# On Detecting Concurrency Defects Automatically at the Design Level

Frank Padberg
and Luis M. Carril
Karlsruhe Institute of Technology KIT
Karlsruhe, Germany
frank.padberg@kit.edu

Oliver Denninger
and Martin Blersch
FZI Forschungszentrum Informatik
Karlsruhe, Germany
denninger@fzi.de

*Abstract*— **We describe an automated approach for detecting concurrency defects from design diagrams of a software, in particular, sequence diagrams. From a given sequence diagram, we automatically infer a formal, parallel specification that generalizes the communication behavior that is designed informally and incompletely in the diagram. We model-check the parallel specification against generic concurrency defect patterns. No additional specification of the software is needed. We present several case-studies to evaluate our approach. The results show that our approach is technically feasible, and effective in detecting nasty concurrency defects at the design level.**

***Automated Defect Detection; Concurrency Defect Modeling; Parallel Specification Inference; Specification Mining***

## I. Introduction

Concurrency defects are notoriously difficult to detect in practice. Whether a defect leads to a failure depends not only on the input data, but also on the scheduling. In practice, many concurrency failures crash or hang the system, see, f.e., the empirical study of 84 defects in the MySQL data base [1] and the empirical study of 105 defects in four major open-source tools [2]. In addition, the runtime conditions under which a concurrency defect manifests itself as a failure are often hard to reproduce. Therefore, finding and fixing concurrency defects can incur a high cost in software development. To support developers, a lot of research is available about race detectors that aid in detecting concurrency defects in the code; research on automated testing of concurrent code is also increasing. But – couldn't we save a substantial amount of development effort if we would detect and avoid concurrency defects *early* during development, *before* they enter the code?

In this paper, we present a novel approach for detecting potential concurrency defects automatically in design diagrams. Concurrency defects occur if objects access data concurrently, with insufficient synchronization. We focus on sequence diagrams as input, since they describe the *dynamics of collaboration* among objects in a natural way. In particular, sequence diagrams are much more adequate for designing object interaction than state diagrams, which are better used for designing the internal workings of an object. Sequence diagrams and state diagrams are most likely to be encountered as design artifacts in real-world software development, if any.

Our approach can be used at any stage during design, even with partial designs, and repeatedly as design proceeds. We do not require any code as input. No special modeling technique is asked from the developer; nor must he specify any application-specific properties for checking. In particular, we do not require diagrams as input that already would be parallel specifications, such as Petri nets, nor any logic formulas.

We automatically infer a formal, parallel specification in CSP calculus from the sequence diagrams, then model-check the formulas for any occurrence of a *concurrency defect pattern*. The defect patterns are generic, specified in CSP, and modeled after concurrency defects that were observed in real applications. We found that CSP provides compact, highly readable specifications that are still close to the design input; in particular, as opposed to automata specs. Currently, we focus on non-deadlock defects since there already is a substantial amount of good work on deadlock detection.

We present a prototype tool that automates our approach. This prototype is part of our QUALICORE research project that started in 2011. The workflow in our tool starts with a design diagram being entered by the developer and ends with a possible match of a concurrency defect pattern.

To evaluate our approach, we present four case-studies. Since there is no established benchmark with design diagrams for concurrent software, we use sequence diagrams that describe collaboration scenarios in well-known open-source software and that actually turned out to contain a concurrency bug. The case studies are *explorative* in nature. Our main goal in this initial study is not a full-grown validation, but to gain a better understanding of the potential of our approach and hints how it could be extended to become more effective.

One might argue that the cutout of the software that is visible in a design diagram is too small for detecting any concurrency defects. There is strong empirical evidence, though, that many (if not most) non-deadlock concurrency defects are fairly "local" in the sense that just 2 threads, 1 or 2 common variables, and 4 read-write accesses in a particular order are required for the defect to lead to a failure; see the empirical study [2].

One might also argue that typical design diagrams are too abstract to reveal concurrency defects. While this may be true for simple atomicity violations that are introduced when coding some variable assignments, we found evidence in our in-depth analysis of published concurrency defects that many of them are introduced already at the design or conceptual level. Concurrency defects often root in incompletely thought-out synchronization, some overlooked interleaving of operations, or a violation of implicit ordering assumptions among operations. Such gaps can be seen and warned about already when designing object interaction, before coding.

## II. Related Work

There is a large amount of work on race detectors [3] [4] [5] [6] [7] [8]. Race detectors work at the code level, hence, can be employed only late in the development process where the cost for detecting and fixing defects is high; dynamic detectors even need executable code. For software of a realistic size, their internal models get very large and their defect analysis is computationally expensive. Similar comments apply to other dynamic analysis-based approaches such as [22] [23] [24] [36] [29]. While race detectors are good at detecting low-level atomicity violations and deadlocks, they have problems detecting more intricate concurrency defects where the programmer implicitly relies on ordering assumptions among concurrent operations.

In contrast, our approach aims at detecting defects early from the *design* when code is not yet available. In addition, the formal models that we generate and analyze are smaller by orders of magnitude since we start from a more abstracted view than code and focus on the concurrent communication. Hence, our defect detection tool is very fast.

There also is a large body of work on deriving formal models for UML or OMT diagrams, the focus being on class, activity, and state diagrams. For activity diagrams, which basically are parallel specs already, Petri nets [32] [31] and CSP [9] [10] [11] are often used as the spec language. The goals are manifold: define a precise formal semantics for diagrams [33] [10] [34]; perform consistency checking and refinement checking of diagrams [30] [9]; perform model-checking of application-specific behavioral properties on diagrams [30] [11] [35]. In contrast, our modeling goal is different, our focus is on object interaction, and our natural input are sequence diagrams.

Inferring a *parallel* spec from sequence diagrams has not received much attention yet. [26] [27] derive state machines to precisely capture sequence diagram semantics. [25] transform UML sequence diagrams into coloured Petri nets for the purpose of consistency checking and model integration. [28] use algebraic semantics to capture different variants of sequence diagrams. [37] transform sequence diagrams into CSP to precisely capture their semantics; they use UML meta-models for sequence diagrams and CSP, and specify the transformation rules with QVT/XSLT.

Our approach naturally shows some overlap in technical aspects with the CSP-based modeling approaches sketched in [10] and [37]; f.e., we also model individual method calls as CSP events. Contrary to all existing formal modeling work for sequence diagrams, we do *not* aim at providing a spec that reflects a given sequence diagram as *faithfully* as possible. Rather, for the purpose of detecting potential concurrency defects, we infer a *generalized* spec from the scenario which deliberately includes potential interleavings of operations that are not shown in the design, but later might make their way into the final code. In addition, we do not require application-specific properties of the software as input for model-checking. Rather, we check our inferred CSP models against *generic* patterns that capture well-known, nasty concurrency defects.

In [12], Briand e.a. pursue a goal similar to ours: detection of certain concurrency issues at the design level. They focus on starvation and deadlocks, though, whereas we aim at detecting complicated race conditions that are not easily handled by race detectors. They start from diagrams that must be enriched with timing and concurrency information. Instead of using CSP or a similar spec language, they use a custom tuple representation. Their detection approach is much different from ours: They apply a genetic algorithm to try and cause starvation or deadlock in their models. The authors claim that their approach can be tailored to race detection by suitably choosing fitness functions in the genetic algorithm, but they do not show this in the paper. Contrary to our tool, their approach can require up to several hours of running time.

## III. Modeling and Defect Detection

Our defect detection approach has two main steps:

1. Generate a formal, parallel model from the sequence diagrams.

2. Check the formal model against generic concurrency defect patterns.

We use the CSP calculus [13] for formal parallel modeling and the FDR2 tool for model checking [14]. FDR is free for academic use.

Our models focus on the communication between objects and abstract away from the details of computations inside objects. This seems natural as we aim at concurrency issues.

The internal logic of a CSP process gets specified using sequences of events, deterministic and non-deterministic choices, recursion, and parallel composition. CSP processes communicate by participating in common events. That means, two (or more) processes share the same event and "fire" this event all at the same time, synchronously. If one of the processes cannot fire the event now because of the logic in his process formula, the event cannot occur – no communication. The other processes have to wait and are blocked if this event comes next in their program logic.

The key *technical* difficulty that we are facing is to map the usual procedure-call semantics in a standard design diagram to the synchronous event semantics of the CSP calculus. This problem is not specific for CSP, but applies to other process calculi as well. We solve this difficulty by introducing an event for each method call (and return) and share this event between the caller and callee.

The key *conceptual* difficulty that we are facing is to infer the parallel model in such a way that it captures not only the one scenario that the designer specified in the diagram, but also other, closely related scenarios that might make their way into the code when implementing the diagram. That is, our model must suitably generalize the information provided in the design cutout. Roughly, what we do is model the concurrent objects individually, according to the operations shown in their lifeline, then parallel-compose the individual models. This way, many more interleavings are feasible in the model than the one particular interleaving that is specified in the sequence diagram. We are aware that this approach can result in models that may over-generalize, but we expect that these models are useful for detecting potential concurrency issues.

In the following subsections, we describe our technique in detail using the parallel compression program Bzip2 [15] as a running example.

## A. *Design Input*

In this paper, we focus on sequence diagrams as input. It is more natural and easier for a developer to draw a familiar sequence diagram for a concurrent scenario – marking objects as active (thread) and using an additional type of arrow for asynchronous calls – than to learn and use a parallel specification language such as Petri nets. Synchronization can be specified easily using the "critical"-construct in UML 2.4, or by explicit calls to lock- and unlock-methods.

For example, Fig. 1 shows the core processing scenario in Bzip2 as a sequence diagram. The diagram specifies the order of method calls as intended by the developer.

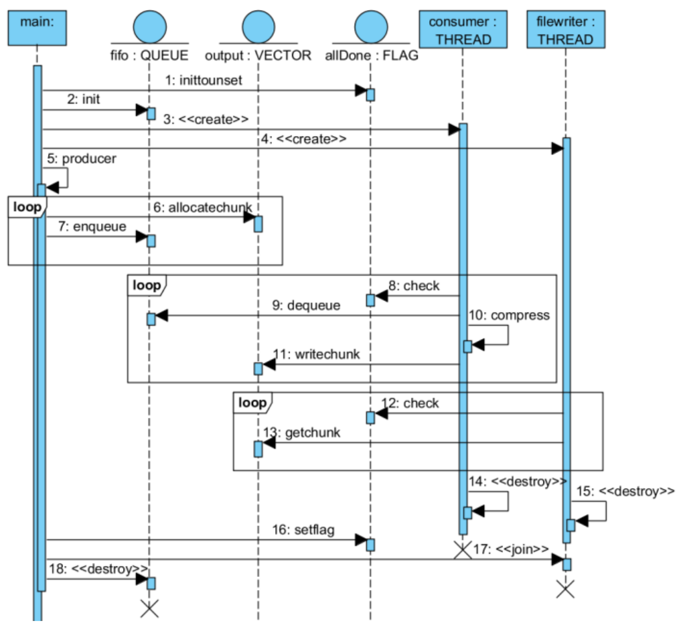

Figure 1. Sequence diagram for processing in Bzip2

In Bzip2, the main thread puts data to be compressed in a fifo and reserves slots for the compressed data in an output data structure. Parallel to that, a "consumer" thread fetches data blocks from the fifo, compresses the data, and puts them into the reserved output slots. Yet another thread, "filewriter," concurrently reads compressed data from the slots and writes them to the file system.

For synchronization, the three threads use a flag ("alldone") that gets set by the main thread to signal that processing should stop; the flag is checked regularly by the other threads. After setting the flag, the main thread waits for the filewriter thread to end.

The Bzip2 sequence diagram is what one would typically encounter during design. It visualizes the core classes, their collaboration, and the main data flow in the software. It might get augmented by class diagrams and state diagrams that specify the behavior of individual objects in more detail.

## B. *Model Generation*

We show step-by-step how to derive a formal CSP model from the sequence diagram for Bzip2. The model captures the behavior specified in the diagram, but also carefully *generalizes* it in order to account for other scenarios of operation that might be implicit in the design. Interleavings are included that could lead to concurrency defects but have not been taken into consideration by the designer when drawing the diagram.

Each object in the diagram (including the data objects) gives rise to one CSP process, named in uppercase letters by CSP convention. Since the objects run concurrently, the application modeled as a parallel composition (operator ǁ) of these processes:

```
BZIP = MAIN ǁ FIFO ǁ OUTPUT ǁ ALLDONE ǁ CONSUMER ǁ FILEWRITER
```

Listing 1. Top-level CSP spec for Bzip2

Each process gets modeled separately now, basically chasing the sequence of method calls to and from the object on its lifeline in the diagram.

Method calls between objects are modeled as CSP events. By CSP convention, event names begin with lower case letters. In our modeling approach, events are *prefixed* to provide more semantics of what the event means and where it originates in the diagram. F.e., when the main thread calls the enqueue-method in the fifo object, we model this as a CSP event mc_MAIN_FIFO.enqueue, naming the caller and callee – that is, the "channel" over which the call is being issued. In addition to "mc" for method calls, we use the prefixes "mr" for method returns, "oc" for object creation, "od" for object destruction, and "oj" for joins. A self-call of a method within an object is annotated slightly differently, because they are less relevant for the communication between objects. F.e., the internal use of the compress-method in the consumer thread is annotated as the event compress_CONSUMER.

For the main thread we get the following CSP process formula:

```
MAIN = (mc_MAIN_ALLDONE.inittounset → mc_MAIN_FIFO.init →
oc_CONSUMER → oc_FILEWRITER → producer_MAIN → MX)
MX = (mc_MAIN_OUTPUT.allocatechunk → mc_MAIN_FIFO.enqueue → MX ⊓
mc_MAIN_OUTPUT.allocatechunk → mc_MAIN_ALLDONE.setflag →
oj_MAIN_FILEWRITER → mc_MAIN_FIFO.destroy → od_FIFO → SKIP)
```

Listing 2. CSP spec for the main thread (Bzip2)

The main thread initializes and creates some objects, then enters its producer-method and internal loop. From the sequence diagram, we cannot tell when the main thread will leave this loop. Hence, we model this as a non-deterministic choice (operator ⊓) between two possible behaviors: One branch allocates output slots, enqueues data, and loops; the other branch leaves the loop, sets the flag, and joins the filewriter thread. SKIP is a standard CSP process that doesn't do anything anymore.

If more information were provided in the sequence diagram about the loop exit condition we could include this in the model. Otherwise, we found that modeling loops using non-deterministic choice is adequate for our purposes.

The other active objects in the diagram (consumer thread, filewriter thread) get modeled analogously:

```
CONSUMER = (oc_CONSUMER → CX)
CX = (mc_CONSUMER_ALLDONE.check → mc_CONSUMER_FIFO.dequeue
→ compress_CONSUMER → mc_CONSUMER_OUTPUT.writechunk → CX ⊓
mc_CONSUMER_ALLDONE.check → od_CONSUMER → SKIP)
```

Listing 3. CSP spec for the consumer thread (Bzip2)

```
FILEWRITER = (oc_FILEWRITER → FX)
FX = (mc_FILEWRITER_ALLDONE.check →
mc_FILEWRITER_OUTPUT.getchunk → FX ⊓
mc_FILEWRITER_ALLDONE.check → od_FILEWRITER →
oj_MAIN_FILEWRITER → SKIP)
```

Listing 4. CSP spec for the filewriter thread (Bzip2)

The other objects in the Bzip2 diagram are modeled differently because they are "passive." They gain control only if one of their methods is called from outside. F.e., the fifo object offers methods that are called by the main thread and consumer thread. The fifo object has no control over when its methods are being called, nor by whom. In particular, its methods may be called *concurrently* from different threads, with all implications and defect risks. Hence, we model this passive object as a *parallel composition* of two sub-processes, each of which covers the communication with one of the active threads:

```
FIFO = FX1 || FX2
FX1 = (mc_MAIN_FIFO.init → FX1 | mc_MAIN_FIFO.enqueue → FX1 |
mc_MAIN_FIFO.destroy → od_FIFO → SKIP)
FX2 = (mc_CONSUMER_FIFO.dequeue → FX2)
```

Listing 5. CSP spec for the fifo object (Bzip2)

In the diagram, the main thread can call fifo's init-, enqueue-, or destroy-method. From the perspective of the FIFO process, these are alternatives (deterministic choice); if the methods should be called in a particular order, this must be specified as part of the caller's logic. Note that this comment also applies to the loop in the diagram, which is part of the main thread, not of the fifo object. Modeling "passive" objects this way complies with the usual behavior of procedural code.

The remaining objects in the diagram are modeled accordingly:

```
OUTPUT = OX1 || OX2 || OX3
OX1 = (mc_MAIN_OUTPUT.allocatechunk → OX1)
OX2 = (mc_FILEWRITER_OUTPUT.getchunk → OX2)
OX3 = (mc_CONSUMER_OUTPUT.writechunk → OX3)
```

Listing 6. CSP spec for the output object (Bzip2)

```
ALLDONE = AX1 || AX2 || AX3
AX1 = (mc_MAIN_ALLDONE.inittounset → AX1 | mc_MAIN_ALLDONE.setflag
→ AX1)
AX2 = (mc_FILEWRITER_ALLDONE.check → AX2)
AX3 = (mc_CONSUMER_ALLDONE.check → AX3)
```

Listing 7. CSP spec for the flag (Bzip2)

## C. *Generator Tool*

We developed a prototype tool – the model generator – that derives CSP formulas *automatically* from a sequence diagram. The tool generates all processes, prefixes, events, etc. from the sequence diagram; no manual intervention by the developer is needed. It is ongoing work to extend the tool to handle state diagrams and combine information from several diagrams. As input for the model generator, we use an intermediate XML representation of the diagram.

## D. *Generic Defect Patterns*

Concurrency defects patterns are modeled after defects that occurred in real applications, see [16] [2] [17]. We analyzed these known defects in depth to extract their core ingredients.

The patterns are short CSP formulas whose events are composed exactly the same way as the events in the formulas of an application; the only difference is that the patterns use *generic* process and method names. F.e., a (faulty) access to an object O that has been destroyed can be formally modeled as an object deletion event (od_O) followed by a method call (mc_P_O), a read access (rd_P_O), or a write access (wr_P_O) from some other process P to the destroyed object.

Table 1 lists our current catalogue of defect patterns. The access-after-deletion pattern is adapted from [16], asynchronous-wait from [2], multiple-initialization from [18], and mutable-lock from [19]. The atomicity violation patterns total to 14 (we show only an excerpt in the table) and are adapted from [17]. They differ only in the number and order of read-write accesses. Their applicability is discussed in detail in [17] [20] [21]. We derived yet another set of defect patterns from them in which the violation occurs indirectly when accessing some data via method calls of an encapsulating object. This helps detect certain defects, see, f.e., the MySQL case study below.

Table 1. Concurrency defect patterns

| | |
|---|---|
| Access after deletion | |
| AD1<br>AD2<br>AD3 | od_SD → mc_P1_SD.method → STOP<br>od_SD → rd_P1_SD → STOP<br>od_SD → wr_P1_SD → STOP |
| Asynchronous wait with flag | |
| AW1<br><br>AW2 | mc_MAIN_ASYNC.method → wr_ASYNC_FLAG →<br>wr_MAIN_FLAG → rd_MAIN_FLAG → STOP<br>oa_ASYNC → wr_ASYNC_FLAG → wr_MAIN_FLAG →<br>rd_MAIN_FLAG → STOP |
| Multiple initialization | |
| MI | rd_T1_REF → rd_T2_REF → oc_REF → oc_REF → STOP |
| Mutable lock | |
| ML | lock_REF → oc_REF → lock_REF → STOP |
| Atomicity violation with one variable | |
| AV1<br>AV2<br>etc. | rd_P1_SD → wr_P2_SD → wr_P1_SD → STOP<br>rd_P1_SD → wr_P2_SD → rd_P1_SD → STOP |
| Atomicity violation with two variables | |
| AV6<br><br>AV7<br><br>etc. | wr_P1_SD1 → wr_P2_SD1 → wr_P2_SD2 → wr_P1_SD2 →<br>STOP<br>wr_P1_SD1 → wr_P2_SD2 → wr_P2_SD1 → wr_P1_SD2 →<br>STOP |
| Indirect atomicity violation | |
| IAV2<br><br><br>etc. | mc_P1_DELEGATE.method1 → rd_DELEGATE_SD →<br>mc_P2_DELEGATE.method2 → wr_DELEGATE_SD →<br>mc_P1_DELEGATE.method3 → rd_DELEGATE_SD → STOP |

Note that the defect patterns need to be specified only once; they are stored in our tool.

## E. Defect Detection

A concurrency defect in an application's CSP model indicates a potential concurrency defect in the design. To check whether one of the defect patterns matches, we use the CSP model checker FDR2 [14]. Basically, we hand FDR the CSP model and have it analyze whether the *traces* of the pattern are observable as part of the traces of the application. A trace is a sequence of CSP events; which event sequences can occur (or not) is specified by the formulas in a model.

Before we can run any checks, we must *instantiate* the defect patterns for the given application. That is, we systematically substitute concrete object and method names from the application for the generic ones in the patterns. Recall that the events in the patterns are composed the same way as the events in the process formulas of an application. F.e., the access-after-deletion pattern "od_SD → mc_P1_SD.method → STOP" contains the generic names SD for a "data" object, P1 for a "process" object, and method for an operation that is called on the data object by the process object. A particular instance of this pattern using Bzip2 names then is "od_FIFO → mc_MAIN_FIFO.enqueue → STOP".

Not all combinations of concrete names make sense for a given pattern; many can safely be left out. F.e., for an error-prone overlapping read-write access, we need at least two active processes. There are only a handful of such sanity checks for each defect pattern, but they help cut down the number of pattern instances that must be considered.

Each concrete pattern instance then is fed into the model checker, together with the application model and some additional code that programs the model checker to compare the two trace spaces.

Let's look at an example. The design of Bzip2 contains a concurrency bug which is not immediate from the sequence diagram: After setting the flag, the main process waits for the filewriter thread, assuming that the consumer process will exit before the filewriter thread. If the consumer thread checks the flag right before it is set by main and checked by filewriter, the threads may exit in the reverse order, and consumer will try to read from the queue that has just been freed by the main thread.

This bug was actually contained in an earlier version of Bzip2 [16]. At the model level, this defect is visible as a particular trace of CSP events:

mc_CONSUMER_ALLDONE.check,
mc_MAIN_ALLDONE.setflag, mc_MAIN_FIFO.destroy, od_FIFO,
mc_CONSUMER_FIFO.dequeue.

The defective trace is detected by the model checker when checking the Bzip2 model against the catalogue of defect patterns. *The model actually allows for detecting the concurrency defect* hidden in the design diagram.

## F. Detector Tool

We have fully automated the pattern instantiation and model-checking process in a prototype tool – the defect detector – that hooks up our model generator tool with FDR. Since our CSP models tend to be slim, model-checking them with FDR is very fast. Typically, we get a complete check of all pattern instances within a few seconds. We provide exact measurements for the case study examples in the next section.

At the model level, there are *no false positives*: If one of the defect patterns matches, there definitely is a concurrency defect *in the model*. Yet, there is not always a corresponding defect in the application design. Since our CSP model might over-generalize the design information to some extent, it may happen that the defective trace actually is excluded in the design. Our experiments show that the number of such false positives tends to be small, though.

## G. Frontends

The model generator takes an XML representation of the design diagram as input. The XML schema is tailored to our tool. Developers will create their diagrams using a common UML modeling tool, though, such as Visual Paradigm, StarUML, or Eclipse MDT. We provide frontends for these UML tools that convert the XML/XMI exports from the modeling tools into our own format, so that input is automated.

Similarly, we are working on backends that feed any detected defects back into the UML modeling tool. Currently, our tool reports any defect matches as the corresponding, concrete CSP trace. Since the event names in our models carry information about the objects and methods involved, we in principle can trace this output back to the scenario in the initial design diagram.

## IV. Evaluation

Evaluating a design-based defect detection approach faces the difficulty that there is no established benchmark suite containing design diagrams for real software applications. Currently, we are working with our industry partners in the QUALICORE research project to compile such a benchmark suite for their industrial systems.

In this section, we provide an initial, *explorative* evaluation of our approach. We analyze sequence diagrams that are cutouts from the open-source applications Mozilla and MySQL. The diagrams are similar to the Bzip2 diagram in section 3. The diagrams describe concurrent collaboration scenarios for which a concurrency bug report was filed. We use the diagrams as input to our tool chain and ask whether we can detect the hidden defect in the diagram automatically.

The examples in our study are named after the application and the corresponding bug id, f.e., Mozilla 97866.

Due to a lack of proper documentation for the applications, we had to create the sequence diagrams manually "in the aftermath," similar to what one would do when trying to understand the code. Deriving the diagrams from the code was not really an option for this study, since our approach aims at the design phase of the development process, when code is not yet available. We honestly tried to resemble the kind of information and level of detail in the diagrams that would be available in the analysis or design phase of the development process.

We would like to point out that the sequence diagrams we provide always specify a "positive" scenario, that is, a non-defective sequence of operations, as the scenario should work. Still, a concurrency defect is lurking in the background and will pop up if the operations are scheduled in a different order.

For each example, we give the computing times for generating the model and for defect-checking the model. The measurements were taken on a standard laptop with a DualCore Intel i7 at 2.8 GHz and 8 GB of RAM, running Win7-64.

### *A. Bzip2*

*Scenario*. We described the processing scenario for Bzip2 in section 3, including a sequence diagram in Fig. 1. The diagram shows the typical sequence of operations as intended by the developer.

*Defect Detection*. Our model generator computes from the sequence diagram the CSP formulas shown in Listings 1-7. The CSP model generalizes the particular sequence of operations provided in the diagram (we discussed this in section 3). The resulting set of CSP traces includes this correct sequence, but also other interleavings of the operations.

When checking the CSP model against our defect patterns, the concurrency bug that is hidden in the diagram is detected: The access-after-deletion pattern matches. Just 3 instances were created for all patterns listed in Table 1, since most read-write patterns don't apply to the diagram. The total computing time was 2.1 seconds (0.6+1.5).

*Discussion*. The sequence diagram specifies the intended operations for a core part of the software, and will occur naturally during design. The level of detail is sufficient to detect the concurrency defect hidden in the scenario.

One might argue that the diagram includes information about how processing finalizes in the given scenario, and that such information is not always included during design; or, it might be part of a separate design artifact. We take this as an indication that we need to merge information from several design diagrams. This is clearly within reach of our technique and can be achieved at both the diagram and the CSP model level. We are working on this.

### *B. Mozilla 97866*

*Scenario*. Our second example is taken from Mozilla. In the scenario, the main thread issues an asynchronous read call to some system library, then enters a loop for a busy wait on the results. The asynchronous call signals that it has finished by setting a flag ("io_pending") that is checked by the main thread inside the wait loop. Fig. 2 shows a sequence diagram that specifies the intended behavior.

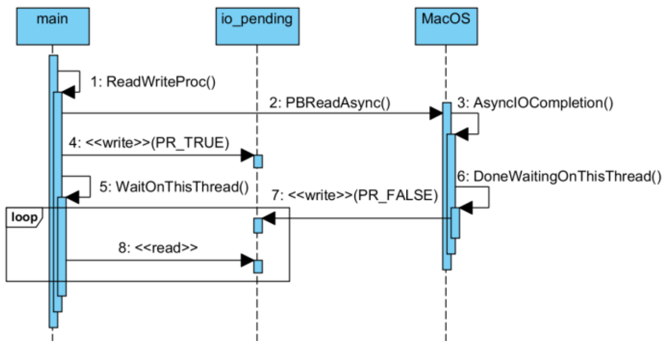

Figure 2. Busy wait on async operation (Mozilla 97866)

*Defect Detection*. Listing 8 shows the CSP formulas that are generated automatically by our tool from the diagram:

```
MOZILLA97866 = (MAIN || IOPENDING || MACOS)
MAIN = (readwriteproc_MAIN → mc_MAIN_MACOS.pbreadasync →
wr_MAIN_IOPENDING.prtrue → waitonthisthread_MAIN → MAINXL)
MAINXL = (rd_MAIN_IOPENDING → MAINXL  Π  rd_MAIN_IOPENDING →
SKIP)
IOPENDING = (IOPENDINGXS1 || IOPENDINGXS2)
IOPENDINGXS1 = (wr_MACOS_IOPENDING.prfalse → IOPENDINGXS1)
IOPENDINGXS2 = (wr_MAIN_IOPENDING.prtrue → IOPENDINGXS2  |
rd_MAIN_IOPENDING → IOPENDINGXS2)
MACOS = (mc_MAIN_MACOS.pbreadasync → asynciocompletion_MACOS →
donewaitingonthisthread_MACOS → wr_MACOS_IOPENDING.prfalse →
MACOS)
```

Listing 8. CSP spec for Mozilla 97866

When checking this model with our defect detector, the asynchronous-wait pattern and two atomicity-violation patterns match, making apparent that the way in which the flag is used is not safe: For certain (rare) schedules, the asynchronous call might return and set the flag before the main thread has had a chance to reset it before entering its loop. As a result, the main thread will loop forever. This bug was actually present in Mozilla and caused the browser to hang. Our tool generated 7 pattern instances. The total computing time was 0.8 seconds (0.5+0.3).

*Discussion*. In the sequence diagram, read and write access to the common flag is specified using stereotypes («read» and «write»). This helps our tool to understand the semantics of the access. We are aware of the fact that designers in practice might be more sloppy when specifying such operations in a diagram. F.e., the stereotypes might be missing, or, operations such as get and set might be used whose semantics are implicitly understood.

A designer might also leave out the explicit lifeline for the flag altogether and specify that the flag is being set and reset in some other way, typically by providing more annotation at certain arrows in the diagram. Fig. 3 shows an example, where access to the flag appears as a self-call within the active object, with the annotation including some assignment statement, such as "io_pending=PR_FALSE". We are extending our parser to handle such cases properly.

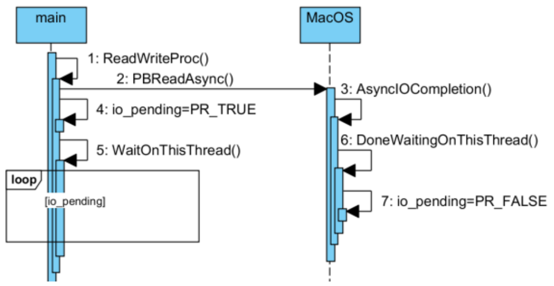

Figure 3. Alternative sequence diagram for Mozilla 97866

### C. *MySQL 3596*

*Scenario*. A central data structure ("proc_info") is managed exclusively by a specific thread. Access to the data is tunneled through this manager thread. Several threads use the manager concurrently. Fig. 4 shows a sample sequence diagram where one thread asks for the data on several occasions; later, another thread resets (part of) the data.

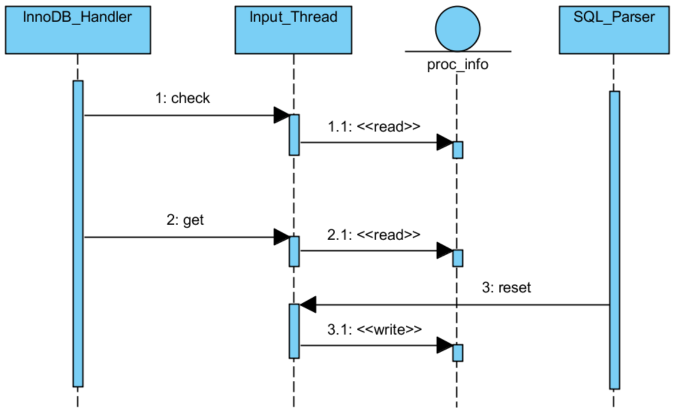

Figure 4. Indirect read/write access (MySQL 3596)

*Defect Detection*. Listing 9 contains the CSP model for the MySQL sequence diagram. Clearly, the problem is that read and write accesses are caused indirectly by different threads and may overlap; this cannot be seen by the clients. The manager thread must provide for proper synchronization.

To detect this kind of defect, we extended our pattern catalogue to cover read/write conflicts that are caused *indirectly* by concurrent external method calls. In the MySQL example, our defect detector created 6 pattern instances, and one of the indirect-atomicity-violation patterns matched. The defect can also be detected if the trailing «write» is substituted with a «destroy». The total computing time was 0.8 seconds (0.5+0.3).

```
MYSQL3596 = (INNODBHANDLER || INPUTTHREAD || SQLPARSER ||
PROCINFO)
INNODBHANDLER = (mc_INNODBHANDLER_INPUTTHREAD.check →
mc_INNODBHANDLER_INPUTTHREAD.get → SKIP)
INPUTTHREAD = (INPUTTHREADXS1 || INPUTTHREADXS2)
INPUTTHREADXS1 = (mc_SQLPARSER_INPUTTHREAD.reset →
wr_INPUTTHREAD_PROCINFO → INPUTTHREADXS1)
INPUTTHREADXS2 = (mc_INNODBHANDLER_INPUTTHREAD.check →
rd_INPUTTHREAD_PROCINFO → INPUTTHREADXS2 |
mc_INNODBHANDLER_INPUTTHREAD.get → rd_INPUTTHREAD_PROCINFO
→ INPUTTHREADXS2)
SQLPARSER = (mc_SQLPARSER_INPUTTHREAD.reset → SKIP)
PROCINFO = (rd_INPUTTHREAD_PROCINFO → PROCINFO |
wr_INPUTTHREAD_PROCINFO → PROCINFO)
```

Listing 9. CSP spec for MySQL 3596

*Discussion*. This synchronization problem may not be obvious at design time. From the perspective of the client threads, simply a few methods are called on the manager thread. This shouldn't cause a problem; after all, delegation is a normal thing in object-oriented software. Hence, it wouldn't make sense to mark all concurrent method calls on an object as error-prone without having additional information on the semantics.

The problem occurs indirectly here, as the manager thread fails to properly synchronize the reads and writes on the data triggered by the method calls. The defect can be detected automatically only if these internals are visible in the design. Similar to previous examples, this might require combining several diagrams in our model.

### D. *Mozilla 73291 Synchronized*

We also studied examples in which a known concurrency defect was repaired by introducing proper synchronization. The corresponding CSP model then includes locking events and does *not* exhibit any defective traces; hence, no defect pattern matches, and our tool handles the repaired diagram correctly. Mozilla bug 73291 provides an example.

*Scenario*. When layouting text, the Mozilla browser uses three distinct variables (content, length, offset) to specify which text segment to access. Several active objects may access the text storage concurrently. The three variables logically belong together; non-atomic concurrent updates would leave the text storage in an inconsistent state. In the original Mozilla code, this scenario caused a concurrency bug that was reported to crash the browser. Fig. 5 shows a sequence diagram in which the bug is repaired: the read accesses are enclosed in a UML 2.4 critical-construct, and so are the write accesses.

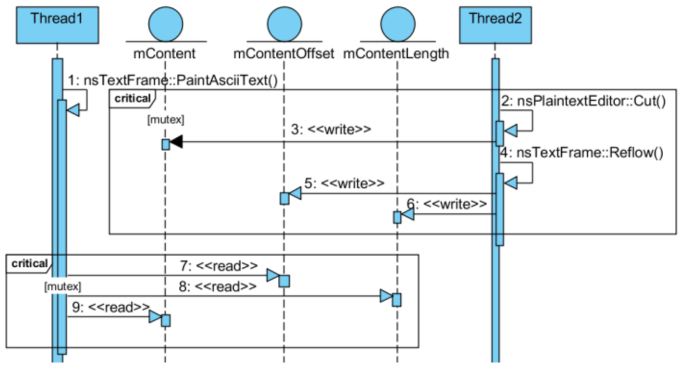

Figure 5. Mozilla 73291 Synchronized

*Defect Detection*. Listing 10 contains the corresponding CSP model. Our model generator automatically includes a mutex process MUTEX, and the thread processes THREAD1 and THREAD2 call the lock- and unlock-operations on the mutex:

```
MOZILLA73291SYNC = (THREAD1 || THREAD2 || MCONTENT ||
MCONTENTOFFSET || MCONTENTLENGTH || MUTEX)
THREAD1 = (nstextframepaintasciitext_THREAD1 → lck_THREAD1_MUTEX →
rd_THREAD1_MCONTENT → rd_THREAD1_MCONTENTOFFSET →
rd_THREAD1_MCONTENTLENGTH → unl_THREAD1_MUTEX → SKIP)
THREAD2 = (lck_THREAD2_MUTEX →  nsplaintexteditorcut_THREAD2 →
nstextframereflow_THREAD2 → wr_THREAD2_MCONTENT →
wr_THREAD2_MCONTENTOFFSET →  wr_THREAD2_MCONTENTLENGTH
→ unl_THREAD2_MUTEX → SKIP)
MCONTENT = (MCONTENTXS1 || MCONTENTXS2)
MCONTENTXS1 = (rd_THREAD1_MCONTENT → MCONTENTXS1)
MCONTENTXS2 = (wr_THREAD2_MCONTENT → MCONTENTXS2)
MCONTENTOFFSET = (MCONTENTOFFSETXS1 || MCONTENTOFFSETXS2)
MCONTENTOFFSETXS1 = (wr_THREAD2_MCONTENTOFFSET →
MCONTENTOFFSETXS1)
MCONTENTOFFSETXS2 = (rd_THREAD1_MCONTENTOFFSET →
MCONTENTOFFSETXS2)
MCONTENTLENGTH = (MCONTENTLENGTHXS1 || MCONTENTLENGTHXS2)
MCONTENTLENGTHXS1 = (wr_THREAD2_MCONTENTLENGTH →
MCONTENTLENGTHXS1)
MCONTENTLENGTHXS2 = (rd_THREAD1_MCONTENTLENGTH →
MCONTENTLENGTHXS2)
MUTEX = (lck_THREAD2_MUTEX → unl_THREAD2_MUTEX → MUTEX  |
lck_THREAD1_MUTEX → unl_THREAD1_MUTEX → MUTEX)
```

Listing 10. CSP spec for Mozilla 73291 synchronized

The synchronization in the CSP model prevents traces that exhibit a concurrency defect. Hence, no defect pattern matches, as expected. Our tool creates 42 pattern instances; the total computing time is 2.4 seconds (0.5+1.9).

*Discussion*. This example illustrates that specifying synchronization in a sequence diagram is straightforward, and that our tool handles synchronization correctly. Alternatively to the critical blocks, the designer might include a lock object and lock/unlock operations in the diagram, but these calls are more difficult to get right than the critical blocks.

In addition, one might well argue that the scenario is low-level, so that there might be no design diagram showing the three variables explicitly. If any, one might see a sequence diagram that specifies the order in which a *single* client should use the variables when accessing text, but this is not sufficient to detect the defect. There seem to be concurrency defects that are so closely tied to the actual code that we cannot expect to see them at the design level. We would guess that many atomicity violations fall into this category. Fortunately, race detectors are good at detecting simple atomicity violations, although one has to wait until coding.

## V. Conclusions

We presented an automated approach for detecting concurrency defects in software designs. Our tool automatically infers formal parallel specifications from standard UML sequence diagrams. We automatically search for concurrency defects in the specifications using generic defect patterns. Any match corresponds to a defective trace at the model level and, hence, points to a potential concurrency defect in the design.

Compared against existing work, our approach is novel in several respects, including: using sequence diagrams as a natural, interaction-centric design input; inferring a formal specification that generalizes the design scenario suitably; checking against generic defect patterns instead of application-specific properties; allowing for checking incomplete design models; producing small models that can be checked fast.

The concurrency defect patterns were adapted from empirical studies of known concurrency defects in large open-source applications. We formally model both the design diagrams and the defect patterns in CSP calculus. Our automated defect detection tool employs the FDR model checker.

We also presented three case studies to provide an initial evaluation of our approach. We modeled and analyzed sequence diagrams for parts of Mozilla, MySQL, and Bzip2. In the examples studied, the concurrency defect that is hidden in the design diagram is detected automatically by our tool, although the diagram only shows a positive scenario, that is, a correct sequence of operations.

More often than not, a design view is incomplete and leaves room for a concurrency defect. The capability of our tool to detect defects in positive scenarios is achieved by generalizing the given scenario carefully when generating the CSP model, as explained in section 3.

As discussed in the evaluation section, our approach can detect a potential concurrency defect only if sufficient information is visible in the design. The necessary information could emerge from combining two or more diagrams, each highlighting part of the picture, such as a sequence diagram and a state diagram from different phases of design. We are currently working on automating the merging of diagrams and of the corresponding formal models.

Developers can support defect detection at the design level in several ways: including initialization and finalization in usage scenarios; modeling central data objects explicitly as separate lifelines; and designing safe concurrent access to data objects explicitly, using locks or the UML critical-construct. This will make the design more complete, easier to understand, and easier to evaluate automatically.

From a development process perspective, our findings provide additional motivation for practitioners to invest more time and effort into designing the concurrent parts of their software. Learning and applying a parallel calculus such as CSP or Petri nets is somewhat difficult. For the purpose of defect detection, there seems to be an easier way for developers than to develop formal specifications manually: With an automated approach, they can start from their familiar design diagrams, generate the formal specs automatically from the diagrams, and search for typical concurrency defects automatically.

Clearly, our automated approach cannot be expected to find all concurrency defects hidden in a design. Some defects are simply too low-level to be visible in the design. Yet, from an in-depth analysis of concurrency defects in real applications that we performed previously, we expect that many defects are being introduced already at the design stage and can be detected from the information available from standard diagrams. It would save substantial development and maintenance effort if the majority were detected before they turn into code.

## ACKNOWLEDGMENT

We would like to thank our industry partners GPP Communications, Acellere, and petaFuel for the joint work in the QUALICORE project. QUALICORE is supported by research grant no. 01|S11011 from the German Federal Ministry of Science and Education BMBF.

## REFERENCES

[1] Fonseca, P., Li, C., Singhal, V., and Rodrigues, R., “A study of the internal and external effects of concurrency bugs,” Int. Conf. on Dependable Systems and Networks (DSN ‘10), 221-230.

[2] Lu, S., Park, S., Seo, E., and Zhou, Y., “Learning from mistakes: a comprehensive study on real world concurrency bug characteristics,” 13th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS’08), 329–339.

[3] Engler, D. and Ashcraft, K., “RacerX: effective, static detection of race conditions and deadlocks,”. 19th ACM Symp. on Operating Systems Principles (SOSP’03), 237–252.

[4] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T., “Eraser: a dynamic data race detector for multithreaded programs,” ACM Trans. Comput. Syst., vol. 15, no. 4 (1997), 391–411.

[5] Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., and Stata, R., “Extended static checking for Java,” ACM Conf. on Programming Language Design and Implementation (PLDI’02), 234–245.

[6] Flanagan, C. and Freund, S.N., “FastTrack: efficient and precise dynamic race detection,” ACM Conf. on Programming Language Design and Implementation (PLDI’09), 121–133.

[7] Bond, M. D, Coons, K. E., and McKinley, K. S., “PACER: proportional detection of data races,” ACM Conf. on Programming Language Design and Implementation (PLDI’10), 255-268.

[8] Serebryany, K., Potapenko, A., Iskhodzhanov, T., and Vyukov, D., “Dynamic Race Detection with LLVM Compiler - Compile-Time Instrumentation for ThreadSanitizer,” Runtime Verification (RV’11) 110-114.

[9] Xu, D. et al., “Model Checking UML Activity Diagrams in FDR,” 8th Int. Conf. on Computer and Information Science (ICIS’09), 1035–1040.

[10] Davies, J. and Crichton, C., “Concurrency and Refinement in the Unified Modeling Language,” Formal Aspects of Computing. vol. 15, no. 2-3 (2003), 118–145.

[11] Möller, M., Olderog, E.-R., Rasch, H., and Wehrheim, H., “Linking CSP-OZ with UML and Java: A Case Study,” 4th Int. Conf. on Integrated Formal Methods (IFM’04), 267-286.

[12] Shousha, M. et al., “A UML/MARTE Model Analysis Method for Uncovering Scenarios Leading to Starvation and Deadlocks in Concurrent Systems,” IEEE Trans. on Softw. Eng., vol. 38, no. 2 (2012), 354–374.

[13] Hoare, C.A.R: *Communicating Sequential Processes*. Prentice Hall, 1985

[14] Formal Systems Software, `fsel.com/software.html`

[15] Gilchrist, J.: *Parallel BZIP2 (PBZIP2) – Data Compression Software*, `compression.ca/pbzip2/`

[16] Yu, J. and Narayanasamy, S., “A case for an interleaving constrained shared-memory multi-processor,” 36th Int. Symp. on Computer Architecture (ISCA’09), 325–336.

[17] Hammer, C., Dolby, J., Vaziri, M., and Tip, F., “Dynamic detection of atomic-set-serializability violations,” 30th Int. Conf. on Softw. Eng. (ICSE’08), 231–240.

[18] Bradbury, J.S. and Jalbert, K., “Defining a Catalog of Programming Anti-Patterns for Concurrent Java,” 3rd Int. Workshop on Softw. Patterns and Quality (SPAQU’09), 6–11.

[19] Luo, Z. Da, Nir-Buchbinder, Y., and Das, R., “Java concurrency bug patterns for multicore systems - Six lesser known Java concurrency bug patterns,” 2010.

[20] Vaziri, M., Tip, F., and Dolby, J., “Associating synchronization constraints with data in an object-oriented language,”.33rd ACM Symp. on Principles of Programming Languages (POPL’06), 334–345.

[21] Lu, S., Tucek, J., Qin, F., and Zhou, Y., “AVIO: Detecting Atomicity Violations via Access Interleaving Invariants,” 11th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS’06), 37–48.

[22] Park, S., Vuduc, R.W., and Harrold, M. J., “Falcon: Fault Localization in Concurrent Programs,” 32nd Int. Conf. on Softw. Eng. (ICSE ’10), 245–254.

[23] Cordeiro, L., and Fischer, B., “Verifying Multi-threaded Software using SMT-based Context-Bounded Model Checking Categories and Subject Descriptors Exploring the Reachability Tree,” 33rd Int. Conf. on Softw. Eng. (ICSE ’11), 331–340.

[24] Wang, C., Chaudhuri, S., Gupta, A., and Yang, Y., “Symbolic pruning of concurrent program executions,” 7th European Softw. Eng. Conf. (ESEC/FSE ’09), 23-32.

[25] Bowles, J., and Meedeniya, D., “Formal Transformation from Sequence Diagrams to Coloured Petri Nets,” 17th Asia Pacific Softw. Eng. Conf. (APSEC ’10), 216–225.

[26] Uchitel, S., Kramer, J., and Magee, J., “Incremental Elaboration of Scenario-Based Specifications and Behaviour Models Using Implied Scenarios,” ACM Trans. Softw. Eng. and Methodology, vol. 13, no. 1 (2004) 37-85.

[27] Uchitel, S., Brunet, G., and Chechik, M., “Behaviour Model Synthesis from Properties and Scenarios,” 29th Int. Conf. Softw. Eng. (ICSE’07), 34-43.

[28] Mitchell, B., “Characterizing Communication Channel Deadlocks in Sequence Diagrams,” IEEE Trans. on Softw. Eng., vol. 34, no. 3 (2008) 305–320.

[29] Zhang, W.E.I., Sun, C., Lim, J., Lu, S., and Madison, W., “ConMem: Detecting Crash-Triggering Concurrency Bugs through an Effect-Oriented Approach,” ACM Trans. on Softw. Eng. and Methodology, vol. 22, no. 2 (2013), 1–33.

[30] Campbell, L.A., Cheng, B.H.C., McUmber, W.E., and Stirewalt, R.E.K., “Automatically detecting and visualising errors in UML diagrams,” Requirements Eng., vol. 7, no. 4 (2002), 264–287.

[31] Thierry-Mieg, Y. and Hillah, L.-M., “UML behavioral consistency checking using instantiable Petri nets,” Innovations in Systems and Softw. Eng., vol. 4, no. 3 (2008), 293–300.

[32] Baresi, L., and Pezze, M., “On Formalizing UML with High-Level Petri Nets,” in Springer LNCS 2001, Concurrent Object-Oriented Programming and Petri Nets, ed. G.A. Agha, F. Cindio, and G. Rozenberg, Springer Verlag (2001), 276–304.

[33] Liu, S., Liu, Y., Choppy, C., Sun, J., Wadhwa, B., and Dong, J.S., “A Formal Semantics for Complete UML State Machines with Communications,” Integrated Formal Methods (IFM’13), 331–346.

[34] Evans, A., France, R., Lano, K., and Rumpe, B., “The UML as a formal modeling notation,” Computer Standards & Interfaces, vol. 19, no. 7 (1998), 325–334.

[35] Gagnon, P., Mokhati, F., and Badri, M., “Applying Model Checking to Concurrent UML Models,” Journal of Object Technology, vol. 7, no. 1 (2008), 59-84.

[36] Sinha, N., and Wang, C., “Staged concurrent program analysis,” 18th ACM Int. Symp. on Foundations of Softw. Eng. (FSE’10) 47-56.

[37] Dan, L., “QVT Based Model Transformation from Sequence Diagram to CSP,” 15th Int. Conf. on Eng. of Complex Computer Systems (ICECCS’10), 349-354