Empir Software Eng
DOI 10.1007/s10664-009-9107-y

Experimental evaluation of a tool for the verification and transformation of source code in event-driven systems

Gürcan Güleşir · Klaas van den Berg · Lodewijk Bergmans · Mehmet Akşit

Editor: Giuliano Antoniol

Abstract 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, whereas the non-reactive part is typically structured according to the concepts of the application domain (e.g., the services provided by the system). In such systems, the non-reactive part of software stimulates the reactive part with *event calls*. Whenever the non-reactive part is modified (e.g., during evolution), the existing *event calls* may become invalid, new *event calls* may become necessary, and the two parts of software may become *incompatible*. Manually finding and repairing these types of defects is a time-consuming and error-prone maintenance task. In this article, we present a solution that combines source code model checking and aspect-oriented programming techniques, to reduce the time spent by developers and to automatically find defects, while performing the maintenance task mentioned above. In addition, we present controlled experiments showing that the solution can reduce the time by 75%, and enable the prevention of one defect per 140 lines of source code.

This work has been carried out as a part of the Ideals and Darwin projects, under the management of the Embedded Systems Institute, and is partially supported by the Netherlands Ministry of Economic Affairs under the Senter and Bsik programs.

G. Güleşir (✉) · K. van den Berg · L. Bergmans · M. Akşit
Department of Computer Science, University of Twente, Enschede, The Netherlands
e-mail: g.gulesir@cs.utwente.nl

K. van den Berg
e-mail: k.g.vandenberg@cs.utwente.nl

L. Bergmans
e-mail: bergmans@cs.utwente.nl

M. Akşit
e-mail: aksit@cs.utwente.nl

Keywords Event-driven systems · Source code verification · Source code transformation · Formal experiment

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 (Harel et al. 1990; Harel and Pnueli 1985; Harel 1987; Harel and Naamad 1996; Harel and Politi 1998), whereas the non-reactive part is typically structured according to the concepts of the application domain (e.g., system services and hardware components). Examples of such systems include semiconductor manufacturing machines, digital televisions, electron microscopes, and MRI[1] scanners (van Engelen and Voeten 2007).

The reactive part of software responds to occurrences of events (Harel 1987); it regulates the execution of the non-reactive part, through *control call*s (see Fig. 1). Some of the events occur due to execution of the non-reactive part. To transmit these occurrences to the reactive part, *event call*s (i.e., calls to the functions that stimulate the reactive part) 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. For instance, the system may be modified for implementing a new feature. Possible human errors during an evolution result in various defects in the system. Whenever the non-reactive part of software evolves,[2] the following types of defects may emerge at runtime:

– There is an execution of an event call, but no corresponding occurrence of the event. Thus, the execution of the event call is invalid.
– There is an occurrence of an event, but no corresponding execution of an event call. Thus, an execution of an event call is necessary.
– The reactive part waits for the execution of an event call, but the execution never happens. Thus, the two parts of software are incompatible with each other.

Manually finding and repairing these types of defects is a time-consuming and error-prone maintenance task. To reduce this time and to automatically find defects, we developed a solution that consists of (a) a graphical language called VisuaL (Güleşir 2008), which can be used for specifying event calls and *compatibility constraints*, (b) a source code analyzer for automatically verifying that the *compatibility constraints* are satisfied, and (c) a source-to-source transformer for automatically generating event calls. This solution is documented in (Güleşir 2008), and the solution is related to multiple fields of software engineering: The graphical language enables concern-shy programming (Lieberherr and Lorenz 2005); the analyzer can be seen as a source code model checker (Visser et al. 2000; Corbett et al. 2000) and the combination of the analyzer and the transformer exhibits some of the fundamental

[1] Magnetic Resonance Imaging

[2] The evolution of the reactive part of software is outside the scope of this article.

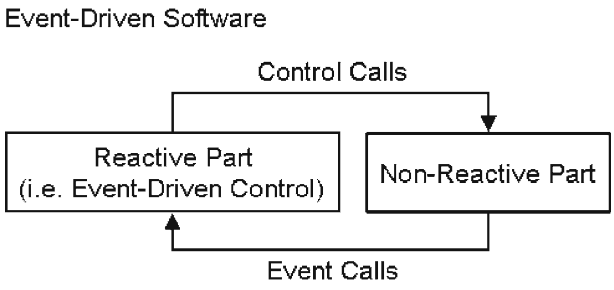


Fig. 1 A conceptual model of event-driven software systems

characteristics of a weaver (Kiczales et al. 1997) in aspect-oriented programming. In this article, we discuss the defect types enumerated above, explain the solution, and report on the controlled experiments we conducted for evaluating the solution.

In Section 2, an industrial application is presented and several terms are defined. This application is the running example that we use for explaining the industrial setting, defects types, the solution, and the experimental setup. In Section 3, we discuss the defect types listed above. In Section 4, a brief overview of our solution is provided. Sections 5–8 contain the solution. Sections 9–12 contain the experimental evaluation, and the remaining sections contain the related work, discussion, and conclusions.

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 Netherlands B.V. (ASML; http://www.asml.com) 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 (Kernighan and Ritchie 1978).

The reactive part of the wafer scanner software is structured according to the states and transitions of the system, using statecharts (Harel 1987). The non-reactive part is structured according to the activities (Harel 1987) (i.e., the services) of the system, using a procedural decomposition. Both parts are implemented in C.

In Section 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.

2.1 Simplified Wafer Scanner

A wafer scanner (Fig. 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.

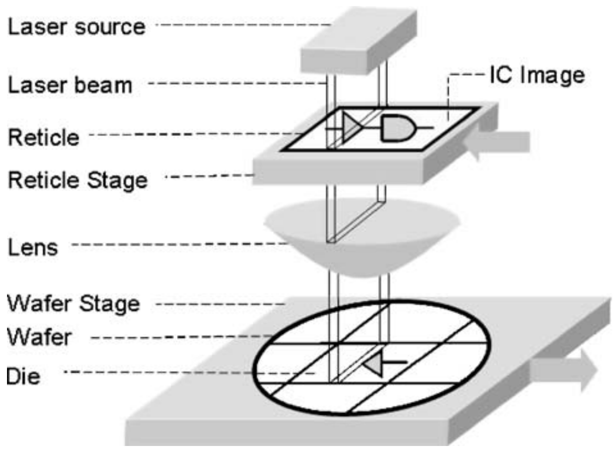


Fig. 2 A snapshot of the wafer scanner during scanning

2.1.1 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*. Figure 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. 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.

2.1.2 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 the wafer scanner 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.

2.1.3 The Requirements of the Wafer Scanner

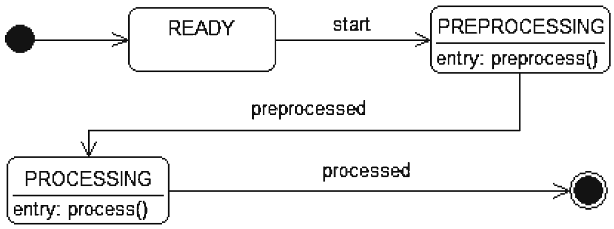
R1 The wafer scanner must start upon an external signal (e.g., an operator presses the start button).

R2 The wafer scanner must completely process the wafer (i.e., the wafer scanner must advance to a die, scan it, and repeat this for each die on the wafer).

R3 After processing, all ICs on the wafer must be faultless (i.e., before processing, the wafer scanner must clean the reticle if it is dirty and then measure the wafer).

R4 The wafer scanner must stop after the wafer is completely processed.

Fig. 3 The wafer scanner's reactive behavior



2.1.4 The Reactive Part of the Event-Driven Software

Based on the requirement R1, let us define an event called start that occurs upon an external signal, e.g., an operator presses the start button. Based on the requirement R2, let us define an event called preprocessed that occurs upon a completion of the preprocessing activity. Finally, based on the requirement R4, let us define an event called processed that occurs upon a completion of the processing activity.

Considering all of the four the requirements and the event definitions above, the reactive part of the wafer scanner software can be structured as the statechart in Fig. 3. This statechart can be interpreted as 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 (i.e., if the preprocessing activity is completed), 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 (i.e., if the processing activity is completed), then the scanner enters the final state (i.e., stops).

Based on a specific formal semantics of statecharts (e.g., (Harel and Naamad 1996)), one can manually implement the statecharts, or an implementation can be generated by a tool (e.g., (Harel et al. 1990)). Since the implementation details of the statechart in Fig. 3 are not important in this article, we assume that there is an implementation in C, which operates as explained above.

2.1.5 The Non-reactive Part of the Event-Driven Software

The preprocessing activity (Section 2.1.2), can be implemented as the C function in Listing 1.

```c
void preprocess()
{
    if(!reticleIsClean)
    {
        cleanReticle();
    }
    measureWafer();
}
```

Listing 1 An implementation of the preprocessing activity in C.

The processing activity (Section 2.1.1) can be implemented as the C function in Listing 2.

```c
void process()
{
    int i;
    for(i = 0; i < numberOfDies; i++)
    {
        advance();
        scan();
    }
}
```

Listing 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 article.

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: Currently, there is no functionality that stimulates the implementation of the statechart, with the occurrences of the preprocessed and processed events. We explain and implement this functionality, in Section 2.2.

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. 1). Harel (1987) refers to this step 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, which we also did: In Fig. 3, entry: preprocess() and entry: process() are the control calls.

The second step for connecting the statechart and the activities is creating the event calls (see Fig. 1): Stimulating the implementation of the statecharts with the events that occur due to the execution of the non-reactive part (i.e., the activities). In the remainder of this section, we present the details of the second step, which is necessary for understanding the remainder of this article.

The second step requires identifying all the points (i.e., locations) in the implementation (i.e., source code) of the activities where the events occur during execution. For example, the preprocessed event occurs when the preprocessing activity is completed (Section 2.1.4). Note that the preprocessing activity consists of cleaning the reticle if it is dirty, and then measuring the shape imperfections of the wafer (Section 2.1.2). Therefore, the preprocessed event occurs if one of the following sequences of function calls is executed: <..., `cleanReticle`, ..., `measureWafer`>, or <..., `measureWafer`>. By analyzing all possible paths through the implementation of the preprocessing activity (i.e., Listing 1), we can find out whether such sequences exist (note that both of the previously mentioned sequences exist in Listing 1). If such a sequence exists, then we can find the syntactic location in the source code where that sequence terminates (i.e., the location after '`;`' in Line 7, Listing 1). We call such a location *event point*, because an event occurs when an execution reaches that point. Now, let us define the term *event point* more precisely:

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 (Sections 2.1.2 and 2.1.1), we can identify the event points, as follows: The preprocessed event is mapped to the point located after '`;`' in Line 7, Listing 1. The processed event is mapped to the point located after '`}`' in Line 8, Listing 2. Hence, there are two event points:

1. The point located after '`;`' in Line 7, Listing 1.
2. The point located after '`}`' in Line 8, Listing 2.

After the identification of the event points, the implementation of the statechart must be stimulated with the occurrences of the events that are mapped to the event points. For this reason, typically a function that transmits the occurrence of an event to the statecharts is called at each event point. Hence, we call such functions *event functions*. Listings 3 and 4 show the implementations of the preprocessing and processing activities after inserting calls to the event functions, say `preprocessed` and `processed`, at the event points enumerated above.

```
void preprocess()
{
    if(!reticleIsClean)
    {
        cleanReticle();
    }
    measureWafer();
    preprocessed();
}
```

Listing 3 The implementation of the preprocessing activity after the connection with the statechart.

```
void process()
{
    int i;
    for(i = 0; i < numberOfDies; i++)
    {
        advance();
        scan();
    }
    processed();
}
```

Listing 4 The implementation of the processing activity after the connection with the statechart.

The definitions of the event functions are typically located in the reactive-part of event-driven software (see Fig. 1). They can be considered as the interface provided by the reactive part to the non-reactive part; *event calls* (see Fig. 1) are the calls to the event functions. The implementation details of the event functions are not important in this article.

The insertion of the event calls concludes the connection of the statechart and the activities. Consequently, each requirement in Section 2.1.3 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. We identified two typical reasons for this:

Modularity To optimize the modularity of the activities, engineers structure the activities according to the concepts of the application domain (e.g., system services and hardware components); and this structure is not always aligned with the structure of the states and transitions of the system.

Concurrency At an intermediate step during an execution of a function, say f (i.e., the implementation of an activity), 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 modularity and concurrency cases in this article, because they 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 these complicated cases, our solution addresses them, as well.

3 Defects During Activity Evolution

An event-driven software system may evolve several times during its lifetime. For instance, the system may be modified for implementing a new feature. Possible human errors 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 a time-consuming and error-prone maintenance task. In this section, we explain these defects, and show how they are manually found and repaired (i.e., without any tool support) currently at ASML. 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.[3]

Note that the ETB presented in Section 2.2 is both sound and complete.

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. Note that, even if there is only one version of activities (i.e., activities are implemented once and they do not evolve), the ETB should still be sound and complete (i.e., defect-free).

3.1.1 ETB Becomes Unsound

Imagine that we remove the call to the `measureWafer` function from Line 7, Listing 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.[4]

3.1.2 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.

3.1.3 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

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

[4]In this particular case, removing the call to `measureWafer` introduces, next to the unsoundness, an additional defect explained in Section 3.2.

can ‘wrap’ the call to `measureWafer` (Line 7, Listing 3) with an `if` block, as shown in Listing 5.

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

Listing 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.[5] To restore the soundness and the completeness, we must move[6] 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 5, one may argue that the ETB is *complete*; because, whenever the preprocessed event occurs (i.e., whenever the `measureWafer` function is called), the `preprocessed` function is executed. However, our definition of completeness is based on the syntactic structure of source code (i.e., at each event point, which is a syntactic location in the source code, there must be a related event-call), not on execution semantics. The rationale for this choice will become clear in the upcoming case.

Now, let us consider an extract-function restructuring (Griswold and Notkin 1993) that involves moving Lines 3–7 in Listing 3 to a new function `newPreprocess`, as shown in Listing 6.

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 (e.g., somewhere within the body of another function that we did not present so far), then the new occurrences of the preprocessed event (i.e., the occurrences due to the newly added call to `newPreprocess`, which is different than the call in Line 3 in Listing 6) are not transmitted the statechart. Therefore, the current location of the event call may result in defects as the system evolves. If the call to `preprocessed`

[5]Note that the addition of the if block simultaneously causes *two* independent defects: one unsoundness defect and one incompleteness defect.

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

in Line 4 is moved to the point located after `;` in Line 13, then the ETB will become sound and complete.

```
1  void preprocess()
2  {
3      newPreprocess();
4      preprocessed();
5      return;
6  }
7  void newPreprocess()
8  {
9      if(!reticleIsClean)
10     {
11         cleanReticle();
12     }
13     measureWafer();
14 }
```

Listing 6 An extract-function restructuring.

If our definition of completeness were based on the execution semantics instead of the syntactic structure, then we could not recognize the fact that the location of the event call (i.e., Line 4) may result in defects as the system evolves; because the restructuring does not change the execution semantics, but it changes the syntactic structure of the source code.

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 3, then the ETB becomes both unsound and incomplete.

Throughout Section 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 3, then the ETB is still sound and complete. In any case, manually verifying that the ETB of a large-scale software system is defect-free is a time-consuming and error-prone maintenance task.

3.2 Activity Becomes Incompatible

Let us reconsider the case in Section 3.1.1, where we remove the calls to `measureWafer` and `preprocessed` in Listing 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 (Section 2.1.3) and the statechart in Fig. 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 human errors during evolution of activities.

Based on the discussion in this section, one can imagine certain constraints on the implementation of activities, such that these 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 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 3, then the activity is still compatible with the statechart. In any case, a manual verification of the compatibility in a large-scale software system is a time-consuming and error-prone maintenance task.

3.3 Other Defects

Possible human errors during the evolution of activities may cause other defects than those we discussed so far in Section 3. For example, a human error during the evolution of the `measureWafer` function, which is the implementation of the wafer measuring activity, may lead to incorrect measurements, hence defective ICs. Addressing these kinds of defects is beyond the scope of this article.

3.4 Problem Summary

Whenever the activities (i.e., the non-reactive part of software) evolve, the compatibility between the activities and the statecharts (i.e., the reactive part of software) has to be verified. If this verification fails, then the compatibility has to be maintained (i.e., one or more compatibility defect 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 (i.e., one or more ETB defects has to be repaired). The problem is that these verification and maintenance tasks are time-consuming and error-prone, if they are manually performed.

3.5 Goals of this Research

To address the problem stated in Section 3.4, we defined the following goals to be reached in this article:

1. Save the time spent by developers for the manual verification of compatibility.
2. Prevent human errors that are possibly made during the manual verification of compatibility.
3. Ensure that the activities and statecharts are compatible.
4. Save the time spent by developers for the manual verification and maintenance of the soundness and completeness of the ETB.
5. Prevent human errors that are possibly made during the manual verification and maintenance of the soundness and completeness of the ETB.

In Section 4, we provide an overview of the solution that we developed for reaching the goals stated above.

4 A Four-Stage Solution

To reach the goals stated in Section 3.5, we developed a solution that

1. Automates the compatibility verification, so that compatibility defects are automatically found and reported to developers. Consequently, developers do not need to manually search for compatibility defects; developers only need to manually repair the reported defects, such that the automatic verification mechanism does not report any compatibility defect any more. Note that the automatic verification of the compatibility brings us to the first three goals stated in Section 3.5.
2. Eliminates the necessity for the manual verification and maintenance of the soundness and completeness of the ETB. This brings us to the last two goals stated in Section 3.5.

As depicted in Fig. 4, our solution consists of 4 stages. In Section 4.1, we provide an overview of these stages by explaining Fig. 4. In Section 4.2, we explain how this solution brings us to the goals stated in Section 3.5.

4.1 An Overview of the Stages

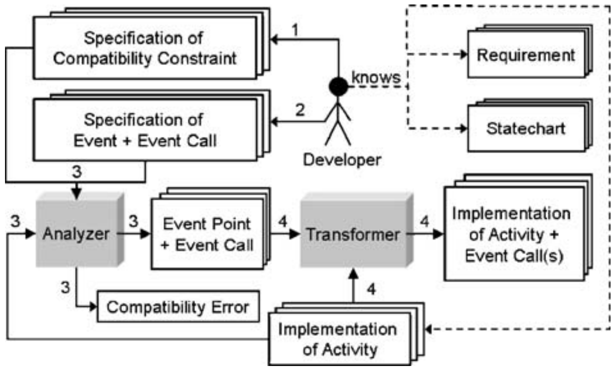
4.1.1 Stage 1: Deriving and Specifying Compatibility Constraints

At this stage, a developer (or multiple developers) who knows the requirements (e.g., Section 2.1.3), the statecharts (e.g., Fig. 3), and the event-call-free implementations of the activities (e.g., Listing 1 and 2), derives and specifies the compatibility constraints. To specify the constraints, the developer uses *VisuaL*, which is a graphical language for expressing logical and temporal properties of the behavior of algorithms. In Section 5, we explain this stage in detail.

4.1.2 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, we explain this stage in detail.

Fig. 4 A four-stage solution to automate the compatibility verification, and to eliminate the necessity for the manual verification and maintenance of the soundness and completeness of the ETB


Specification of Compatibility Constraint
1
knows
Requirement
Specification of Event + Event Call
2
Statechart
Developer
3
3
Analyzer
3
Event Point + Event Call
4
Transformer
4
Implementation of Activity + Event Call(s)
3
4
Compatibility Error
Implementation of Activity

4.1.3 Stage 3: Analysis

At this stage, a source code analyzer is provided with the compatibility constraints from Stage 1, the event specifications and the bindings from Stage 2, and the event-call-free implementations of the activities (e.g., Listings 1 and 2). All of these input artifacts are stored in an pre-specified input directory (i.e., the folder containing the input of the analyzer and transformer) that the analyzer and transformer know where to find.

If the compatibility constraints are not satisfied (e.g., the case in Section 3.2), then the analyzer outputs a compatibility error that is valuable for understanding the incompatibility and repairing it. Here, the "repairing" means "modifying the event-call-free implementation of activities (e.g., Listings 1 and 2), the specifications of compatibility constraints (e.g., Figs. 5 and 6), or statecharts (e.g., Fig. 3); such that the compatibility error disappears". To repair an incompatibility, the developer should decide which artifact(s) to modify. For example, a compatibility defect may indicate that the activities and compatibility constraints are correct, but the statecharts are no longer correct, because the activities have evolved and now the statecharts also need to evolve to fulfill the new or changing requirements for the event-driven system. In that case, the developer needs to modify statecharts to repair the incompatibility.

If the analyzer reports no error at this stage, then the analyzer outputs each event point together with a related event call. In Section 7, we explain this stage in detail.

4.1.4 Stage 4: Transformation

At this stage, a source-to-source transformer is provided with the event-call-free implementations of the activities (e.g., Listings 1 and 2) and the output of the analyzer from Stage 3. At each event point, the transformer inserts the related event call, which results in a sound and complete ETB (e.g., Listings 3 and 4). In Section 8, we explain this stage in detail.

The transformer deposits the source code resulting from this stage to an output folder that is different than the input folder mentioned in Section 4.1.3. The source code that is deposited to the output folder is the input for the C compiler, and this source code should not be modified by software engineers.

4.2 The Benefit of the Solution (i.e., How the Goals are Reached)

When software engineers implement the activities for the first time (without event calls), the four stages can be performed to verify that the compatibility constraints are satisfied, and to insert valid event calls, so that a sound and complete ETB is generated.

Whenever software engineers modify the event-call-free implementation of the activities (e.g., Listings 1 and 2) that is in the input folder mentioned in Section 4.1.3, the Stages 3 and 4 can be *automatically* repeated (a) to re-verify that the compatibility constraints are satisfied, and (b) to re-insert valid event calls; so that (a) a sound and complete ETB is re-generated, and (b) the resulting source code is deposited to the output folder mentioned in Section 4.1.3.[7] This automatic repetition of the Stages 3 and 4 brings us to the goals stated in Section 3.5.

[7]The content of the output folder is deleted each time before Stage 4 is performed.

Now, let us see our solution in more detail. In Section 5, we explain a way of working for deriving and specifying compatibility constraints. This way of working is the first stage of our solution.

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 such 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.

5.1 The 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 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. 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. 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 2.1.4) 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 the scope of this article.

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 completed, (b) the preprocessing activity is "cleaning the reticle if it is dirty, *and then* measuring the shape imperfections of the wafer" (Section 2.1.2), (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 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 article, we only present stricter-than-necessary constraints due to a limitation of the current analyzer implementation: the analyzer 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., CodeSurfer; http://www.grammatech.com) are already capable of reasoning about the nesting of function calls. The current implementation of our analyzer has proven to be useful despite this limitation (Sections 9–12).

The compatibility constraints that are related to the mandatory event processed (i.e., the other event in Fig. 3) are provided in (Güleşir 2008).

5.3 Specifying Compatibility Constraints

After the developer derives the compatibility constraints, she needs to specify them in VisuaL, so that the analyzer (see Fig. 4) can verify the implementations of the activities. In this section, we explain how to specify C1 and C2 in VisuaL, and informally discuss the syntax and semantics of VisuaL, using examples. If the readers are interested in the general syntax and formal semantics of VisuaL, then we kindly request them to see (Güleşir 2008), where an entire chapter is allocated to this topic. Broadly speaking, a VisuaL specification can be considered as a special type of Moore machine (Hopcroft and Ullman 1990).

5.3.1 Specifying C1

The specification of the compatibility constraint C1 is shown in Fig. 5. Informally, the larger rectangle represents the source code of the `preprocess` function, the arrows represent the function calls from `preprocess`, and the smaller rectangles represent the source code points located after the function calls. Below, we explain the specification in Fig. 5.

Inside the larger rectangle, there is a structure represented by the smaller rectangles and the arrows. Such a structure is called a *pattern*.

The <<from>> preprocess label means the following: Each possible sequence of function calls from the `preprocess` function must be *matched by the pattern*,[8] else there is an error.

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

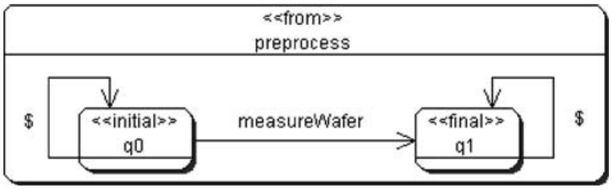


Fig. 5 The specification of the compatibility constraint C1 in VisuaL

The rectangle q0 represents the beginning of a given sequence of function calls, because it has the <<initial>> label.

The $-labelled arrow originating from q0 matches each function call from the beginning of a sequence, until a call to the `measureWafer` function is reached. This 'until condition' is due to the existence of another arrow with measureWafer label originating from the same rectangle. Hence, a $-labelled arrow is a 'context-sensitive wildcard' whose matching excludes the function calls that can be matched by the other arrows *originating from the same rectangle*. In VisuaL, no two arrows originating from the same rectangle can have the same label.

During the matching of a given sequence of function calls, when a call to the `measureWafer` function is 'reached', this call is matched by the measureWafer-labelled arrow. If the sequence terminates upon this match, then the sequence is *matched by the pattern*, because the measureWafer-labelled arrow points to a rectangle that has the <<final>> label. If there are additional function calls in the sequence, then each of them are matched by the $-labelled arrow originating from q1. As a result, the pattern matches any sequence containing at least one call to the `measureWafer` function. Hence, Fig. 5 is a specification of the compatibility constraint C1.

The readers who are familiar with LTL (Clarke et al. 1999) may have noticed that the VisuaL specification in Fig. 5 is in fact very similar to the LTL specification $eventually(measureWafer)$. In Section 13, we discuss the similarities and differences between VisuaL and LTL, in detail.

5.3.2 Specifying C2

The specification of the compatibility constraint C2 is shown in Fig. 6. q0 and q1 in this figure are different rectangles than q0 and q1 in Fig. 5, because they are in different specifications.

The <<initial-final>> label indicates that q0 has both the <<initial>> and the <<final>> labels. The lack of an arrow originating from q2 indicates that the matching stops

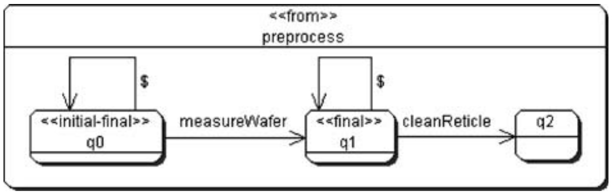


Fig. 6 The specification of the compatibility constraint C2 in VisuaL

when it 'reaches' q2. In such a case, the sequence is not matched by the pattern, because q2 does not have the <<final>> label. In all other cases, either q0 or q1 is 'reached', and the sequence is matched by the pattern, because both q0 and q1 have the <<final>> label. As a result, the pattern matches any sequence in which there is no call to the `cleanReticle` function after a possible call to the `measureWafer` function. Hence, Fig. 6 is a formal specification of C2.

Note that the $-labelled arrows in Fig. 6 are necessary for some of the desired matchings: For example, <cleanReticle, measureWafer> is one of the possible sequence of function calls from the `preprocess` function in Listing 1, and this sequence is 'legal' according to C2. If there were not a $-labelled arrow originating from q0, then the cleanReticle call (i.e., the first call in the sequence) would not be matched by any arrow, consequently the sequence would not be matched by the pattern.

The specifications of the compatibility constraints that are related to the mandatory event processed (i.e., the other event in Fig. 3) are provided in (Güleşir 2008).

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 2.1.2). With this definition in mind, the preprocessed event can be specified as the pattern shown in Fig. 7.

This pattern matches any sequence of function calls, because (a) all small rectangles have the <<final>> label, and (b) a $-labelled arrow originates from each small rectangle. 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 arrow originates from each rectangle; each measureWafer-labelled arrow points to the same rectangle (i.e., q1); and no arrow with a different label points to this rectangle. Thus, q1 is the 'hook' for binding `preprocessed();`. The binding is done by placing an ellipse containing the text preprocessed();, inside q1.

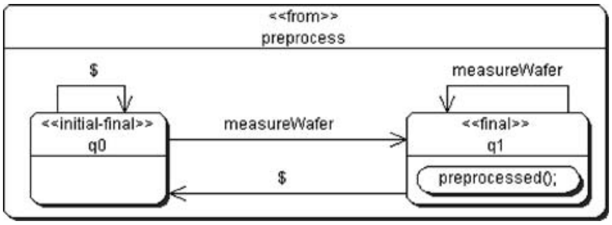


Fig. 7 The specification of the preprocessed event and the binding of the event call `preprocessed();`, in VisuaL

Although an occurrence of the preprocessed event involves a possible execution of the `cleanReticle` function, we do not need to include any cleanReticle-labelled arrow in the pattern shown in Fig. 7, because the satisfaction of the constraint C2, which is specified in Fig. 6, guarantees that any call to `cleanReticle`, if it exists, comes before any call to `measureWafer`.

Throughout Section 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 mandatory event preprocessed is currently distributed over three VisuaL specifications: Figs. 5, 6, and 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. For example, the whole information about the preprocessed event (i.e., the information captured in Figs. 5, 6, and 7) can also be captured in one concise specification, as shown in Fig. 8.

This specification is concise, because (a) it has less number of rectangles and arrows than the total number of rectangles and arrows in Figs. 5, 6, and 7; and (b) none of the rectangles and arrows in Fig. 8 is unnecessary.

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. 8). In (Güleşir 2008), we show that VisuaL specifications are closed under the boolean composition operators "not", "and", "or"; and computation-theoretic composition operators "concatenation", and "Kleene closure". These closure properties enable us to systematically compose multiple (composite) specifications to create a single composite specification. In (Güleşir 2008), we also provide an algorithm for minimizing (i.e., reducing as much as possible) the number of rectangles and arrows of a given VisuaL specification, without altering its semantics.

Using the current version of VisuaL, one cannot specify the processed event (i.e., the other event in Fig. 3), which is mapped to the point located after '}' in Line 8, Listing 2. The current version of VisuaL is not expressive enough for identifying this point. We revisit this limitation in Section 14.1.

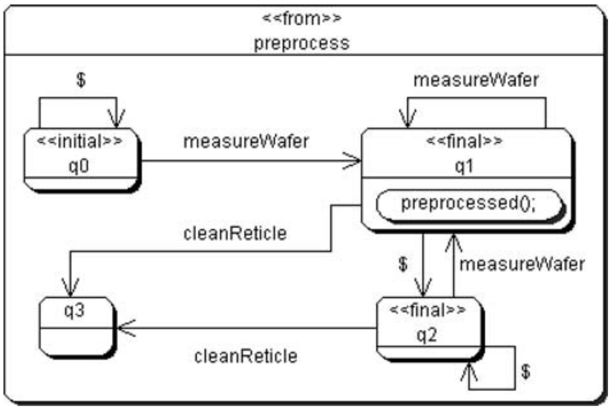


Fig. 8 The composite specification in which the compatibility constraints C1 and C2 are specified, preprocessed is specified, and `preprocessed();` is bound

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 5), and the event points are identified based on the event specifications from Stage 2 (Section 6). In this section, we provide the details of Stage 3, which consists of three steps. For the sake of conciseness in this section, we are going to use the composite specification (Fig. 8), instead of the other specifications (Figs. 5, 6, and 7).

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

If the `preprocess` function (Listing 1) is given to the analyzer as an input, then the analyzer parses the `preprocess` function, and constructs an abstract syntax tree (Aho et al. 1986) $AST_{preprocess}$ shown at the top of Fig. 9.

The rectangles labelled with *FDef* or *FCall* are the abstract nodes denoting a function definition or a function call, respectively.

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 (Fenton and Pfleeger 1998) between the function calls. Based on this assumption, only a part of the information that is in the AST is needed during the analysis. Therefore, the analyzer constructs a model (of the AST) that contains only the function calls and the 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 (Fenton and Pfleeger

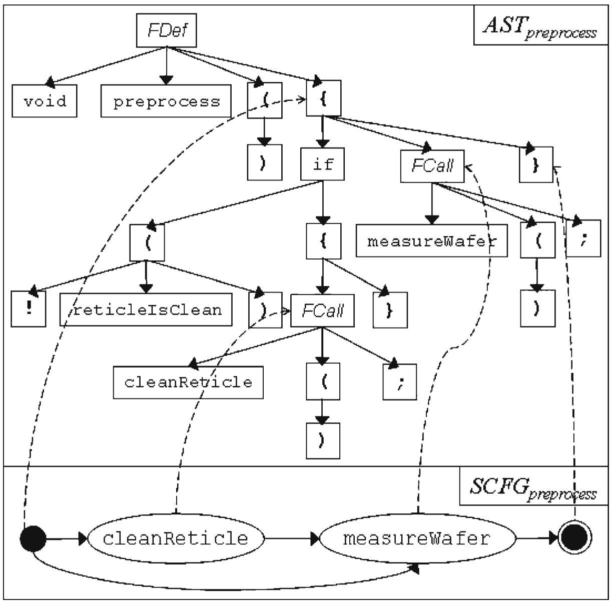


Fig. 9 The abstract syntax tree ($AST_{preprocess}$) and the simplified control flow graph ($SCFG_{preprocess}$) of the `preprocess` function in Listing 1

1998). A formal definition of a SCFG and its relation to source code are provided in (Güleşir 2008).

A SCFG decouples the analysis algorithm (explained in Section 7.3) from the implementation language of the activities, 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 activities 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.

The analyzer traverses $AST_{preprocess}$ in the depth-first (Cormen et al. 2001) manner to create $SCFG_{preprocess}$ depicted at the bottom of Fig. 9. The black dot on the left represents the *initial node*, which denotes the beginning of the `preprocess` function; The `cleanReticle`-labelled ellipse represents an *internal node* that denotes the call to `cleanReticle`; The `measureWafer`-labelled ellipse represents an internal node that denotes the call to `measureWafer`; The circled black dot represents the *final node*, which denotes the end of the `preprocess` function; 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, the analyzer maintains a one-to-one mapping from the nodes of $SCFG_{preprocess}$ to the related nodes of $AST_{preprocess}$.

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 5.2, the analyzer has to check whether all possible sequences of function calls from the `preprocess` function are matched by the pattern depicted in Fig. 8. To generate the function call sequences, the analyzer traverses $SCFG_{preprocess}$ in a depth-first manner, such that if the analyzer detects a cycle in the SCFG, the analyzer backtracks. 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 5.2. The analysis of these sequences reveals that each sequence 'ends in' q1 (see Fig. 8). Since this rectangle 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 the analyzer 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. 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 rectangles (i.e., q2 and q3) are not mapped to any node of $SCFG_{preprocess}$. This partial[9] mapping from the set of the rectangles 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. 4, this output is the input for the transformation (i.e., Stage 4), which is explained in Section 8.

Statecharts are proposed for expressing the event-driven and continuous (non-terminating) behavior of reactive systems (Harel et al. 1990; Harel and Pnueli 1985;

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

Harel 1987; Harel and Naamad 1996; Harel and Politi 1998). 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. Given this fact, the verification algorithm can be explained as follows: Let *ptrn* denote a pattern that represents a compatibility constraint *cns*. *ptrn* can be translated to a Moore machine 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 $S_f \subseteq S_{ptrn}$. We compute this by finding the intersection of the set of sequences accepted by Moore machines. A mathematical explanation of the analysis algorithm and its asymptotic time complexity (polynomial) are provided in (Güleşir 2008).

8 Stage 4: Transformation

At Stage 4, the transformer 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 7.3.

First, the transformer is provided with the partial mapping created during the analysis (Section 7.3). Second, the transformer selects `q1` (see Fig. 8), because `q1` contains the event call to be inserted (i.e., `preprocessed();`). Third, the transformer parses `preprocessed();` and creates $AST_{preprocessed();}$ shown in Fig. 10. Fourth, the transformer inserts $AST_{preprocessed();}$ as a sibling next to the upper *FCall* node in Fig. 9, because during the analysis (Section 7.3), `q1` was mapped to the `measureWafer`-labelled ellipse in Fig. 9, and this ellipse is mapped to the upper *FCall* node (see the dotted arrow between the `measureWafer`-labelled ellipse and the upper *FCall* node). Finally, the transformer traverses the modified $AST_{preprocess}$ to output the source code shown in Listing 3. A mathematical explanation of the transformation algorithm is provided in (Güleşir 2008).

Whenever the event-call-free implementation of the activities (e.g., Listings 1 and 2) is modified, the Stages 3 and 4 can be *automatically* repeated to re-verify that the compatibility constraints are satisfied, and to re-insert valid event calls, so that a sound and complete ETB is re-generated. Consequently, the goals stated in Section 3.5 can be reached.

IMPLEMENTATION We implemented both the analyzer and the transformer in Java. We used an open source parser generator (ANTLR; http://www.antlr.org) together with an open source grammar (CGRAM; http://www.antlr.org/grammar/cgram)

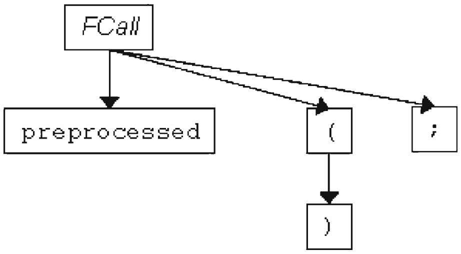


Fig. 10 The abstract syntax tree $AST_{preprocessed();}$ of the `preprocessed();` event call

to generate a parser for C. Thus, we used ANTLR as the platform on which we build our analyzer and transformer. Alternatively, TXL (Cordy 2006) could also be used for this purpose.

We implemented a plug-in for (BORLAND TOGETHER; http://www.borland.com/us/products/together), so that the VisuaL specifications can be drawn in the activity diagram editor of Borland Together, and they can be recognized by the analyzer as input. The source code of the analyzer and the transformer is available upon request.

We tested the analyzer using an Intel(R) Pentium(R) M 1,700 MHz processor with 1 GB of RAM. With an industrial specification in VisuaL, which has 11 rectangles and 23 arrows, the analyzer can process industrial functions containing 280, 127, and 83 lines of code, in 70, 40, and 32 ms, respectively. The cyclomatic complexity number (McCabe 1976) of these functions is 51, 27, and 20, respectively.

9 Experiment Definition and Planning

To see whether the analyzer and transformer can actually save time and prevent human errors in real-life, we conducted controlled 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. (2002) and Wohlin et al. (2000).

9.1 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 2. In the past, one of the software teams developing and maintaining such a component informed us about the excessive maintenance time they spend due to the defects explained in Section 3. Therefore, we conducted this research.

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

After we developed the solution, our purpose was to find answers to these questions: Is VisuaL expressive enough to specify events and compatibility constraints in real-life? Can a professional software developer efficiently use VisuaL? To find preliminary answers to these questions, we trained the domain expert of the component, who is a software developer with 15 years of professional experience. During the 1-h training, we used the material presented in this article. 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. This selection and identification was purely based on the existing daily work that the expert had to carry out at that time.

Using the heuristic presented in Section 5.1, the developer identified three mandatory events. Then, the developer created three composite specifications in VisuaL,

Table 1 The size and complexity of the VisuaL specifications, and the time that the domain expert spent for creating the specifications

Specifications	# Nodes	# Edges	Cyclomatic complexity	Time (min)
Specification1	11	19	10	80
Specification2	11	23	14	47
Specification3	10	20	12	28

each of which consists of one compatibility constraint, one event definition, and one event call binding.

In total, the developer worked 8 h under daily conditions: he was interrupted by colleagues, phone calls, lunch and coffee breaks, etc. Using two video cameras, we captured the developer, his screen, and his desk while he was working. The resulting footage enabled us to accurately calculate the net amount of time he spent for creating the specifications, which was 155 min.

The first specification created by the developer contains 11 nodes and 19 edges. To create this specification, and to draw it using Borland Together, the developer spent 80 min in total. In Table 1, the data for each of the three specifications is listed.

Using the data presented in Table 1, one can calculate that the developer spent on the average 160, 83, and 56 s per node or edge while creating Specification1, Specification2, and Specification3, respectively. This calculation indicates that the developer quickly gained speed in creating specifications. To be able to generalize this conclusion to other developers however, we need to repeat this study with more number of software developers.

To create the specifications, the developer had to rigorously analyze the relationship between the implementation, the detailed design, the architecture, and the requirements of the software component. This rigorous analysis enabled him to find one defect, which had to be repaired in the next release, four design anomalies that required restructuring and maintenance, and one undocumented feature. Two weeks earlier, the component in which the developer found these problems had been maintained by himself, and reviewed by two of his colleagues.

9.2 Motivation and Overview

The purpose of this experiment is (a) to test whether the analyzer and transformer can reduce the time spent by humans and prevent human errors, while humans are removing incompatibilities and repairing ETB defects in industrial software, and (b) to quantify the time reduction and error prevention.

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 9.1), 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.

9.3 Hypotheses

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

- ${H_0}^1$: The tools (i.e., the analyzer and the transformer) do not have any effect on the amount of time spent by M.Sc. students.
- ${H_0}^2$: The tools do not have any effect on the number of defects that M.Sc. students leave unrepaired (i.e., not repaired) in source code.

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 time spent by professional developers.
- ${H_0}^4$: The tools do not have any effect on the number of defects that professional developers leave unrepaired in source code.

We chose 0.01 as the significance level for rejecting the hypothesis above.

In this article, we *do not* investigate the differences between M.Sc. students and professional developers, because the experiments were executed under different conditions, as we will explain in Section 10.2.

9.4 The Variables of the Experiment

9.4.1 Factors

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

9.4.2 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 (McCabe 1976) 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 2, the size and complexity of the three functions are listed.

The domain expert selected these three functions, because he needed to maintain the compatibility and ETB of these functions, at the time of the selection. These

Table 2 The size and complexity of the C functions

Functions	# Lines of code	Cyclomatic complexity
Function1	88	20
Function2	127	27
Function3	280	51

Table 3 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

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 9.1. For a better understanding of the file, the descriptive statistics about the 55 functions can be found in Table 3.

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 1, 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.

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

We injected the following defects into each of the three functions:

– We removed a first possible function call whose removal creates an incompatibility defect (e.g., the case in Section 3.2). The partial ordering (implied by the control flow) of the function calls determined which function call is "a first possible function call". For example, if we wanted to inject such an incompatibility defect to the `preprocess` function in Listing 1, then we would remove the call to `measureWafer` from Listing 1.
– We added one control statement, such that this statement created an unsoundness and incompleteness defect (e.g., the case in Section 3.1.3). For example, if we would like to inject such an unsoundness and incompleteness defect to the `preprocess` function in Listing 3, then we would modify this function to obtain the function in Listing 5.

We injected one incompatibility defect (see above) into each function that was given to any participant. Thus, each participant was treated with 3 incompatibility defects.

The unsoundness and incompleteness defects were injected only into the functions that we gave to the participants who did not have the tool support, because only these participants had to manually verify and maintain the soundness and completeness of the ETB. Thus, there were three unsoundness and three incompleteness defects given to the participants that did not have the tool support.

The participants who had the tool support did not have to repair unsoundness and incompleteness defects, because they worked with event-call-free source code; the tools generated the sound and complete ETB each time the participants modified the source code to fix the incompatibility defect. Since there was only one event call in each original function, removing the event calls from the source code of the tool-supported participants did not make any significant difference in terms of the size and complexity of the source code that we gave to the participants.

9.4.3 Dependent Variables

There are two dependent variables in this experiment:

– Amount of *time* is a dependent variable measured in the ratio scale. We measure this variable in terms of minutes. To clearly explain what this time measure consists of, let us briefly revisit the steps of our approach:
  As visible in Fig. 4, developers create three kinds of artifacts:

  1. Developers write source code to implement activities without event calls; e.g., Listings 1 and 2.
  2. Stage 1: Developers create VisuaL diagrams that express temporal and logical constraints, which we call compatibility constraints; e.g., Figs. 5 and 6.
  3. Stage 2: Developers create VisuaL diagrams that express events and binding of the event calls to the events; e.g., Fig. 7.

  After these artifacts are created, the following steps are taken:

  1. Stage 3: The event-call-free implementations of the activities (e.g., Listings 1 and 2) are checked against the compatibility constraints (e.g., Figs. 5 and 6).
  2. Stage 4: If the source code conforms to the constraints, then the event definitions and event call bindings (e.g., Fig. 7) are used for injecting event calls (i.e., function calls) to the correct locations in the event-call-free implementations of the activities (e.g., Listings 1 and 2). The result is the source code that contains both the correct implementation of activities and correct event calls (e.g., Listings 3 and 4).

  Our time measure consists of the time that is spent for performing Stages 3 and 4, either with or without tool support. This time measure excludes the time that is spent for the implementation of the activities, Stage 1, and Stage 2.
– Number of *unrepaired defects* (i.e., not repaired defects) is a dependent variable measured in the absolute scale.

9.5 Selection of Participants

9.5.1 Selection of Students

This experiment was an integral part of the 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 1,000 lines of source code
3. More than 1,000, less than 5,000 lines of source code
4. More than 5,000, less than 10,000 lines of source code
5. More than 10,000 lines of source code

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

None of the students had any previous experience with the instruments listed in Section 9.7.

9.5.2 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 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. Twenty three 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.

Before we conducted the experiment with these developers, we asked the permission of the managers at ASML. Fortunately, we were permitted by the managers, because they were also interested in knowing more about the potential benefits of the tools.

None of the developers had any previous experience about the instruments listed in Section 9.7, except two of the developers have previously seen the C functions used in the experiment.

9.6 Experiment Design

As visible in Table 4, we designed an experiment that has one factor and two treatments. The factor and its levels are already explained in Section 9.4.1. Each 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 article, 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.

Table 4 The experiment has one factor with two levels each of which is one of the two treatments

	Factor: tool support	
	Level: exists	Level: not exists
Experiment 1	11 students	10 students
Experiment 2	12 developers	11 developers

The number of participants per treatment in each of the experiments is also shown in this table

9.7 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., the analyzer and the transformer),
- 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.

10 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.

10.1 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., analyzer and transformer). 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 11, 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 criterion was performing the tasks with least number of unrepaired defects in least amount of time, where the number of unrepaired defects 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.

10.2 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.

For practical reasons, each the participant had a time limit of 3 h for performing the tasks of the experiment.

10.3 Data Validation

As explained in Section 10.1, 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).

11 Data Analysis

By investigating the log files created during the experiment, we found out that each tool-supported participant did not leave any unrepaired defects in the source code. On the other hand, the manual participants left some unrepaired defects (see Section 3). To calculate the number of unrepaired defects, we used the following criteria: For each unsoundness situation (e.g., Section 3.1.1), we counted one defect. For each incompleteness situation (e.g., Section 3.1.2), we counted one defect. For each incompatibility situation (e.g., Section 3.2), we counted one defect.

By investigating the log files created during the experiment, we found out that some of the manual participants introduced new defects while trying to repair the defects that we originally injected. Since we counted the defects that remain in the source code, this count includes both the unrepaired defects that we originally injected, and the unrepaired defects that the participants introduced during the experiment.

The raw data of the experiment is provided in Appendix. 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 9.3.

We used SPSS Version 12.0.1 for Windows (SPSS; http://www.spss.com/spss/) for analyzing our data, and testing the hypotheses.

11.1 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 3 h. Therefore, we excluded his data, too.

11.2 Descriptive Statistics

11.2.1 The Experiment with the Students

In Table 5, the descriptive statistics of the data collected from the experiment with the students is presented.

Since each tool-supported student did not leave any unrepaired defects, the descriptive statistics of the number of unrepaired defects in the existence of tool support is omitted in Table 5.

Table 5 Thedescriptive statistics of the data collected from the experiment with the students

	Tool support			Statistic	Std. error
Time	Exists	Mean		32.44	4.661
		95% confidence	Lower bound	21.70	
		Interval for mean	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	0.717
		Kurtosis		–0.166	1.400
	Not exists	Mean		64.11	4.929
		95% confidence	Lower bound	52.75	
		Interval for mean	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		–0.268	0.717
		Kurtosis		–0.930	1.400
Unrepaired defects	Not exists	Mean		4.67	1.027
		95% confidence	Lower bound	2.30	
		Interval for mean	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		0.205	0.717
		Kurtosis		–1.568	1.400

The data consists of time measured in minutes, and the number of unrepaired defects. Since the number of unrepaired defects is constant when the tool support exists, the related statistics is omitted in this table

The mean amount of time spent by the tool-supported students is 32 min,[10] whereas the mean amount of time spent by the manual students is 64 min. Hence, we can conclude that the tools reduced the time spent by an average student approximately by 50% in this experiment.

[10]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 Table 5.

Table 6 The results of the normality tests for the data collected from the experiment with the students

	Tool support	Kolgomorov–Smirnov[a]			Shapiro–Wilk		
		Statistic	df	Sig.	Statistic	df	Sig.
Time	Exists	0.264	9	0.071	0.881	9	0.161
	Not exists	0.133	9	0.200[b]	0.965	9	0.853
Unrepaired defects	Not exists	0.194	9	0.200[b]	0.898	9	0.243

Since the number of unrepaired defects is constant when the tool support exists, the related statistics is omitted in this table

[a]Lilliefors Significance Correction

[b]This is a lower bound of the true significance

The mean number of unrepaired defects left by the tool-supported students is 0, whereas the mean number of unrepaired defects left by the manual students is 5. Since each participant worked with 500 lines of source code in total (see Table 2), we can conclude that the tools enabled the prevention of approximately one defect 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 time in the existence of tool support is 32.44 min, and the corresponding trimmed mean is 31.88 min. 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 time in the existence of tool support (1.005) indicates that the majority of the tool-supported students spent less than 32 min during the experiment. The negative skewness of the time in the lack of tool support (−0.268) indicates that the majority of the manual students spent more than 64 min 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).

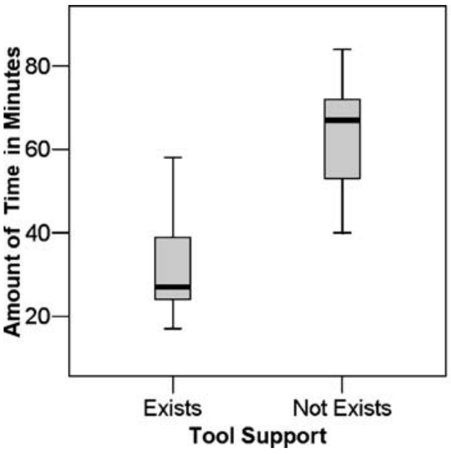
In Table 6, the results of the normality tests for the data collected from the experiment with the students are shown. It is very likely for the amount of time and the number of unrepaired defects to have a normal distribution, because the significance values (shown as "Sig." in Table 6) are greater than 0.05.

In Fig. 11a and b, the box plots of the amount of time versus tool support, and the number of unrepaired defects 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).

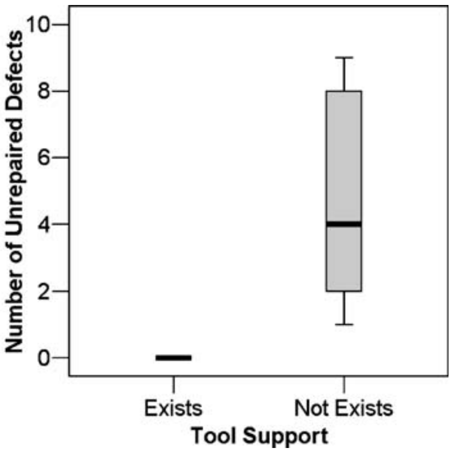
11.2.2 The Experiment with the Developers

In Table 7, the descriptive statistics of the data collected from the experiment with the developers is presented.

Since each tool-supported developer did not leave any unrepaired defects, the descriptive statistics of the number of unrepaired defects in the existence of tool support is omitted in Table 7.



(a) Time vs. tool support.



(b) Unrepaired defects vs. tool support.

Fig. 11 **a**, **b** Box plots of the results obtained from the experiment with M.Sc. students

The mean amount of time spent by the tool-supported developers is 12 min, whereas the mean amount of time spent by the manual developers is 50 min. Hence, we can conclude that the tools reduced the time spent by an average developer approximately by 75% in this experiment.

The mean number of unrepaired defects left by the tool-supported developers is 0, whereas the mean number of unrepaired defects left by the manual developers is 3.5. Hence, we can conclude that the tools enabled the prevention of approximately one defect per $500 \div 3.5 = 140$ lines of source code in this experiment.

Note that the 5% trimmed means are very close to the original means. For instance, the original mean of the amount of time in the existence of tool support is 11.75 min, and the corresponding trimmed mean is 11.67 min. 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 time in the existence of tool support (0.573) indicates that the majority of the tool-supported developers spent less than 12 min during the experiment. The negative skewness of the time in the lack of tool support (–0.542) indicates that the majority of the manual developers spent more than 50 min 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 Table 8, 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 time and the number of unrepaired defects have a normal distribution, because the significance values are greater than 0.05.

According to the Kolgomorov–Smirnov test, it is not likely that the amount of time 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 time in the

Table 7 The descriptive statistics of the data collected from the experiment with the developers

	Tool support			Statistic	Std. error
Time	Exists	Mean		11.75	0.889
		95% confidence	Lower bound	9.79	
		Interval for mean	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		0.573	0.637
		Kurtosis		–1.270	1.232
	Not exists	Mean		49.73	4.200
		95% confidence	Lower bound	40.37	
		Interval for mean	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		–0.542	0.661
		Kurtosis		–0.854	1.279
Unrepaired defects	Not exists	Mean		3.64	0.704
		95% confidence	Lower bound	2.07	
		Interval for mean	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		0.422	0.661
		Kurtosis		–0.737	1.279

The data consists of time measured in minutes, and the number of unrepaired defects. Since the number of unrepaired defects is constant when the tool support exists, the related statistics is omitted in this table

lack of tool support and the number of unrepaired defects have a normal distribution, because the significance value 0.2 is greater than 0.05.

In Fig. 12a and b, the box plots of the amount of time versus tool support, and the number of unrepaired defects 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).

Table 8 The results of the normality tests for the data collected from the experiment with the developers

	Tool support	Kolgomorov–Smirnov[a]			Shapiro–Wilk		
		Statistic	df	Sig.	Statistic	df	Sig.
Time	Exists	0.298	12	0.004	0.879	12	0.086
	Not exists	0.178	11	0.200[b]	0.924	11	0.349
Unrepaired defects	Not exists	0.175	11	0.200[b]	0.912	11	0.254

Since the number of unrepaired defects is constant when the tool support exists, the related statistics is omitted in this table

[a]Lilliefors Significance Correction

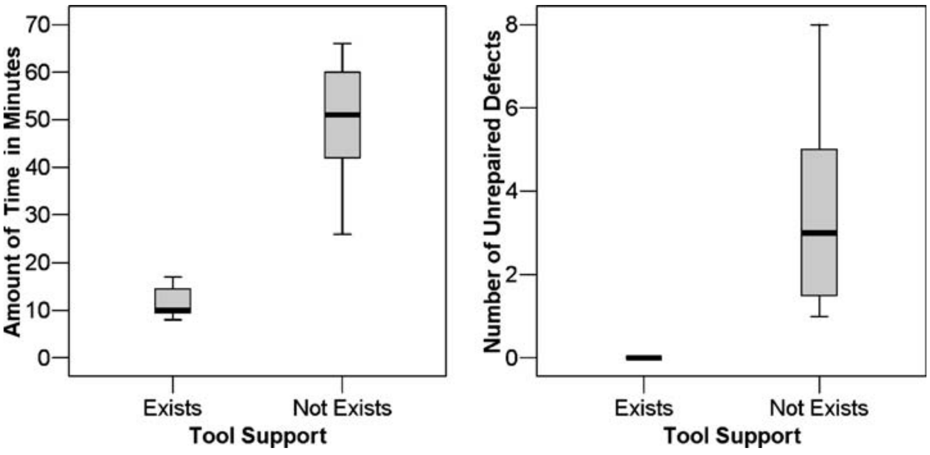
[b]This is a lower bound of the true significance

11.3 Hypothesis Testing

For testing the hypotheses stated in Section 9.3, 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 9.4.3); each participant is randomly assigned to either the tool-supported or the manual group (see Section 9.6); the observations made during the experiment are independent of each other (see Section 10.2); it is likely that the dependent variables have a normal distribution (see Section 11.2).

11.3.1 Testing $H_0{}^1$

An independent-samples *t*-test was conducted to compare the amount of time spent by the tool-supported students versus the manual students (see Table 9). 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 Table 9 is considered). There was a significant difference in the amount of time spent by the tool-supported students (Mean = 32;



(a) Time vs. tool support. (b) Unrepaired defects vs. tool support.

Fig. 12 **a**, **b** Box plots of the results obtained from the experiment with professional developers

Table 9 The results of the independent samples *t*-test for assessing the differences between the tool-supported and manual students

		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
Time	Equal variances assumed	0.059	0.811	−4.668	16	0.000	−31.667	6.783	−46.047	−17.286
	Equal variances not assumed			−4.668	15.950	0.000	−31.667	6.783	−46.051	−17.283
Unrepaired defects	Equal variances assumed	24.146	0.00	−4.542	16	0.000	−4.667	1.027	−6.845	−2.489
	Equal variances not assumed			−4.542	8.000	0.002	−4.667	1.027	−7.036	−2.297

Std. Dev. = 14) and the manual students (Mean = 64; Std. Dev = 15; $t(16) = -4.66$; $p = 0.01$). Therefore, we can reject $H_0{}^1$.

11.3.2 Testing $H_0{}^2$

An independent-samples *t*-test was conducted to compare the number of unrepaired defects left by the tool-supported students versus the manual students (see Table 9). 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 Table 9 is considered). There was a significant difference in the number of unrepaired defects left 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 $H_0{}^2$.

11.3.3 Testing $H_0{}^3$

An independent-samples *t*-test was conducted to compare the amount of time spent by the tool-supported developers versus the manual developers (see Table 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 second row in Table 10 is considered). There was a significant difference in the amount of time 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 $H_0{}^3$.

11.3.4 Testing $H_0{}^4$

An independent-samples *t*-test was conducted to compare the number of unrepaired defects left by the tool-supported developers versus the manual developers (see Table 10). 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 Table 10 is considered). There was a significant difference in the number of unrepaired defects left 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 $H_0{}^4$.

12 Validity Evaluation

In this section, we discuss the threats to the validity of the experiment. We organized these threats using the categorization proposed in (Cook and Campbell 1979); 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 (Wohlin et al. 2000).

12.1 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 (Cook and Campbell 1979).

Table 10 The results of the independent samples *t*-test for assessing the differences between the tool-supported and manual developers

		Levene's test for equality of variances		*t*-test for equality of means						
		F	Sig.	*t*	df	Sig. (2-tailed)	Mean difference	Std. error difference	95% confidence interval of the difference	
									Lower	Upper
Time	Equal variances assumed	17.141	0.000	−9.221	21	0.000	−37.977	4.119	−46.542	−29.412
	Equal variances not assumed			−8.847	10.896	0.000	−37.977	4.293	−47.437	−28.518
Unrepaired defects	Equal variances assumed	38.896	0.000	−5.405	21	0.000	−3.636	0.673	−5.035	−2.237
	Equal variances not assumed			−5.164	10.000	0.000	−3.636	0.704	−5.205	−2.067

12.1.1 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[11] to find the actual power we achieved in our statistical tests. This analysis revealed that the power we achieved is more than or equal to 0.80, for each hypothesis. Since 0.80 is the commonly accepted minimum level of power, we can conclude that the power level of our statistical tests is not a major threat to the validity of our conclusions. The power we achieved also indicates that the number of participants was sufficient for our experiments.

12.1.2 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 12.2.3. 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.

12.2 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 (Cook and Campbell 1979).

12.2.1 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 11.

[11] We used G*Power (Faul et al. 2007) for this analysis.

12.2.2 Instrumentation

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

We conducted preliminary runs of the experiment to test the quality of the instruments listed in Section 9.7. 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 11, we excluded the data of the preliminary runs.

If the VisuaL specifications that were created by the domain expert were wrong, then the results of the experiment would be skewed. This threat was partially addressed by the fact that the specifications were reviewed by the colleagues of the domain expert. Nevertheless, it was not possible for us to be absolutely sure that the specifications were correct. Consequently, possible defects in the VisuaL specifications is a threat to the internal validity.

The instructors used slides to give the tutorials. The slides of different instructors (i.e, the instructor of the tool-supported students versus the instructor of the manual students) were as similar as possible, but not exactly the same. The differences were due to the fact that one instructor had to teach the tool-supported students how to use the tools, and the other instructor had to teach the manual students how to repair unsoundness and incompleteness defects. The fact that the tutorial slides of different instructors were not exactly the same can be considered as a threat to the internal validity.

12.2.3 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 10.2, we avoided this threat in the experiment with the students, and we took effective precautions in the experiment with the developers so that the developers do not prematurely inform each other about the experiment.

As explained in Section 9.5.2, we presented the solution summarized in Section 4 to the software developers of ASML, and then some of these developers volunteered to participate in our experiment. This presentation, of course, prematurely informed the developers about the experiment. We do not think that this premature information is a serious threat to the internal validity, because (a) both the tool-supported and manual developers were equally informed, and (b) we do not compare developers with students (developers were informed more than the students).

12.3 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 three types of threats to the construct validity: confounding constructs and levels of constructs, restricted generalizability across constructs, and experimenter expectancy (Cook and Campbell 1979).

12.3.1 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 9.5.1, 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 9.5.1), 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 9.5.2, 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.

12.3.2 Restricted Generalizability Across Constructs

These kinds of threats arise if the treatment may affect the studied construct positively, but unintentionally affect the other constructs negatively.

In our experiments, the tool-supported participants were asked to remove only incompatibility defects, because the tools ensured that there is no unsoundness or incompleteness defect. Whereas, the manual participants were asked to remove both incompatibility, and unsoundness and incompleteness defects. Thus, the participants of different groups had to deal with different number of defects. This difference is an inevitable consequence of our solution, and it can be considered as a threat to the construct validity of our experiments.

12.3.3 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 first author of this article. Hence, the experimenter expected that the tools are beneficial. To eliminate this threat, we planned, conducted, and analyzed this experiment together with the second author of this article, who did not have any specific expectations from this experiment.

12.4 External Validity

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

12.4.1 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 that the tools are beneficial both for a homogenous set of students, and a homogenous set of developers.

12.4.2 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 two 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 h 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.

The domain expert specifically selected the three functions into which we injected defects, because the expert had to maintain the compatibility and ETB of these three functions, at the time of the selection. Therefore, we do not think that the representativeness of the functions is a serious threat to the validity of our experiments.

12.4.3 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 article, we did not present such a conclusion; i.e., any conclusion that is presented in this article 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 time spent by manual developers, whereas the tool-supported students spent 267% (=32/12) of the time 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 article, 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 are not suitable for comparing students and developers.

13 Related Work

The solution presented in this article (Sections 4–8) can be summarized as (a) verifying given source code, and (b) inserting additional source code at well-defined points (i.e locations) in the verified source code. Accordingly, the related work is organized under these sections:

13.1 Formal Verification and Source Code Model Checking

VisuaL is a graphical language that is suitable 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.

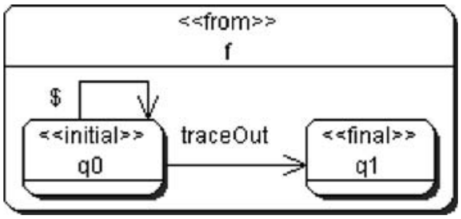


Fig. 13 A VisuaL specification indicating that *the last call* from the function `f` must be a call to `traceOut`

Since an algorithm does not execute indefinitely (otherwise it would not be an algorithm by definition (Linz 2001)), 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. 13.

C3 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 (Magee and Kramer 1999) or reactive systems (Harel and Pnueli 1985) are often infinite. Therefore, we call such systems *non-terminating systems*. To express the logical and temporal properties of non-terminating systems, several temporal logic formalisms are available: LTL (Clarke et al. 1999), CTL (Clarke et al. 1999), CTL* (Clarke et al. 1999), FLTL (Giannakopoulou and Magee 2003), (Letier et al. 2005). To be able to express the constraint C3 (see above) using one of these formalisms, one has to first translate the finite executions of the algorithm to infinite executions. For example, if *seq* =<`g`, `h`, `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`,⊙, ⊙, ⊙, ... >, where ⊙ 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* ⊙)). This LTL formula is semantically different than the VisuaL specification shown in Fig. 13. However, the formula 'mimics' the specification, provided that the finite sequences are extended to infinite sequences as explained above. FLTL, CTL, CTL* can also be used for creating formulas similar to the LTL formula stated above.

Event-based systems can be modelled as labelled transition systems (LTS) (Magee and Kramer 1999), and these models can be checked by the LTSA tool (Magee and Kramer 1999), based on the properties specified in FLTL. The nature of the models that our analyzer checks is different from the nature of the models that LTSA tool checks: Our analyzer checks a given SCFG (see Section 7.3), which is not an LTS. However, one can transform a SCFG to an LTS, by labelling each edge with the label of the target node. Upon this transformation, one can also perform the verification using the LTSA tool. Nevertheless, LTSA neither can analyze nor transform source code. The combination of our analyzer and transformer is capable of doing these.

Bandera (Corbett et al. 2000) is an integrated collection of analysis tools. Bandera can (a) automatically extract finite-state models from Java code, for the verification of LTL properties, (b) perform the automatic verification, and (c) automatically map counter-examples to Java code. From this perspective, our analyzer is similar to Bandera: Our analyzer can (a) automatically create the SCFG of a given C function

(b) automatically perform the verification, (c) enumerate the function calls along a path that is a counter example. Bandera can analyze Java source code but cannot transform it. The combination of our analyzer and transformer can both analyze and transform C code.

We perform static program analysis to verify source code. There is a substantial body of research in static analysis (Ball and Rajamani 2002; Chen and Wagner 2002; Das et al. 2002; Evans et al. 1994; Ashcraft and Engler 2002; Engler et al. 2000; CodeSurfer, http://www.grammatech.com; CodeSonar, http://www.grammatech.com; CoverityPrevent, http://www.coverity.com). This line of research does not focus on source code transformation. Therefore, static analysis and verification tools do not provide mechanisms for specifying events and binding event calls. Consequently, using existing static analysis tools, one cannot solve the problems presented in Section 3.1, at least in the way we solve them.

Interval logic, which is originally introduced by Schwartz et al. (1983), is a type of temporal logic that is specifically designed for expressing abstract requirements that a program must satisfy. Dillon et al. (1994a, b) have recognized the need for graphical languages for expressing interval logic formulas, and they introduced Graphical Interval Logic. Later, Bates (1995) introduced Event-Based Behavioral Abstraction (EBBA) for debugging heterogenous distributed systems. EBBA provides a textual way to express interval logic expressions.

Rosenblum (1991), 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 (Luckham and Vera 1995), which provides extensive support for event-based specifications of software architectures.

Hendrickson et al. (2005) propose an approach that aids in understanding, debugging, and visualizing the reactive behavior of event-driven systems. This approach can address the problems due to possible mistakes during the evolution of statecharts, which is mentioned as future work in Section 14.4.

13.2 Aspect-Oriented Programming (AOP)

In AOP terms, the ETB of a system is scattered (Kiczales et al. 1997) over and tangled (Kiczales et al. 1997) with the implementations of the activities. Hence, the ETB is a crosscutting concern (Kiczales et al. 1997). The event specifications (e.g., the pattern in Fig. 7) correspond to pointcuts (Kiczales et al. 1997), the event points (e.g., the point located after '`;`' in Line 7, Listing 1) correspond to joinpoints (Filman et al. 2005), the event calls (e.g., `preprocessed();`) correspond to advices (Kiczales et al. 1997), and the VisuaL specifications in which an event is specified and an event call is bound (e.g., Fig. 7) correspond to aspects (Kiczales et al. 1997). Thus, part of the solution presented in this article 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 9.1, we stated 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 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 (Aldrich 2005; Sullivan et al. 2005) 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 (2005) proposed to restrict joinpoint models and advising mechanisms. His proposal has been incorporated to AspectJ (http://www.eclipse.org/aspectj/) by Ongkingco et al. (2006). On the other hand, (Sullivan et al. 2005) 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 5) are interface specifications (Sullivan et al. 2005) (or contracts (Beugnard et al. 1999)) 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. (1999), 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 (Allan et al. 2005; Douence et al. 2001, 2002, 2004). 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 Chen and Wagner (2002). 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 and Fig. 8). If the existing trace-based languages (Allan et al. 2005; Douence et al. 2001, 2002, 2004) are used however, it is necessary to create at least one declare error (AJ5, http://www.eclipse.org/aspectj/) pointcut for the compatibility constraints (e.g., C1 and C2 in Section 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 (Lieberherr and Lorenz 2005). The $\$$-labelled arrows (see Figs. 5, 6, 7, and 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 (Douence et al. 2005; Goldsmith et al. 2005; Martin et al. 2005). 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 Sullivan et al. (2005).

14 Discussion

In this section, we discuss VisuaL, the tools, and the generality of our approach, in three separate sections.

14.1 VisuaL

In Section 2.2, we explained that the `processed` event is mapped to the point located after '}' in Line 8, Listing 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 the language can express context-free patterns. Using VisuaL however, one can express only regular patterns. Therefore, VisuaL is not suitable for identifying the point located after '}' in Line 8, Listing 2; hence to define the `processed` event.

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.

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 compatibility constraints and events can be defined using relatively less rectangles and arrows.

14.2 Using Our Approach in Legacy Systems

In this article, we focused on explaining how our solution could be applied from the beginning of a new software project where the event calls are always automatically inserted by our tools (i.e., never manually inserted by software engineers). Nevertheless, for existing software systems where the event calls have been manually inserted, our solution could still be beneficial: One can create VisuaL diagrams for expressing (a) the compatibility constraints (e.g., C1 and C2 in Section 5.2), and (b) the soundness and completeness properties of ETB. For example, two of the soundness and completeness properties of the simplified wafer scanner that is presented in Section 2 are as follows:

P1 In each possible sequence of function calls from `preprocess`, each call to `preprocessed` must come immediately after a call to `measureWafer`.

P2 In each possible sequence of function calls from `preprocess`, each call to `measureWafer` must be immediately be followed by a call to `preprocessed`.

The ETB of the simplified wafer scanner is (a) unsound if P1 is not satisfied, and (b) incomplete if P2 is not satisfied. Note that Listing 3 satisfies these properties.

If both the compatibility constraints and the soundness and completeness properties are specified using VisuaL, then each time software engineers modify the source code containing the event calls, our analyzer can verify both the compatibility and the soundness and completeness of the ETB of the system. In this way, our solution can be beneficial for existing event-driven systems where engineers manually maintain the ETB of the system.

Alternatively, one may also try using VisuaL, analyzer, and transformer to automatically find and remove the event calls from an existing software system. Afterwards, the same VisuaL specifications can be used for re-inserting the event calls each time the event-call-free implementations of the activities are modified, as explained throughout this article. Such a migration in a large scale legacy system is probably easier imaginable than successfully done; we do not believe that such a complete migration is practically feasible for most of the large-scale legacy systems today.

14.3 Tools

The SCFG generator of the current analyzer has limitations that can be overcome through further development. For example, the analyzer cannot recognize calls through a function pointer. To overcome this limitation, we need to incorporate pointer analysis capabilities to the current analyzer.

The current implementation of our analyzer cannot rule out infeasible paths through a function, because it does not analyze the flow of data. This may result in false positives during compatibility analysis: Some infeasible paths may indicate an incompatibility for which our analyzer outputs an error. Although we did not come across such paths while testing the solution in the limited scope defined in Section 9.1, it is a limitation to be addressed in the future. This can be done using data flow analysis. Existing commercial tools such as (CodeSonar, http://www.grammatech.com) and (CoverityPrevent, http://www.coverity.com) are already capable of ruling out infeasible paths.

The current implementation of our analyzer examines all paths through a function. If a path does not conform to a corresponding VisuaL diagram, then the analyzer reports a compatibility error containing the path. Since the analyzer examines *all* paths, it is guaranteed that there are no false negatives (i.e. if the analyzer does not report a compatibility error, then there is indeed no compatibility error).

Our transformer does not have the functionality to bind free variables (Allan et al. 2005) 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.

ASML requested us to preserve the layout of the source code upon transformation; because they were interested in clearly seeing that the injected event calls are the only modifications to the original source code. Therefore, we take care of the location of comments, indentation, white spaces etc., as follows: For each token that is encountered during parsing, we keep (a) the physical location of the token in terms of line and column numbers, and (b) the size of the token. Comments are parsed through a different token channel, so that we could exclude them from the actual AST. We keep a link between a comment and the closest non-comment token that comes before the comment in the source code. Whenever we inject an event call by modifying the AST, we accordingly update the line and column numbers of the tokens that follow the injected event call. To calculate the line and column numbers of the injected event call, we do the following: If the previous token has no link to a comment, then we use the location and size of the previous token in the AST to calculate the location of the injected event call, otherwise we use the location and size of the related comment.

14.4 Generality

In the industrial application, the C programming language was chosen to implement the activities. Therefore, the *implementation* of our solution 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 (Wirth 1975)), then porting the solution boils down to adapting the SCFG creation functionality of the analyzer.

In the object-oriented paradigm, the reactive behavior of the instances of a class is typically modelled in UML statecharts (UML, http://www.uml.org/), and the activities performed at a given state are typically implemented in the methods of the class. Hence, the solution presented in this article 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.

In this article, 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.

15 Conclusions

During the evolution of the non-reactive part of an event-driven system, several types of defects may emerge. Manually finding and repairing these types of defects is time-consuming and error-prone. To reduce the time and to prevent errors, we used a solution that integrates source code model checking and aspect-oriented programming techniques. This solution consists of a graphical language called VisuaL, a source code analyzer, and a source-to-source transformer. The graphical language is suitable for (a) specifying the compatibility constraints that ensure the correct communication of the reactive and non-reactive parts of software, and (b) defining the sequence of operations that result in an event to be responded by the reactive part of software. The analyzer can verify that the implementation of the non-reactive part satisfies the compatibility constraints; thus the analyzer can be seen as a source code model checker. The combination of the analyzer and transformer can identify the source code locations where the events occur, and then automatically insert the function calls that stimulate the reactive part upon such occurrences. Hence, this combination can be seen as a weaver in aspect-oriented programming.

We conducted two controlled experiments to understand whether the solution is beneficial in terms of time reduction and error prevention. Twenty-one M.Sc. students participated in the first experiment, and 23 professional developers participated in the second experiment. In each experiment, the participants were randomly divided into two balanced groups. Each group was treated with the same industrial specifications, source code, and defects. We instructed one group to repair the defects with the help of the analyzer and transformer, whereas the other group had to repair the defects manually (i.e., without the help of the tools).

During the experiments, we observed that the participants did not have any difficulty in understanding the specifications written in VisuaL; hence the 15-min tutorial was sufficient for them to understand and use VisuaL specifications. Based on this fact, we think that VisuaL can be a practical alternative to temporal logic formalisms such as LTL, especially for writing temporal and logical properties of algorithmic systems. Thus, VisuaL addresses the requirements specification problem stated by Hatcliff and Dwyer (Hatcliff and Dwyer 2001): "... 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."

By analyzing the data that we collected during the experiments, we observed that an average tool supported student repaired all defects in 32 min; whereas an average manual student spent 64 min to repair the defects, and left 5 not-repaired defects in 500 lines of source code. These results indicate that the tools reduce the time of an average student by 50% and prevent one error per 100 lines of source code. An average tool supported developer repaired all defects in 12 min; whereas an average manual developer spent 50 min to repair the defects, and left 3.5 not-repaired defects in 500 lines of source code. These results indicate that the tools reduce the time of an average developer by 75% and prevent one error per 140 lines of source code. The results of both experiments are statistically significant. Moreover, a post-hoc power analysis reveals that we achieved sufficient power in the experiments.

After we developed VisuaL and the tools, we proposed the managers at ASML to adopt our tools to their organization, but the managers raised many questions such as

1. Can an average developer read, understand, and create VisuaL diagrams?
2. Can an average developer efficiently and effectively use the output of the tools to repair defects in source code?
3. How much benefit do the tools bring in?
4. What is the cost of creating VisuaL diagrams?
5. How should we store and version control VisuaL diagrams?
6. What do we need to do to integrate the tools into the existing build process involving multiple development teams and multiple code streams?
7. How mature are the tools to be used in production?
8. Once the tools are adopted to the organization, how much maintenance is needed to keep the tools up and running?

9. How do VisuaL and the tools fit together with the existing tools and techniques already used in the organization?
10. What percentage of the overall software development cost can be reduced by the tools?
11. Can the benefit of VisuaL diagrams and the tools quickly exceed their cost?

Our experiments could answer only the first three questions. At least one case study is needed to be able to answer the remaining questions. Such a case study could be as follows: VisuaL and the tools could be used in a pilot project at ASML, where multiple teams collaborate to maintain software on the basis of CRPRs mentioned in Section 12.4.2. In such a pilot project, we could collect both quantitative and qualitative data to answer the remaining questions of the managers at ASML.

Acknowledgements The comments of the reviewers played the key role in improving the quality, content, and scope of this article. We cannot thank the reviewers enough for their substantial contribution. In addition, the editorial office was very responsive and helpful during the review process. We also would like to thank Cees Terlouw for his help in designing and analyzing the experiments.

Appendix: Experimental Data

The data we collected during the experiment with the students is provided in Tables 11 and 12; and the data we collected during the experiments with the professional developers is provided in Tables 13 and 14.

The student S6 (see Table 11) could not finish the task within the given time frame, which was 3 h. Therefore, we omitted the related data. In addition, the logged data of the student S5 (see Table 11) and S17 (see Table 12) was corrupted. Therefore, we excluded this data from our calculations.

Table 11 The data of the tool-supported M.Sc. students

Student	# Unrepaired defects	Time 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

Table 12 The data of the manual M.Sc. students

Student	Incompat.	Unsound.	Incomplete.	Total def.	Time (min)
S12	3	3	3	9	84
S13	1	0	1	2	60
S14	0	2	2	4	67
S15	1	2	2	5	49
S16	1	3	4	8	40
S17	N.A.	N.A.	N.A.	N.A.	N.A.
S18	0	0	1	1	72
S19	0	4	4	8	71
S20	1	2	1	4	81
S21	0	0	1	1	53

Table 13 The data of the tool-supported professional developers

Developer	# Unrepaired defects	Time 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 14 The data of the manual professional developers

Developer	Incompat.	Unsound.	Incomplete.	Total def.	Time (min)
D13	1	4	3	8	66
D14	1	1	1	3	26
D15	0	1	0	1	47
D16	0	1	1	2	44
D17	0	1	0	1	51
D18	2	2	2	6	61
D19	0	0	1	1	66
D20	0	1	2	3	59
D21	0	3	2	5	58
D22	0	2	3	5	29
D23	1	2	2	5	40

References

Aho AV, Sethi R, Ullman JD (1986) Compilers: principles, techniques, and tools. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. ISBN 0-201-10088-6

Aldrich J (2005) Open modules: modular reasoning about advice. In: European conference on object-oriented programming

Allan C, Avgustinov P, Christensen AS, Hendren L, Kuzins S, Lhotàk O, de Moor O, Sereni D, Sittampalam G, Tibble J (2005) 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. ACM, New York, NY, USA, pp 345–364. ISBN 1-59593-031-0

Ashcraft K, Engler D (2002) Using programmer-written compiler extensions to catch security holes. In: IEEE symposium on security and privacy. Oakland, California, May

Ball T, Rajamani SK (2002) 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. ACM, New York, NY, USA, pp 1–3. ISBN 1-58113-450-9

Bates PC (1995) Debugging heterogeneous distributed systems using event-based models of behavior. ACM Trans Comput Syst 13(1):1–31

Beugnard A, Jézéquel J-M, Plouzeau N, Watkins D (1999) Making components contract aware. Computer 32(7):0 38–45. ISSN 0018-9162

Chen H, Wagner DA (2002) Mops: an infrastructure for examining security properties of software. Technical report, Berkeley, CA, USA

Clarke EM, Grumberg O, Peled DA (1999) Model checking. MIT, Cambridge

Cook TD, Campbell DT (1979) Quasi-experimentation: design and analysis issues for field settings. Rand McNally Collage, Chicago

Corbett JC, Dwyer MB, Hatcliff J, Laubach S, Păsăreanu CS, Zheng H (2000) Bandera: extracting finite-state models from java source code. In: ICSE '00: proceedings of the 22nd international conference on software engineering. ACM, New York, NY, USA, pp 439–448. ISBN 1-58113-206-9

Cordy JR (2006) The txl source transformation language. Sci Comput Program 61(3):190–210. ISSN 0167-6423

Cormen TH, Lieserson CE, Rivest RL (2001) Introduction to algorithms. MIT, Cambridge, MA, USA. ISBN 0-262-03293-7

Das M, Lerner S, Seigle M (2002) Esp: path-sensitive program verification in polynomial time. In: PLDI '02: proceedings of the ACM SIGPLAN 2002 conference on programming language design and implementation. ACM, New York, NY, USA, pp 57–68. ISBN 1-58113-463-0

Dillon LK, Kutty G, Moser LE, Melliar-Smith PM, Ramakrishna YS (1994a) A graphical interval logic for specifying concurrent systems. ACM Trans Softw Eng Methodol 3(2):131–165

Dillon LK, Kutty G, Melliar-Smith PM, Moser LE, Ramakrishna YS (1994b) Visual specifications for temporal reasoning. J Vis Lang Comput 5(1):61–81

Douence R, Motelet O, Südholt M (2001) A formal definition of crosscuts. In: REFLECTION '01: proceedings of the 3rd international conference on metalevel architectures and separation of crosscutting concerns. Springer, London, UK, pp 170–186

Douence R, Fradet P, Südholt M (2002) A framework for the detection and resolution of aspect interactions. In: GPCE '02: the ACM SIGPLAN/SIGSOFT conference on generative programming and component engineering. Springer, London, UK, pp 173–188. ISBN 3-540-44284-7

Douence R, Fradet P, Südholt M (2004) Composition, reuse and interaction analysis of stateful aspects. In: AOSD '04: proceedings of the 3rd international conference on aspect-oriented software development. ACM, New York, NY, USA, pp 141–150.

Douence R, Fradet P, Südholt M (2005) Trace-based aspects. In: Filman RE, Elrad T, Clarke S, Akşit M (eds) Aspect-oriented software development. Addison-Wesley, Boston, pp 201–217. ISBN 0-321-21976-7

Engler D, Chelf B, Chou A, Hallem S (2000) Checking system rules using system-specific, programmer-written compiler extensions. In: Proceedings of the 4th symposium on operating systems design and implementation. San Diego, CA, October

Evans D, Guttag J, Horning J, Tan YM (1994) Lclint: a tool for using specifications to check code. In: SIGSOFT '94: proceedings of the 2nd ACM SIGSOFT symposium on foundations of software engineering. ACM, New York, NY, USA, pp 87–96. ISBN 0-89791-691-3

Faul F, Erdfelder E, Lang A-G, Buchner A (2007) Gpower 3: a flexible statistical power analysis program for the social, behavioral, and biomedical sciences. Behav Res Methods 39(2):175–191. ISSN 1554-351X

Fenton NE, Pfleeger SL (1998) Software metrics: a rigorous and practical approach. PWS, Boston, MA, USA. ISBN 0534954251

Filman RE, Elrad T, Clarke S, Akşit M (eds) (2005) Aspect-oriented software development. Addison-Wesley, Boston. ISBN 0-321-21976-7

Giannakopoulou D, Magee J (2003) Fluent model checking for event-based systems. SIGSOFT Softw Eng Notes 28(5):257–266. ISSN 0163-5948

Goldsmith S, O'Callahan R, Aiken A (2005) Relational queries over program traces. In: OOPSLA '05: proceedings of the 20th annual ACM SIGPLAN conference on object oriented programming, systems, languages, and applications. ACM, New York, NY, USA, pp 385–402

Griswold WG, Notkin D (1993) Automated assistance for program restructuring. ACM Trans Softw Eng Methodol 2(3):228–269. ISSN 1049-331X

Güleşir G (2008) Evolvable behavior specifications using context-sensitive wildcards. PhD thesis, University of Twente, Enschede, March. http://purl.org/utwente/58767

Harel D, Pnueli A (1985) On the development of reactive systems. In: Logics and models of concurrent systems. Springer, New York, NY, USA, pp 477–498. ISBN 0-387-15181-8

Harel D (1987) Statecharts: a visual formalism for complex systems. Sci Comput Program 8(3): 231–274

Harel D, Naamad A (1996) The statemate semantics of statecharts. ACM Trans Softw Eng Methodol 5(4):293–333. ISSN 1049-331X

Harel D, Politi M (1998) Modeling reactive systems with statecharts: the statemate approach. McGraw-Hill, New York, NY, USA. ISBN 0070262055

Harel D, Lachover H, Naamad A, Pnueli A, Politi M, Sherman R, Shtull-Trauring A, Trakhtenbrot MB (1990) STATEMATE: a working environment for the development of complex reactive systems. IEEE Trans Softw Eng 16(4):403–414

Hatcliff J, Dwyer MB (2001) 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. Springer, London, UK, pp 39–58. ISBN 3-540-42497-0

Hendrickson SA, Dashofy EM, Taylor RN (2005) An (architecture-centric) approach for tracing, organizing, and understanding events in event-based software architectures. In: IWPC '05: proceedings of the 13th international workshop on program comprehension. IEEE Computer Society, Washington, DC, USA, pp 227–236. ISBN 0-7695-2254-8

Hopcroft JE, Ullman JD (1990) Introduction to automata theory, languages, and computation. Addison-Wesley Longman, Boston, MA, USA

Kernighan BW, Ritchie DM (1978) The C programming language. Prentice-Hall, Englewood Cliffs, New Jersey

Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes C, Loingtier J-M, Irwin J (1997) Aspect-oriented programming. In: European conference on object-oriented programming, pp 220–242

Kitchenham BA, Pfleeger SL, Pickard LM, Jones PW, Hoaglin DC, El Emam K, Rosenberg J (2002) Preliminary guidelines for empirical research in software engineering. IEEE Trans Softw Eng 28(8):721–734. ISSN 0098-5589

Letier E, Kramer J, Magee J, Uchitel S (2005) 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. ACM, New York, NY, USA, pp 70–79. ISBN 1-59593-014-0

Lieberherr K, Lorenz DH (2005) Coupling aspect-oriented and adaptive programming. In: Filman RE, Elrad T, Clarke S, Akşit M (eds) Aspect-oriented software development. Addison-Wesley, Boston, pp 145–164. ISBN 0-321-21976-7

Linz P (2001) An introduction to formal languages and automata. Jones and Bartlett, USA. ISBN 0763714224

Luckham DC, Vera J (1995) An event-based architecture definition language. IEEE Trans Softw Eng 21(9):717–734

Magee J, Kramer J (1999) Concurrency: state models & Java programs. Wiley, New York, NY, USA. ISBN 0-471-98710-7

Martin M, Livshits B, Lam MS (2005) 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. ACM, New York, NY, USA, pp 365–383

McCabe TJ (1976) A complexity measure. IEEE Trans Softw Eng SE(2):308–320

Ongkingco N, Avgustinov P, Tibble J, Hendren L, de Moor O, Sittampalam G (2006) Adding open modules to aspectj. In: AOSD '06: proceedings of the 5th international conference on aspect-oriented software development. ACM, New York, NY, USA, pp 39–50. ISBN 1-59593-300-X

Rosenblum DS (1991) Specifying concurrent systems with tsl. IEEE Softw 8(3):52–61. ISSN 0740-7459

Schwartz RL, Melliar-Smith PM, Vogt FH (1983) An interval logic for higher-level temporal reasoning. In: PODC '83: proceedings of the second annual ACM symposium on principles of distributed computing. ACM, New York, NY, USA, pp 173–186. ISBN 0-89791-110-5

Sullivan K, Griswold WG, Song Y, Cai Y, Shonle M, Tewari N, Rajan H (2005) 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. ACM, New York, NY, USA, pp 166–175. ISBN 1-59593-014-0

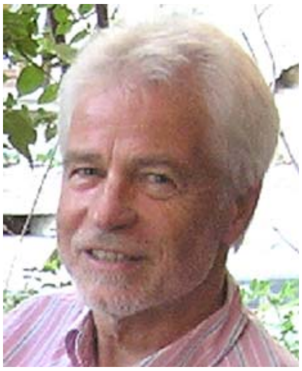
van Engelen R, Voeten J (eds) (2007) Ideals: evolvability of software-intensive high-tech systems. Embedded Systems Institute, Eindhoven, The Netherlands.

Visser W, Havelund K, Brat G, Park S (2000) Model checking programs. In: ASE '00: proceedings of the 15th IEEE international conference on Automated software engineering. IEEE Computer Society, Washington, DC, USA, p 3. ISBN 0-7695-0710-7

Wirth N (1975) An assessment of the programming language pascal. IEEE Trans Softw Eng 1(2):192–198

Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A (2000) Experimentation in software engineering. Kluwer Academic, Dordrecht

Gürcan Güleşir is a post-doc researcher at the Computer Science Department of the University of Twente in The Netherlands, where he also received his Ph.D. degree in 2008. Güleşir received his B.Sc. degree in 2001, M.Sc. degree in 2003, from the Computer Engineering Department of Bilkent University in Turkey. His research interests include software composition, evolution, verification, and measurement.



Klaas van den Berg is an Assistant Professor in the Software Engineering Group of the University of Twente (the Netherlands). He received his BSc and MSc in Electrical Engineering and his PhD degree in Computer Science from the University of Twente. He has been lecturer for some years at the University of Zambia and the University of Dar es Salaam. In the Department of Computer Science at the University of Twente, he teaches graduate and undergraduate courses on Software Engineering, Software Management and Software Architectures. His research interests include software measurement and quality metrics, software process modelling, software evolvability, traceability, aspect-oriented software development and model-driven engineering. As responsible and senior researcher, he is involved in several research projects related to these subjects. He participated in several workshops and conferences as organizer and member of the program committee.

Lodewijk Bergmans is an Assistant Professor at the Computer Science Department of the University of Twente in The Netherlands. He holds an M.Sc. and Ph.D. degree from the same institute. He has ample industrial experience in object-oriented software engineering, most notably on analysis and design, and software architecture of embedded systems. Lodewijk's long-term research goal is to achieve a deeper understanding of software composition. Most of his recent work is focusing on aspect-oriented composition.

Mehmet Akşit holds an M.Sc. degree from the Eindhoven University of Technology and a Ph.D. degree from the University of Twente. Currently, he is working as a full professor at the Department of Computer Science, University of Twente. He is the head of the Software Engineering chair and the leader of the Twente Research and Education on Software Engineering (TRESE) Group. His research interests include aspect-oriented software development, synthesis based software design, application of fuzzy logic to software design processes, and design algebra for managing large design spaces.