Towards Reliable Concurrent Software

Marieke Huisman and Sebastiaan J. C. Joosten

Abstract As the use of concurrent software is increasing, we urgently need techniques to establish the correctness of such applications. Over the last years, significant progress has been made in the area of software verification, making verification techniques usable for realistic applications. However, much of this work concentrates on sequential software, and a next step is necessary to apply these results also on realistic concurrent software. In this paper, we outline a research agenda to realise this goal. We argue that current techniques for verification of concurrent software need to be further developed in multiple directions: extending the class of properties that can be established, improving the level of automation that is available for this kind of verification, and enlarging the class of concurrent programs that can be verified.

1 Introduction

Software is everywhere! Every day we use and rely upon enormous amounts of software, without even being aware of it [33]. This includes the obvious applications, such as mobile phone apps and all kinds of office software, but also the software in our cars, household equipment, airplanes etc. It has become impossible to imagine what life would be like without software. What we are not aware of, is how much software is actually safety-critical or business-critical, and how big the risk is that one day software failures will bring this everyday life to a grinding halt. In fact, all software contains errors that cause it to behave in unintended ways [32, 49]. Studies have shown that software applications have on average between 1 and 16 errors per 1000 lines of code, even when tested and deployed [58, 59], and substantial research is needed to reduce this number and to make software that is reliable under all circumstances, without compromising its performance.

M. Huisman · S. J. C. Joosten (✉)
University of Twente, Enschede, The Netherlands
e-mail: m.huisman@utwente.nl; s.j.c.joosten@utwente.nl

P. Müller, I. Schaefer (eds.), *Principled Software Development*,
https://doi.org/10.1007/978-3-319-98047-8_9

A commonly used approach to improve software performance is the use of *concurrency* and *distribution*. For many applications, a smart split into parallel computations can lead to a significant increase in performance. Unfortunately, parallel computations make it more difficult to guarantee *reliability* of the software. The consequence is unsettling: the use of concurrent and distributed software is widespread, because it provides efficiency and robustness, but the unpredictability of its behaviour makes that errors can occur at unexpected, seemingly random moments.

As we will see below, the quest for reliable software builds on a long history, and significant progress has already been made. Nevertheless, ensuring reliability of efficient concurrent and distributed software remains an open challenge. Ultimately, it is our dream that program verification techniques are built into software development environments. When a software developer writes a program, he explicitly writes down the crucial desired properties about the program, as well as the assumptions under which the different program components may be executed. Continuously, an automatic check is applied to decide whether the desired properties are indeed established, and whether the assumptions are respected. If this is not the case, this is shown to the developer—with useful feedback on why the program does not behave as intended.

This paper outlines a research agenda, discussing what we believe are the crucial step towards reaching this goal. First, in Sect. 2 we will discuss the state-of-the-art in verification of concurrent software, focusing in particular on deductive verification techniques. In Sect. 3 we identify three main directions of research, and discusses challenges and a possible approach for each of those directions. This section discusses how abstract models can be combined with deductive verification techniques to reason about global functional correctness properties of programs. Section 4 discusses the need for and possible approaches to further increase the level of automation in deductive verification. Finally, Sect. 5 sketches the steps and chances that exist to adapt existing verification techniques to other concurrent programming models.

2 State-of-the-Art in Verification of Concurrent Software

2.1 Software Correctness

The quest for software correctness is an old tale (see Fig. 1 for a historic overview). Already in the sixties, in the early days of computing, Floyd and Hoare realised that it is actually possible to prove that a program behaves as intended [30, 37]. Given a small code fragment, and a specification of what the fragment is supposed to do, a collection of simple proof rules was devised, which can be used to establish whether a program behaves as specified. By applying the proof rules, auxiliary proof obligations in first-order logic are generated. If the proof obligations can be

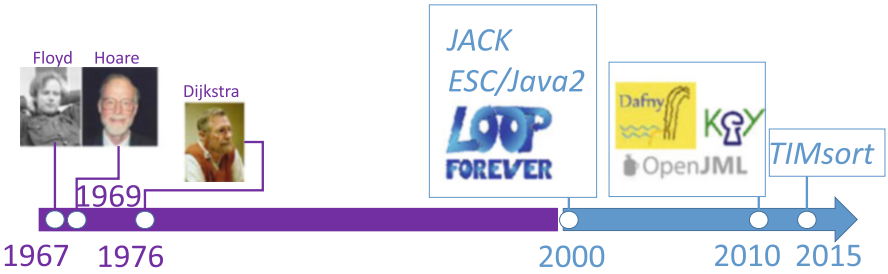

Fig. 1 Development of Sequential Software Verification

proven, we can conclude that the program satisfies its specification. This approach, called Floyd-Hoare or Hoare logic, still forms the basis for many techniques to reason about program behaviour (usually implemented using Dijkstra's predicate transformer semantics [24]).

For a long time, program verification remained a pen-and-paper activity. However, around the year 2000, several groups started working on the development of tools to support this kind of verification [8–10, 20, 38, 50]. There are several technical reasons behind the coordination of these developments:

- the emergence of Java meant that there was a popular and widely-used programming language with a reasonably well-defined semantics, amenable to formal reasoning;
- in addition, computing power had increased, which made it actually feasible to build efficient tools to reason about non-trivial programs; and
- there was tremendous progress in automated verification technology for first-order logic, which enabled automatic discharge of auxiliary proof obligations, culminating in modern, very powerful SMT solvers.

Since then, work on these program verification tools has progressed, resulting in tools such as OpenJML [21], CodeContracts [28, 47], and the most recent versions of KeY [1], which are now being used in teaching, integrated in standard development environments, and to verify or find bugs in non-trivial algorithms, such as TIMsort [23].

Despite the enormous progress that has been made, there are still many open challenges in this area [35]. One important open challenge for program verification is that it still requires a substantial level of expertise, in particular because of the high number of auxiliary annotations that have to be provided to guide the proving process (see for example the solutions to the VerifyThis program verification challenges at http://www.verifythis.org).

2.2 *Verification of Concurrent Software*

All techniques mentioned above focus on proving local safety properties of sequential programs, i.e., with a single thread of execution, but cannot specify or effectively prove properties on the global program behaviour of concurrent or distributed software. Thus, extending program verification techniques to enable reasoning about programs with multiple threads of execution is a necessary step to ensure the reliability of realistic programs, see Fig. 2 for a historic overview.

Already in the 70s, Owicki and Gries proposed a technique to extend program logic to reason about concurrent programs [60]. Their technique required annotations for each atomic step in the program, and a proof that these annotations could not be invalidated by any atomic step made by other program threads, thus resulting in a non-modular verification technique with an exponential number of proof obligations. In particular, if a verified program is extended with a new thread, also the existing threads have to be reverified. In 1980, Jones proposed a modular verification technique for concurrent programs, called rely-guarantee reasoning [41]. In rely-guarantee reasoning the verifier explicitly specifies the steps that are allowed for the environment, which requires thorough understanding of the application at hand.

About 10 years ago, Concurrent Separation Logic (CSL) was invented [17, 53] (Brookes and O'Hearn received the Gödel prize 2016 for this achievement). This was an important step for the verification of concurrent software, as it enabled thread-modular verification. Originally, separation logic was proposed as an extension of classical Hoare logic to reason about pointer programs, by explicitly considering which memory locations are relevant for what part of the program [54, 55]. This characteristic makes it also extremely suitable to reason about concurrent programs: if we can prove that two threads work on disjoint parts of the memory, then we know that they cannot interfere with each other.

The invention of concurrent separation logic led to a whole plethora of techniques and logics to reason about concurrent software, focusing on different aspects, see [18] for an overview.

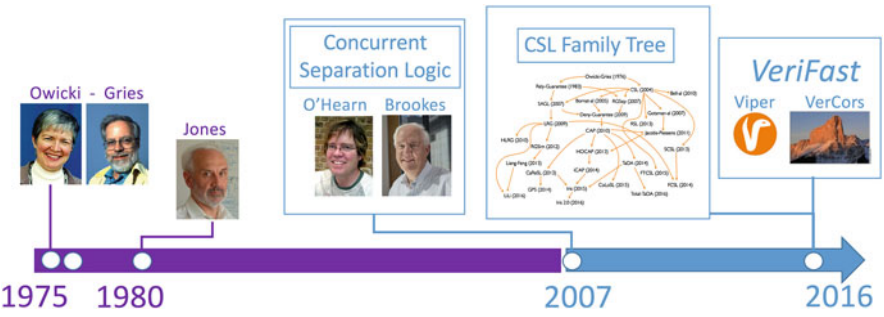

Fig. 2 Development of Concurrent Software Verification

In one line of work, more and more advanced logics are proposed, grouped in the CSL family tree [18]. This contains for example a combination of rely guarantee and separation logic [67], (impredicative) concurrent abstract predicates [25, 66], TaDa (a logic for time and data abstraction) [61, 62], fine-grained concurrent separation logic [52, 63], a combination of monoids and invariants [44, 45], and reasoning based on linearisation points [36, 68], with the aim of finding a generic logic, which can be used to verify the behaviour of all concurrent programs. So far, these approaches are still fairly theoretic, and require a high level of expertise. Some of these logics are formalised in Coq, with suitable tactics to use them inside Coq. Further, they are usually developed for relatively simple core programming languages, and focus on small but intricate examples.

In another line of work, the focus is on developing practical techniques to reason about commonly used programs, using various synchronisation methods, support for dynamic thread creation, reentrant locks etc. This has been the focus of our work on the VerCors tool set [3–5, 14], where we developed techniques (with tool support) to reason about multi-threaded Java and OpenCL programs. This is also the aim of the VeriFast tool, for verification of single- and multithreaded C and Java programs [39, 65] and the Viper framework, which provides support for separation logic-based reasoning for a low-level intermediate language [43, 51]. In particular, our VerCors tool is build on top of the Viper framework. Some of the more theoretical results on verification of concurrent software are (partially) integrated in these techniques.

By now, there is a plethora of logics to verify specific core properties about concurrent software, such as that the program is free of data races. The next challenge is to efficiently prove properties about the *global functional behaviour* of a realistic concurrent program.

2.3 Concurrent Software in Industrial Practice

Because of high demands on software performance, industry is using concurrency more and more in their daily practice. However, for many companies, reliability of the software they develop is very important: if their software is misbehaving, they risk losing the confidence of their customers. Therefore, we see that companies are often quite conservative in their use of concurrency: they use well-known programming patterns, reuse existing libraries as much as possible, and try to isolate the concurrency-related aspects to a small part of their application.

Software developers need effective verification techniques to improve the quality and reliability of their concurrent software. We believe that to develop these techniques, the ultimate challenge is not in finding a logic that can reason about all possible concurrent programs. Instead, the challenge is to develop techniques that can be used efficiently on many common concurrent programming patterns, and that can be used to detect bugs quickly and effectively, without requiring too many user interventions, and without too many false positives. This conviction is

what drives our research: we aim at developing techniques that can help software developers in their daily software development practice to improve the quality of the software they are producing.

3 Abstraction Techniques for Functional Verification

One of the main open challenges for the verification of concurrent software that we consider is how to to develop techniques to *automatically verify global functional correctness properties of concurrent and distributed software*, i.e., to ensure that an application has the expected behaviour and does not experience failures.

To reach this goal, we advocate an approach where a *mathematical model* of a concurrent application is constructed, which provides an *abstract view* of the program's behaviour, leaving out details that are irrelevant for the properties being checked, see Fig. 3. The main verification steps in this approach are

1. *algorithmic verification* over the mathematical model to reason about global program behaviour, and
2. *program logics* to verify the formal connection between the software and its mathematical model.

Typically, the basic building blocks of the abstract mathematical model are *actions*, for which we can prove a correspondence between abstract actions and concrete code fragments, which is then used to prove the formal connection between the software and its mathematical model. Moreover, this has the advantage that if a global property does not hold at the abstract level, the abstract-level counterexample corresponds to a concrete candidate counterexample at the software level.

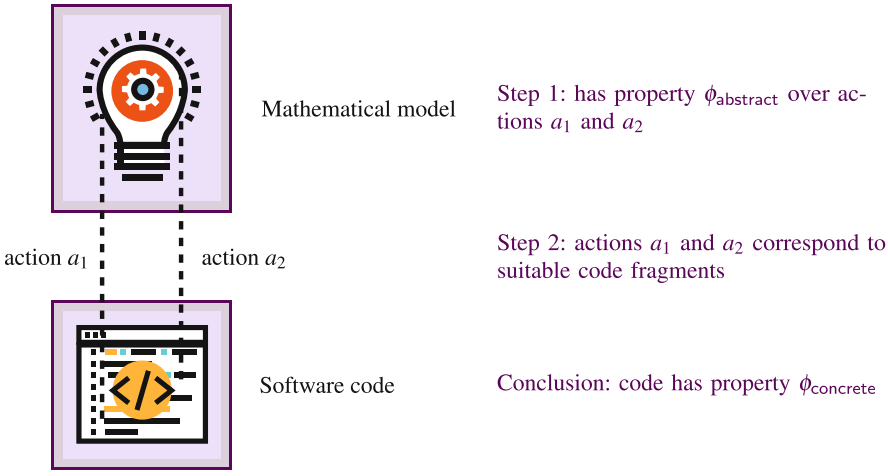

Fig. 3 Using abstraction for the verification of concurrent and distributed software

Within this approach, a *software designer* specifies the desired *global properties* for a given application in terms of abstract actions. The *software developer* should then specify how these *abstract actions map to concrete program state*: in which states is the action allowed, and what will be its effect on the program state. Global properties may be safety properties, e.g., an invariant relation between the values of variables in different components, or a complicated protocol specifying correct interface usage, but we believe that extensions of the approach to liveness and progress properties are also possible.

To make this approach possible, we believe the following challenges should be addressed:

1. identify a good abstraction theory,
2. extend the abstraction theory to reason about progress and liveness properties of code, and
3. use the abstraction theory to guide the programmer to develop working code through refinement.

We discuss these challenges in more detail, and discuss our first results in this direction.

3.1 The Right Mathematical Model

The purpose of a good abstraction is that it reduces the verification effort in two ways: it makes it easier for the software designer to reason about the essential parts of his program, and automated verification methods can be used, because the verification effort is used on a model that is smaller than the original program. Moreover, the abstraction should support modular and compositional verification.

To find such a level of abstraction, we need to look at what currently hinders verification. One problem is the large state space of a program, as we have to consider all possible values of all program variables. Thus, a suitable abstraction needs to be able to describe a reachable configuration of variables as a single mathematical object.

Moreover, verification of concurrent software needs to consider all possible interleavings of the threads. Thus, we need to find ways to group actions, in particular also actions that do not occur inside atomic blocks of code. The theory of linearisation points will be a good starting point [36, 68] for this, but it needs to be further generalised, as abstract actions also could correspond to method calls, and not just to memory writes.

Summarising, a good abstraction should have the following properties:

- it can accurately capture low-level implementation details,
- it is modular,
- it can abstract over a sequence of multiple actions, and
- it can abstract over a valuation of multiple variables.

When developing this abstraction theory, we use the logics in the CSL family tree [18] as an important source of inspiration. In particular, the notion of views has been advocated as a general framework that captures many commonalities in the verification of concurrent software [26], and we believe it is important that our basic abstraction theory can be described in terms of views. However, to further the state of the art in program verification, we believe there are two more additional requirements that should be considered, namely that the abstraction should be able to reason about time-dependent properties, and it should facilitate reasoning in a top-down manner, as will be motivated below.

3.2 Reasoning About Liveness and Progress

Most of program verification concerns the verification of safety properties: functional correctness is interpreted as ensuring that under a certain precondition, the postcondition will hold after executing some code. Checking that a postcondition is not violated corresponds to verifying that variable assignments that violate the postcondition are not reachable, which is a safety property. However, when designing concurrent code, safety is only one issue software designers need to deal with, they also need to make sure that their program will not deadlock, and will eventually do the right thing.

The latter property is called progress: if an action is enabled, it will eventually happen. Whether or not a progress property holds depends on a program scheduler, which depends on hardware, firmware, drivers and software. We therefore need an approach in which the assumptions about the scheduler can be made explicit.

We wish to use the abstraction theory to support reasoning about global liveness properties. This means that the abstractions need to incorporate a notion that an action must happen. This has been explored earlier by Larsen *et al.* who defined modal transition systems as an extension of standard LTSs with must- and may-transitions [6, 46].

Variant-based reasoning allows us to show that an action indeed will happen. As the actions might happen in different parts of the program, using this variant-based technique might not always be straightforward. We may develop a nested approach for variant-based reasoning, where at the lowest level we show that individual methods terminate, and at higher levels we show that a sequence of method calls terminates (assuming that the method calls themselves terminate). In the rewriting community there is a substantial amount of work on termination of rewrite systems, some of which has been applied to sequential programming languages and transition systems [16, 34, 42], and it should be investigated if and how these techniques can be used in the context of concurrent software verification.

3.3 Unification of Model and Code

Finally, we believe that if the abstraction theory is fully developed, it should also be usable in the opposite direction: if we take an abstract model as a starting point, can we use *refinement* to transform this into correct working code?

The basic refinement process can be divided into two phases. In the first phase, the global property can be separated into properties about individual processes. This might introduce some communication steps, and we need to resort here to the notion of process of equivalence under hiding of actions. The big challenge in this step is to decide how the property should be split. When verifying a concrete program, the program code dictates how this should be done. But in this case, where we wish to generate the program, we need other ways to do this. We plan to investigate different possibilities, for example maximising parallelisation, where each thread is responsible for an individual action, or even multiple threads execute the same action, all in parallel; minimising parallelisation, and grouping sets of related actions. For all these possibilities, different splitting strategies can be developed, but typically some user intervention will be necessary here to indicate the intent of the program.

In the second phase, we have process algebra terms that describe the behaviour of an individual thread. The process algebra term itself describes the control flow of the thread, and we will develop a technique to transform each process algebra term into a sequence of program instructions. The abstraction typically defines some variables, which model the synchronisation between the different threads and are used to capture the effect of the actions. These variables should be mapped into concrete program variables, and as a last step the actions are translated into concrete program code, executing the action's specified effect. Typically, the guards will be fulfilled by construction, and do not have to be incorporated in the generated program code.

In the long run, the results of these investigations might lead to a theory that unifies models and programs and removes the borders between the two. This would allow us to reason about systems where some components are already implemented, and others are only specified by a model, which later might be refined into an implementation.

3.4 First Steps Towards a Solution

In our earlier work on abstract models [13, 56, 69], we have shown that it is possible to use process algebra terms to describe the abstract control flow of a program. This allow us to show that the program behaves according to a certain protocol (for example preventing unwanted flow of information by ensuring that a send may never occur after a receive) or that a variable evolves according to a particular pattern (for example, a variable only increases, a queue never becomes empty etc.). The unique

characteristic of this approach is that we can prove the correspondence between the abstract model and the program code using standard program logic, by linking the actions that are the basic building blocks of the model to concrete program statements.

Below, this approach is sketched on a very simple example. Suppose we have a shared variable `x` protected by a lock `lck`, and we have two threads that manipulate `x`: one thread multiplies `x` by 4, the other thread adds 4 to `x`. The specification of the thread that performs the multiplication captures that the multiplication has happened. For this we use the notion of history [13]: an abstraction of the actions a thread has taken up-to now. If before the thread is executed, the history is equal to H (written *Hist(H)*), then afterwards the action *mult(4)* is added at the end of the history ($P.a$ is notation for a process P, followed by action a). Similarly the specification of the addition thread captures that the addition has happened. The *action* annotation inside the method body indicates the concrete code fragment that corresponds to this abstract action.

```
class Mult extends Thread {

//@ requires Hist(H);
//@ ensures Hist(H.mult(4));
public void run() {
    //@ action mult(4) {
      lock(lck);
      x = x * 4;
      unlock(lck);
    //@ }
  }
}
```

```
class Add extends Thread {

//@ requires Hist(H);
//@ ensures Hist(H.add(4));
public void run() {
    //@ action add(4) {
      lock(lck);
      x = x + 4;
      unlock(lck);
    //@ }
  }
}
```

Next, we have action specifications that describe the effect of the actions *mult* and *add*. Using program logic, we can prove that the action implementations (in the thread bodies) indeed behave as specified.

```
//@ assume true;
//@ guarantee x == \old(x) * k;
action mult(k);

//@ assume true;
//@ guarantee x == \old(x) + k;
action add(k);
```

Suppose we have a `main` method, which starts the two threads and then waits for them to terminate. For this main method we can specify and verify (using a history-aware program logic) that it will execute the *mult* and the *add* action in any order (where $P + Q$ denotes a non-deterministic choice between P and Q and *empty* denotes an empty history).

```
//@ requires Hist(empty) & x == 0;
//@ ensures Hist(mult(4).add(4) + add(4).mult(4));
public void main(...) {
  Thread t1 = new Mult();  Thread t2 = new Add();
  t1.fork(); t2.fork();
  t1.join(); t2.join();
}
```

From the history specification of the `main` method and the action specifications, we can derive the possible values of variable x after termination of the `main` method, i.e., x can be either 4 or 16.

This example is very simple, but the same approach can be used in many different settings: for larger programs, non-terminating programs, distributed programs etc. In particular, for non-terminating programs, an abstraction can be used to predict the (abstract) behaviour, and correctness of the abstraction boils down to showing that the program flow never moves out of the predicted behaviour [56, 69]. We have used this approach to prove properties such as: in a concurrent queue, the order of elements is preserved [2]; adherence to protocols that are commonly used to capture essential security properties, such as 'no send after receive' [56]; and correctness of an active object implementation using MPI operations [70]. Our own VerCors tool set provides support to reason in this way, but also the VeriFast tool can reason about histories (personal communication by Bart Jacobs, KU Leuven).

Note that an essential difference with other existing abstraction-based approaches such as CEGAR and IC3 [15, 19, 48] is how the correctness of the abstraction is proven. Usually, the relation between the original program and the abstract program is proven as a meta-theorem, and one has to trust the implementation of the algorithm that performs the abstractions (or check it manually), while in our approach the program logic is used to prove correctness of the abstraction.

4 Automating the Verification Process

Another major challenge that we need to consider is how to automate the verification process. At the moment, program verification requires many user annotations, explicitly describing properties which are often obvious to developers. We believe that many of the required annotations can be generated automatically, using a combination of appropriate static analyses and smart heuristics. We advocate a very pragmatic approach to annotation generation, where any technique that can be used to reduce the annotation burden is applied, combined with a smart algorithm to evaluate the usability of a generated annotation, removing any annotations that do not help automation. This will lead to a framework where for a large subset of non-trivial programs, we can automatically verify many common safety properties (absence of null-pointer dereferencing, absence of array out of bounds indexing, absence of data races etc.), and if we wish to verify more advanced functional

properties, the developer might have to provide a few crucial annotations, but does not have to spell out in detail what happens at every point in the program (in contrast to current program verification practice). However, it should be stressed that with this approach, we will never be able to automatically verify correctness of all programs; there will always be programs using unusual patterns, which need additional manual annotations in order to be verified.

We believe that efficient annotation generation should build on existing static analyses and heuristics [27, 29, 31, 40] extended with tailor-made new generation techniques, aiming for an optimal verification result within a minimal amount of time.

There is a plethora of tools and techniques available which can be used to derive properties about the program state. However, many of these tools work on simple idealised languages, and these results will have to be extended to a more realistic programming setting. In particular, some approaches do not consider aliasing, which is often essential for the correctness of a program.

Moreover, if we use any technique that is available, this might lead to an overload of annotations, which can have a negative impact on verifiability. We thus need to find an optimal balance in how and when to generate annotations automatically. This will be an incremental process, where we use different analyses or heuristics to generate annotations and then select those that help towards our goal. Some of the generated annotations will need other auxiliary annotations to be verified automatically, thus we need to find a suitable order in which to apply the annotation generation algorithms. For example, if an analysis is sensitive to aliasing, we might first want to use an analysis which can derive some annotations about when two variables may or may not be aliased. Note that if we use unsound heuristics to generate annotations, this may lead to conflicting annotations, which might actually give a false impression of program correctness. Therefore, we also need to investigate efficient ways to avoid conflicting annotations. In some cases, a syntactic check will be sufficient to conclude that two annotations are not conflicting. Making optimal use of these cases will help to make this conflict check efficient.

Lastly, if an annotation cannot be verified, we have to investigate how to provide the most suitable feedback. It is important to distinguish between the two following cases:

1. a counterexample exists, which thus means that the annotation is incorrect. In this case, either the annotation is removed, or if a counterexample exists for the property the developer wanted to show for the program, the counterexample has to be presented to the user. In that case, it is important that the counterexample is intelligible, and helps the developer to understand why the program does not have the intended behaviour, and how to fix this.
2. there is not sufficient information to prove the annotation. In this case, the annotation might still be kept as a candidate annotation, because when more annotations are generated, it might become possible to prove it. An intelligent strategy will be needed to keep potentially interesting annotations (for example, if the annotation would help to prove the globally desired property, it is potentially interesting), while ignoring others.

5 Verification of Programs Using Different Concurrency Paradigms

To support software developers in practice, verification techniques need to support different programming languages, and different concurrency paradigms. Most work on the verification of concurrent software focuses on shared memory concurrency with heterogeneous threading, as can be found, e.g., in Java or C. In this model, all threads have access to a shared memory, and all threads execute their own program code. However, in practice there are many other concurrency models in use (and there is also more and more hardware that supports these concurrency models directly). Therefore, we need to investigate how our verification approaches can be used for these other concurrency models as well.

In particular, we believe that it is important to investigate how to reason about programs written using the structured parallel programming model (or vector-based programming), where all threads execute the same instructions, as this model is growing in popularity. Recently, we have shown how the verification techniques for Java's shared-memory concurrency can be adapted in a straightforward manner to GPUs (including atomic update instructions) [11, 12]. On GPUs, there is a shared memory, but all threads execute the same program instructions (but operate on a disjoint part of the memory). It turns out that this restricted setting has a positive impact on verification: the same verification techniques can be used, and verification actually gets simpler. Because of the simpler concurrency paradigm, reasoning about many functional properties can be done without the need for abstraction, because the behaviour of all the other threads is more predictable. However, to reason about the interaction of the vector program in interaction with the host program, which invokes the vector program (the kernel), we are again back to the heterogeneous setting, and the abstraction theory can be used to give an abstract specification of the behaviour of the vector program. We believe that this direction should be explored further, as typical GPU programs are usually quite low-level, which makes them more error-prone. Thus, there is a high need to further develop automated techniques to reason about such applications.

It is also interesting to look at how these programs are developed. One commonly-used approach is that a developer writes a sequential program and gives compiler hints about possible parallelisations [7]. When this approach is used, a programmer is greatly helped by automated verification of these compiler directives. For basic compiler directives, we developed verification techniques to prove the correctness of these parallelisations, i.e., to prove that if the program is parallelised, its behaviour will be equivalent to the behaviour of the sequential program [11, 22], but this approach is still in its early stages, and needs to be developed further, allowing for more advanced compilation patterns.

Further, we believe that a promising line of work is to combine these techniques, in such a way that one can automatically transform a verified sequential program with annotations into an annotated vector program, which will be directly verifiable. We believe this idea can also be used for other compiler optimisations. For example,

vector programs written in OpenCL (a platform-independent programming language for GPUs) can be executed both on CPUs and GPUs, but experiments have shown that to optimise performance, the data format should be different [64]. This idea can be defined as a standard compiler transformation, that transforms not only the program, but also the correctness annotations, such that the result is a (hopefully) verifiable program again. Instead of proving correctness of the transformation, both the program and the annotations are transformed, such that after the transformation the resulting program with annotations can be reverified.

Another interesting paradigm that deserves more attention is the area of distributed software, where we need techniques to reason about programs without shared memory. One particular instance of these are distributed programs, but there are also concurrency paradigms, such as the message-box concurrency model of Scala, where each parallel computation works on its own memory. We have shown that reasoning about distributed programs using the message passing interface (MPI) builds on the same principles [57] as reasoning about shared memory concurrency, but here the abstraction plays an even more important role, because it models the communication between the different computations. By adding a notion of synchronisation to the actions, we can model the communication. By defining variations in the action synchronisation, it should be possible to model other distributed programming models, such as the actor model, as well as the Scala concurrency model (on a single computer, with instantaneous send and receive), or variations of the MPI model, where the sending of messages can take time, and messages can bypass each other. It would even be possible to use this kind of reasoning at a lower level, for example to prove the correctness of an MPI implementation, where we take into account that messages might be lost.

6 Concurrent Software in 10 Years

As we have seen, over the last years, there has been enormous progress in the area of program verification, and in particular concerning the verification of concurrent software. By now, the theory behind verification of concurrent software is reasonably well understood, even though there are still open ends, but a large step is still needed to make the results usable for all programmers, in their every-day software development.

In this paper, we discussed some challenges that need to be addressed to achieve this, and we also outlined possible approaches to tackle them. In the coming years, we plan to develop techniques to address these questions, which should lead to a situation where software verification techniques will be an integral part of the software development practice, also for highly complicated concurrent software.

When verification is an integral part of software development, developing code that is formally correct will be deemed easier than developing code without formal verification. If correctness is built into the software compile chain, checking correctness and occasionally getting verification errors will be as commonplace

as dealing with type checking errors. In ten years, writing code without static verification might be seen as this obscure workaround that can be okay to use if you really know what you are doing. Using automated verification will be as normal as structured programming and static type checking is now.

Acknowledgement This work is supported by the NWO VICI 639.023.710 Mercedes project.

References

1. Wolfgang Ahrendt et al. *Deductive Software Verification – The KeY Book* Vol. 10001. Lecture Notes in Computer Science. Springer International Publishing, 2016. ISBN: 9783319498126.
2. A. Amighi, S. Blom, and M. Huisman. "VerCors: A Layered Approach to Practical Verification of Concurrent Software". In: *PDP* 2016, pp. 495–503.
3. Afshin Amighi et al. "Verification of Concurrent Systems with VerCors". In: *Formal Methods for Executable Software Models 14th International School on Formal Methods for the Design of Computer Communication, and Software Systems, SFM 2014, Bertinoro, Italy June 16–20, 2014, Advanced Lectures* 2014, pp. 172–216.
4. A. Amighi et al. "Permission-based separation logic for multithreaded Java programs". In: *LMCS* 11.1 (2015).
5. A. Amighi et al. "The VerCors Project: Setting Up Basecamp". In: *Programming Languages meets Program Verification (PLPV 2012)* ACM Press, 2012, pp. 71–82. https://doi.org/10.1145/2103776.2103785
6. A. Antonik et al. "20 years of modal and mixed specifications". In: *Bulletin of the EATCS* 95 (2008), pp. 94–129.
7. R. Baghdadi et al. "PENCIL: Towards a Platform-Neutral Compute Intermediate Language for DSLs". In: *CoRR* abs/1302.5586 (2013).
8. G. Barthe et al. "JACK: A Tool for Validation of Security and Behaviour of Java Applications". In: *Formal Methods for Components and Objects (FMCO 2006)* Vol. 4709. LNCS. Springer, 2007, pp. 152–174.
9. B. Beckert, R. Hähnle, and P.H. Schmitt, eds. *Verification of Object-Oriented Software: The KeY Approach* Vol. 4334. LNCS. Springer, 2007.
10. J. van den Berg and B. Jacobs. "The LOOP compiler for Java and JML". In: *Tools and Algorithms for the Construction and Analysis of Systems* Ed. by T. Margaria and W. Yi. Vol. 2031. LNCS. Springer, 2001, pp. 299–312.
11. S. Blom, S. Darabi, and M. Huisman. "Verification of loop parallelisations". In: *FASE* Vol. 9033. LNCS. Springer, 2015, pp. 202–217.
12. S. Blom, M. Huisman, and M. Mihelvić "Specification and Verification of GPGPU programs". In: *Science of Computer Programming* 95 (3 2014), pp. 376–388. ISSN: 0167–6423.
13. S. Blom, M. Huisman, and M. Zaharieva-Stojanovski. "History-based verification of functional behaviour of concurrent programs". In: *SEFM*. Vol. 9276. LNCS. Springer, 2015, pp. 84–98.
14. S. Blom et al "The VerCors Tool Set: Verification of Parallel and Concurrent Software". In: *iFM* Vol. 10510. LNCS. Springer, 2017, pp. 102–110.
15. A.R. Bradley. "SAT-Based Model Checking without Unrolling". In: *Verification, Model Checking and Abstract Interpretation (VMCAI)* LNCS. Springer, 2011.
16. Marc Brockschmidt et al. "Certifying safety and termination proofs for integer transition systems". In: *International Conference on Automated Deduction* Springer. 2017, pp. 454–471.
17. S. Brookes. "A Semantics for Concurrent Separation Logic". In: *Theoretical Computer Science* 375.1–3 (2007), pp. 227–270.
18. Steve Brookes and Peter O'Hearn. "Concurrent Separation Logic". In: *ACM SIGLOG News* 3.3 (2016), pp. 47–65.

19. E. Clarke et al. “Counterexample-Guided Abstraction Refinement”. In: *Computer-Aided Verification (CAV)* Vol. 1855. LNCS. Springer, 2000.
20. D. Cok and J. R. Kiniry. “ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an Internet voting tally system”. In: *Proceedings, Construction and Analysis of Safe Secure and Interoperable Smart devices (CASSIS’04) Workshop* Ed. by G. Barthe et al. Vol. 3362. LNCS. Springer, 2005, pp. 108–128.
21. David Cok. “OpenJML: Software verification for Java 7 using JML, OpenJDK, and Eclipse”. In: *1st Workshop on Formal Integrated Development Environment, (F-IDE)* Ed. by Catherine Dubois, Dimitra Giannakopoulou, and Dominique Méry. Vol. 149. EPTCS. 2014, pp. 79–92. https://doi.org/10.4204/EPTCS.149.8. URL: http://dx.doi.org/10.4204/EPTCS.149.8
22. S. Darabi, S.C.C. Blom, and M. Huisman. “A Verification Technique for Deterministic Parallel Programs”. In: *NASA Formal Methods (NFM)* Ed. by C. Barrett, M. Davies, and T. Kahsai. Vol. 10227. LNCS. 2017, pp. 247–264.
23. S. De Gouw et al. “OpenJDK’s java.utils.Collection.sort() is broken: The good, the bad and the worst case”. In: *Proc. 27th Intl. Conf on Computer Aided Verification (CAV), San Francisco* Ed. by D. Kroening and C. Pasareanu. Vol. 9206. LNCS. Springer, July 2015, pp. 273–289.
24. Edsger W. Dijkstra. *A Discipline of Programming* Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1976.
25. T. Dinsdale-Young et al. “Concurrent Abstract Predicates”. In: *ECOOP* Ed. by Theo D’Hondt. Vol. 6183. LNCS. Springer, 2010, pp. 504–528.
26. T. Dinsdale-Young et al. “Views: Compositional Reasoning for Concurrent Programs”. In: *POPL’13* ACM, 2013, pp. 287–300.
27. J. Dohrau et al. “Permission Inference for Array Programs”. In: *Computer Aided Verification (CAV)* LNCS. Springer, 2018.
28. Manuel Fahndrich et al. “Integrating a Set of Contract Checking Tools into Visual Studio”. In: *Proceedings of the 2012 Second International Workshop on Developing Tools as Plug-ins (TOPI)* IEEE, June 2012. URL: https://wwwmicrosoftcom/en-us/research/publication/integrating-a-set-of-contract-checking-tools-into-visual-studio/.
29. P. Ferrara and P. Müller. “Automatic inference of access permissions”. In: *Proceedings of the 13th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2012)* LNCS. Springer, 2012, pp. 202–218.
30. R. W. Floyd. “Assigning Meanings to Programs”. In: *Proceedings Symposium on Applied Mathematics* 19 (1967), pp. 19–31.
31. J.P. Galeotti et al. “Inferring Loop Invariants by Mutation, Dynamic Analysis, and Static Checking”. In: *IEEE Transactions on Software Engineering* 41 (10 2015), pp. 1019–1037.
32. Archana Ganapathi and David A. Patterson. “Crash Data Collection: A Windows Case Study.” In: *Dependable Systems and Networks (DSN)* IEEE Computer Society, Aug. 1, 2005, pp. 280–285. ISBN: 0-7695-2282-3.
33. Michiel van Genuchten and Les Hatton. “Metrics with Impact”. In: *IEEE Software* 30 (4 July 2013), pp. 99–101.
34. Jürgen Giesl et al. “Proving termination of programs automatically with AProVE”. In: *International Joint Conference on Automated Reasoning* Springer. 2014, pp. 184–191.
35. R. Hähnle and M. Huisman. “Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools”. In: *Computing and Software Science* Vol. 10000. LNCS. 2018.
36. Nir Hemed, Noam Rinetzky, and Viktor Vafeiadis. “Modular Verification of Concurrency-Aware Linearizability”. In: *Symposium on Distributed Computing (DISC)* Springer, 2015.
37. C. A. R. Hoare. “An Axiomatic Basis for Computer Programming”. In: *Communications of the ACM* 12.10 (Oct. 1969), pp. 576–580, 583. URL: http://doi.acmorg/10.1145/363235.363259.
38. Marieke Huisman. “Reasoning about Java Programs in higher order logic with PVS and Isabelle”. IPA Dissertation Series, 2001-03. University of Nijmegen, Holland, Feb 2001. URL: ftp://ftpsop.inria.fr/lemme/Marieke.Huisman/thesis.ps.gz
39. B. Jacobs and F. Piessens. *The VeriFast program verifier* Tech. rep. CW520. Katholieke Universiteit Leuven, 2008.

40. M. Janota. “Assertion-based Loop Invariant Generation”. In: *1st International Workshop on Invariant Generation (WING)* 2007.
41. Cliff B. Jones. “Tentative Steps Toward a Development Method for Interfering Programs”. In: 5.4 (1983), pp. 596–619.
42. Sebastiaan JC Joosten, René Thiemann, and Akihisa Yamada. “CeTA–Certifying Termination and Complexity Proofs in 2016”. In: *15th International Workshop on Termination* Ed. by Aart Middeldorp and René Thiemann. 2016.
43. U. Juhasz et al. *Viper: A Verification Infrastructure for Permission-Based Reasoning* Tech. rep. ETH Zurich, 2014.
44. R. Jung et al. “Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning”. In: *Principles of Programming Languages (POPL)* 2015.
45. R. Krebbers et al. “The Essence of Higher-Order Concurrent Separation Logic”. In: *ESOP* Vol. 10201. LNCS. Springer, 2017, pp. 696–723.
46. K.G. Larsen and B. Thomsen. “A modal process logic”. In: *Logic in Computer Science (LICS)* IEEE Computer Society, 1988, pp. 203–210.
47. Francesco Logozzo. “Practical verification for the working programmer with CodeContracts and Abstract Interpretation”. In: *Verification, Model Checking and Abstract Interpretation (VMCAI)* Springer, 2011.
48. A. Malkis, A. Podelski, and A. Rybalchenko. “Thread-Modular Counterexample-Guided Abstraction Refinement”. In: *Static Analysis (SAS)* Vol. 6337. LNCS. Springer, 2010.
49. Rivalino Matias et al. “An Empirical Exploratory Study on Operating System Reliability”. In: *29th Annual ACM Symposium on Applied Computing (SAC)* Gyeongju, Republic of Korea: ACM, 2014, pp. 1523–1528. ISBN: 978-1-4503-2469-4. https://doi.org/10.1145/2554850.2555021
50. Jörg Meyer and Arnd Poetzsch-Heffter. “An Architecture for Interactive Program Provers”. In: *Tools and Algorithms for Construction and Analysis of Systems, 6th International Conference TACAS 2000* Ed. by Susanne Graf and Michael I. Schwartzbach. Vol. 1785. Lecture Notes in Computer Science. Springer, 2000, pp. 63–77.
51. P. Müller, M. Schwerhoff and A.J. Summers. “Viper A Verification Infrastructure for Permission-Based Reasoning”. In: *VMCAI* 2016.
52. Aleksandar Nanevski et al. “Communicating State Transition Systems for Fine-Grained Concurrent Resources” In: *European Symposium on Programming (ESOP)* 2014, pp. 290–310.
53. P. W. O’Hearn, J. Reynolds, and H. Yang. “Local Reasoning about Programs that Alter Data Structures”. In: *Computer Science Logic* Ed. by L. Fribourg. Vol. 2142. LNCS. Paris: Springer, 2001, pp. 1–19. https://doi.org/10.1007/3540448020_1
54. P. W. O’Hearn, H. Yang, and J. C. Reynolds. “Separation and Information Hiding”. In: *Principles of Programming Languages* Venice, Italy: ACM Press, 2004, pp. 268–280.
55. Peter W. O’Hearn. “Resources, concurrency and local reasoning”. In: 375.1-3 (2007), pp. 271–307. ISSN: 0304-3975. http://dx.doi.org/10.1016/j.tcs.2006.12.035.
56. W. Oortwijn, S. Blom, and M. Huisman. “Future-based Static Analysis of Message Passing Programs”. In: *PLACES* 2016, pp. 65–72.
57. W. Oortwijn et al. “An Abstraction Technique for Describing Concurrent Program Behaviour”. In: *VSTTE* Vol. 10712. LNCS. 2017, pp. 191–209.
58. Thomas J. Ostrand and Elaine J. Weyuker. “The Distribution of Faults in a Large Industrial Software System”. In: *2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)* Roma, Italy: ACM, 2002, pp. 55–64. ISBN: 1-58113-562-9. https://doi.org/10.1145/566172.566181
59. Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell. “Where the Bugs Are”. In: *2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISTTA)*. Boston, Massachusetts, USA: ACM, 2004, pp. 86–96. ISBN: 1-58113-820-2. https://doi.org/10.1145/1007512.1007524
60. S. Owicki and D. Gries. “An Axiomatic Proof Technique for Parallel Programs”. In: *Acta Informatica Journal* 6 (1975), pp. 319–340. https://doi.org/10.1007/BF00268134

61. P. da Rocha Pinto, T. Dinsdale-Young, and P. Gardner. “Steps in Modular Specifications for Concurrent Modules”. In: *Mathematical Foundations of Programming Semantics (MFPS)*. 2015.
62. P. da Rocha Pinto, T. Dinsdale-Young, and P. Gardner. “TaDA: A Logic for Time and Data Abstraction”. In: *European Conference on Object-Oriented Programming (ECOOP)* LNCS. Springer, 2014.
63. I. Sergey, A. Nanevski, and A. Banerjee. “Specifying and Verifying Concurrent Algorithms with Histories and Subjectivity”. In: *ESOP* Vol. 9032. LNCS. Springer, 2015, pp. 333–358.
64. J. Shen. “Efficient High Performance Computing on Heterogeneous Platforms”. PhD thesis. Technical University of Delft, 2015.
65. Jan Smans, Bart Jacobs, and Frank Piessens. “VeriFast for Java: A Tutorial”. In: *Aliasing in Object-Oriented Programming* Ed. by Dave Clarke, Tobias Wrigstad, and James Noble. Vol. 7850. LNCS. Springer, 2013.
66. K. Svendsen and L. Birkedal. “Impredicative Concurrent Abstract Predicates”. In: *ESOP* Vol. 8410. LNCS. Springer, 2014, pp. 149–168.
67. V. Vafeiadis and M.J. Parkinson. “A Marriage of Rely/Guarantee and Separation Logic”. In: *CONCUR* Ed. by Luís Caires and Vasco Thudichum Vasconcelos. Vol. 4703. LNCS. Springer, 2007, pp. 256–271.
68. Viktor Vafeiadis. “Automatically Proving Linearizability”. In: *Computer Aided Verification* Ed. by Tayssir Touili, Byron Cook, and Paul Jackson. Vol. 6174. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 450–464. ISBN: 978-3-642-14294-9. https://doi.org/10.1007/978-3-642-14295-6_4. URL: http://dxdoiorg/10.1007/978-3-642-142956_40.
69. M. Zaharieva-Stojanovski. “Closer to Reliable Software: Verifying Functional Behaviour of Concurrent Programs”. PhD thesis. University of Twente, 2015. https://doi.org/10.3990/1.9789036539241.
70. J. Zeilstra. “Reasoning about Active Object Programs”. MA thesis. University of Twente, 2016.