulates the execution of the non-reactive part through **control call**s (see Fig. 6.1). Some events occur due to execution of the non-reactive part. To transmit these occurrences to the reactive part, **event call**s are inserted into (the source code of) the non-reactive part. Hence, the control- and event calls connect the two parts of software.

An event-driven software system may evolve several times during its lifetime. Possible mistakes during an evolution result in various defects in the system. Whenever the non-reactive part of software evolves[1], the following defects may emerge:

- There is an event call, but no corresponding occurrence of the event. Thus, the event call is invalid.
- There is an occurrence of an event, but no corresponding event call. Thus, a new event call is necessary.

[1]The evolution of the reactive part of software is outside the scope of this thesis.

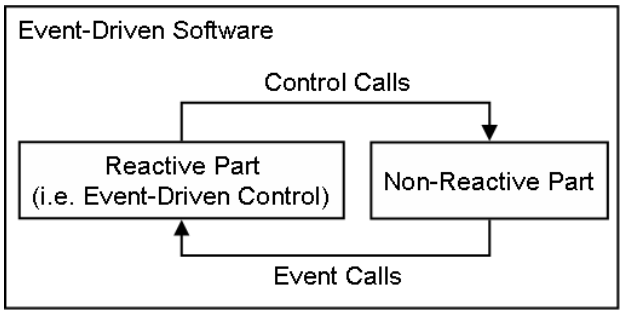

Figure 6.1: A conceptual model of event-driven software systems.

- The reactive part waits for an event call, but the call never happens. Thus, the two parts of software are **incompatible** with each other.
- The target (function, method, procedure) of a control call is missing. Thus the control call is invalid.
- The non-reactive part does not provide a required service.

Manually repairing these defects is effort-consuming and error-prone. In this chapter, we present a solution that substantially reduces the effort, and prevents errors while repairing the first three types of defects; the last two types of defects are outside the scope of this thesis. The solution consists of (a) an extended version of VisuaL, which can be used for specifying event calls and *compatibility constraints*, (b) an extended version of CheckSource for automatically verifying that the *compatibility constraints* are satisfied, and (c) a source-to-source transformer called TransformSource, for automatically generating event calls.

To quantify the benefits of the solution, we conducted a controlled experiment with 23 professional software developers and 21 M.Sc. students. In this experiment, the solution reduced the effort by 75%, and prevented one error per 140 lines of source code. The experiments show that the use of CheckSource and TransformSource makes it easier to detect and fix the first three types of defects listed above, in certain types of C functions. Compared to a manual approach, the experiments show less effort, and a reduced likelihood of introducing new errors while fixing others.

The solution is related to multiple fields of software engineering: The graphical language enables concern-shy programming [62]; the analyzer can be considered as a source code verification tool; and the combination of CheckSource and TransformSource exhibits some of the fundamental characteristics of a weaver [39] in aspect-oriented programming.

In Section 6.2, an industrial application is presented and several terms are defined.

In Section 6.3, we discuss the first three defects listed above. In Section 6.4, an overview of our solution approach is provided. Throughout Sections 6.5-6.8, the solution is presented in detail. Section 6.9 contains conclusions. The experiments are presented in Chapter 7.

6.2 An Example Application

A **silicon wafer** is a circular slice of silicon that is used for producing integrated circuits (ICs). A **wafer scanner** is a semiconductor manufacturing machine that exposes IC images onto silicon wafers. ASML [4] is a company that produces wafer scanners, and an ASML wafer scanner is a large-scale embedded system that has approximately 400 sensors, 300 actuators, 50 processors, and event-driven software containing around 15 million lines of source code written in C [58].

The reactive part of the wafer scanner software is structured according to the states and transitions of the system, using statecharts [47]. The non-reactive part is structured according to the activities [47] (i.e. the services) of the system, using a procedural decomposition. Both parts are implemented in C.

In Section 6.2.1, we present a simplified version of an ASML wafer scanner and its event-driven software. We explain the activities that the simplified wafer scanner can perform, and describe how these activities are controlled by a statechart, which represents the reactive part of the event-driven software.

6.2.1 Simplified Wafer Scanner

A wafer scanner (Fig. 6.2) **expose**s an IC image on rectangular segments of a wafer. Such a segment is called **die**. During an exposure, the wafer scanner uses a laser beam to **scan** the image, and uses a lens to project the laser beam onto the die.

Processing (An Activity of the Wafer Scanner)

A **reticle** is the material that contains the IC image to be exposed on dies. Before scanning, a reticle must be loaded onto a platform called **reticle stage**, and a wafer must be loaded onto a platform called **wafer stage**. Fig. 6.2 shows a snapshot of the wafer scanner during **scanning**: the lens and the laser source are fixed, the laser source is emitting a laser beam, and the wafer stage and the reticle stage are moving in opposite directions. Consequently, the IC image is being exposed on a die.

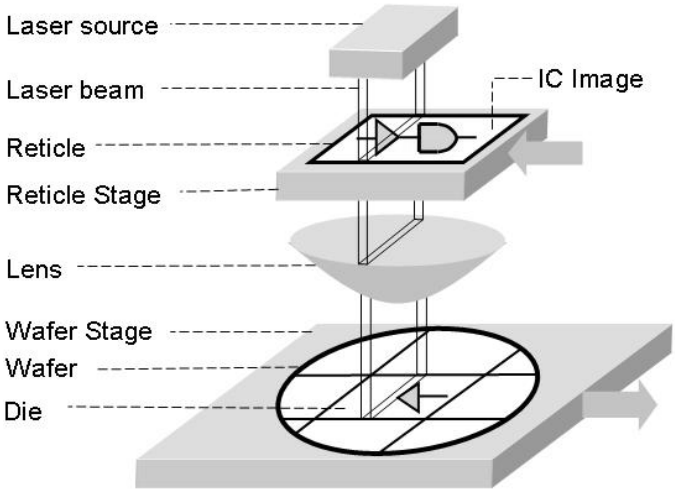

Figure 6.2: A snapshot of the wafer scanner during scanning.

When the IC image is completely exposed on the die, the wafer stage will be moved to align the next die with the lens. This activity is called **advancing**. The wafer **processing** activity consists of advancing to a die, then scanning it, and repeating this (i.e. advancing and scanning) for each die on the wafer.

Preprocessing (An Activity of the Wafer Scanner)

To produce faultless ICs, the wafer scanner's actuators need to operate at a level of precision that is measured in terms of nanometers. To attain this precision level, two issues must be resolved: (a) The reticle must be clean, and (b) the wafer scanner must know the shape imperfections of the wafer, so that it can compensate accordingly during processing. Therefore, before processing, the wafer scanner must carry out the **preprocessing** activity, which consists of **cleaning** the reticle if it is dirty, *and then* **measuring** the shape imperfections of the wafer.

Requirements of the Wafer Scanner

R1: The wafer scanner must start upon an external signal.
R2: The wafer scanner must process the wafer.
R3: After processing, all ICs on the wafer must be faultless.
R4: The wafer scanner must stop after the wafer is processed.

Reactive Part of the Event-Driven Software

Considering the requirements, the reactive part of the wafer scanner software can be structured as the statechart in Fig. 6.3. This statechart can be interpreted as

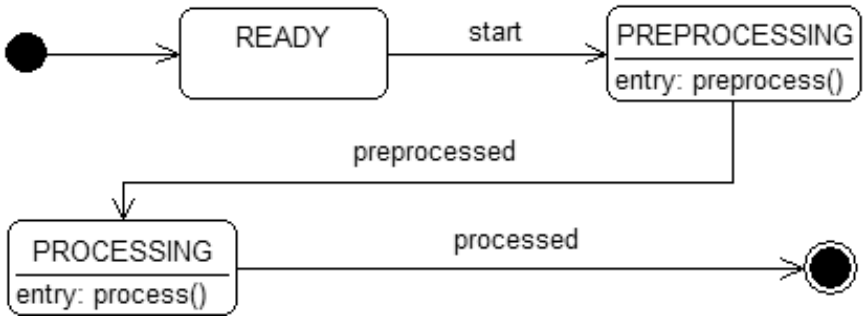

Figure 6.3: The wafer scanner's reactive behavior.

follows: When the scanner is in the READY state, if the start event occurs (e.g. an operator presses the start button), then the scanner enters the PREPROCESSING state, where it starts the preprocessing activity (i.e. calls the `preprocess` function, which will be defined later). If the scanner is in the PREPROCESSING state, and if the preprocessed event occurs, then the scanner enters the PROCESSING state, where it starts the processing activity (i.e. calls the `process` function, which will be defined later). If the scanner is in the PROCESSING state, and if the processed event occurs, then the scanner enters the final state (i.e. stops).

Based on a specific formal semantics of statecharts (e.g. [49]), one can manually implement the statecharts, or an implementation can be generated by a tool (e.g. [48]). Since the implementation details of the statechart in Fig. 6.3 are not important in this chapter, we assume that there is an implementation in C, which operates as explained above.

Non-Reactive Part of the Event-Driven Software

The preprocessing activity (Section 6.2.1), can be implemented as the C function in Listing 6.1.

```
1 void preprocess()
2 {
3     if(!reticleIsClean)
4     {
5         cleanReticle();
6     }
```

```
7     measureWafer();
8 }
```

Listing 6.1: An implementation of the preprocessing activity in C.

The processing activity (Section 6.2.1) can be implemented as the C function in Listing 6.2.

```
1 void process()
2 {
3     int i;
4     for(i = 0; i < numberOfDies; i++)
5     {
6         advance();
7         scan();
8     }
9 }
```

Listing 6.2: An implementation of the processing activity in C.

The definitions of the global variables `reticleIsClean` and `numberOfDies`, and the definitions of the functions `cleanReticle`, `measureWafer`, `advance`, and `scan` are not provided in the listings, because they are not important in the context of this chapter.

The event-driven software resulting from the implementation of the statechart and the activities fulfils R1, but not R2, R3 and R4; because the connection between the statechart and the activities is currently incomplete. This connection is completed in Section 6.2.2.

6.2.2 Connecting the Statechart and the Activities

Connecting the statechart and the activities consists of two steps: The first step is creating the control calls (see Fig. 6.1). In [47], this step is referred as "linking activities to states", which can be explained as follows: Upon entrance to a state, calling a function realizing an activity. Control calls are usually specified while creating the statecharts (e.g. see Fig. 6.3). The second step is creating the event calls (see Fig. 6.1): Stimulating the implementation of the statecharts with the events that occur due to execution of the non-reactive part. In the remainder of this section, we present the details of the second step, which is necessary for understanding the remainder of this chapter.

The second step requires identifying all the points (i.e. locations) in the source code

where the events occur during execution. To explain this more precisely, we need to define the following terms:

Definition: An event *e* is **mapped** to a source code point *pnt*, if and only if *e* occurs when an execution reaches *pnt*.

Definition: A source code point *pnt* is an **event point**, if and only if an event is mapped to *pnt*.

Using the definitions of the preprocessing and processing activities (see Section 6.2.1), we can identify the event points: The preprocessed event is mapped to the point located after '`;`' in Line 7, Listing 6.1. The processed event is mapped to the point located after '}' in Line 8, Listing 6.2.

After the identification of the event points, the implementation of the statechart must be stimulated with the occurrences of events. For this reason, a function that transmits the occurrence of an event to the statecharts is called at each event point. Hence, we term such functions **event function**s. Listings 6.3 and 6.4 show the implementations of the preprocessing and processing activities after inserting calls to the event functions `preprocessed` and `processed` at the event points.

The definitions of the event functions are typically located in the reactive-part of event-driven software (see Fig. 6.1). They can be considered as the interface provided by the reactive part to the non-reactive part; **event call**s (see Fig. 6.1) are the calls to the event functions. The implementation details of the event functions are not important in this chapter.

```
1 void preprocess()
2 {
3     if(!reticleIsClean)
4     {
5         cleanReticle();
6     }
7     measureWafer();
8     preprocessed();
9 }
```

Listing 6.3: The implementation of the preprocessing activity after the connection with the statechart.

```
1 void process()
2 {
3     int i;
4     for(i = 0; i < numberOfDies; i++)
5     {
6         advance();
```

```
 7          scan();
 8      }
 9      processed();
10  }
```

Listing 6.4: The implementation of the processing activity after the connection with the statechart.

The insertion of the event calls concludes the connection of the statechart and the activities. Consequently, each requirement in Section 6.2.1 is fulfilled by the event-driven software resulting from the connection explained in this section.

Note that the event points of the simplified wafer scanner are at the end of the functions. In the software of the actual wafer scanner, however, there are usually several other function calls between an event point and the end of a function. There are several reasons for this. For instance, at an intermediate step during an execution of a function, say f, an event e occurs; e stimulates a statechart s; s performs a transition to a next state, and calls another function g (i.e. the implementation of another activity), in which case f and g are executed concurrently. We do not illustrate such a case in this chapter, because it would unnecessarily complicate our example application.

In the software of the actual wafer scanner, the following cases exist, too: An event is mapped to multiple points in one function; An event is mapped to multiple points in multiple functions; Multiple events are mapped to multiple points in one function. Although we do not illustrate such cases, our solution addresses them, as well.

6.3 Defects During Activity Evolution

An event-driven software system may evolve several times during its lifetime. Possible mistakes during an evolution result in various defects in the system. Whenever the activities evolve, event call- and compatibility defects may occur. Manually finding and repairing these defects is effort-consuming and error-prone. In this section, we explain these defects, and show how they are manually found and repaired. To precisely explain the defects, we first need to define the following terms:

Definition: The **event transmitting behavior (ETB)** of a system is the behavior that is implemented by the event calls.

Definition: Let *ep* be an event point, and *ec* be an event call that transmits an occurrence of a specific event *e*. *ep* and *ec* are **related**, if and only if *e* is mapped to *ep*.

Definition: The ETB of a system is **sound**, if and only if each event call is located at a related event point.

Definition: The ETB of a system is **complete**, if and only if at each event point a related event call is located[2].

Note that the ETB presented in Section 6.2.2 is both sound and complete.

6.3.1 ETB Becomes Defective

If the activities evolve, then the ETB may become unsound or incomplete (i.e. defective), as we exemplify in this section.

ETB Becomes Unsound

Imagine that we remove the call to the `measureWafer` function from Line 7, Listing 6.3. As a result, the call to `preprocessed` is located at a point to which the preprocessed event is no longer mapped, because the wafer is not measured at that point. Hence, the ETB is no longer sound, and the requirement R3 is no longer fulfilled: the processing activity starts before the wafer is measured, which results in defective ICs. Therefore, we must remove the call to `preprocessed` to restore the soundness of the ETB.[3]

ETB Becomes Incomplete

Adding a new function call to the source code may result in a new event point (i.e. a new mapping of an existing type of event to a source code point). In such a case, the ETB becomes incomplete. To restore the completeness, adding a related event call at the new event point is necessary. Otherwise, the system cannot sense some occurrences of the event, and react to them. Consequently, (some of) the requirements may not be fulfilled.

[2]If multiple events are mapped to a point, then an ordering among the event calls is necessary.

[3]In this particular case, removing the call to `measureWafer` introduces, next to the unsoundness, an additional defect explained in Section 6.3.2.

ETB Becomes both Unsound and Incomplete

Consider a new requirement stating "the wafer must be measured only if the reticle is clean", because the reticle cleaning activity may fail. To fulfill the requirement, we can 'wrap' the call to `measureWafer` (Line 7, Listing 6.3) with an `if` block, as shown in Listing 6.5.

```
1  void preprocess()
2  {
3      if(!reticleIsClean)
4      {
5          cleanReticle();
6      }
7      if(reticleIsClean)
8      {
9          measureWafer();
10     }
11     preprocessed();
12 }
```

Listing 6.5: The preprocessing activity after adding a new control statement.

In this case, the preprocessed event is mapped to the point located after `;` in Line 9 where a call to `preprocessed` does not exist. Hence, the ETB is incomplete. In addition, the call to `preprocessed` (Line 11) is located at a point to which preprocessed is not mapped. Thus, the ETB is unsound. To restore the soundness and the completeness, we must move[4] the call to `preprocessed` from Line 11 to the point located after `;` in Line 9. Otherwise, the requirement R3 may not be fulfilled, because the wafer may not be measured, and the reticle may be dirty.

Considering the execution semantics of the source code in Listing 6.5, one may argue that the ETB is *complete*; because, whenever the preprocessed event occurs, the `preprocessed` function is executed. However, our definition of completeness is based on the syntactic structure of source code, not on execution semantics. The rationale for this choice will become clear in the upcoming case.

Now, let us consider an extract-function restructuring [45] that involves moving Lines 3-7 in Listing 6.3 to a new function `newPreprocess`, as shown in Listing 6.6.

```
1  void preprocess()
2  {
3      newPreprocess();
```

[4]Doing this conflicts with the requirement R2, but we ignore this fact for the sake of illustration in this chapter.

```
 4     preprocessed();
 5     return;
 6 }
 7 void newPreprocess()
 8 {
 9     if(!reticleIsClean)
10     {
11         cleanReticle();
12     }
13     measureWafer();
14 }
```

Listing 6.6: An extract-function restructuring.

In this case, the ETB is both unsound and incomplete, similar to the previous case explained in this section. Nevertheless, all the requirements are still fulfilled. This is certainly what is expected from a restructuring. However, if the system evolves further, and the `newPreprocess` function is called from an additional place different than Line 3, then the new occurrences of the preprocessed event are not transmitted. Clearly, the current location of the event call is not 'future-proof'. If the call to `preprocessed` in Line 4 is moved to the point located after `;` in Line 13, then the ETB will become sound and complete.

If our definition of completeness were based on execution semantics, then we could not recognize the fact that the location of the event call (i.e. Line 4) is not 'future-proof'.

Another case in which the ETB becomes both unsound and incomplete is as follows: If a new statement (e.g. a function call) is inserted after `;` in Line 7 in Listing 6.3, then the ETB becomes both unsound and incomplete.

Throughout Section 6.3.1, we discussed some of the evolution cases in which the ETB of a system becomes defective. In contrast, one can imagine other cases in which the ETB remains defect-free. For instance, if one or more statements are inserted after `;` in Line 5 in Listing 6.3, then the ETB is still sound and complete. In any case, manually verifying that the ETB is defect-free is effort-consuming and error-prone in large-scale software.

6.3.2 Activity Becomes Incompatible

Let us reconsider the case in Section 6.3.1, where we remove the calls to `measureWafer` and `preprocessed` in Listing 6.3. After removing the call to `preprocessed`, the

statechart is no longer stimulated with an occurrence of the preprocessed event. Consequently, the wafer scanner cannot perform the transitions from the PREPROCESSING state to the PROCESSING state, and from the PROCESSING state to the final state. Hence, the requirements R2 and R4 cannot be fulfilled despite the sound and complete ETB. Thus, we can conclude that an occurrence of the preprocessed event is mandatory whenever the `preprocess` function is executed. Therefore, the call to `measureWafer` must not be removed, as 'dictated' by the requirements and the statechart in Fig. 6.3. Otherwise, the preprocessing activity becomes incompatible with the statechart.

In general, if the activities of a given system are not **compatible** with the statecharts of the system, then the system may not behave as intended (i.e. requirements may not be fulfilled), despite a sound and complete ETB. The scenario explained above illustrates a typical incompatibility arising from possible mistakes during evolution of activities.

Based on the discussion in this section, one can imagine certain constraints on the implementation of activities, such that the constraints are satisfied, if and only if the activities are compatible with the statecharts. For example, the constraint in this case would be "the preprocessed event must occur whenever the `preprocess` function is executed". We call such constraints **compatibility constraints**. In Section 6.5, we explain these constraints in detail.

In contrast to the case presented in this section, one can imagine other cases in which the activities remain compatible with the statecharts. For instance, if one or more statements are inserted after `;` in Line 5 in Listing 6.3, then the activity is still compatible with the statechart. In any case, manual verification of compatibility is effort-consuming and error-prone in large-scale software.

6.3.3 Other Defects

Possible mistakes during the evolution of activities may cause other defects than those we discussed so far in Section 6.3. For example, a mistake during the evolution of the `measureWafer` function, which is the implementation of the wafer measuring activity, may lead to incorrect measurements, hence defective ICs. Furthermore, due to evolution of activities, the existing control calls (see Fig. 6.1) may become invalid. Addressing these kinds of defects is beyond the scope of this chapter.

6.3.4 Our Goal

Whenever the activities (i.e. the non-reactive part) evolve, the compatibility between the activities and the statecharts (i.e. the reactive part) has to be verified. If this verification fails, then the incompatibility has to be repaired. Next, the soundness and completeness of the ETB has to be verified. If this verification fails, then the ETB has to be maintained such that it remains sound and complete. Our goal is (a) to automate the compatibility verification, and (b) to *eliminate* the necessity of the ETB verification and maintenance; so that effort is reduced, and human errors are prevented. Our approach to reach this goal is presented in Section 6.4.

6.4 A 4-Stage Approach

As depicted in Fig. 6.4, we developed a 4-stage approach to automate the compatibility verification, and to eliminate the necessity of the ETB verification and maintenance. In Section 6.4.1, we provide an overview of this approach by explaining Fig. 6.4. In Section 6.4.2, we explain why this approach brings us to the goal stated in Section 6.3.4.

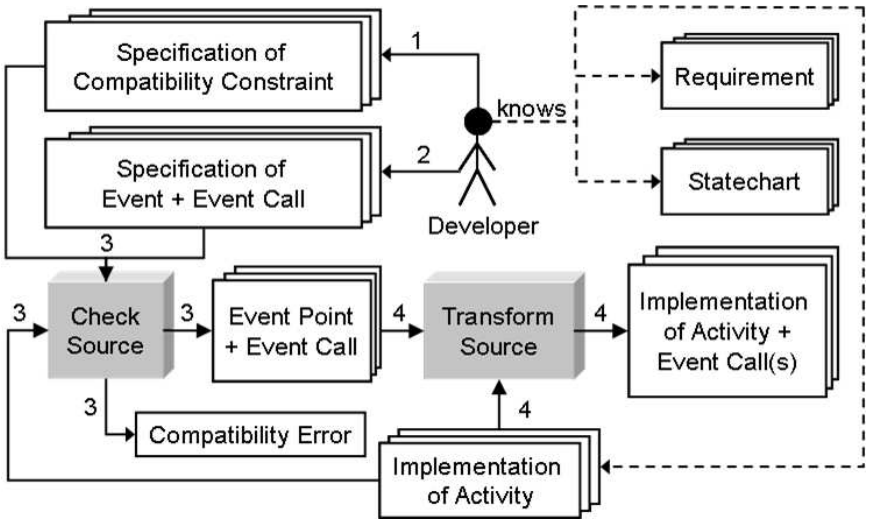

Figure 6.4: A 4-stage approach to automate the compatibility verification, and to eliminate the necessity of the ETB maintenance.

6.4.1 An Overview of the Stages

Stage1: Deriving and Specifying Compatibility Constraints

At this stage, a developer (or multiple developers) who knows the requirements (Section 6.2.1), the statecharts (Fig. 6.3), and the implementations of the activities (Listing 6.1 and 6.2), derives and specifies the compatibility constraints. To specify the constraints, the developer uses an extended version of VisuaL. Although the syntax and formal semantics of the extended version of VisuaL are provided in Appendix A.2, the reader can follow the remainder of this chapter without reading the appendix. In Section 6.5, Stage 1 is explained in detail.

Stage 2: Specifying Events and Binding Event Calls

At this stage, using VisuaL, the developer specifies the events, and binds the event calls to the event specifications. In Section 6.6, Stage 2 is explained in detail.

Stage 3: Analysis

At this stage, CheckSource is provided with the compatibility constraints from Stage 1, the event specifications and the bindings from Stage 2, and the ETB-free implementations of the activities. If the compatibility constraints are not satisfied (e.g. the case in Section 6.3.2), then CheckSource outputs a compatibility error that is valuable for understanding the incompatibility and repairing it. Here, the "repairing" means "either modifying the activities, or the statecharts and compatibility constraints, such that the compatibility error disappears". If there is no error, then CheckSource outputs each event point together with a related event call. In Section 6.7, Stage 3 is explained in detail.

Stage 4: Transformation

At this stage, a TransformSource, which is a source-to-source transformer, is provided with the ETB-free implementations of the activities and the output of the analyzer from Stage 3. At each event point, TransformSource inserts the related event call, which results in sound and complete ETB. In Section 6.8, Stage 4 is explained in detail.

6.4.2 The Benefit of our Approach (i.e. How the goal is reached)

Whenever the activities evolve, then the Stages 3 and 4 can be *automatically* repeated to verify that the compatibility constraints are satisfied, and to re-insert valid event calls, so that sound and complete ETB is re-generated. Consequently, effort can be reduced, and human errors can be prevented.

6.5 Stage 1: Deriving and Specifying Compatibility Constraints

At Stage 1, a developer (or multiple developers) derives and specifies the compatibility constraints. In this section, we explain Stage 1 in detail: First, we present a systematic way to collect hints for deriving compatibility constraints. Next, we explain how the hints can be used for deriving compatibility constraints from requirements, statecharts, and source code. Finally, we explain how the derived constraints can be specified using VisuaL.

6.5.1 Hints for Deriving Compatibility Constraints

The hints for deriving the compatibility constraints are the events whose lack of occurrence indicates an incompatibility exemplified in Section 6.3.2. We call such events **mandatory events**, because if such an event does not occur, then some of the requirements are not fulfilled. After the identification of mandatory events, the compatibility constraints can be derived in such a way that the satisfaction of the constraints guarantees the occurrences of the mandatory events.

The developer (see Fig. 6.4), who knows the requirements, the statecharts, and the implementations of the activities, can identify the mandatory events: she picks an event, say preprocessed, from the statechart in Fig. 6.3, and imagines what would happen if this event would not occur: the wafer scanner could not perform the transitions from the PREPROCESSING state to the PROCESSING state, and from the PROCESSING state to the final state. Hence, the system could not fulfill the requirements R2 and R4. With this line of reasoning, the developer realizes that the preprocessed event is mandatory. Note that the processed event (see Section 6.2.1) is a mandatory event, too.

If the requirements are formally specified and 'linked' to the states and transitions,

then automating the identification of the mandatory events becomes possible. However, this automation falls outside our scope; requirements engineering is extensively discussed in [83].

Identifying mandatory events should be considered as a *heuristic* for deriving compatibility constraints.

6.5.2 Deriving Compatibility Constraints

In this section, we explain how a developer can derive the compatibility constraints whose satisfaction guarantees the occurrence of the mandatory event preprocessed.

These constraints can be derived from the following facts: (a) the preprocessed event occurs when the preprocessing activity is carried out, (b) the preprocessing activity is "cleaning the reticle if it is dirty, *and then* measuring the shape imperfections of the wafer" (Section 6.2.1), (c) the reticle cleaning activity, the wafer measuring activity, and the preprocessing activity are respectively implemented within the `cleanReticle`, `measureWafer`, and `preprocess` functions. Using these facts, the developer can derive the following compatibility constraints:

C1: In each possible sequence of function calls from `preprocess`, there must be at least one call to `measureWafer`.
C2: In each possible sequence of function calls from `preprocess`, a call to `cleanReticle` must not come later than a call to `measureWafer`.

There are two possible sequences of function calls from the `preprocess` function in Listing 6.1: seq_1 = <`cleanReticle`, `measureWafer`>, and seq_2 =<`measureWafer`>. With these sequences in mind, note that the `preprocess` function satisfies both C1 and C2. As a result, the mandatory event preprocessed occurs each time the `preprocess` function is executed.

In fact, C1 and C2 are stricter than necessary: they enforce that the preprocessed event is mapped to the source code point(s) within the definition of the `preprocess` function. However, it would equally be fine if the preprocessed event were mapped to the source code point(s) in the definition of another function, say f, such that the event occurs each time f is executed, and f is called each time the `preprocess` function is executed. In this chapter, we only present stricter-than-necessary constraints due to a limitation of the current implementation of CheckSource: CheckSource can reason about function definitions as a single block, but it cannot reason about the nesting of function calls. This is not a fundamental limitation; some of the existing program analysis tools (e.g. [2]) are already capable of reasoning about the nesting of function calls. The current implementation of CheckSource has proven to be

useful despite this limitation, as indicated by the empirical evidence (Section 5.7).

The compatibility constraints related to the mandatory event processed (i.e. the other event in Fig. 6.3) are in Appendix A.3.

6.5.3 Specifying Compatibility Constraints

After the developer derives the compatibility constraints, she needs to specify them in VisuaL, so that CheckSource (see Fig. 6.4) can verify the implementations of the activities. Below, we provide the VisuaL specifications that express C1 and C2.

Specifying C1

The specification of the compatibility constraint C1 is shown in Fig. 6.5.

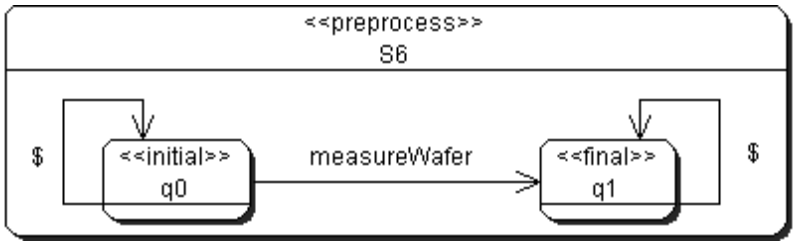

Figure 6.5: The specification of the compatibility constraint C1 in VisuaL.

Specifying C2

The specification of the compatibility constraint C2 is shown in Fig. 6.6. q0 and

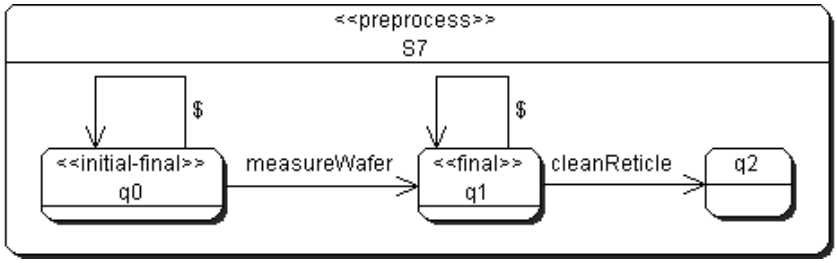

Figure 6.6: The specification of the compatibility constraint C2 in VisuaL.

q1 in this figure are different nodes than q0 and q1 in Fig. 6.5, because they are in different specifications.

The specifications of the compatibility constraints related to the mandatory event processed (i.e. the other event in Fig. 6.3) are provided in Appendix A.3.

6.6 Stage 2: Specifying Events and Binding Event Calls

At Stage 2, a developer (or multiple developers) formally specifies the events, and binds the event calls to the event specifications. In this section, we explain the details of Stage 2, by specifying the preprocessed event, and binding the `preprocessed();` event call, using VisuaL.

The preprocessed event occurs when the preprocessing activity is completed; and the preprocessing activity is defined as "cleaning the reticle if it is dirty, *and then* measuring the shape imperfections of the wafer" (Section 6.2.1). With this definition in mind, the preprocessed event can be specified as the pattern shown in Fig. 6.7.

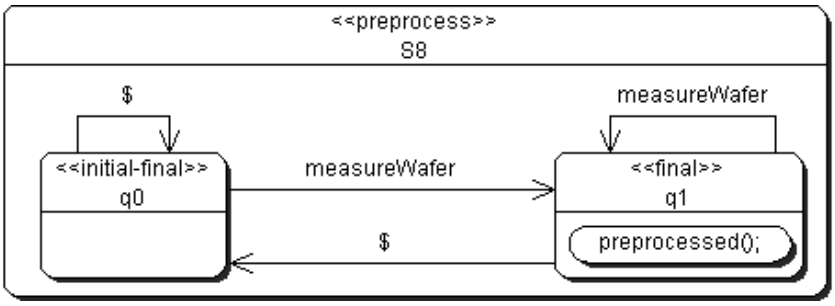

Figure 6.7: The specification of the preprocessed event and the binding of the event call `preprocessed();`, in VisuaL.

This pattern matches any sequence of function calls, because (a) all inner nodes have the <<final>> label, and (b) a $-labelled edge originates from each inner node. This unconstrained matching is intentional, because the pattern is created for specifying an event, which may or may not occur; in both cases there is no error. In contrast, a compatibility constraint must be satisfied, otherwise there is an error.

Each time `measureWafer` is executed during an execution of `preprocess`, the preprocessed event occurs. To detect such an occurrence, a measureWafer-labelled edge originates from each node; each measureWafer-labelled edge points to the same

node (i.e. `q1`); and no edge with a different label points to this node. Thus, `q1` is the 'hook' for binding `preprocessed();`. The binding is done by placing an ellipse containing the text preprocessed();, inside `q1`. If there is at least one such ellipse in a VisuaL specification, then the DFA denoted by the specification is a variant of Moore machine [54]. This is an extension to VisuaL, which is originally defined in Section 2. The details about the formal semantics of the extended VisuaL can be found in Appendix A.1.

Although an occurrence of the preprocessed event involves possible execution of the `cleanReticle` function, we do not need to include any cleanReticle-labelled edge in the pattern, because the satisfaction of the constraint C2, which is specified in Fig. 6.6, guarantees that any call to `cleanReticle`, if exists, comes before any call to `measureWafer`.

Throughout Section 6.5, and so far in this section, we explained that the information about a mandatory event consists of (a) specifications of the compatibility constraints, (b) the specification of the event, and (c) the binding of the event call to the event specification. Note that the information about the preprocessed event is currently distributed over three specifications: Figures 6.5, 6.6, and 6.7. To benefit from the advantages of the locality of information, one may prefer to capture the whole information about a mandatory event in one *concise* specification, using a single language. To our best knowledge, VisuaL is the only language that is suitable for this purpose. For example, the whole information about the preprocessed event (i.e. the information captured in Figures 6.5, 6.6, and 6.7) can also be captured in one concise specification, as shown in Fig. 6.8. This specification is concise, because (a) it has less number of nodes and edges than the total number of nodes and edges in Figures 6.5, 6.6, and 6.7; and (b) none of the nodes and edges in Fig. 6.8 is redundant.

In general, if a specification in VisuaL consists of at least one compatibility constraint and event call binding, then the specification is a **composite specification** (e.g. Fig. 6.8). A systematic way to compose multiple specifications to create a single composite specification is already discussed in Chapter 4.

Using the current version of VisuaL one cannot specify the processed event (i.e. the other event in Fig. 6.3), which is mapped to the point located after '}' in Line 8, Listing 6.2. The current version of VisuaL is not expressive enough for identifying this point. We revisit this limitation in Section 8.2.1.

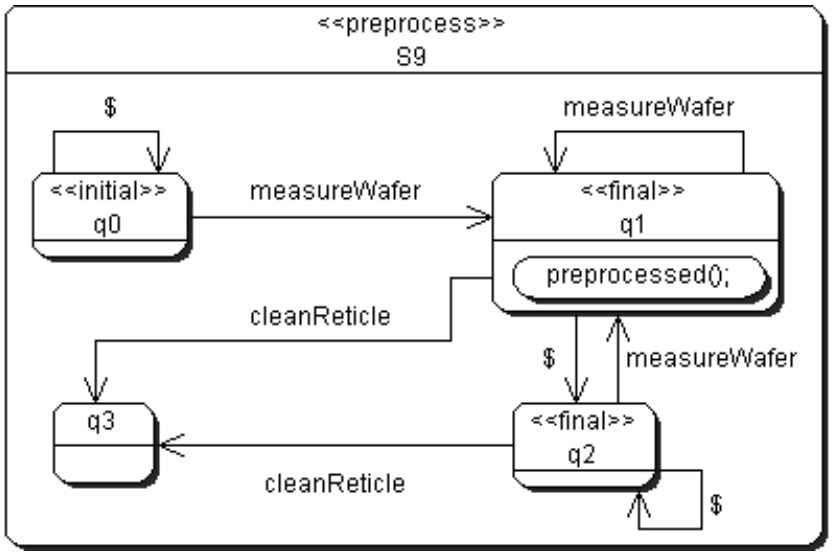

Figure 6.8: The composite specification in which the compatibility constraints C1 and C2 are specified, preprocessed is specified, and `preprocessed();` is bound.

6.7 Stage 3: Analysis

At Stage 3, the ETB-free implementations of the activities are analyzed with respect to the compatibility constraints from Stage 1 (Section 6.5), and the event points are identified based on the event specifications from Stage 2 (Section 6.6). In this section, we provide the details of Stage 3, which consists of three steps. The details of these steps are already presented in Chapter 5. For the sake of conciseness in this section, we are going to use the composite specification (Fig. 6.8), instead of the other specifications (Figures 6.5, 6.6, and 6.7).

6.7.1 Step 1: Creation of Abstract Syntax Tree (AST)

If the `preprocess` function (Listing 6.1) is given to the CheckSource as an input, then CheckSource parses the `preprocess` function, and constructs an abstract syntax tree [9] $AST_{preprocess}$ shown at the top of Fig. 6.9.

6.7.2 Step 2: Derivation of Simplified Control Flow Graph

We assume that the compatibility constraints and the events can be specified in terms of function calls and the possible flow of control [38] between the function

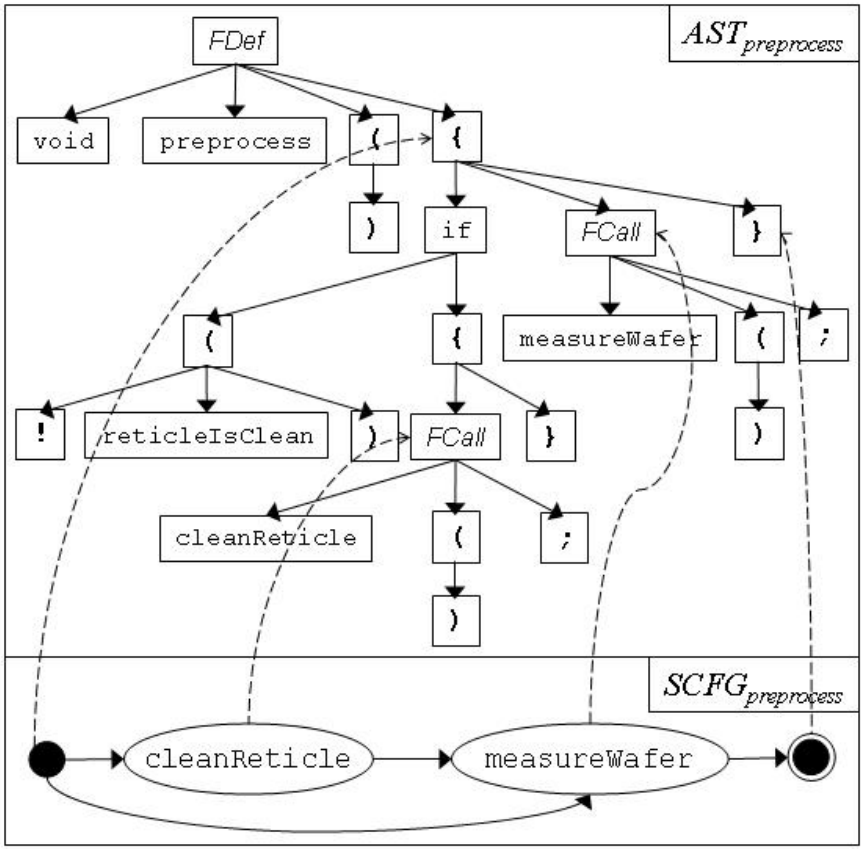

Figure 6.9: The abstract syntax tree ($AST_{preprocess}$) and the simplified control flow graph ($SCFG_{preprocess}$) of the `preprocess` function in Listing 6.1.

calls. Based on this assumption, only a part of the information that is in the AST is needed during the analysis. Therefore, CheckSource constructs the simplified control flow graph $SCFG_{preprocess}$ depicted at the bottom of Fig. 6.9. This construction is already explained in Chapter 5. As visualized by the dashed edges, CheckSource maintains a one-to-one mapping from the nodes of $SCFG_{preprocess}$ to the related nodes of $AST_{preprocess}$.

6.7.3 Step 3: Analysis of Simplified Control Flow Graph with respect to VisuaL Specification

To verify that the `preprocess` function satisfies the compatibility constraints C1 and C2 presented in Section 6.5.2, CheckSource has to check whether all possible sequences of function calls from the `preprocess` function are matched by the pattern

depicted in Fig. 6.8. To generate the function call sequences, CheckSource traverses $SCFG_{preprocess}$ in a depth-first manner. As understandable from $SCFG_{preprocess}$, there are two possible sequences of function calls: seq_1 and seq_2, both of which are already presented in Section 6.5.2. The analysis of these sequences reveals that each sequence 'ends in' `q1` (see Fig. 6.8). Since this node has the <<final>> label, each sequence is matched by the pattern, which means the `preprocess` function satisfies both C1 and C2. If the constraints were not satisfied, then CheckSource would output a compatibility error containing a sequence that violates the constraint. Such an error is valuable for finding and repairing an inconsistency.

During the analysis of seq_1, `q0` (see Fig. 6.8) is mapped to the `cleanReticle`-labelled internal node of $SCFG_{preprocess}$, and `q1` is mapped to the `measureWafer`-labelled internal node. During the analysis of seq_2, `q1` is once more mapped to the `measureWafer`-labelled internal node. Other nodes (i.e. `q2` and `q3`) are not mapped to any node of $SCFG_{preprocess}$. This partial[5] mapping from the set of the nodes of the specification to the set of the internal nodes of $SCFG_{preprocess}$ is the output of the analysis (i.e. Stage 3). As depicted in Fig. 6.4, this output is the input for the transformation (i.e. Stage 4), which is explained in Section 6.8.

Statecharts are proposed for expressing the event-driven and continuous behavior of reactive systems [47, 48, 49, 46, 50]. According to this proposal, the activities must terminate upon execution. Hence, the possible sequences of function calls from a function realizing an activity must be finite. The verification algorithm is already explained in Section 5.4.1.

6.8 Stage 4: Transformation

At Stage 4, TransformSource inserts the event calls at the event points identified at Stage 3. In this section, we explain the details of Stage 4 using the output of the example analysis presented in Section 6.7.3.

First, TransformSource is provided with the partial mapping created during the analysis (Section 6.7.3). Second, TransformSource selects `q1` (see Fig. 6.8), because `q1` contains the event call to be inserted (i.e. `preprocessed();`). Third, TransformSource parses `preprocessed();` and creates $AST_{preprocessed();}$ shown in Fig. 6.10. Fourth, TransformSource inserts $AST_{preprocessed();}$ as a sibling next to the upper *FCall* node in Fig. 6.9, because during the analysis (Section 6.7.3), `q1` was mapped to the `measureWafer`-labelled ellipse in Fig. 6.9, and this ellipse is mapped to the upper *FCall* node (see the dotted arrow between the `measureWafer`-labelled

[5] In a general case, such a mapping is not necessarily partial.

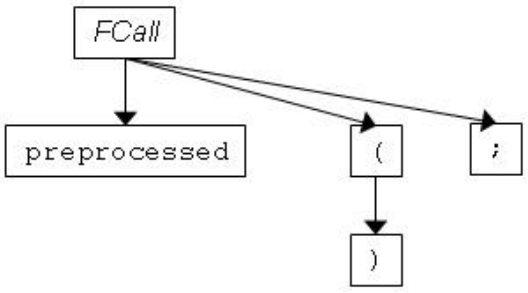

Figure 6.10: The abstract syntax tree $AST_{preprocessed();}$ of the `preprocessed();` event call.

ellipse and the upper *FCall* node). Finally, TransformSource traverses the modified $AST_{preprocess}$ to output the source code shown in Listing 6.3. A mathematical explanation of the transformation algorithm is provided in Appendix A.4.

Whenever the activities evolve, then the Stages 3 and 4 can be *automatically* repeated to verify that the compatibility constraints are satisfied, and to re-insert valid event calls, so that sound and complete ETB is re-generated. Consequently, effort can be reduced, and human errors can be prevented. In Section 7.3, the effort reduction and the error prevention are quantified.

IMPLEMENTATION: We implemented both CheckSource and TransformSource in Java. We used the open source parser generator ANTLR [3] together with the open source grammar cgram [6] to generate a parser for C. In addition, we implemented a plug-in to Borland Together [5], so that the VisuaL specifications can be drawn in the activity diagram editor of Borland Together, and they can be recognized by CheckSource as input. The source code of CheckSource and TransformSource is available upon request.

We tested the analyzer using an Intel(R) Pentium(R) M 1700 MHz processor with 1 GB of RAM. With an industrial specification in VisuaL, which has 11 nodes and 23 edges, CheckSource can process industrial functions containing 280, 127, and 83 lines of code, in 70, 40, and 32 milliseconds, respectively. The cyclomatic complexity number [67] of these functions is 51, 27, and 20, respectively.

6.9 Conclusions

In this chapter, we studied the evolution of event-driven software. We analyzed the problems arising from the evolution of the non-reactive part of software, and presented a novel solution that combines source code verification and aspect-oriented

programming techniques.

The solution consists of (a) a simple heuristic for deriving compatibility constraints from requirements, statecharts, and source code; (b) usage of VisuaL for specifying event calls and compatibility constraints; (c) usage of CheckSource for automatically verifying that the compatibility constraints are satisfied, and for identifying event points; and (d) usage of TransformSource for automatically inserting event calls.

To quantify the benefits of the solution, we conducted a controlled experiment with 23 professional software developers and 21 M.Sc. students. In this experiment, the solution reduced the effort by 75%, and prevented one error per 140 lines of source code. The benefits of the solution are statistically significant at level 0,01. In Chapter 7, we present the details of these experiments.

Chapter 7

Experimental Evaluation of CheckSource and TransformSource

This chapter has a lot in common with Chapter 5. However, there are many small variations scattered over the entire chapter. To avoid excessive cross-references to Chapter 5. We decided to repeat the contents of Chapter 5, in this chapter.

7.1 Experiment Definition and Planning

In Section 6.4.2, we claimed that CheckSource and TransformSource can reduce human effort and prevent errors. To test this claim, we conducted formal experiments. In this section, we present the definition and planning of these experiments. For preparing, conducting, and documenting the experiments, we followed the guidelines proposed by Kitchenham et. al. [59] and Wohlin et. al. [86].

7.1.1 Background Information

The software of the actual wafer scanner consists of around 200 software components, most of which are designed and implemented in the way explained in Section 6.2. In the past, one of the software teams developing and maintaining such a component have informed us about the excessive maintenance effort they spend due to the defects explained in Section 6.3. Therefore, we conducted this research.

The solutions presented in Chapter 6 were tested within the context of the component mentioned above. This component has 15 statecharts, each having on average

10 states and 15 transitions. The component contains 55.000 lines of source code, and 55 events mapped to 102 source code points distributed over 81 function definitions. Hence, the component has 55 event functions, and 102 event calls distributed over 81 functions. Among the 200 components of the actual wafer scanner, this component is a mid-sized one.

In the controlled experiments, we decided to use real-life functions and real-life VisuaL specifications. Therefore, we trained the domain expert of the component, so that he can create some VisuaL specifications corresponding to some of the functions in the component. After the training, the developer selected a statechart that has 18 states and 22 transitions, and identified the part of the software component in which the corresponding activities are implemented. Next, using the heuristic presented in Section 6.5, the developer identified three mandatory events. Then, the developer created three composite specifications in VisuaL, each of which consists of one compatibility constraint and the specification of one mandatory event.

7.1.2 Motivation and Overview

The purpose of this experiment is to test the claimed benefits of CheckSource and TransformSource while removing the incompatibilities and repairing the ETB defects in industrial software.

We conducted this experiment twice. In the first experiment, 21 M.Sc. computer science students from the University of Twente participated. In the second experiment, 23 professional software developers from ASML participated.

During both experiments, the participants worked with three C functions (i.e. implementations of three activities) selected by the domain expert from the component of the wafer scanner software (see Section 7.1.1), and the corresponding specifications that were created by the expert using VisuaL.

We injected an incompatibility defect, an unsoundness defect, and an incompleteness defect into each function, and then we requested the participants to repair these defects by modifying the functions, such that each function would conform to the corresponding specification.

7.1.3 Hypotheses

We formulated the following hypotheses to be tested in the first experiment:

- $H_0{}^1$: The tools (i.e. CheckSource and TransformSource) do not have any effect

on the amount of effort spent by M.Sc. students.
- $H_0{}^2$: The tools do not have any effect on the number of errors made by M.Sc. students.

We formulated the following hypotheses to be tested in the second experiment:

- $H_0{}^3$: The tools do not have any effect on the amount of effort spent by professional developers.
- $H_0{}^4$: The tools do not have any effect on the number of errors made by professional developers.

We chose 0,01 as the significance level for rejecting the hypothesis above.

7.1.4 The Variables of the Experiment

Factors

- **Tool support** is the only factor of this experiment. This factor is measured in the nominal scale, at two levels: exists, not exists.

Non-factor Independent Variables

There are two independent variables that we kept at fixed levels in this experiment. The first one is the function-specification pair, and the second one is the injected defect. Below, we explain these variables in detail.

- **Function-Specification Pair** is an independent variable kept at a fixed level:

Each participant was treated with the same set of three C functions and the corresponding VisuaL specifications. We measured the size and cyclomatic complexity [67] of both the functions and the specifications.

For a given function, the size is measured by counting the physical lines of code, and the complexity is measured by calculating the cyclomatic complexity number. In Table 7.1, the size and complexity of the three functions are listed. These functions are originally located in a file that has 55 functions. This file is one of the several files in the software component mentioned in Section 7.1.1. For a better understanding of the file, the descriptive statistics about the 55 functions can be found in Table 7.2.

For a given VisuaL specification, the size is measured by counting the nodes and the edges, and the complexity is measured by calculating the cyclomatic complexity

Table 7.1: The size and complexity of the C functions.

Functions	# Lines of Code	Cyclomatic Complexity
Function1	88	20
Function2	127	27
Function3	280	51

Table 7.2: Descriptive statistics of the 55 functions in the file.

	Avg.	Min.	Max.	Std. Dev.
Lines of Code	133	24	390	89
Cyclomatic Complexity	28	4	114	20

number. In Table 7.3, the size and complexity of the three specifications are listed. These specifications were created by the domain expert at ASML. Specification1,

Table 7.3: The size and complexity of the VisuaL specifications.

Specifications	# Nodes	# Edges	Cyclomatic Complexity
Specification1	11	19	10
Specification2	11	23	14
Specification3	10	20	12

Specification2, and Specification3 respectively corresponds to Function1, Function2, and Function3.

- **Injected defect** is an independent variable kept at a fixed level:

We injected the same kind of defect into each of the three functions: We removed the first possible function call to inject an incompatibility (e.g. the case in Section 6.3.2), and we added one control statement to inject unsoundness and incompleteness (e.g. the case in Section 6.3.1).

Dependent Variables

There are two dependent variables in this experiment:

- Amount of **effort** is a dependent variable measured in the ratio scale. We measure this variable in terms of minutes.
- Number of **errors** is a dependent variable measured in the absolute scale.

7.1.5 Selection of Participants

Selection of Students

This experiment was an integral part of the 2006 spring semester Software Management course at the University of Twente. Hence, the students of this course participated in the experiment. These students were M.Sc. computers science students.

To collect some information about the software development experience of these students, we asked them the size of the largest computer program they have written using one of the imperative languages (e.g. C, Java). The students had to select one of the following answers:

1. Less than 100 lines of source code
2. More than 100, less than 1000 lines of source code
3. More than 1000, less than 5000 lines of source code
4. More than 5000, less than 10000 lines of source code
5. More than 10000 lines of source code

No student selected 1, four students selected 2, six students selected 3, seven students selected 4, four students selected 5.

None of the students had any previous experience about the instruments listed in Section 7.1.7.

Selection of Developers

After we conducted the first experiment with the students, we presented the results of this experiment together with the solution summarized in Section 6.4 to the software developers of ASML. Out of approximately 500 software developers of ASML, around 130 developers attended this presentation. After the presentation, we invited these developers to participate in this experiment. 23 developers (voluntarily) participated in the experiment. At the beginning of the experiment, we requested the developers to indicate their professional software development experience with the imperative programming languages (e.g. C, Java), in terms of years. It turned out that each developer has at least four years of professional experience.

None of the developers had any previous experience about the instruments listed in Section 7.1.7, except two of the developers have previously seen the C functions used in the experiment.

7.1.6 Experiment Design

As visible in Fig. 7.1, we designed an experiment that has one factor and two treatments. The factor and its levels are already explained in Section 7.1.4. Each

	Factor: Tool Support	
	Level: Exists	Level: Not Exists
Experiment 1	11 Students	10 Students
Experiment 2	12 Developers	11 Developers

Figure 7.1: The experiment has one factor with two levels each of which is one of the two treatments. The number of participants per treatment in each of the experiments is also shown in this figure.

level of the factor is a treatment in this experiment.

The participants were randomly assigned to one of the two treatments (i.e. there were two independent groups of participants). We balanced the design by assigning (almost) equal number of participants per treatment. In the remainder of this chapter, we will use **tool-supported participant** for referring to a participant treated with the tool support, and **manual participant** for referring to a participant treated without tool support.

7.1.7 Instrumentation

The instruments of this experiment are

- the C functions into which we injected defects,
- the VisuaL specifications,
- the tools using which the participants repaired the defects (i.e. CheckSource and TransformSource),
- the tutorial slides that we presented to the participants to train them for repairing the defects,
- the documents containing the stepwise instructions for the participants to repair the defects, and
- the facilitating software that we developed for automatic data collection.

Interested readers can request the instruments from us by providing personal details and affiliation. If ASML approves the request, then we can send a non-disclosure

agreement (NDA). After the NDA is signed and returned, we can provide the instruments.

7.2 Experiment Operation

The operation phase of the experiment consisted of three steps: preparation, execution, and data validation. In this section, we explain these steps in detail.

7.2.1 Preparation

We prepared a tutorial for teaching the participants how to (a) interpret the specifications, (b) relate the specifications to the source code, and (c) repair the defects in the source code using the specifications. For the tool-supported participants, the tutorial also included how to use the tools. We presented this tutorial before the experiment as a slide show, and we distributed hard copies of the slides to the participants, after the presentation.

We prepared step-wise instructions for the participants. By following these instructions, a participant could find the source code in the directory structure of the computer, run the tools, etc.

We implemented facilitating software that puts a time stamp on the source code modified by a participant, and logs the source code in a file. The manual participants ran this software twice: once at the beginning of the treatment, and once at the end of the treatment. The facilitating software was integrated with the tool support (i.e. CheckSource and TransformSource). Consequently, the tool-supported participants ran the facilitating software at least twice: once at the beginning of the treatment (i.e. when they initially used the tool to find and understand the defects), once at the end of the treatment (i.e. after they modified the source code), and zero or more times during the treatment (i.e. each additional time they used the tool to see whether they could successfully repair the defects).

We prepared an example treatment for the participants, so that they get used to the tasks they are required to perform. In this way, we aimed at improving the accuracy of our measurements, by decreasing the learning overhead in the actual treatments. The example treatment was the first treatment of each participant.

We conducted preliminary runs of the experiment to test the artifacts explained above. These runs enabled us to improve the instruments of the experiment. The four participants of these preliminary runs were different than the participants of

the actual experiment. During the analysis presented in Section 7.3, we excluded the data of the preliminary runs.

To motivate the students for performing the tasks as carefully and quickly as they can, we rewarded the first, second, and the third best performers in each of the tool-supported and manual groups with 50 EUR, 40 EUR, and 30 EUR, respectively. The ranking criteria was performing the tasks with least number of errors in least amount of time, where the number of errors had priority over the amount of time. Besides the top three prizes, each student received 10 EUR for his participation. Before the students started the experiment, we informed them about the prizes and the ranking criteria. The results of the students were kept anonymous, and these results did not have any impact on their course grade.

We assumed that the developers were self-motivated, because they volunteered. Therefore, we did not reward them with a prize.

7.2.2 Execution

During the experiment, the students worked at the computer laboratories of the university, and they used the computers in the laboratories to modify source code, and to run the tools.

To ensure the independence of the observations, each student participated in the experiment at the same time. This required an instructor to give the tutorial for the tool-supported group in a laboratory, and another instructor to give the tutorial for the manual group in another laboratory. Moreover, the instructors and two additional assistants were present at the laboratories.

The developers participated in the experiment at their offices at ASML, and they participated in the experiment not at the same time but at various dates and times. We could not avoid this due to the busy agendas of the developers. We ensured the independence of observations as good as possible: We kept the participant list secret (note that 23 out of 500 developers participated), and collected the material of the experiment after the participation of each developer. During the experiment, the developers used a computer whose setting was identical to the setting of the computers used by the students in the laboratories.

7.2.3 Data Validation

As explained in Section 7.2.1, each participant ran a facilitating software that logs the source code with a time stamp. The participants were not authorized to modify the clock of the operating system.

To validate the data contained in the files, we compared the latest time stamp in a file with the last modified time of the file. If they were different, this would indicate that the participant had manually modified the file, hence the data is invalid. In this way, we found two invalid log files, and we did not include their data in the analysis.

We informed each participant about his result, and asked whether the result is as he expected; each participant informed us that his result is as he expected. This supports the claim that the participants have understood the instructions, and followed them properly (i.e. this is a positive indication about the validity of data).

7.3 Data Analysis

By investigating the log files created during the experiment, we realized that each tool-supported participant worked until CheckSource gave no more error messages. Therefore, after a tool-supported participant finished a treatment, the resulting source code was free of incompatibility, unsoundness, and incompleteness defects. On the other hand, the manual participants made errors while repairing the defects (see Section 6.3). To calculate the number of errors, we used the following criteria: For each unsoundness situation (e.g. Section 6.3.1), we counted one error. For each incompleteness situation (e.g. Section 6.3.1), we counted one error. For each incompatibility situation (e.g. Section 6.3.2), we counted one error.

The raw data of the experiment is provided in Appendix B.2. In the remainder of this section, we analyze the data in three steps: First, we discuss the screening and cleaning of the raw data, second we present the descriptive statistics of the clean data, and third we present the statistical tests we applied to the hypotheses stated in Section 7.1.3.

We used SPSS Version 12.0.1 for Windows [7] for analyzing our data, and testing the hypotheses.

7.3.1 Screening and Cleaning the Data

Our investigations on the log files revealed that the logged data of one tool-supported and one manual student were manually modified (i.e. corrupted). We understood this by comparing the time stamps in the files with the last modified time of the files. Consequently, we excluded their data from our calculations.

One of the tool-supported students could not finish the treatment within the given amount of time, which was three hours. Therefore, we excluded his data, too.

7.3.2 Descriptive Statistics

The experiment with the students

In Fig. 7.2, the descriptive statistics of the data collected from the experiment with the students is presented.

Since each tool-supported student worked until CheckSource gave no more error messages, the descriptive statistics of the number of errors in the existence of tool support is omitted in Fig. 7.2.

The mean amount of effort spent by the tool-supported students is 32 minutes[1], whereas the mean amount of effort spent by the manual students is 64 minutes. Hence, we can conclude that the tools reduced the effort spent by an average student approximately by 50% in this experiment.

The mean number of errors made by the tool-supported students is 0, whereas the mean number of errors made by the manual students is 5. Since each participant worked with 500 lines of source code in total (see Table 7.1), we can conclude that the tools prevented approximately one error per $500 \div 5 = 100$ lines of source code in this experiment.

Note that the 5% trimmed means (i.e. the means calculated upon excluding 5% of the data at the extremes) are very close to the original means. For instance, the original mean of the amount of effort in the existence of tool support is 32 minutes, and the corresponding trimmed mean is 33 minutes. Due to the closeness of each trimmed mean to the corresponding original mean, we can conclude that the extreme values of the dependent variables do not have a strong influence on the original means.

[1]Wherever it is appropriate, we present rounded numbers for increasing the readability of the text. More accurate numbers are presented in the figures. For example, this number (i.e. 32) is presented as 32,44 in Fig. 7.2.

	Tool Support			Statistic	Std. Error
Effort	Exists	Mean		32,44	4,661
		95% Confidence Interval for Mean	Lower Bound	21,70	
			Upper Bound	43,19	
		5% Trimmed Mean		31,88	
		Median		27,00	
		Variance		195,528	
		Std. Deviation		13,983	
		Minimum		17	
		Maximum		58	
		Range		41	
		Interquartile Range		23	
		Skewness		1,005	,717
		Kurtosis		-,166	1,400
	Not Exists	Mean		64,11	4,929
		95% Confidence Interval for Mean	Lower Bound	52,75	
			Upper Bound	75,48	
		5% Trimmed Mean		64,35	
		Median		67,00	
		Variance		218,611	
		Std. Deviation		14,786	
		Minimum		40	
		Maximum		84	
		Range		44	
		Interquartile Range		26	
		Skewness		-,268	,717
		Kurtosis		-,930	1,400
Errors	Not Exists	Mean		4,67	1,027
		95% Confidence Interval for Mean	Lower Bound	2,30	
			Upper Bound	7,04	
		5% Trimmed Mean		4,63	
		Median		4,00	
		Variance		9,500	
		Std. Deviation		3,082	
		Minimum		1	
		Maximum		9	
		Range		8	
		Interquartile Range		7	
		Skewness		,205	,717
		Kurtosis		-1,568	1,400

Figure 7.2: The descriptive statistics of the data collected from the experiment with the students. The data consists of effort measured in minutes, and the number of errors. Since the number of errors is constant when the tool support exists, the related statistics is omitted in this figure.

The positive skewness of the effort in the existence of tool support (1,005) indicates

that the majority of the tool-supported students spent less than 32 minutes during the experiment. The negative skewness of the effort in the lack of tool support (-0,268) indicates that the majority of the manual students spent more than 64 minutes during the experiment.

The negative values of Kurtosis indicate that the distributions of the values are relatively flat (i.e. too many values at the extremes).

In Fig. 7.3, the results of the normality tests for the data collected from the experiment with the students are shown. It is very likely for the amount of effort and the number of errors to have a normal distribution, because the significance values (shown as "Sig." in Fig. 7.3) are greater than 0,05.

In Figures 7.4 and 7.5, the box plots of the amount of effort versus tool support, and the number of errors versus tool support are respectively shown. The grey rectangles represent 50% of the values, with the whiskers (i.e. the lines below and above the rectangles) going to the minimum and the maximum values. SPSS did not detect any outliers (i.e. there is no data point outside the minimum and maximum ranges).

The experiment with the developers

In Fig. 7.6, the descriptive statistics of the data collected from the experiment with the developers is presented. Since each tool-supported developer worked until CheckSource gave no more error messages, the descriptive statistics of the number of errors in the existence of tool support is omitted in Fig. 7.6.

The mean amount of effort spent by the tool-supported developers is 12 minutes, whereas the mean amount of effort spent by the manual developers is 50 minutes. Hence, we can conclude that the tools reduced the effort spent by an average developer approximately by 75% in this experiment.

		Kolmogorov-Smirnov[a]			Shapiro-Wilk		
	Tool Support	Statistic	df	Sig.	Statistic	df	Sig.
Effort	Exists	,264	9	,071	,881	9	,161
	Not Exists	,133	9	,200*	,965	9	,853
Errors	Not Exists	,194	9	,200*	,898	9	,243

*. This is a lower bound of the true significance.

a. Lilliefors Significance Correction

Figure 7.3: The results of the normality tests for the data collected from the experiment with the students. Since the number of errors is constant when the tool support exists, the related statistics is omitted in this figure.

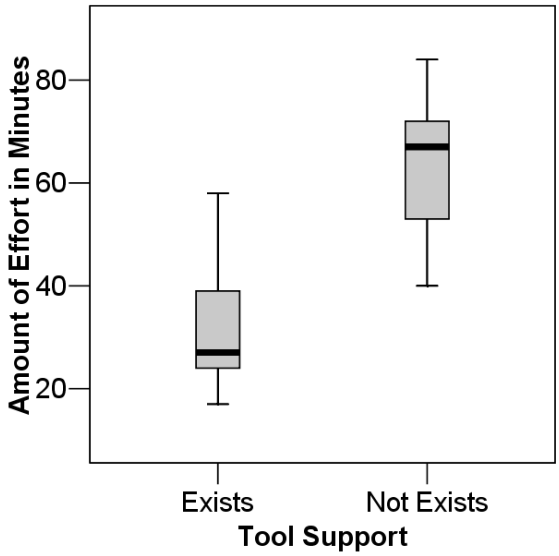

Figure 7.4: Box plot of effort vs. tool support in the experiment with the students.

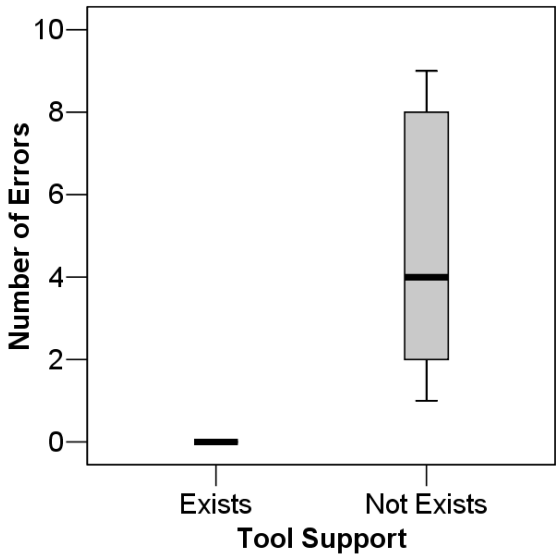

Figure 7.5: Box plot of errors vs. tool support in the experiment with the students.

The mean number of errors introduced by the tool-supported developers is 0, whereas the mean number of errors introduced by the manual developers is 3,5. Hence, we can conclude that the tools prevented approximately one error per $500 \div 3,5 = 140$ lines of source code in this experiment.

	Tool Support			Statistic	Std. Error
Effort	Exists	Mean		11,75	,889
		95% Confidence Interval for Mean	Lower Bound	9,79	
			Upper Bound	13,71	
		5% Trimmed Mean		11,67	
		Median		10,00	
		Variance		9,477	
		Std. Deviation		3,079	
		Minimum		8	
		Maximum		17	
		Range		9	
		Interquartile Range		6	
		Skewness		,573	,637
		Kurtosis		-1,270	1,232
	Not Exists	Mean		49,73	4,200
		95% Confidence Interval for Mean	Lower Bound	40,37	
			Upper Bound	59,08	
		5% Trimmed Mean		50,14	
		Median		51,00	
		Variance		194,018	
		Std. Deviation		13,929	
		Minimum		26	
		Maximum		66	
		Range		40	
		Interquartile Range		21	
		Skewness		-,542	,661
		Kurtosis		-,854	1,279
Errors	Not Exists	Mean		3,64	,704
		95% Confidence Interval for Mean	Lower Bound	2,07	
			Upper Bound	5,21	
		5% Trimmed Mean		3,54	
		Median		3,00	
		Variance		5,455	
		Std. Deviation		2,335	
		Minimum		1	
		Maximum		8	
		Range		7	
		Interquartile Range		4	
		Skewness		,422	,661
		Kurtosis		-,737	1,279

Figure 7.6: The descriptive statistics of the data collected from the experiment with the developers. The data consists of effort measured in minutes, and the number of errors. Since the number of errors is constant when the tool support exists, the related statistics is omitted in this figure.

Note that the 5% trimmed means are very close to the original means. For instance,

the original mean of the amount of effort in the existence of tool support is 11,75 minutes, and the corresponding trimmed mean is 11,67 minutes. Due to the closeness of each trimmed mean to the corresponding original mean, we can conclude that the extreme values of the dependent variables do not have a strong influence on the original means.

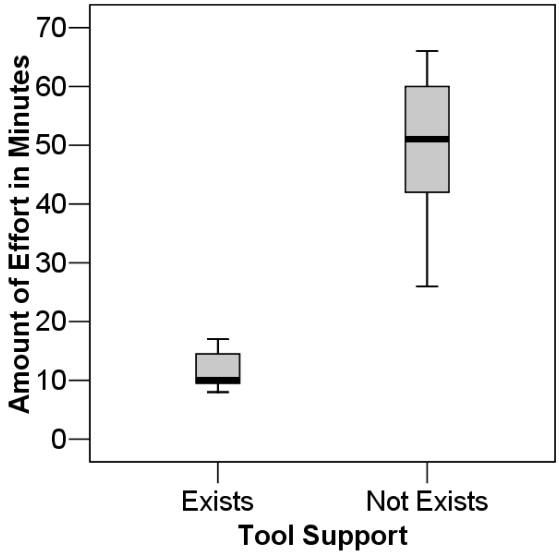

Figure 7.7: Box plot of effort vs. tool support in the experiment with developers.

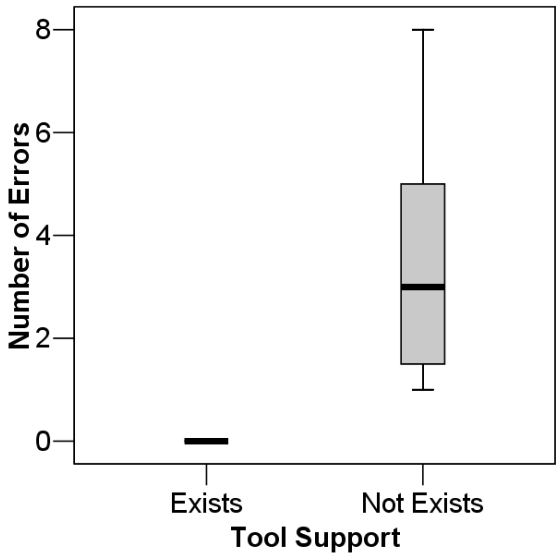

Figure 7.8: Box plot of errors vs. tool support in the experiment with developers.

		Kolmogorov-Smirnov[a]			Shapiro-Wilk		
	Tool Support	Statistic	df	Sig.	Statistic	df	Sig.
Effort	Exists	,298	12	,004	,879	12	,086
	Not Exists	,178	11	,200*	,924	11	,349
Errors	Not Exists	,175	11	,200*	,912	11	,254

*. This is a lower bound of the true significance.

a. Lilliefors Significance Correction

Figure 7.9: The results of the normality tests for the data collected from the experiment with the developers. Since the number of errors is constant when the tool support exists, the related statistics is omitted in this figure.

The positive skewness of the effort in the existence of tool support (0,573) indicates that the majority of the tool-supported developers spent less than 12 minutes during the experiment. The negative skewness of the effort in the lack of tool support (-0,542) indicates that the majority of the manual developers spent more than 50 minutes during the experiment.

The negative values of Kurtosis indicate that the distributions of the values are relatively flat (i.e. too many values at the extremes).

In Fig. 7.9, the results of the normality tests for the data collected from the experiment with the developers are shown. According to the Shapiro-Wilk test, it is likely that the amount of effort and the number of errors have a normal distribution, because the significance values are greater than 0,05.

According to the Kolmogorov-Simmov test, it is not likely that the amount of effort has a normal distribution in the existence of tool support, because the significance value 0,004 is less than 0,05. However, it is very likely that the amount of effort in the lack of tool support and the number of errors have a normal distribution, because the significance value 0,2 is greater than 0,05.

In Figures 7.7 and 7.8, the box plots of the amount of effort versus tool support, and the number of errors versus tool support are respectively shown. SPSS did not detect any outliers (i.e. there is no data point outside the minimum and maximum ranges in the box plots).

7.3.3 Hypothesis Testing

For testing the hypotheses stated in Section 7.1.3, we used the independent-samples t-test provided by SPSS. The assumptions for using the t-test hold in our experiment: each dependent variable is measured in the ratio scale (see Section 7.1.4); each

		Levene's Test for Equality of Variances		t-test for Equality of Means						
									95% Confidence Interval of the Difference	
		F	Sig.	t	df	Sig. (2-tailed)	Mean Difference	Std. Error Difference	Lower	Upper
Effort	Equal variances assumed	,059	,811	-4,668	16	,000	-31,667	6,783	-46,047	-17,286
	Equal variances not assumed			-4,668	15,950	,000	-31,667	6,783	-46,051	-17,283
Errors	Equal variances assumed	24,146	,000	-4,542	16	,000	-4,667	1,027	-6,845	-2,489
	Equal variances not assumed			-4,542	8,000	,002	-4,667	1,027	-7,036	-2,297

Figure 7.10: The results of the independent samples t-test for assessing the differences between the tool-supported and manual students.

participant is randomly assigned to either the tool-supported or the manual group (see Section 7.1.6); the observations made during the experiment are independent of each other (see Section 7.2.2); it is likely that the dependent variables have a normal distribution (see Section 7.3.2).

Testing $H_0{}^1$

An independent-samples t-test was conducted to compare the amount of effort spent by the tool-supported students versus the manual students (see Fig. 7.10). Since the significance value of Levene's test (0,81) is greater than 0,05, the equality of variances is assumed (i.e. the first row in Fig. 7.10 is considered). There was a significant difference in the amount of effort spent by the tool-supported students (Mean = 32; Std. Dev. = 14) and the manual students (Mean = 64; Std. Dev = 15; t(16) = -4,66; p = 0,01). Therefore, we can reject $H_0{}^1$.

Testing $H_0{}^2$

An independent-samples t-test was conducted to compare the number of errors made by the tool-supported students versus the manual students (see Fig. 7.10). Since the significance value of Levene's test (0,00) is less than 0,05, the equality of variances is not assumed (i.e. the fourth row in Fig. 7.10 is considered). There was a significant difference in the number of errors made by the tool-supported students (Mean = 0; Std. Dev. = 0) and the manual students (Mean = 5; Std. Dev = 3; t(8) = -4,54; p = 0,01). Therefore, we can reject $H_0{}^2$.

		Levene's Test for Equality of Variances		t-test for Equality of Means						
									95% Confidence Interval of the Difference	
		F	Sig.	t	df	Sig. (2-tailed)	Mean Difference	Std. Error Difference	Lower	Upper
Effort	Equal variances assumed	17,141	,000	-9,221	21	,000	-37,977	4,119	-46,542	-29,412
	Equal variances not assumed			-8,847	10,896	,000	-37,977	4,293	-47,437	-28,518
Errors	Equal variances assumed	38,896	,000	-5,405	21	,000	-3,636	,673	-5,035	-2,237
	Equal variances not assumed			-5,164	10,000	,000	-3,636	,704	-5,205	-2,067

Figure 7.11: The results of the independent samples t-test for assessing the differences between the tool-supported and manual developers.

Testing $H_0{}^3$

An independent-samples t-test was conducted to compare the amount of effort spent by the tool-supported developers versus the manual developers (see Fig. 7.11). Since the significance value of Levene's test (0,00) is less than 0,05, the equality of variances is not assumed (i.e. the second row in Fig. 7.11 is considered). There was a significant difference in the amount of effort spent by the tool-supported developers (Mean = 12; Std. Dev. = 3) and the manual developers (Mean = 50; Std. Dev = 14; t(10,89) = -8,84; p = 0,01). Therefore, we can reject $H_0{}^3$.

Testing $H_0{}^4$

An independent-samples t-test was conducted to compare the number of errors made by the tool-supported developers versus the manual developers (see Fig. 7.11). Since the significance value of Levene's test (0,00) is less than 0,05, equality of variances is not assumed (i.e. the fourth row in Fig. 7.11 is considered). There was a significant difference in the number of errors made by the tool-supported developers (Mean = 0; Std. Dev. = 0) and the manual developers (Mean = 3,6; Std. Dev = 2,3; t(10) = -5,16; p = 0,01). Therefore, we can reject $H_0{}^4$.

7.4 Validity Evaluation

In this section, we discuss the threats to the validity of the experiment. We organized these threats using the categorization proposed in [24]; each title in this section is a category of validity threats. For each category, we first provide a short explanation,

and then discuss how we addressed this category of threats in our experiment. Most of the short explanations are adopted from [86].

7.4.1 Conclusion Validity

This category of threats effect the ability to draw correct conclusion about the relation between the treatment and the outcome of the experiment.

In our experiment, we identified two categories of threats to the conclusion validity: low statistical power, and reliability of treatment implementation [24].

Low Statistical Power

The power of a statistical test is the ability of the test to reveal a true pattern in the data. If the power is low, then there is a high risk that an erroneous conclusion is drawn. Therefore, we performed a post-hoc analysis to find the actual power we achieved in our statistical tests. We used G*Power [37] for calculating the power. For each hypothesis, the input and output parameters of the power analysis are listed in Fig. 7.12.

As visible in the last row of Fig. 7.12, the power we achieved is either very close to or more than 0,80. Since 0,80 is the commonly accepted minimum level of power, we can conclude that the power level of our analysis is not a major threat to the validity of our conclusions. The power we achieved also indicates that the number of participants was quite sufficient for our experiments.

In the power calculation, G*Power required us to provide non-zero and positive values for the means and standard deviations. We could not give the real values of (a) the mean number of errors (i.e. μ_2 of ${H_0}^2$ and ${H_0}^4$) and (b) the standard deviations of the errors (i.e. σ_2 of ${H_0}^2$ and ${H_0}^4$) of the tool-supported participants; because these values are all 0. Nevertheless, we approximated these values by giving the lowest possible values for the mean number of errors (i.e. 0,000001) and the standard deviation of errors (i.e. 1), regarding the tool-supported participants. This approximation can be considered as a threat to the validity of the conclusions drawn from the hypotheses ${H_0}^2$ and ${H_0}^4$.

Note that, the threat mentioned above does not arise from the preparation, design, or operation of the experiments; it arises from the outcome of the experiments. Therefore, this threat could not be predicted and avoided before the experiment was conducted. In principle, the manual participants could have repaired all the defects, in which case the number of errors of manual participants would have been

	Hypotheses			
	$H_0{}^1$	$H_0{}^2$	$H_0{}^3$	$H_0{}^4$
Input				
μ_1	32,44	4,67	11,75	3,64
μ_2	64,11	0,000001	49,73	0,000001
σ_1	13,983	3,082	3,079	2,335
σ_2	14,786	1	13,929	1
d	2,3687463	1,788854	4,027992	1,897367
α	0,01	0,01	0,01	0,01
n_1	9	9	11	11
n_2	9	9	12	12
Output				
δ	5,024870	3,794733	9,649652	4,545423
Critical t	2,920782	2,920782	2,831360	2,831360
Df	16	16	21	21
$1-\beta$	0,971638	0,793631	1,000000	0,945239

Figure 7.12: The input and output parameters of the post-hoc power analysis for independent-samples and two-tailed t-tests (per hypothesis). In this analysis, the equality of the sample sizes is assumed. μ_1 and μ_2 denote the means of the first and second samples. σ_1 and σ_2 denote the standard deviations of the first and second samples. d denotes the effect size calculated by G*Power based on the means and standard deviations. α denotes the significance level we have chosen for rejecting the null hypotheses. n_1 and n_2 denote the sizes of the samples. δ denotes the non-centrality parameter calculated by G*Power. *Critical t* denotes the critical t-value calculated by G*Power. Df denotes the degree of freedom. $1-\beta$ denotes the power we achieved in our statistical analysis.

0; or the tool-supported participants could have made some errors, in which case the number of errors of tool-supported participants would have been non-zero. The fact that the tool-supported participants did not make any errors is the outcome of the experiment.

Reliability of Treatment Implementation

The implementation of a treatment means the application of the treatment to a subject. To improve the reliability of treatment implementation, the implementation

must be as standard as possible over different participants and occasions.

In the experiment with the students, each student participated in the experiment at the same time. This was important to avoid information exchange between the students, hence to prevent the threat explained in Section 7.4.2. Consequently, this required an instructor to give the tutorial for the tool-supported group in a laboratory, and another instructor to give the tutorial for the manual group in another laboratory. Since different instructors gave the tutorial, there may be a threat to the reliability of treatment implementation.

7.4.2 Internal Validity

Internal validity threats are issues that can affect the measurements of the independent variable, without the researcher's knowledge. Therefore, these kinds of threats may influence the validity of conclusions about a possible causal relationship between a treatment and the corresponding outcome.

In our experiment, we identified and addressed three types of threats to the internal validity: maturation, instrumentation, and diffusion or imitation of treatments [24].

Maturation

The maturation threat arises when subjects are affected negatively (e.g. tired or bored), or positively (unintended learning) during the experiment.

To reduce the unintended learning effect in our experiment, we prepared an example (i.e. preliminary) treatment for the participants, so that they got used to the tasks they were required to perform. In this way, we aimed at improving the accuracy of our measurements. The example treatment was the first treatment of each participant, and the related data is excluded during the analysis presented in Section 7.3.

Instrumentation

This type of threat arises from an improper design of instruments such as data collection forms, document to be inspected in an inspection experiment, etc.

We conducted preliminary runs of the experiment to test the quality of the instruments listed in Section 7.1.7. These runs enabled us to improve the quality of these instruments. The four participants of these preliminary runs were different than the

participants of the actual experiment. During the analysis presented in Section 7.3, we excluded the data of the preliminary runs.

Diffusion or Imitation of Treatments

This threat arises if participants are prematurely informed about the treatments, and behave differently due to this information.

As explained in Section 7.2.2, we avoided this threat in the experiment with the students, and we took effective precautions in the experiment with the developers.

7.4.3 Construct Validity

Threats to construct validity influence the ability to draw correct conclusions about the relation between the results of the experiment and the hypotheses that are being tested using these results. Some of such threats are related to the experimental design, and others are related to social factors.

In our experiment, we identified and addressed two types of threats to the construct validity: confounding constructs and levels of constructs, and experimenter expectancy [24].

Confounding Constructs and Levels of Constructs

These kinds of threats arise from the fact that there are confounding constructs (e.g. experience of subjects) that are not taken into account in an experiment.

As explained in Section 7.1.5, we measured the programming experience of the students to understand their background. However, we did not balance the tool supported v.s. manual groups according to the experience of the students, because we did not have any means to validate their programming experience. Instead, we divided them randomly. As a result, in the tool supported group there were three students with experience level 2 (see Section 7.1.5), three students with experience level 3, three students with experience level 4, and three students with experience level 5. Whereas, in the manual group there were two students with experience level 2, three students with experience level 3, four students with experience level 4, and one student with experience level 5. The lack of balance in the experience may be a threat to the validity of the results related to the students. However, we do not think that this threat is severe, because the weighted average of the experience in

the tool-supported and manual groups were not too different (i.e. respectively 3,5 and 3).

We do not think that there is an important lack of balance due to the differences in the programming experience of developers. In section 7.1.5, we indicated that each developer had at least 4 years of professional software development experience. Considering the nature of the tasks in the experiment, we think that a developer with 4 years of experience can perform as well as a developer with more than 4 years of experience.

Experimenter Expectancy

The experimenters may bias the result of an experiment based on what they expect from the experiment. This is a threat to the construct validity.

The purpose of our experiment was to evaluate the tools developed by the author of this thesis. Hence, the experimenter expected that the tools are beneficial. To eliminate this threat, we planned, conducted, and analyzed this experiment together with Klaas van den Berg, who did not have any specific expectations from this experiment.

7.4.4 External Validity

The threats to external validity limit the ability to generalize the results of the experiment.

Interaction of Selection and Treatment

This threat arises if the selection of subjects do not adequately represent the population for which the results need to be generalized.

The participants of this experiment are not randomly selected from a large population of developers and students. The developers were the volunteers at ASML, and the students were the participants of a course at the university. Therefore, the results of this experiment cannot be generalized for a larger population of students and developers. However, this does not devaluate the results of this experiment, because our purpose was to evaluate the tools, and we have empirical evidence indicating that the tools are beneficial both for a homogenous set of students, and a homogenous set of developers.

Interaction of Setting and Treatment

This threat arises if the experimental setting or the instruments are not representative of, for example, industrial practice.

In our experiments, we used real-life source code and real-life VisuaL specifications, but we injected relatively simple defects. Below, we explain why we could not use real-life evolution scenarios instead of injecting simple defects.

At ASML, developers maintain source code upon receiving a "change request / problem report (CRPR)". Implementing a CRPR typically involves several modifications to the existing source code. Hence, a real evolution scenario typically consists of several additions, deletions, and modifications of function calls, control statements, variables etc. Using a real CRPR in our experiments was infeasible, because

- A CRPR is informally written in English. Therefore, different participants might have (mis)interpreted the CRPR differently. Consequently, we would have lost the control in the experiment, and the results would have been inconclusive.
- No matter there is tool support or not, domain expertise is necessary for implementing a CRPR. Hence, the students could not have implemented the CRPRs. Moreover, only 2 out of 23 developers were in the team that was developing the software component we investigated. Hence, the remaining 21 developers were not domain experts, either.
- The implementation of a CRPR involves multiple changes to the source code. The changes that were not due to the ETB defects or incompatibilities would have been confounding factors in our experiment. In other words, we would have lost control in the experiment, and the results would have been inconclusive.
- Due to the domain expertise required for implementing a CRPR, we cannot estimate how much time is necessary for an average person to implement a given CRPR. Since we could not occupy the participants for more than 3 hours during the experiment, we could not have used a real CRPR.

Due to the reasons listed above, we had to inject relatively simple defects that can be repaired without any domain knowledge. This may be a threat to the generalization of our results to the industrial practice. Frankl et. al. [40] discuss the challenges in creating defect models that enable systematic investigation of reliability in software systems.

The functions that were used during our experiments were chosen by the domain expert of the component in which the functions were placed. The expert chose these functions, because on that day he had to maintain them. Hence, the functions were

not randomly chosen. This is a threat to the generality of our results to arbitrary functions.

Background of Participants

The background of the student participants is classified based on the size of the largest program they have written using one of the imperative languages, whereas the background of the developer participants is classified based on the number of years of programming experience. Using a uniform criteria to classify the background of both the students and the developers would not make sense, since students typically do not have professional experience, and the professional developers typically write programs on a daily basis over multiple years.

The non-uniform criteria for classifying the background of students and developers would be a threat to the validity of any conclusion that would compare the performance of students with the performance of the developers. In this thesis, we did not present such a conclusion; i.e. any conclusion that is presented in this thesis is either about the students or the developers, but not the combination.

Intuitively, the use of tool support should reduce the difference between developers and students. However, this is not the case according to the results of the experiments. The results suggest that the difference between the students and the developers is smaller if there is no tool support: The manual students spent 128% (=64/50) of the effort spent by manual developers, whereas the tool-supported students spent 267% (=32/12) of the effort that is spent by the tool-supported developers. This counter-intuitive result would be a threat to the validity of any conclusion that compares the performance of the students with the developers. In this thesis, we do not present such a conclusion, because comparing the students with developers was not our goal. If this were our goal, then we would have stated additional hypotheses about the difference between students and developers, and we would have designed the experiment differently. The existing design and execution of the experiments is not suitable for comparing students and developers.

Chapter 8

Related and Future Work, Discussion, and Conclusions

8.1 Related and Future Work

8.1.1 Software Documentation

A software system is by nature an abstract, mathematical product, which is described, specified, and implemented in documents (i.e. text, diagrams, etc.). Parnas [74] provides the following definitions for different types of software documents:

- A *description* is a statement of some of the actual attributes of a product.
- A *specification* is a statement of some of the properties required of a product.
- A *model* is a simplified or reduced sized version of a product.
- A *prototype* is an early, full-scale version of a product.

According to these definitions, the documents created using VisuaL are specifications. Each VisuaL specification contains a constraint (i.e. requirement) on the possible sequences of function calls from a given function.

Parnas [74] states that different notations are necessary for documenting different types of software products. He distinguishes four types of software products: Real-time and interactive systems, terminating programs, modules, and objects. VisuaL is best suited for specifying some of the properties of the behavior of terminating programs.

Since VisuaL is a language for specifying constraints on the possible sequences of function calls from a given function, it is related to the trace assertion method

(TAM) [16, 82, 53, 55], and sequence-based software specification [75]. Using TAM, one can specify the legal traces (e.g. function calls) of a system, together with the values of the legal traces. The notation of TAM is textual, hence it is different than the notation of VisuaL, which is graphical. Furthermore, VisuaL does not have any language construct for specifying the values of the traces yet. Our intention is to keep VisuaL as simple as possible, hoping that it can be more easily adopted by the software engineers at ASML. Upon possible demand in the future, we may consider extending VisuaL for handling data.

The documents created using VisuaL are fundamentally different than the traditional flowcharts [20, 71] and UML activity diagrams [8]. For a given specification in VisuaL, there can be multiple different implementations of a function, where each implementation embodies a different flow of control (i.e. a flowchart or an activity diagram). Hence, VisuaL specifications are more abstract than flowcharts and activity diagrams. This higher level of abstraction is enabled by the context-sensitive wildcards explained in Section 2.2.1.

It is possible to semi-automatically construct VisuaL specifications based on existing source code. This is explained in [77].

8.1.2 Temporal Logics and Model Checkers

VisuaL is a graphical language for specifying design constraints on the behavior of algorithms. Such a constraint is a logical or temporal property that must be satisfied by each possible execution of the corresponding algorithm.

Since an algorithm does not execute indefinitely (otherwise it would not be an algorithm by definition [63]), each possible execution of an algorithm is finite. Thus, VisuaL is a language that is suitable for expressing properties of finite executions. For example, the following constraint can be expressed using VisuaL, as shown in Fig. 8.1.

C6: In each possible sequence of function calls from the function `f`, there must be at least one call, and *the last call* must be a call to the function `traceOut`.

In contrast to the executions of algorithms, the executions of finite-state concurrent systems [65] or reactive systems [46] are often infinite. Therefore, we call such systems **non-terminating systems**. To express the logical and temporal properties of non-terminating systems, several model checking formalisms are available: FLTL [42], LTL [23], CTL [23]. To be able to specify the constraint C6 (see above) using one of these model checking formalisms, one has to first translate the finite executions of the algorithm to infinite executions. For example, if *seq* =<`g`, `h`,

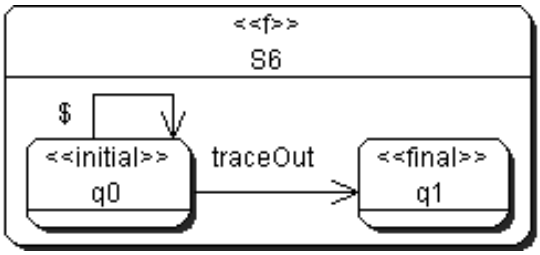

Figure 8.1: A VisuaL specification indicating that *the last call* from the function `f` must be a call to `traceOut`.

`traceOut`$>$ is a finite sequence of function calls representing an execution of the function `f`, then *seq* can be translated into the infinite sequence $seq' =<$ `g`, `h`, `traceOut`,$\odot, \odot, \odot, ... >$, where $\odot$ is a special symbol marking the end of *seq*. Upon this translation, the infinite sequence seq' can be checked against the LTL formula *eventually*(`traceOut` *and* (*next* $\odot$)). This LTL formula is semantically different than the VisuaL specification shown in Fig. 8.1. However, the formula 'mimics' the specification, provided that the finite sequences are extended to infinite sequences as explained above. FLTL and CTL can also be used for creating formulas similar to the LTL formula stated above.

The semantics of a temporal logic formula is defined with respect to a model of the underlying system [23]. For example, the semantics of an LTL formula is often defined with respect to a Büchi automaton that represents the underlying system [23]. Therefore, if the underlying system evolves, then the semantics of the formula also evolves 'automatically' (i.e. one does not need to manually update the formula for this evolution). Hence, the evolvability that is exhibited by DARs (see Section 2.5) is also exhibited by temporal logic formulas.

FLTL is a formalism for specifying state-based LTL properties of event-based systems. An atomic proposition of an FLTL formula is called fluent [42], which is a property that holds after it is initiated by an action, and no longer holds when terminated by another action. "A well-formed FLTL formula is an LTL formula whose atomic propositions are fluents" [61]. If a property of an event-based system is specified as an FLTL formula, and the system is modelled as a labelled transition system (LTS) [65], then the LTSA tool [65] can check whether the LTS satisfies the FLTL formula.

Fluent model checking using FLTL and the LTSA tool is primarily intended for verifying the properties of systems at the architecture level. The LTSA tool takes an LTS [65] as the architectural model of the system, and verifies whether the model satisfies the given properties that are expressed in FLTL. CheckSource however is a source code verification tool. It automatically extracts the model from C code,

and verifies whether the model (hence the C code) satisfies the given properties expressed in VisuaL.

The nodes of a Simplified Control-Flow Graph (SCFG) are labelled with the identifiers of functions, and the edges are not labelled. One can transform a SCFG to an LTS [65], by labelling each edge with the label of the target node. Upon this transformation, one can also perform the verification using the LTSA tool and the FLTL language.

Bandera is an integrated collection of analysis tools for the verification of Java programs. Bandera has a slicer component that compresses the paths in a program by removing control points, variables, and data structures that are irrelevant for checking a given property. CheckSource performs a similar compression, too. It extracts the SCFGs of C functions, where the nodes represent function calls, and the edges represent the possible flow of control between the function calls. Hence, a path through a SCFG does not contain any control points, variables, and data structures. Bandera has an abstraction engine that allows the users to configure the level of abstraction in the extracted models. CheckSource does not have such a feature; the level of abstraction of a SCFG cannot be configured. Bandera can automatically map counter-examples to Java code. CheckSource also maps counter examples to C code. This mapping is done by reporting a path through a function, such that this path violates the constraint expressed by the VisuaL specification. Bandera has a graphical user interface. CheckSource is a command prompt tool, but it is integrated with Borland Together [5], which is used for creating the VisuaL specifications.

CheckSource performs static program analysis to verify source code. There is a substantial body of research in static analysis (e.g. [15, 22, 27, 36, 13, 35]). This line of research is mainly focused on finding general types of bugs (e.g. deadlocks, memory leaks) in source code. In contrast, we provide a language for defining the desired properties of software behavior, and CheckSource finds the bugs based on the desired behavior specified by software engineers.

As we explained in Section 6.1, embedded software is usually a combination of reactive and non-reactive (i.e. algorithmic) parts. If it is necessary to verify the properties of both parts, then one can use the existing model checking formalisms (e.g. FLTL, LTL) for specifying the properties, and let the state-of-the-art model checkers such as LTSA [65, 42] and Bandera [25, 51, 34] perform the automatic verification. Since we are interested in verifying the properties of only the non-reactive part (i.e. finite executions), we perform the model checking using the language inclusion algorithms of finite-word automata theory [63].

The problems presented Section 6.3.1 cannot be satisfactorily solved only by source

code verification techniques; it is necessary to perform source code transformation, too. Therefore, we integrated a source code transformer to our analyzer. This integration enables the combination of verification and transformation.

Using VisuaL, one can express both the properties to be verified and the rules for transformations. To our best knowledge, there is no obvious way to express the transformation rules using the existing temporal logic formalisms.

The state-of-the-art model checking tools such as Bandera and LTSA support temporal logics (e.g. LTL, FLTL) for expressing the properties of software systems. Temporal logics are textual formalisms. In today's industrial practice however, there is a large group of practitioners who prefer graphical formalisms (e.g. statecharts) for specifying their software. Hence, there is a gap between the preferences of these practitioners and the specification languages supported by the state-of-the art model checking tools. VisuaL can be considered as an initial effort to bridge this gap, as we explain in Section 8.1.3.

8.1.3 Extending the VisuaL Language for Expressing the Logical and Temporal Properties of Non-Terminating Systems

In Section 8.1.2, we mentioned some of the existing formalisms for expressing the logical and temporal properties of non-terminating systems. Yet another formalism for expressing such properties is Büchi automata [23, 12]. There are three differences between a Büchi automaton and a DFA:

1. A Büchi automaton with an input alphabet Σ either accepts or rejects any *infinite* sequence of symbols from Σ; whereas a DFA with an input alphabet Σ either accepts or rejects any *finite* sequence of symbols from Σ.
2. A Büchi automaton can perform transitions non-deterministically; whereas a DFA performs transitions deterministically.
3. A Büchi automaton has multiple initial states; whereas a DFA has exactly one initial state.

The set of infinite sequences of symbols accepted by a Büchi automaton is called ω-regular language [23].

Based on the contents of this chapter, one can define a new type of automata, say *Non-deterministic Abstract Infinite-sequence Recognizer* (*NAIR*), such that the differences between a NAIR and a DAR is the same as the differences between a Büchi automaton and a DFA:

1. A NAIR either accepts or rejects any *infinite* sequence of symbols, whereas a DAR either accepts or rejects any *finite* sequence of symbols.
2. A NAIR can perform transitions non-deterministically; whereas a DAR performs transitions deterministically.
3. A NAIR has multiple initial states; whereas a DAR has exactly one initial state.

Since a NAIR would accept or reject any infinite sequence of symbols, it could be used for expressing logical or temporal properties of non-terminating systems. Accordingly, the syntax of VisuaL could be extended, such that each VisuaL specification represents a NAIR, and each NAIR can be represented by a VisuaL specification. Hence, the extended VisuaL could be used for expressing the logical and temporal properties of non-terminating systems.

If the extended VisuaL would be integrated as a front-end to the state-of-the-art verification tools such as Bandera [25, 51, 34] and LTSA [65, 42], then we believe that these tools would become more accessible to the large audience of practitioners who prefer graphical formalisms for specifying their software.

The recent implementation of LTSA [42] translates FLTL formulas to "tester automata with * transitions". These automata are already very similar to, if not the same as, NAIRs. Hence, LTSA already has a suitable foundation for supporting a graphical formalism similar to the extended VisuaL.

In Section 2.4.1, we have shown that the set of ORLs is a proper superset of the set of RLs. The key reason for this was as follows: Since a DAR either accepts or rejects *any* finite sequence of symbols (i.e. without any restriction to a specific alphabet), the set of sequences symbols accepted by a DAR cannot be a regular language. This line of reasoning could be reused for showing that the set of sequences symbols accepted by a NAIR is not necessarily an ω-regular language. Hence, one could show that NAIRs define a new family of formal languages, say *open ω-regular languages*, such that the set of open ω-regular languages is a proper superset of the set of ω-regular languages.

Büchi automata are known to be more expressive than LTL [87]. That is, there is at least one ω-regular language that cannot be expressed by an LTL formula, whereas any LTL formula expresses an ω-regular language. Since NAIRs would express open ω-regular languages, NAIRs would also be more expressive than LTL. Consequently, if the extended VisuaL would be integrated with the existing verification tools such as Bandera, then the users of these tools could (a) express all the properties that can also be expressed using LTL, (b) express additional properties that cannot be expressed using LTL, (c) enjoy a familiar (statechart-like) diagrammatic notation for expressing the properties, and (d) benefit from the improved evolvability explained

in Sections 2.5 and 2.8.

8.1.4 Wildcards in Automata-Based Testing and Verification

The idea of using wildcard transitions in automata is not new. The * transitions [42] facilitate partial order reduction for the properties that are closed under stuttering. These transitions are similar to the # transitions of DARs (Section 2.3.4). Hence, * transitions enable the evolvability explained in Section 2.5. This evolvability is also mentioned by Giannakopoulou and Magee [42], as follows: "An advantage of tester automata that use * transitions is that they are independent of the system, and can be reused across systems with the same set of fluents". The * transitions are wildcards, but they are not context-sensitive.

The "*"-transitions [56], and the *other*-transitions [21] seem to be similar to CSWs. However, the semantics of the "*"-transitions and the *other*-transitions are not formally defined in the respective literature. Therefore, it is not clear whether the symbols matched by such transitions are from a given alphabet or from the universal set of symbols. We have formally defined the semantics of CSWs, and according to this definition, the symbols matched by a CSW are from the universal set of symbols. The fact that CSWs can match symbols from the universal set of symbols has both theoretical and practical implications, which are explained in this chapter.

8.1.5 Adding CSW to Existing Graphical Languages

There are several graphical languages for expressing the behavioral design of software systems: statecharts [47], activity diagrams [8], sequence diagrams [8], collaboration diagrams [8], etc. Wieringa [84] surveys such graphical languages.

In this chapter, we provided theoretical and practical insights about CSW. We believe that these insights can be reused for adding CSW to the existing graphical languages. For example, the set of abstraction mechanisms offered by statecharts could be extended with CSW. This extension would enable the users of statecharts to selectively abstract from the events responded by a reactive system.

8.1.6 Defining a Hierarchy of Open Languages

In section 2.4.2, we have explained that $(\Upsilon \setminus \{g\})^*$ is an ORL but not a CFL. We think that it is possible to generalize this explanation for showing that $(\Upsilon \setminus \{g\})^*$ is not an element of any language family in the Chomsky hierarchy [63]. Subsequently, it is possible to show that the set of ORLs is not a subset of any language family in the Chomsky hierarchy.

In this chapter, we introduced the open regular languages. We think that it is also possible to define 'open context-free languages', 'open context-sensitive languages', etc.; and discover the relation between these 'open languages'. One could also discover the relation between 'open languages' and the language families in the Chomsky hierarchy. Based on our current intuition, we imagine a hierarchy of 'open languages', such that this hierarchy includes the Chomsky hierarchy.

8.1.7 Automata for Strings Over Infinite Sets of Symbols

In Section 2.3.4, we have explained the key difference of DARs from DFAs: DARs can accept or reject finite sequences of symbols from the universal set of symbols, which is an infinite set. Recognizing finite sequences of symbols from an infinite set of symbols has also been studied by Kaminski and Frances [57], Globerman and Harel [43], Milo et. al. [69], and Neven et. al. [72].

Kaminski and Frances [57] introduced *register automata*: A register automaton is a finite-state machine equipped with a finite number of registers. Each register can store a symbol. While processing the input tape, a register automaton compares the current symbol of the tape with the symbols in the registers. Based on the current state and the result of the comparison, the automaton decides (a) the next state, (b) whether to store the current symbol (of the tape) in a register (by overwriting the existing symbol in the register), and (c) whether to move to the left or the right position on the tape, or to stay at the current position of the tape. The symbols on the input tape of a register automaton can be from an infinite set of symbols.

A DAR has a built-in wildcard symbol (i.e. the # symbol), which matches infinite number of symbols. Whereas, a register automaton has finite number of registers each of which contains exactly one 'regular' (i.e. non-wildcard) symbol. Therefore, the symbol in a register of a register automaton cannot match infinite number of symbols. As a result, a register automaton cannot 'simulate' the #-transitions of a DAR. A register automaton can modify the contents of the registers, while processing the input tape. Whereas, a DAR cannot modify the contents of the abstract input alphabet. These differences between a DAR and a register automaton suggest that

DARs and register automata possibly have different expressive power.

Globerman and Harel [43], and Milo et. al. [69] introduced different variations of *pebble automata*. In this section, we consider the variation introduced by Milo et. al. [69]. A pebble automaton is a finite-state machine equipped with a finite number of consecutively numbered pebbles. A pebble can be placed on a position of the input tape, so that the symbol at that position is marked by the automaton. While processing the input tape, a pebble automaton can drop down the pebbles or pick them up. This is regulated according to the stack convention: the i^{th} pebble can be dropped down only if the $(i-1)^{th}$ pebble is already on the tape, and the i^{th} pebble can be picked up only if the $(i+1)^{th}$ pebble is already picked up. While processing the input tape, a pebble automaton compares the current symbol of the input tape with the symbol marked by the most recently dropped pebble. Based on the current state and the result of the comparison, the automaton decides (a) the next state, (b) whether to drop down or pick up a pebble, and (c) whether to move to the left or to the right position on the tape, or to stay at the current position of the tape. The symbols on the input tape of a pebble automaton can be from an infinite set of symbols.

A pebble automaton has a finite number of pebbles each of which can be used for marking exactly one 'regular' (i.e. non-wildcard) symbol. Therefore, the symbols marked by the pebbles of a pebble automaton cannot match infinite number of symbols. As a result, a pebble automaton cannot 'simulate' the #-transitions of a DAR. By dropping down and picking up the pebbles, a pebble automaton can modify the set of marked symbols. Whereas, a DAR cannot modify the contents of the input abstract alphabet. These differences between a DAR and a pebble automaton suggest that DARs and pebble automata possibly have different expressive power. A formal comparison of DARs with register and pebble automata is outside the scope of this article. Therefore, we leave such a comparison as a part of the future work.

Neven et. al. [72] have investigated the expressive power of different types of register and pebble automaton, and compared them with first order logic and monadic second-order logic. They investigated variations of register and pebble automaton: one way or two way tape readers; and deterministic, non-deterministic, or alternating (i.e. hybrid) versions.

8.1.8 Aspect-Oriented Programming (AOP)

In AOP terms, the ETB of a system (Section 6.3) is scattered [39] over and tangled [39] with the implementations of the activities. Hence, the ETB is a crosscutting concern [39]. The event specifications (e.g. the pattern in Fig. 6.7) correspond to

pointcuts [39], the event points (e.g. the point located after '`;`' in Line 7, Listing 6.1) correspond to joinpoints [39], the event calls (e.g. `preprocessed();`) correspond to advices [39], and the VisuaL specifications in which an event is specified and an event call is bound (e.g. Fig. 6.7) correspond to aspects [39]. Thus, part of the solution presented in this chapter exhibits the fundamental characteristics of the AOP technology. Note that our approach and the tools can be used for weaving not only event calls but also arbitrary advice code.

In Section 7.1.1, we explain that there are 55 events mapped to 102 source code points in the component using which we tested our the solution. Hence, a full-blown application of our solution requires creation of 55 aspects for weaving 102 calls to 55 event functions at 102 joinpoints. That is, 1 aspect needs to be created per 1.9 joinpoints on the average. This means that the ETB of the system is not highly-replicated, in contrast to the classical examples of crosscutting concerns: tracing, parameter checking, etc. By addressing the problems presented in Section 6.3.1, we have shown a case in which AOP can be useful for improving the evolvability of a crosscutting concern that is not highly-replicated.

Recent research [10, 80] has raised awareness about the problems of aspect-oriented systems in the context of software evolution. It is argued that seemingly harmless modifications to base code may break the functionality of aspects, which increases the workload of aspect developers, and sometimes makes it infeasible to realize a working system. To address such problems, on one hand Aldrich [10] proposed to restrict joinpoint models and advising mechanisms. His proposal has been incorporated to AspectJ [1] by Ongkingco et. al. [73]. On the other hand, Sullivan et. al. [80] proposed to constrain the implementation of the base programs, so that the aspects can properly function. Our solution follows the latter approach: The compatibility constraints (Section 6.5) are interface specifications [80] (or contracts [18]) that base code developers implement. The analyzer verifies whether these interface specifications (or contracts) are correctly implemented. According to the four levels of contracts proposed by Beugnard et. al. [18], the compatibility constraints can be classified under the third level: "Constraints on the temporal ordering of system services and method calls."

In a nutshell, a VisuaL specification is a Moore machine-like automaton that generates output strings from an output alphabet, while recognizing regular patterns of input symbols from an input alphabet. Therefore, using VisuaL one can specify history-sensitive pointcuts that can identify function call joinpoints, based on regular patterns of function calls. Hence, VisuaL is related to some of the existing trace-based and history-sensitive approaches [11, 30, 31, 33]. In these approaches, the state of the pointcut advances only if the encountered input symbol is in the input alphabet. In VisuaL, however, one always explicitly specifies the next state for

the symbols that are not in the input alphabet as well. In this respect, our language is similar to MOPS [22]. Using this feature, one can naturally express "Function call c_1 has to come immediately after function call c_2", or "Whenever function call c_1 comes immediately after function call c_2, weave advice A".

VisuaL, enables *concisely* localizing the information about a mandatory event (see Section 6.6 and Fig. 6.8). If the existing trace-based languages [11, 30, 31, 33] are used however, it is necessary to create at least one declare error [1] pointcut for the compatibility constraints (e.g. C1 and C2 in Section 6.5.2), and another one for the event specification. So, the information about a mandatory event would be distributed over multiple pointcuts, in which case conciseness and locality-of-information would be suboptimal.

The programming technique enabled by VisuaL can be considered as concern-shy programming [62]. The `$`-labelled edges (see Figures 6.5, 6.6, 6.7, 6.8) abstract away from the function calls that are not parts of the concerns represented by the VisuaL specifications. In our case, these concerns are either compatibility constraints, or the events of a system.

Property checking and program queries are other applications of trace-based approaches [32, 44, 66]. In these approaches, one writes queries over execution traces of programs, often for detecting errors, flaws, etc. Hence, weaving is not the purpose of these applications. In contrast, our purpose is *both* weaving additional behavior, *and* enforcing design rules [80].

Some constraints may apply to multiple functions in the system. For example, "In each possible sequence of function calls from any function, `traceIn` must be the first function call, and `traceOut` must be the last function call". Such constraints can be specified by writing a regular expression in the stereotype of the container node of the specification, as explained in Section 2.2.1. In such a case, the regular expression can be seen as a pointcut [39], and the specification can be seen as an aspect [39].

In Chapter 6, we addressed some of the problems arising from the possible mistakes during the evolution of activities. The problems due to possible mistakes during the evolution of statecharts is a part of future work.

8.1.9 State Machines, Interval Logic, TSL, and Rapide

State machines have long been used for modelling both software systems, and the properties that must be satisfied by these systems [19, 41]. Binder [19] mentions that complete state machines are necessary for testing purposes. That is, one has to

define the next state for each possible incoming symbol from each state. Therefore, the state machines created in this way suffer from the evolvability problem explained in Section 2.5. VisuaL addresses this problem by enabling the engineers to express complete state machines in a concise and evolvable way.

Interval logic, which is originally introduced by Schwartz et. al. [78], is a type of temporal logic that is specifically designed for expressing abstract requirements that a program must satisfy. Dillon et. al. [29, 28] have recognized the need for graphical languages for expressing interval logic formulas, and they introduced Graphical Interval Logic. Later, Bates [17] introduced Event-Based Behavioral Abstraction (EBBA) for debugging heterogenous distributed systems. EBBA provides a textual way to express interval logic expressions.

Rosenblum [76], presents Task Sequencing Language (TSL), which is designed for expressing design constraints on the behavior of concurrent programs. TSL is later evolved in to the architecture description language Rapide [64], which provides extensive support for event-based specifications of software architectures.

8.2 Discussion and Limitations

8.2.1 VisuaL

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 specification can be mapped to a formal representation of automata), and evolution-proof (through the use of wildcards, one can specify only necessary ordering constraints).

The VisuaL specifications are drawn in the activity diagram editor of Borland Together [5]. We have implemented a plug-in for Borland Together so that the VisuaL specifications drawn using Borland Together can be recognized by CheckSource as input.

VisuaL is not expressive enough for constraining the possible sequences of function calls using data values. For example, one cannot specify the following constraint: If a possible sequence of function calls from `preprocess` contains a subsequence in which the value of `reticleIsClean` is 0 (i.e. false), then the last function call in this subsequence must be a call to `cleanReticle`. To enable the specification of such constraints, VisuaL must be extended with new constructs that enable data analysis.

In Section 6.2.2, we explained that the processed event is mapped to the point located

after '}' in Line 8, Listing 6.2. One can identify this point if and only if one can match '{' in Line 5 with '}' in Line 8. Parenthesis matching can be implemented in a given language, if and only if one can specify a non-deterministic push-down automaton in that language. Using VisuaL however, one can specify only deterministic finite-state automaton. Therefore, VisuaL is not suitable for identifying the point located after '}' in Line 8, Listing 6.2; hence to define the processed event.

To determine the level of expressive power for VisuaL, we investigated the descriptions[1] of the events in the design documents of the software component mentioned in Section 7.1.1. We did not find any non-regular description (e.g. "The event e occurs if function b is called exactly the same number of times as function a."). We also discussed the event descriptions with some of the key developers and the architect of the component. The conclusion was "The application domain of this component does not contain any event that requires a non-regular pattern". Therefore, we decided to keep VisuaL simple, and limit its expressive power to regular patterns. If it is necessary in the future, then we can extend VisuaL by (a) adding new syntactic elements to manipulate a stack with push and pop operations, and (b) making VisuaL non-deterministic. If VisuaL is extended, then context-free patterns can also be specified using VisuaL, and the processed event can also be defined using VisuaL.

VisuaL is a graphical language whose syntactic elements are labelled rectangles and arrows. As the size and complexity of a VisuaL specification increases, the comprehensibility and the ease of layout decreases. Therefore, it is essential that constraints can be defined using relatively less rectangles and arrows. This is possible only if software is decomposed into relatively small functions. Since this was the case for the software component that we investigated, VisuaL was suitable for specifying the constraints.

Graphical languages such as UML activity diagrams [8] or flowcharts [71] are frequently used for designing the flow of control within procedural programs such as C functions. Although VisuaL specifications are also graphical artifacts of behavioral design, they are fundamentally different than activity diagrams. An activity diagram is a control-flow *model* [74] of a function (or procedure, method, subroutine); different functions that implement the same activity diagram have the same control-flow. Whereas, a VisuaL specification is a *constraint* (i.e. formally specified requirement) on the control-flow of a function; different implementations that conform to a VisuaL specification may have different control-flow. Thus, VisuaL specifications are typically more abstract than activity diagrams: a VisuaL specification is a constraint on not only the implementation of a procedure but also the activity diagram that is the control-flow model of the procedure.

[1]These descriptions were written in English.

A rigorous comparison of VisuaL and process algebra [52], in particular Π-calculus [68], is a part of our future work

8.2.2 CheckDesign

One can write multiple VisuaL specifications for a given C function. Once CheckDesign verifies that a the specifications are consistent, it is possible to generate 'skeleton code' of the function, using the analysis report created by CheckDesign: To verify that specifications are consistent, CheckDesign constructs the shortest sequence of function calls that is matched by the pattern of each specification, and outputs this sequence in the analysis report. It is possible to generate code such that the body of the function consists of only this sequence. Such an implementation of the function is consistent with each specification. If there are multiple shortest sequences of function calls, then it is possible to select a specific sequence, based on some optimization criteria (e.g. select the sequence that has the minimum number of calls to a specific function).

Since VisuaL and DARs are equivalent, it is possible to implement the algorithms of CheckDesign such that they operate directly on VisuaL specifications, without converting the specifications to DARs. This may arguably improve the runtime performance. However, the standard proofs and algorithms of automata theory has to be adapted for VisuaL, so that the operations are performed directly on VisuaL specifications.

8.2.3 CheckSource

The SCFG generator of the current CheckSource has limitations that can be overcome through further development. For example, CheckSource cannot recognize calls through a function pointer. To overcome this limitation, we need to incorporate pointer analysis capabilities to CheckSource. Such limitations are neither fundamental, nor they are severe regarding the actual wafer scanner's software: We could manually prepare 55.000 lines of source code for our tools in less than one hour, because ASML has well-applied coding conventions.

Execution of some paths through a function may be infeasible. The current implementation of CheckSource cannot rule out infeasible paths through a function, because it does not analyze the flow of data. This may result in false positives during consistency analysis: Some infeasible paths may indicate an inconsistency for which CheckDesign outputs an error. Although we did not come across such paths while testing CheckDesign in an industrial context (Section 5.5.1), it is a limitation to be

addressed in the future.

In the industrial application, the C programming language was chosen to implement the activities. Therefore, the *implementation* of CheckSource is specific for C. But, the problems we discussed and the solution approach we proposed are general: If the implementation language of the activities is changed to another procedural programming language (e.g. Pascal [85]), then porting the solution boils down to adapting the SCFG creation functionality of CheckSource. The design decisions while developing call graph extractors are studied by Murphy et. al. [70].

In the object-oriented paradigm, the behavior of the instances of a class is typically implemented in the methods of the class. Hence, the solution presented in this chapter can be applied in an object-oriented context, too. But, certain issues need to be addressed: For example, due to dynamic binding, statically resolving a method call to a unique method definition may not be possible. Further research is necessary to address such issues if the solution is applied in an object-oriented context. One may try to use "Rapid Type Analysis" [14] for a static approximation of dynamic binding.

8.2.4 TransformSource

TransformSource does not have the functionality to bind free variables [11] in the event calls to the variables in the context of the event points in the source code. Therefore, the event calls are separately parsed and directly inserted. In case of unresolved variables, we rely on the error mechanism of the C compiler that compiles the transformed source code. The functionality to bind free variables is a part of our future work.

8.3 Conclusions

8.3.1 Problems

The development and maintenance of today's software systems is an increasingly effort-consuming and error-prone task. A major cause of the effort and errors is the lack of formal *and* human-readable documentation of software design. In practice, software design is often informally documented, or not documented at all. Therefore, the following problems occur:

1. The design cannot be properly communicated between software engineers.

2. The design cannot be automatically analyzed for finding and removing faults.
3. The conformance of an implementation to the design cannot be automatically verified.
4. Source code maintenance tasks have to be manually performed, although some of these tasks can be automated using formal documentation.

8.3.2 Solutions

In this thesis, we addressed these problems for the design and documentation of the behavior implemented in procedural programs. We introduced the following solutions:

1. A graphical language called VisuaL, which enables engineers to specify constraints on the possible sequences of function calls from a given program. VisuaL addresses the first problem stated above, and it enables us to address the remaining problems.
2. An algorithm called CheckDesign, which automatically verifies the consistency between multiple specifications written in VisuaL. CheckDesign addresses the second problem stated above.
3. An algorithm called CheckSource, which automatically verifies the consistency between a given implementation and a corresponding specification written in VisuaL. CheckSource addresses the third problem stated above.
4. An algorithm called TransformSource, which uses VisuaL specifications for automatically inserting additional source code at the well-defined locations in existing source code. TransformSource addresses the fourth problem stated above.

8.3.3 Results

We conducted three controlled experiments:

1. The first experiment was conducted with 27 M.Sc. computer science students for evaluating CheckSource. The results of this experiment indicates that CheckSource is beneficial during some of the typical maintenance tasks: 60% effort reduction, and prevention of one error per 250 lines of source code. These results are statistically significant at the level 0,05.
2. The second experiment was conducted with 21 M.Sc. computer science students for evaluating the combination of CheckSource and TransformSource. The results of this experiment indicates that the combination of CheckSource and TransformSource is beneficial during some of the typical maintenance

tasks: 50% effort reduction, and prevention of one error per 100 lines of source code. These results are statistically significant at the level 0,01.

3. The third experiment was conducted with 23 professional developers from ASML, for evaluating the combination of CheckSource and TransformSource. The results of this experiment indicates that the combination of CheckSource and TransformSource is beneficial during some of the typical maintenance tasks: 75% effort reduction, and prevention of one error per 140 lines of source code. These results are statistically significant at the level 0,01.

8.3.4 Contributions

The main contribution of this thesis is the graphical language VisuaL with its key feature called *context-sensitive wildcard*. The purpose of context-sensitive wildcards is to make VisuaL specifications more evolvable (i.e. less susceptible to changes), and more concise.

In this thesis we also provide a formal underpinning for VisuaL specifications, called *Deterministic Abstract Recognizers (DARs)*. DARs define a new family of formal languages called *Open Regular Languages (ORLs)*:

- The set of regular languages [63] is a proper subset of the set of ORLs.
- There are ORLs that are not in the set of context-free languages (CFLs) [63].
- There are CFLs that are not in the set of ORLs.

Additional contributions are

- An algorithm for checking the consistency between multiple VisuaL specifications (i.e. CheckDesign),
- An algorithm for verifying the conformance of source code to the corresponding VisuaL specifications (i.e. CheckSource), and
- An algorithm for inserting additional source code at well-defined locations in given source code (i.e. TransformSource).

Finally, the controlled experiments we conducted for evaluating CheckSource and TransformSource confirm the benefits we claimed for the concepts introduced in this thesis.

Appendix A

Data Structures and Algorithms

A.1 Deterministic Abstract Transducers

We introduce a formalism called *deterministic abstract transducers* (*DAT*s) that is a variant of Moore machines [54]. The only difference of a *DAT* from a Moore machine is its ability to operate on input strings [63] from any (possibly infinite) alphabet. Hence, a *DAT* is not specific to a predefined input alphabet (i.e. it is 'abstract').

In the remainder of this section, first, we formally define *DAT*. Next, based on the definition, we precisely explain how *DAT*s consume input strings and produce output strings.

A **DAT** M is a nonuple $\langle Q, \Sigma_a, \Lambda, \delta, \theta, q_0, F, \Xi, \eta \rangle$, where

$Q = \Omega \cup \{q_t\}$ is a finite set of **states**, where Ω is the set of **user-defined states**, q_t is the **default trap state**, and $q_t \notin \Omega$.

$\Sigma_a = \Sigma_b \cup \{\#\}$ is the **abstract input alphabet**, where Σ_b is a finite set of symbols such that $\# \notin \Sigma_b$, which is called the **base input alphabet**. $\#$ is a wildcard-like symbol that will become clear in this section.

Λ is a finite set of symbols called the **output alphabet**.

$\delta : Q \times \Sigma_a \to Q$ is the **transition function**. $\forall \sigma (\sigma \in \Sigma_a \Rightarrow \delta(q_t, \sigma) = q_t)$.

$\theta : Q \to \Lambda \cup \{\lambda\}$ is the **output function**, where λ is the empty string, and $\lambda \notin \Lambda$. $\theta(q_t) = \lambda$.

$q_0 \in Q$ is the **initial state**.

$F \subseteq \Omega$ is a set of **final states**.

Ξ is a regular expression called **scope expression** that matches a non-empty set of strings, such that this set is the set of **scopes** of M. A scope of M is a function name in the C programming language.

η is the **name** of M, which is a string of English characters and natural numbers.

In the remainder of this section, we explain how M consumes input strings and produces output strings.

Let Σ be a (possibly infinite) set of symbols.

Σ^* denotes the set of strings obtained by concatenating zero or more symbols from Σ.

Let $str \in \Sigma^*$.

$|str|$ denotes the number of symbols in str.

$str[i]$ denotes the i^{th} symbol of str.

$str_1 \cdot str_2$ denotes the string obtained by concatenating strings str_1 and str_2.

$error$ denotes an **error string**, such that $error \notin \Lambda^*$.

The operation of M is defined by the total function $Operate_M : \Sigma^* \rightarrow (\Lambda^* \cup \{error\})$ that is specified in the following pseudocode [26]:

function $Operate_M(str)$
(1) $i \leftarrow 1$
(2) $q \leftarrow q_0$
(3) $output \leftarrow \theta(q)$
(4) **while** $i \leq |str|$
(5) **do if** $str[i] \in \Sigma_b$
(6) **then** $q \leftarrow \delta(q, str[i])$
(7) **else** $q \leftarrow \delta(q, \#)$
(8) $output \leftarrow output \cdot \theta(q)$
(9) $i \leftarrow i + 1$
(10) **if** $q \notin F$
(11) **then** $output \leftarrow error$
(12) **return** $output$

Note that Line 7 provides the semantics of the # symbol.

$L(M) = \{str | Operate_M(str) \neq error\}$ denotes the language accepted by M.

A.2 Syntax and Formal Semantics of Extended VisuaL

In this section, we introduce a graphical syntax for the extended version of VisuaL, and provide the formal semantics of the syntactic constructs. The original version of VisuaL is presented in Chapter 2.

A specification in extended VisuaL represents a *deterministic abstract transducer (DAT)*, as introduced in Appendix A.1. Thus, understanding Appendix A.1 is the prerequisite for understanding the formal semantics of VisuaL.

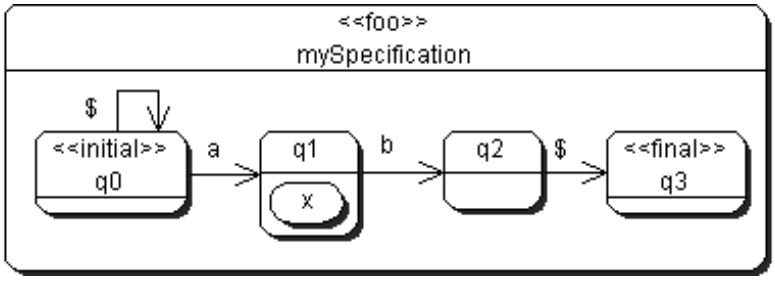

Figure A.1: An example specification in VisuaL.

Let M be a DAT. $G(M)$ denotes the VisuaL specification that represents M. In Fig. A.1, there is an example specification $G(M)$ that represents a specific DAT $M = \langle Q = \Omega \cup \{q_t\}, \Sigma_a = \Sigma_b \cup \{\#\}, \Lambda, \delta, \theta, q0, F, \Xi, \eta \rangle$ whose contents are defined by the syntax of VisuaL:

A.2.1 Rectangles

A specification in VisuaL has a rounded rectangle that is stereotyped and labelled. This rectangle is called **container rectangle** (e.g. the rectangle with label mySpecification in Fig. A.1). The label of the container rectangle defines the name of the DAT represented by the specification, and the stereotype of the container rectangle defines the scope expression of the DAT represented by the specification. Hence, $\eta = mySpecification$ and $\Xi = foo$. Multiple scopes can be specified by writing a regular expression in the label of the container rectangle, such that the regular expression matches the identifiers of all the functions that are intended to be a scope of the DAT.

Inside the container rectangle, there can be multiple, uniquely-labelled, rounded-rectangles. Such a rectangle is called **inner rectangle** (e.g. the rectangles with label q0, q1, q2, and q3, in Fig.A.1). The set of the labels of the inner rectangles define the set of user-defined states. Hence, $\Omega = \{q0, q1, q2, q3\}$.

If an inner rectangle is stereotyped as <<initial>>, or <<final>>, or <<initial-final>>, then the label of the rectangle respectively defines the initial state that is not a final state, or a final state that is not the initial state, or the initial state that is also a final state. Hence, the initial state of M is $q0$, and $F = \{q3\}$.

Inside an inner rectangle, there can be at most one[1] ellipse, which is labelled. Such an ellipse is called **output ellipse** (e.g. the ellipse with label x in Fig. A.1). The set of the labels of the output ellipses define the output alphabet. Hence, $\Lambda = \{x\}$.

Let q be a state defined by the label of an inner rectangle ν. Inside ν, if there is an output ellipse that has a label, say lbl, then the value of the output function at q is equal to lbl. Otherwise, the value of the output function at q is equal to λ. Hence, $\theta(q1) = x$, and $\theta(q0) = \theta(q2) = \theta(q3) = \lambda$.

A.2.2 Arrows

Arrows can be drawn between inner rectangles. The arrows are labelled. The labels of the outgoing arrows of an inner rectangle must differ from each other. This enforces determinism.

The set of arrow labels that are different than $ defines the base input alphabet. Hence, $\Sigma_b = \{a, b\}$.

Let ν_1 and ν_2 be two inner rectangles whose labels define states q_1 and q_2, respectively. If there is an arrow (ν_1, ν_2) with a label lbl that is different than $, then this arrow indicates that the value of the transition function at (q_1, lbl) is equal to q_2. Hence, $\delta(q0, a) = q1$, and $\delta(q1, b) = q2$. If there is an arrow (ν_1, ν_2) with the label $, then this arrow indicates that the value of the transition function at $(q_1, \#)$ is equal to q_2. Hence, $\delta(q0, \#) = q0$, and $\delta(q2, \#) = q3$.

If ν_1 does not have any outgoing arrow that is labelled with $, then the value of the transition function at $(q_1, \#)$ equals to the default trap state. Hence, $\delta(q1, \#) = q_t$, $\delta(q3, \#) = q_t$.

Let lbl $\neq$ $ be the label of an existing arrow in the specification. If ν_1 does not

[1]This limit on the number of ellipses is due to the fact that there can be at most one output symbol of a given state of DAT. If necessary, the DAT formalism can be extended for allowing multiple outputs for one state.

have any outgoing arrow that is labelled with lbl, then the value of the transition function at (ν_1, lbl) equals to the value of the transition function at $(\nu_1, \#)$. Hence, $\delta(q0, b) = \delta(q0, \#) = q0$, $\delta(q1, a) = \delta(q1, \#) = q_t$, $\delta(q2, a) = \delta(q2, b) = \delta(q2, \#) = q3$, and $\delta(q3, a) = \delta(q3, b) = \delta(q3, \#) = q_t$.

A.3 Compatibility Constraints for Processed Event

The compatibility constraints related to the mandatory event processed are

C3: In each possible sequence of function calls from `process`, there must be at least one call to `advance`.

C4: In each possible sequence of function calls from `process`, there must be at least one call to `scan`.

C5: If at least one call to `advance` and at least one call to `scan` exist in a possible sequence of function calls from `process`, then the first call to `advance` must come before the first call to `scan`.

C6: If at least one call to `advance` and at least one call to `scan` exist in a possible sequence of function calls from `process`, then the last call to `advance` must come before the last call to `scan`.

C7: In each possible sequence of function calls from `process`, exactly one call to `scan` must exist between consecutive calls to `advance`.

C8: In each possible sequence of function calls from `process`, exactly one call to `advance` must exist between consecutive calls to `scan`.

The specifications of C3 and C4 are similar to the specification of C1 (Section 6.5.3): In Fig. 6.5, if measureWafer is replaced with advance or scan, then the result is the specification of C3 or C4, respectively. C5, C6 and C7 can respectively be specified as in Figures A.2, A.3, and A.4. The specification of C8 is similar to the specification of C7: If each scan in Fig. A.4 is replaced with an advance, and each advance in Fig. A.4 is replaced with a scan, then the result is the specification of C8.

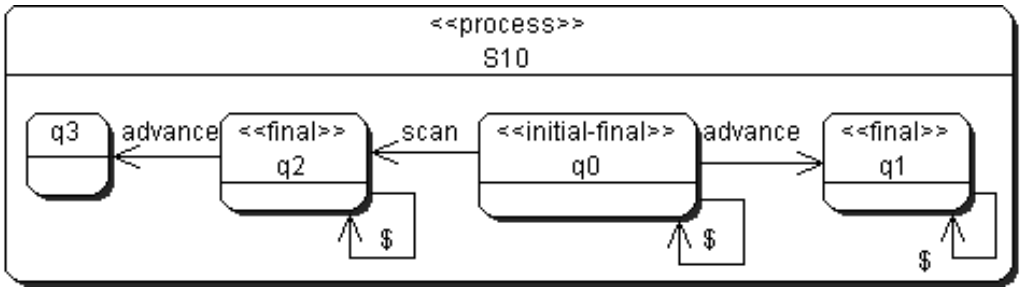

Figure A.2: The specification of the compatibility constraint C5.

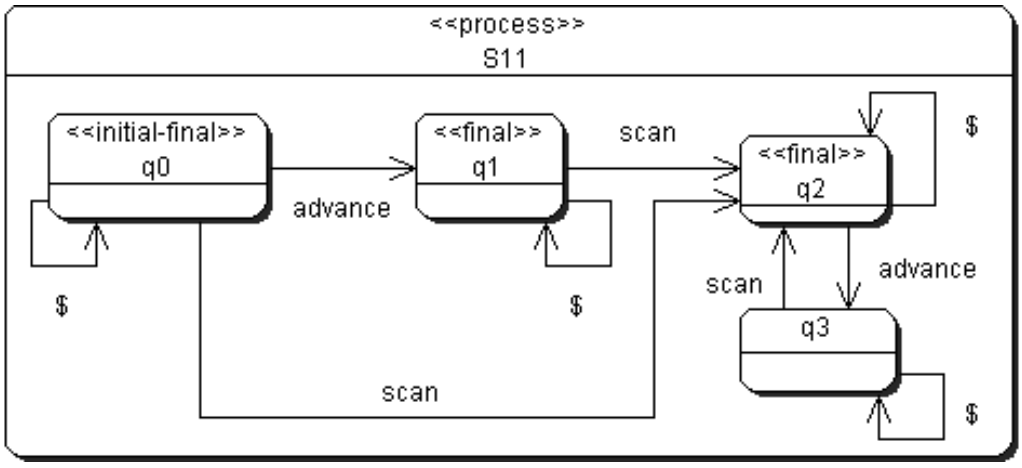

Figure A.3: The specification of the compatibility constraint C6.

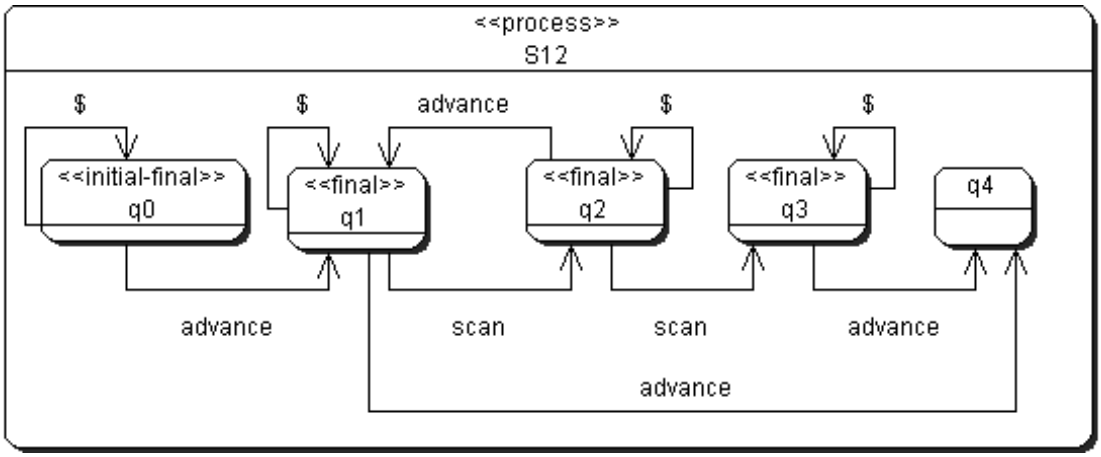

Figure A.4: The specification of the compatibility constraint C7.

A.4 Transformation

Appendix A.1 and Section 5.3.1 are the prerequisites for this appendix.

Let f be a function definition in the C programming language, and $M = \langle Q, \Sigma_a, \Lambda, \delta, \theta, q_0, F, f, \eta \rangle$ be a DAT that is mapped to $SCFG_f = \langle V = \{\nu_0\} \cup V_I \cup \{\nu_F\}, E \rangle$ as a result of the verification (Section 6.7.3). $mappedTo(q, \nu)$ denotes that state $q \in Q$ is mapped to node $\nu \in V$. $callsOf(q) =$

$\{c|\forall\nu(\nu \in V_I \wedge mappedTo(q,\nu) \Rightarrow c = nodeOfCall^{-1}(\nu))\}$ is the set of AST nodes designated by q.

If M is given as the input to TransformSource, then it iterates over each state $q \in Q$, and carries out the following operations at each iteration: (a) constructs $AST_{\theta(q)}$, and (b) inserts $AST_{\theta(q)}$ after each $c \in callsOf(q)$. After the last iteration, TransformSource emits the source code corresponding to the transformed AST_f.

Appendix B

Experimental Data

B.1 Experimental Data for CheckSource

Table B.1: The data of the tool-supported M.Sc. students

Student	#Errors	Effort in minutes
S1	0	14
S2	0	8
S3	0	13
S4	N.A.	N.A.
S5	0	25
S6	0	13
S7	0	4
S8	0	10
S9	0	33
S10	0	11
S11	0	18
S12	0	16
S13	0	5
S14	N.A.	N.A.

The data of the students S4, S14, S19, and S25 were manually modified (i.e. corrupted). Therefore, we excluded their data from our calculations.

Table B.2: The data of the manual M.Sc. students.

Student	#Errors	Effort in minutes
S15	3	28
S16	0	27
S17	2	12
S18	0	42
S19	N.A.	N.A.
S20	5	21
S21	0	41
S22	4	43
S23	0	30
S24	0	33
S25	N.A.	N.A.
S26	2	54
S27	2	45

B.2 Experimental Data for the Combination of CheckSource and TransformSource

Table B.3: The data of the tool-supported M.Sc. students

Student	#Errors	Effort in minutes
S1	0	39
S2	0	29
S3	0	26
S4	0	27
S5	N.A.	N.A.
S6	N.A.	N.A.
S7	0	24
S8	0	58
S9	0	21
S10	0	51
S11	0	17

The student S6 could not finish the task within the given time frame, which was three hours. Therefore, we omitted the related data. In addition, the logged data of the student S5 and S17 was corrupted. Therefore, we excluded this data from our calculations.

Table B.4: The data of the manual M.Sc. students.

Student	**#Errors**	**Effort in minutes**
S12	9	84
S13	2	60
S14	4	67
S15	5	49
S16	8	40
S17	N.A.	N.A.
S18	1	72
S19	8	71
S20	4	81
S21	1	53

Table B.5: The data of the tool-supported professional developers

Developer	**#Errors**	**Effort in minutes**
D1	0	16
D2	0	8
D3	0	9
D4	0	15
D5	0	10
D6	0	9
D7	0	10
D8	0	17
D9	0	13
D10	0	10
D11	0	10
D12	0	14

Table B.6: The data of the manual professional developers

Developer	**#Errors**	**Effort in minutes**
D13	8	66
D14	3	26
D15	1	47
D16	2	44
D17	1	51
D18	6	61
D19	1	66
D20	3	59
D21	5	58
D22	5	29
D23	5	40

Bibliography

[1] AspectJ TM 5 Development Kit Developer's Notebook. http://www.eclipse.org/aspectj/doc/released/adk15notebook/.

[2] CodeSurfer. http://www.grammatech.com.

[3] ANTLR. http://www.antlr.com.

[4] ASML. http://www.asml.com.

[5] Borland Together. http://www.borland.com/us/products/together.

[6] cgram. http://www.antlr.org/grammar/cgram.

[7] SPSS Version 12.0.1 for Windows. http://www.spss.com/spss/.

[8] UML. http://www.uml.org/.

[9] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[10] Jonathan Aldrich. Open modules: Modular reasoning about advice. In *European Conference on Object-Oriented Programming*, 2005.

[11] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to aspectj. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 345–364, New York, NY, USA, 2005. ACM Press.

[12] Bowen Alpern and Fred B. Schneider. Verifying temporal properties without temporal logic. *ACM Trans. Program. Lang. Syst.*, 11(1):147–167, 1989.

[13] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes, May 2002. In IEEE Symposium on Security and Privacy, Oakland, California.

[14] David Francis Bacon. *Fast and Effective Optimization of Statically Typed Object-Oriented Languages*. Ph.D. Thesis, University of California Berkeley, 1997.

[15] Thomas Ball and Sriram K. Rajamani. The slam project: debugging system software via static analysis. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–3, New York, NY, USA, 2002. ACM Press.

[16] Wolfram Bartussek and David Lorge Parnas. Using assertions about traces to write abstract specifications for software modules. In *Proceedings of the 2nd Conference of the European Cooperation on Informatics*, pages 211–236, London, UK, 1978. Springer-Verlag.

[17] Peter C. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. *ACM Trans. Comput. Syst.*, 13(1):1–31, 1995.

[18] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making components contract aware. *Computer*, 32(7):38–45, 1999.

[19] Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[20] Corrado Böhm and Giuseppe Jacopini. Flow diagrams, turing machines and languages with only two formation rules. *Commun. ACM*, 9(5):366–371, 1966.

[21] Hao Chen and David Wagner. Mops: an infrastructure for examining security properties of software. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 235–244, New York, NY, USA, 2002. ACM.

[22] Hao Chen and David A. Wagner. Mops: an infrastructure for examining security properties of software. Technical report, Berkeley, CA, USA, 2002.

[23] Edmund M. Clarke, Orna Grumberg, and Doron a. Peled. *Model Checking*. The MIT Press, 1999.

[24] Thomas D. Cook and Donald T. Campbell. *Quasi-Experimentation: Design and Analysis Issues for Field Settings*. Rand McNally Collage Publishing Company, 1979.

[25] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 439–448, New York, NY, USA, 2000. ACM Press.

[26] Thomas H. Cormen, Charles E. Lieserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, USA, 2001.

[27] Manuvir Das, Sorin Lerner, and Mark Seigle. Esp: path-sensitive program verification in polynomial time. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 57–68, New York, NY, USA, 2002. ACM Press.

[28] L. K. Dillon, G. Kutty, L. E. Moser, P. M. Melliar-Smith, and Y. S. Ramakrishna. A graphical interval logic for specifying concurrent systems. *ACM Transactions on Software Engineering and Methodology*, 3(2):131–165, 1994.

[29] Laura K. Dillon, G. Kutty, P. M. Melliar-Smith, Louise E. Moser, and Y. S. Ramakrishna. Visual specifications for temporal reasoning. *Journal of Visual Languages and Computing*, 5(1):61–81, 1994.

[30] Rèmi Douence, Pascal Fradet, and Mario Südholt. A framework for the detection and resolution of aspect interactions. In *GPCE '02: The ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, pages 173–188, London, UK, 2002. Springer-Verlag.

[31] Rèmi Douence, Pascal Fradet, and Mario Südholt. Composition, reuse and interaction analysis of stateful aspects. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 141–150, New York, NY, USA, 2004. ACM Press.

[32] Remi Douence, Pascal Fradet, and Mario Südholt. Trace-based aspects. In Filman et al. [39], pages 201–217.

[33] Rèmi Douence, Olivier Motelet, and Mario Südholt. A formal definition of crosscuts. In *REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 170–186, London, UK, 2001. Springer-Verlag.

[34] Matthew B. Dwyer, John Hatcliff, Roby Joehanes, Shawn Laubach, Corina S. Pasareanu, Robby, Hongjun Zheng, and W Visser. Tool-supported program abstraction for finite-state verification. In *International Conference on Software Engineering*, pages 177–187, 2001.

[35] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation, San Diego, CA*, October 2000.

[36] David Evans, John Guttag, James Horning, and Yang Meng Tan. Lclint: a tool for using specifications to check code. In *SIGSOFT '94: Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, pages 87–96, New York, NY, USA, 1994. ACM Press.

[37] Faul, Franz, Erdfelder, Edgar, Lang, Albert-Georg, Buchner, and Axel. Gpower 3: A flexible statistical power analysis program for the social, behavioral, and biomedical sciences. *Behavior Research Methods*, 39(2):175–191, May 2007.

[38] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., Boston, MA, USA, 1998.

[39] Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005.

[40] Phyllis G. Frankl, Richard G. Hamlet, Bev Littlewood, and Lorenzo Strigini. Evaluating testing methods by delivered reliability. *IEEE Trans. Softw. Eng.*, 24(8):586–601, 1998.

[41] Susumu Fujiwara, Gregor von Bochmann, Ferhat Khendek, Mokhtar Amalou, and Abderrazak Ghedamsi. Test selection based on finite state models. *IEEE Trans. Softw. Eng.*, 17(6):591–603, 1991.

[42] Dimitra Giannakopoulou and Jeff Magee. Fluent model checking for event-based systems. *SIGSOFT Softw. Eng. Notes*, 28(5):257–266, 2003.

[43] Noa Globerman and David Harel. Complexity results for two-way and multi-pebble automata and their logics. *Theor. Comput. Sci.*, 169(2):161–184, 1996.

[44] Simon Goldsmith, Robert O'Callahan, and Alex Aiken. Relational queries over program traces. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 385–402, New York, NY, USA, 2005. ACM Press.

[45] William G. Griswold and David Notkin. Automated Assistance for Program Restructuring. *ACM Trans. Softw. Eng. Methodol.*, 2(3):228–269, 1993.

[46] D. Harel and A. Pnueli. On the development of reactive systems. In *Logics and models of concurrent systems*, pages 477–498, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[47] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[48] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark B. Trakhtenbrot. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, 1990.

[49] David Harel and Amnon Naamad. The statemate semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, 1996.

[50] David Harel and Michal Politi. *Modeling Reactive Systems with Statecharts: The Statemate Approach*. McGraw-Hill, Inc., New York, NY, USA, 1998.

[51] John Hatcliff and Matthew B. Dwyer. Using the bandera tool set to model-check properties of concurrent java software. In *CONCUR '01: Proceedings of the 12th International Conference on Concurrency Theory*, pages 39–58, London, UK, 2001. Springer-Verlag.

[52] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[53] D. Hoffman and Richard Thomas Snodgrass. Trace specifications: Methodology and models. *IEEE Trans. Softw. Eng.*, 14(9):1243–1252, 1988.

[54] John E. Hopcroft and Jeffrey D. Ullman. *Introduction To Automata Theory, Languages, And Computation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[55] Ryszard Janicki and Emil Sekerinski. Foundations of the trace assertion method of module interface specification. *IEEE Trans. Softw. Eng.*, 27(7):577–598, 2001.

[56] Claude Jard and Thierry Jéron. Tgv: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Int. J. Softw. Tools Technol. Transf.*, 7(4):297–315, 2005.

[57] Michael Kaminski and Nissim Francez. Finite-memory automata. *Theor. Comput. Sci.*, 134(2):329–363, 1994.

[58] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[59] Barbara A. Kitchenham, Shari Lawrence Pfleeger, Lesley M. Pickard, Peter W. Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Trans. Softw. Eng.*, 28(8):721–734, 2002.

[60] Balachander Krishnamurthy and David S. Rosenblum. Yeast: A general purpose event-action system. *IEEE Trans. Softw. Eng.*, 21(10):845–857, 1995.

[61] Emmanuel Letier, Jeff Kramer, Jeff Magee, and Sebastian Uchitel. Fluent temporal logic for discrete-time event-based models. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 70–79, New York, NY, USA, 2005. ACM Press.

[62] Karl Lieberherr and David H. Lorenz. Coupling aspect-oriented and adaptive programming. In Filman et al. [39], pages 145–164.

[63] Peter Linz. *An Introduction to Formal Languages and Automata*. Jones and Bartlett Publishers, Inc., USA, 2001.

[64] David C. Luckham and James Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734, 1995.

[65] Jeff Magee and Jeff Kramer. *Concurrency: state models & Java programs*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[66] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using pql: a program query language. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 365–383, New York, NY, USA, 2005. ACM Press.

[67] Thomas J. McCabe. A complexity measure. *IEEE Trans. Softw. Eng.*, SE(2):308–320, 1976.

[68] Robin Milner. *Communicating and mobile systems: the* Π*-calculus*. Cambridge University Press, New York, NY, USA, 1999.

[69] Tova Milo, Dan Suciu, and Victor Vianu. Typechecking for XML transformers. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 11–22. ACM, 2000.

[70] Gail C. Murphy, David Notkin, William G. Griswold, and Erica S. Lan. An empirical study of static call graph extractors. *ACM Transactions on Software Engineering and Methodology*, 7(2):158–191, 1998.

[71] I. Nassi and B. Shneiderman. Flowchart techniques for structured programming. *SIGPLAN Not.*, 8(8):12–26, 1973.

[72] Frank Neven, Thomas Schwentick, and Victor Vianu. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Logic*, 5(3):403–435, 2004.

[73] Neil Ongkingco, Pavel Avgustinov, Julian Tibble, Laurie Hendren, Oege de Moor, and Ganesh Sittampalam. Adding open modules to aspectj. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 39–50, New York, NY, USA, 2006. ACM Press.

[74] D. L. Parnas. Precise description and specification of software. In *Software fundamentals: collected papers by David L. Parnas*, pages 93–106. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[75] Stacy J. Prowell and Jesse H. Poore. Foundations of sequence-based software specification. *IEEE Trans. Softw. Eng.*, 29(5):417–429, 2003.

[76] David S. Rosenblum. Specifying concurrent systems with tsl. *IEEE Softw.*, 8(3):52–61, 1991.

[77] Henk Schoemaker. *Automated Construction of Compatibility Constraints in VisuaL from Existing Source Code.* M.Sc. Thesis, University of Twente, 2007.

[78] Richard L. Schwartz, P. M. Melliar-Smith, and Friedrich H. Vogt. An interval logic for higher-level temporal reasoning. In *PODC '83: Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 173–186, New York, NY, USA, 1983. ACM.

[79] Robert W. Sebesta. *Concepts of Programming Languages.* Addison Wesley, 4 edition, 1999.

[80] Kevin Sullivan, William G. Griswold, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, Nishit Tewari, and Hridesh Rajan. Information hiding interfaces for aspect-oriented design. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 166–175, New York, NY, USA, 2005. ACM Press.

[81] Remco van Engelen and Jeroen Voeten, editors. *Ideals: evolvability of software-intensive high-tech systems.* Embedded Systems Institute, Eindhoven, The Netherlands, 2007.

[82] Yabo Wang and David Lorge Parnas. Simulating the behavior of software modules by trace rewriting. *IEEE Trans. Softw. Eng.*, 20(10):750–759, 1994.

[83] R. J. Wieringa. *Requirements engineering: frameworks for understanding.* John Wiley & Sons, Inc., New York, NY, USA, 1996.

[84] Roel Wieringa. A survey of structured and object-oriented software specification methods and techniques. *ACM Comput. Surv.*, 30(4):459–527, 1998.

[85] Niklaus Wirth. An assessment of the programming language pascal. *IEEE Transactions on Software Engineering*, 1(2):192–198, 1975.

[86] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering.* Kluwer Academic Publishers, 2000.

[87] Pierre Wolper. Temporal logic can be more expressive. *Information and Control*, 56(1/2):72–99, 1983.

Samenvatting

De ontwikkeling en het onderhoud van hedendaagse software systemen kost steeds grotere inspanningen waarbij bovendien steeds meer fouten worden gemaakt. Het ontbreken van formele en voor mensen leesbare documentatie is een belangrijke reden voor deze toenemende inspanningen en fouten. In de praktijk worden software-ontwerpen informeel, of in het geheel niet, gedocumenteerd. Als gevolg hiervan: (a) kan het ontwerp niet goed worden gecommuniceerd tussen software-ontwikkelaars, (b) kan het ontwerp niet geautomatiseerd worden geanalyseerd voor het opsporen en verwijderen van fouten, (c) kan niet geautomatiseerd worden geverifierd of een implementatie ook conform het ontwerp is, en (d) moet onderhoud aan de source code handmatig worden uitgevoerd, terwijl sommige onderhoudstaken zich lenen voor geautomatiseeerde verwerking op basis van een formele documentatie.

In dit proefschrift addresseren we bovenstaande problemen met betrekking tot het ontwerp en de documentatie van gedrag dat in procedurele talen is geprogrammeerd. We presenteren de volgende oplossingen, welke elk overeenkomen met de hierboven genoemde problemen: (a) de grafische taal VisuaL, waarmee softwareontwikkelaars beperkingen kunnen specificeren op de mogelijke sequenties van functie-aanroepen door een gegeven procedureel programma, (b) een algorithme genaamd CheckDesign, welke automatisch de consistentie tussen meerdere VisuaL specificaties kan controleren, (c) een algorithme genaamd CheckSource, welke automatisch de consistentie tussen gegeven programmacode en de bijbehorende VisuaL specificaties kan controleren, en (d) een algorithme genaamd TransformSource, welke VisuaL specificaties kan gebruiken om automatisch programmafragmenten in te voegen op welgedefinieerde locaties in bestaande programmacode.

Empirische resultaten laten zien dat het gebruik van CheckSource nuttig is tijdens een aantal van de veelvoorkomende onderhoudstaken met betrekking tot de control-flow van programma's: het kan 60% minder tijdsbesteding en 1 fout minder in elke 250 regels programmacode betekenen. Deze resultaten zijn statistisch significant op het niveau 0,05. Bovendien is de combinatie van CheckSource en TransformSource eveneens nuttig tijdens een aantal van de veelvoorkomende onderhoudstaken met

betrekking tot de control-flow: dit kan 75% minder tijdsbesteding en 1 fout minder in elke 140 regels programmacode betekenen. Deze resultaten zijn statistisch significant op het niveau 0,01.

De belangrijkste bijdrage van dit proefschrift is de grafische taal VisuaL met zijn mathematische onderbouwing op basis van zogenoemde Deterministic Abstract Recognizers (DARs). Deze laatste definiren een nieuwe familie van formele talen welke open reguliere talen worden genoemd (ofwel Open Regular Languages–ORLs). De belangrijkste eigenschap van VisuaL is de zogenoemde 'context-sensitive wildcard'. Deze maakt VisuaL specificaties beter aanpasbaar (d.w.z. minder gevoelig voor veranderingen), en meer compact.

Titles in the IPA Dissertation Series since 2002

M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects*. Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02

T. Kuipers. *Techniques for Understanding Legacy Software Systems*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03

S.P. Luttik. *Choice Quantification in Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04

R.J. Willemen. *School Timetable Construction: Algorithms and Complexity*. Faculty of Mathematics and Computer Science, TU/e. 2002-05

M.I.A. Stoelinga. *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-06

N. van Vugt. *Models of Molecular Computing*. Faculty of Mathematics and Natural Sciences, UL. 2002-07

A. Fehnker. *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

R. van Stee. *On-line Scheduling and Bin Packing*. Faculty of Mathematics and Natural Sciences, UL. 2002-09

D. Tauritz. *Adaptive Information Filtering: Concepts and Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2002-10

M.B. van der Zwaag. *Models and Logics for Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11

J.I. den Hartog. *Probabilistic Extensions of Semantical Models*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2002-12

L. Moonen. *Exploring Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-13

J.I. van Hemert. *Applying Evolutionary Computation to Constraint Satisfaction and Data Mining*. Faculty of Mathematics and Natural Sciences, UL. 2002-14

S. Andova. *Probabilistic Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2002-15

Y.S. Usenko. *Linearization in μCRL*. Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. *Random Redundant Storage for Video on Demand*. Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. *To Reuse or To Be Reused: Techniques for component composition and construction*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02

J.M.W. Visser. *Generic Traversal over Typed Source Code Representations*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-03

S.M. Bohte. *Spiking Neural Networks*. Faculty of Mathematics and Natural Sciences, UL. 2003-04

T.A.C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing*. Faculty of Mathematics and Computer Science, TU/e. 2003-05

S.V. Nedea. *Analysis and Simulations of Catalytic Reactions*. Faculty of Mathematics and Computer Science, TU/e. 2003-06

M.E.M. Lijding. *Real-time Scheduling of Tertiary Storage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-07

H.P. Benz. *Casual Multimedia Process Annotation – CoMPAs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08

D. Distefano. *On Modelchecking the Dynamics of Object-based Software: a Foundational Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-09

M.H. ter Beek. *Team Automata – A Formal Approach to the Modeling of Collaboration Between System Components*. Faculty of Mathematics and Natural Sciences, UL. 2003-10

D.J.P. Leijen. *The λ Abroad – A Functional Approach to Software Components*. Faculty of Mathematics and Computer Science, UU. 2003-11

W.P.A.J. Michiels. *Performance Ratios for the Differencing Method*. Faculty of Mathematics and Computer Science, TU/e. 2004-01

G.I. Jojgov. *Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving*. Faculty of Mathematics and Computer Science, TU/e. 2004-02

P. Frisco. *Theory of Molecular Computing – Splicing and Membrane systems*. Faculty of Mathematics and Natural Sciences, UL. 2004-03

S. Maneth. *Models of Tree Translation*. Faculty of Mathematics and Natural Sciences, UL. 2004-04

Y. Qian. *Data Synchronization and Browsing for Home Environments*. Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05

F. Bartels. *On Generalised Coinduction and Probabilistic Specification Formats*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-06

L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications*. Faculty of Science, Mathematics and Computer Science, KUN. 2004-07

E.H. Gerding. *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications*. Faculty of Technology Management, TU/e. 2004-08

N. Goga. *Control and Selection Techniques for the Automated Testing of Reactive Systems*. Faculty of Mathematics and Computer Science, TU/e. 2004-09

M. Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs*. Faculty of Science, Mathematics and Computer Science, RU. 2004-10

A. Löh. *Exploring Generic Haskell*. Faculty of Mathematics and Computer Science, UU. 2004-11

I.C.M. Flinsenberg. *Route Planning Algorithms for Car Navigation*. Faculty of Mathematics and Computer Science, TU/e. 2004-12

R.J. Bril. *Real-time Scheduling for Media Processing Using Conditionally Guaranteed Budgets*. Faculty of Mathematics and Computer Science, TU/e. 2004-13

J. Pang. *Formal Verification of Distributed Systems*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14

F. Alkemade. *Evolutionary Agent-Based Economics*. Faculty of Technology Management, TU/e. 2004-15

E.O. Dijk. *Indoor Ultrasonic Position Estimation Using a Single Base Station*. Faculty of Mathematics and Computer Science, TU/e. 2004-16

S.M. Orzan. *On Distributed Verification and Verified Distribution*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-17

M.M. Schrage. *Proxima - A Presentation-oriented Editor for Structured Documents*. Faculty of Mathematics and Computer Science, UU. 2004-18

E. Eskenazi and A. Fyukov. *Quantitative Prediction of Quality Attributes for Component-Based Software Architectures*. Faculty of Mathematics and Computer Science, TU/e. 2004-19

P.J.L. Cuijpers. *Hybrid Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2004-20

N.J.M. van den Nieuwelaar. *Supervisory Machine Control by Predictive-*

Reactive Scheduling. Faculty of Mechanical Engineering, TU/e. 2004-21

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue*. Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms*. Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications*. Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures*. Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability*. Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions*. Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments*. Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression*. Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages*. Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations*. Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics*. Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems*. Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication*. Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell*. Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model*. Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems*. Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality*. Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting*. Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs*. Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations*. Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data*. Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space*. Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices*. Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization*. Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML*. Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models*. Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications*. Faculty of Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Faculty of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition*. Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems*. Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation.* Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures.* Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. *Silent Steps in Transition Systems and Markov Chains.* Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use.* Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development Processes.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

R. Boumen. *Integration and Test plans for Complex Manufacturing Systems.* Faculty of Mechanical Engineering, TU/e. 2007-12

A.J. Wijs. *What to do Next?: Analysing and Optimising System Behaviour in Time.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

C.F.J. Lange. *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML.* Faculty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. *Component-based Configuration, Integration and Delivery.* Faculty of Natural Sciences, Mathematics, and Computer Science,UvA. 2007-15

B.S. Graaf. *Model-Driven Evolution of Software Architectures.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16

A.H.J. Mathijssen. *Logical Calculi for Reasoning with Binding.* Faculty of Mathematics and Computer Science, TU/e. 2007-17

D. Jarnikov. *QoS framework for Video Streaming in Home Networks.* Faculty of Mathematics and Computer Science, TU/e. 2007-18

M. A. Abam. *New Data Structures and Algorithms for Mobile Data.* Faculty of Mathematics and Computer Science, TU/e. 2007-19

W. Pieters. *La Volonté Machinale: Understanding the Electronic Voting Controversy.* Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automaton Proofs in PVS.* Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multi-disciplinary Systems*. Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates*. Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines*. Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras*. Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit*. Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks*. Faculty of Mathematics and Computer Science, TU/e. 2008-12

G. Gulesir. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13