One of the requirements for software programs is the correctness. Varying techniques, such as testing, are used as an attempt to ensure the correctness of written software. However, in order to prove that the program is correct, one would have to test it with every possible input.

Doing that is not feasible for most of the practical programs. What we can do instead, is to prove that program has the desired properties for any input. Software programs are a series of machine instructions executed by a machine. Instructions and memory access in a machine are implemented using logical gates in hardware. These logical gates can be modelled as boolean formulas, which means that programs can be modelled as boolean formulas as well.

We can take advantage of this fact, and reduce the problem of proving the correctness of software programs to the boolean satisfiability problem. During the conversion of program to a boolean formula (the model) we can add constraints that model the desired properties of the program. Solving such model gives us the answer whether the program satisfies all of the constraints, and thus whether it possesses the desired properties.

Generating such models can be computed linearly in the size of the program (number of instrucitons). But since the model is a reduction of the correcntess problem to a satisfiability problem, solving the model means solving the satisfiability problem. The satisfiability problem is, however, NP-Complete. This means that solving of such models is computationally expensive.

When generating models, we can tweak different parameters in order to influence its complexity. For example, machines contain circuits for memory access. Depending on the size of each addressable memory block (memory granularity), we need to access a different number of addresses. This means that, for smaller memory blocks, more addresses are needed to access the whole memory, which results in more complicated model of memory access. Larger memory granularity is better in this regard, but worse in others. In general, extracting values from memory that differ in number of bits from the used memory granularity increases complexity. We might need to use bit-shifting and bit-masking to extract the desired value. If our program has frequent access to values in memory smaller than the configured memory granularity, then this adds more complexity for each such memory access. So memory granularity might affect the solving performance.

We can tweak parameters of models to analyze how these shifts in complexity affect the performance of solvers. In this thesis we're particularly interested in solving performance when using different combinations of memory granularity for code and non-code memory segments. We also aim for a general setup that can be used to test and benchmark models of the same program generated using different parameters.