

Subscribe to DeepL Pro to edit this document.  
Visit [www.DeepL.com/pro](https://www.deepl.com/pro?cta=edit-document) for more information.

Software is everywhere today. Whether in smartphones, cars or ATMs, it ensures that things run smoothly everywhere. It is therefore all the more important that this software works reliably. Errors can not only be annoying, but can also be dangerous in safety-critical areas.

Normally, you try to find such errors with testing by executing the programme with selected inputs and checking whether everything works as expected. But tests have their limits: they only check certain scenarios and never cover all possible inputs. A programme can therefore pass all tests and still be faulty in other cases. Especially with complex or safety-critical systems, testing is therefore often not enough to have real confidence in the software. This is where formal methods come into play. They help us to prove programmes mathematically, just as you prove a theorem in mathematics. Instead of just testing a few examples, they ensure that the programme always works correctly. No matter what inputs you choose.

This sounds complicated at first, but modern tools such as Dafny make the whole thing much more accessible. Dafny is a language that makes it possible to add conditions such as pre- and post-conditions, loop invariants and termination checks directly to programmes. This allows you to check whether the programme is correct when writing it, without even executing it. This allows many errors to be recognised and rectified at an early stage before they cause major problems later on. This can make a big difference, especially in safety-critical systems.

Unfortunately, real-life examples repeatedly show that software errors are not just theoretical. Some time ago, Tesla had to recall around two million vehicles due to software errors in the Autopilot[1](#user-content-fn-1). The system failed to recognise certain obstacles correctly, which in the worst case could have led to accidents. Another example is NASA. On the Mars Rover Spirit, a memory leak caused the system to crash repeatedly[2](#user-content-fn-2). The problem was difficult to find and almost caused the mission to fail.

Such cases show how important it is not only to test programmes, but also to check their correctness mathematically from the outset. A test might not have found the Tesla errors because it only covers a fraction of all possible scenarios. Formal methods, on the other hand, can ensure that the programme works correctly in all conceivable cases. This not only gives the developers more security, but above all the people who have to rely on these systems.

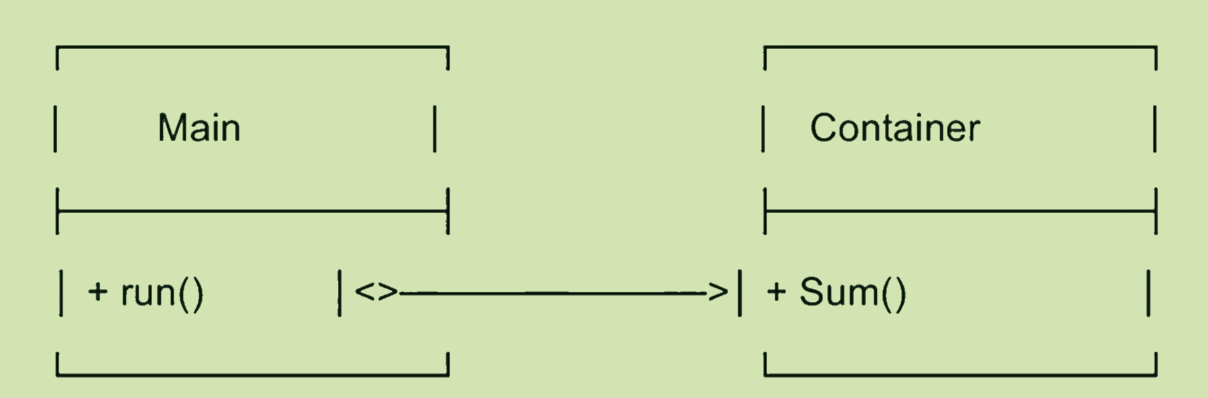
**Problem**

In practice, developing correct software is one of the biggest challenges of all. Even when writing code, developers have to make sure that no errors occur that are difficult to find later on. Particularly in safety-critical systems, where software can mean the difference between life and death, errors are not only annoying but can have serious consequences.

While tests and code reviews can help to detect errors, these methods are often not sufficient to ensure the correctness of a programme in all conceivable situations. This is because tests can only ever check a selection of inputs, but never cover all possibilities. A programme can therefore pass all tests and still be faulty.

This is precisely where formal verification comes in. The aim is to develop programmes at the writing stage in such a way that their correctness can be proven mathematically. This guarantees that programmes work correctly for all conceivable inputs.

This is where Dafny comes into play. Dafny is a modern tool that has been specially developed for the verification of programmes. It allows developers not only to write their programmes, but also to prove their correctness at the same time. This means that the software is checked for possible errors before it is executed. This facilitates the development of reliable software enormously and ensures that you don't have to worry about bugs that are difficult to trace later on.

This paper therefore examines how Dafny specifically helps to guarantee the correctness of programmes at the writing stage. To illustrate the functionality of the example programme (under DAFY/code ), a class diagram is shown in the following figure:

**Description:**

Main calls Container.Sum().

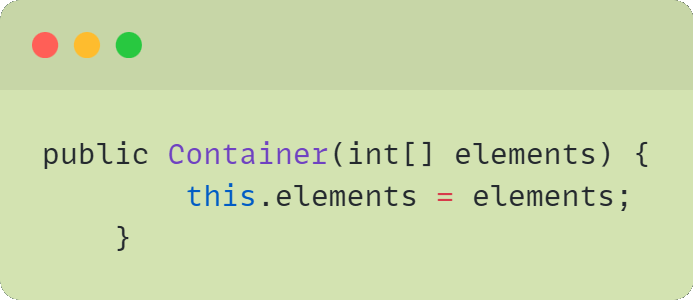
* Container is a class that has a method.
* Sum() is a method that adds Sequenze.

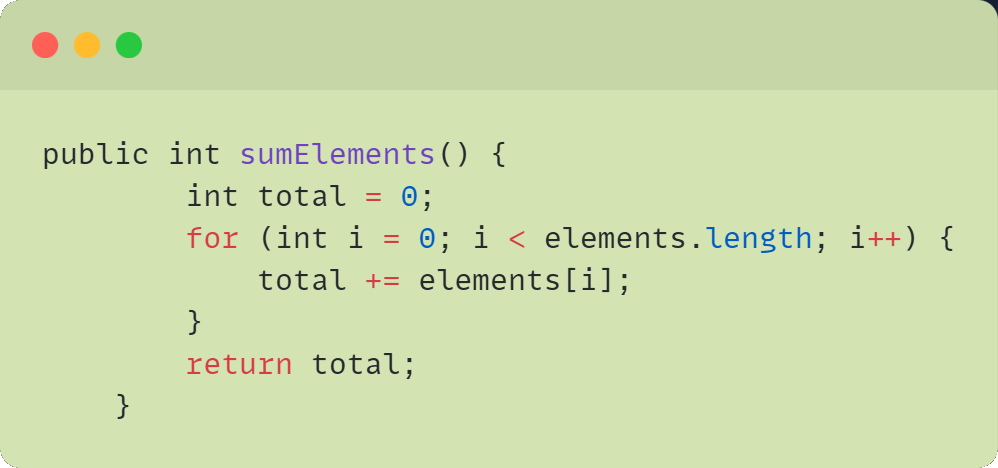
**Dafny Features:**

Dafny enhances programme development with practical functions that ensure the correctness of programmes as they are being written. These features make it easier to mathematically prove programmes and detect potential errors at an early stage:

* Pre- and postconditions:  
  They define conditions that must be fulfilled before and after the execution of a function. This precisely defines what a function may do and what it is guaranteed to return (1).
* Loop invariants:  
  They describe a condition that must be fulfilled at the beginning and after each loop pass. This can be used to mathematically prove that the loop works correctly at all times (2).
* Termination Checking:  
  This feature checks whether the programme or a loop is guaranteed to end at some point and not get stuck in an infinite loop (3).
* SMT solver (Z3):  
  A powerful tool that automatically checks compliance with all specified conditions and thus supports the verification of the code (4).

Examples:

Z1:

Z2:

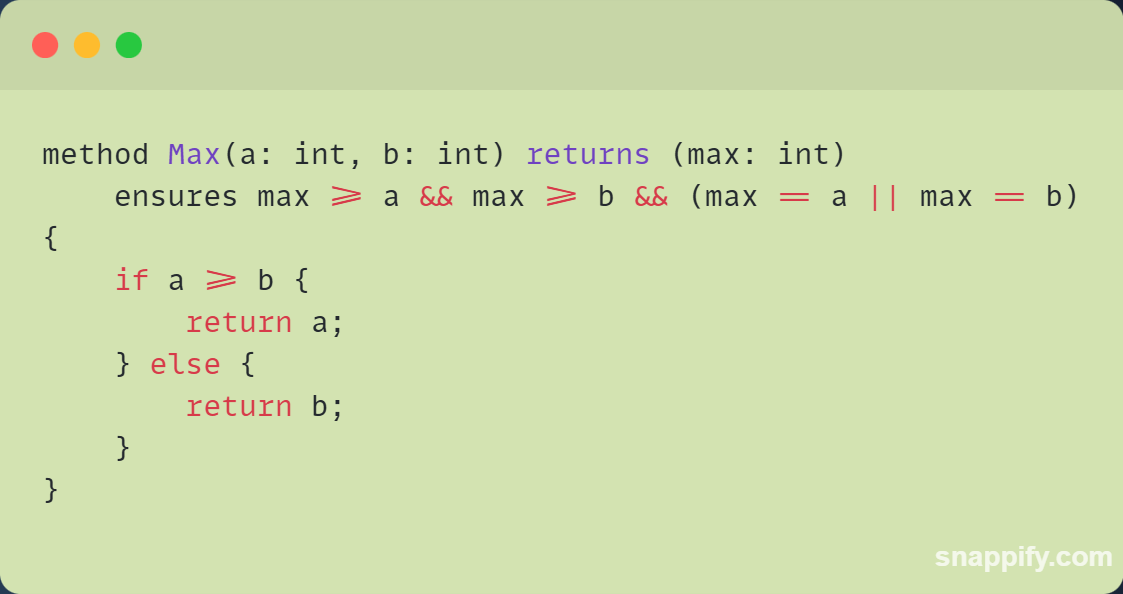
Z3:

Main Idea & Details:

In order to better understand the practical application of Dafny, the central language constructs are shown below using a simple example. Of particular importance here are the options for formulating mathematical conditions directly in the code and thus having the correctness checked automatically.

**Syntax and language constructs (code as an example)**

Dafny is based on familiar syntax patterns of modern programming languages and supplements them with special keywords for specifications. For example, a function can be written that calculates the maximum of two numbers:



This example shows how a postcondition (ensures) can be defined directly in the code so that verification takes place when the code is written.

**requires and ensures**

requires describes preconditions that must be fulfilled before a function is called. ensures defines the conditions that should apply after the function has been executed. These two keywords can be used to precisely define and check the desired behaviour of a function.

**assert and invariant**

The keyword assert is used to formulate conditions at any point in the code that must always apply. If a condition is not fulfilled, the verifier reports an error. With invariant, loop conditions can be defined that must be fulfilled before, during and after each iteration. This ensures that loops do not reach any undesired states.

**Z3 integration**

Dafny uses the SMT solver Z3 in the background to automatically check the defined conditions. This means that even complex conditions can be checked quickly and reliably without developers having to perform each proof manually.

**Own experience**

I have not yet been able to gain any personal experience with Dafny, so I cannot report any hurdles or challenges at the moment. This section serves as a placeholder and will be added to in the next submission as soon as I have worked with Dafny in practice.

**Literature Critique**

This literature review looks at the paper **"An Automatic Program Verifier for Functional Correctness"** by **K. Rustan M. Leino (2010, Microsoft Research).** Many of our statements in this paper are based on the results and findings of this paper, as it describes Dafny very comprehensively and explains it using case studies. Although there are other papers available, we found this one to be the most appropriate as it provides a good mix of practice and theory and covers many basic Dafny features in detail.

Literature criticism is important in order to understand the strengths and weaknesses of a paper and to assess the extent to which it is relevant for your own work and applications. Structured assessments help you to recognise your own research gaps and provide an objective overview of the literature used.

**was particularly easy to understand:**

* **Overview of Dafny features:** The paper explains central concepts such as pre-/postconditions, loop invariants and termination checking in an accessible way, so that even beginners gain a good overview.
* **Integration of the SMT solver Z3:** The functionality of the SMT solver is presented in a practical way and makes it easier to understand how Dafny automatically checks mathematical conditions.
* **Case studies with code and metrics:** The detailed analysis of the case studies, including lines of code and proof obligations, illustrates the practical use of Dafny very clearly.

**was difficult to access:**

* **Dynamic Frames**: This concept was only briefly mentioned and not explained in a way that is easy to grasp for beginners.
* **Proof strategies:** The presentation of the lemmas and invariants sometimes required a higher level of formal logic, which can be a hurdle for less experienced readers (like us).

So far, we have not found a better source online that picks up both beginners and advanced users so well. One possible reason for this is that other papers are often either too theoretical or too specialised, whereas this paper offers a good overview and provides practical examples. In addition, Dafny has only been used more intensively worldwide since around 2010, which is why there are not yet so many easy-to-understand papers on it.

**Future Work & Conclusion**

**Conclusion**  
This paper has shown that Dafny can be a valuable addition to classic tests as a modern verification tool. Particularly in safety-critical systems, where errors have serious consequences, Dafny makes it possible to detect errors at an early stage during writing. Dafny is not only interesting for small projects, but also offers students a practical introduction to formal verification.

**Outlook**  
In the future, the focus should be on integrating Dafny even more closely into larger teams and agile development processes. The coordination of code reviews and tests is becoming increasingly complex, especially in projects where many developers are working in parallel. With its formal specifications, Dafny can create a reliable basis here, which has often been lacking up to now. Currently, its use in large teams is still a challenge, as many companies focus on quick results and are reluctant to plan additional resources for formal verification. It is often underestimated that the costs of errors during operation or in safety-critical applications are later significantly higher than the investment in thorough verification.

Dafny is particularly relevant at a time when artificial intelligence has long been an integral part of our everyday lives. Today, systems with AI decide on processes that can have a major impact. This is precisely why it is important that the code works without errors and that the results are reliable. This is where Dafny can help to ensure the correctness of such systems - whether in industrial projects, critical systems or even software for space missions.

At the same time, you should bear in mind that Dafny also consumes resources and is not absolutely necessary for every project. Not all software is safety-critical. It is therefore up to each team to decide when a project is so critical that no errors should be allowed. This is precisely when Dafny should be used.

Another exciting prospect would be to integrate Dafny even more closely into modern DevOps environments in future, for example with Docker, in order to avoid errors in databases or complex systems.

In conclusion, it remains a challenge to convince companies, large teams and the community that the initial extra effort is an investment in quality and security. Especially today, when many companies are focussing on quick profits and cost reduction, it is often overlooked that a later error can be more expensive than solid verification. This is where Dafny can build a bridge between modern development processes and secure software - a task that the community should drive forward together.

1 See Die Presse, "Problems with Autopilot: Tesla recalls two million cars", 2025.

2 Cf. NASA Mars Exploration Rover Spirit - Technical Report (NASA JPL), 2004.