Symbolic Fault Injection

Daniel Larsson and Reiner Hähnle

Chalmers University of Technology, Department of Computer Science and Engineering
S-412 96 Gothenburg, Sweden, {reiner,danla}@chalmers.se

Abstract. Fault tolerance mechanisms are a key ingredient of dependable systems. In particular, software-implemented hardware fault tolerance (SIHFT) is gaining in popularity, because of its cost efficiency and flexibility. Fault tolerance mechanisms are often validated using fault injection, comprising a variety of techniques for introducing faults into a system. Traditional fault injection techniques, however, lack coverage guarantees and may fail to activate enough injected faults. In this paper we present a new approach called *symbolic fault injection* which is targeted at validation of SIHFT mechanisms and is based on the concept of symbolic execution of programs. It can be seen as the extension of a formal technique for formal program verification that makes it possible to evaluate the consequences of *all* possible faults (of a certain kind) in given memory locations for *all* possible system inputs. This makes it possible to formally prove properties of fault tolerance mechanisms.

1 Introduction

One of the most common and important ways to ensure the dependability of computer systems and to analyse their fault tolerance mechanisms is *fault injection*. This includes a variety of techniques for deliberately introducing faults into a computer system and monitoring the system's behavior in the presence of these faults.

From a methodological point of view, fault injection is an experimental technique similar to *testing*: individual runs of a system are executed with input test data which in the case of fault injection is additionally instrumented with specific locations for fault injection.

During the last decade *formal methods* were increasingly used to ensure the absence of (or to detect the presence of) permanent software faults. Formal techniques such as model checking [11], extended static checking [10], and deductive verification [5] are able to find bugs or verify safety properties of industrial software. The common advantage of these methods is that they are *symbolic* and work on a logic-based representation of software properties. In consequence, one single correctness proof of a system property represents system runs for *all* admissible inputs.

Formal methods are not a replacement, but a complement of conventional software testing, because they typically work on source or bytecode and do not cover faults in machine code, compilers, or runtime environments. In order to verify the latter, testing is indispensable. Formal methods are also too expensive (or unsuitable) to cover all aspects of a system such as the user interface or I/O. For safety-critical segments of source code, on the other hand, formal verification is an increasingly cost efficient and extremely reliable alternative to testing [24, 2].

In the existing approaches to formal software verification, a program is proven to have certain properties under the assumption that no hardware faults occur (that are not detected and handled by the hardware or the operating system) during execution of the program. In other words, nothing is proven about the fault tolerance of the program. This is clearly a limitation of formal methods in the area of safety-critical systems.

The main contribution of this paper is to show that symbolic techniques such as formal software verification can be extended to symbolic analysis of fault injection and to software fault tolerance mechanisms. In contrast to conventional fault injection, this establishes the possibility to *prove* that a given fault tolerance mechanism achieves the desired behaviour for all inputs and *all modeled faults*. In particular, it is possible to guarantee that all injected faults are actually activated. Even when a fault tolerance mechanism fails to contain the injected faults and, therefore, a proof is not possible, the verification system allows to investigate the effects of the introduced faults. The method presented in this paper is applicable to *node-level* fault tolerance mechanisms, i.e., mechanisms for achieving fault tolerance withing a single node (or in an non-distributed environment).

To the best of our knowledge, this is the first presentation of a formal verification framework for software-implemented hardware fault tolerance (SIHFT). Related work is discussed in Sect. 7. We call our approach *symbolic fault injection*. It is based on *symbolic execution* of source code [8], a technique where program execution is simulated using symbolic representations rather than actual values for input data, and the effect of program execution is expressed as logical expressions over these symbols.

The central idea is to inject symbolic faults (representing whole classes of concrete faults) during symbolic execution which then reflects the consequences of the injected faults. This has been prototypically implemented and evaluated in a tool for formal verification of (JAVA) software, the KeY [1, 5] tool.

The paper is organized as follows: in the following section we review SIHFT, our main target application. We discuss our fault model in Sect. 3. Readers unfamiliar with formal verification find the necessary background in Sect. 4. The core of the paper is Sect. 5, where we explain how symbolic fault injection is modeled and implemented in the logic of the verification system. In Sect. 6 we present a case study showing the potential of symbolic fault injection. We close with related work, a discussion of the achieved results and future work.

2 SIHFT

It is impossible to guarantee that a given computer system is free of faults. Even using the best available techniques for manufacturing hardware components, the best available processes for the design of the hardware and the software, and the best available techniques for testing a system, it may still contain defects. Moreover, it is impossible to guarantee that no transient faults occur during operation of the computer system. Therefore, in order to construct dependable computer systems we need to equip them with mechanisms for detecting and recovering from faults. Faults can be classified into hardware and software faults. An orthogonal classification divides faults into transient, intermittent, and permanent faults. In this paper the focus is on *transient hardware faults*, specifically, bit-flips in data memory locations.

Fault tolerance mechanisms can be based on hardware (for example, redundant components) or on software. From a cost perspective it is often beneficial to use software-implemented fault tolerance whenever possible, because (i) commercial, standardized components can be used; (ii) hardware redundancy can be avoided; and (iii) high flexibility can be obtained.

The scenario, where mechanisms for handling hardware faults are implemented in software, is called SIHFT (Software-Implemented Hardware Fault Tolerance) (for example, [7]). Common SIHFT techniques include assertions, algorithm-based fault tolerance (ABFT), control-flow checking, and data duplication and comparison. Other examples are checksum algorithms like CRC (Cyclic Redundancy Check). We apply our symbolic technique to the latter in Sect. 6 below.

Our method for formal verification in the presence of faults operates on the source code level of high-level programming languages and hence is restricted to software mechanisms. The type of faults we tried to emulate so far are transient hardware faults, specifically, bit-flips in the data area of memory (this is not an inherent limitation: other kinds of faults could be modelled, see Sect. 8). SIHFT is a natural target application for our method. The fact that hardware-implemented fault detection mechanisms rarely detect faults in the data area of the memory [4] further motivates the choice of our fault model which is described in detail in the following section.

3 Fault Injection and Fault Model

The purpose of using fault injection is to provoke the occurrence of errors in a system in order to validate the system's dependability. Errors occur too infrequently during normal operation of a computer system to be able to perform such a validation within reasonable time.

Existing fault injection approaches can be classified into hardware-implemented and software-implemented fault injection (SWIFI). Examples of the first are techniques, where integrated circuits are exposed to heavy-ion radiation [16] or electromagnetic interference [15], and the injection of faults directly on the pins of an integrated circuit [3]. Software-implemented fault injection can be further classified into prototype-based [9] and simulation-based [14] fault injection. In the first case the actual computer system to be validated (or a prototype thereof) is running while faults are introduced into the system through software. In the second case a simulation of the system is used when the faults are introduced.

A fault injection approach is based on a *fault model* which specifies the exact kind of faults to be injected or emulated. In many fault injection approaches/tools only single bit-flips are used since they are considered to be efficient in revealing dependability weaknesses.

The next question to consider is *where* and *when* the faults are to be injected. For the purpose of evaluating the relative effectiveness of fault tolerance mechanisms on different levels (hardware level, operating system level, and application level), it is useful to be able to inject faults, with high precision, in specific parts of the hardware. For example, faults might be injected into the MMU (Memory Management Unit) to evaluate a system's robustness against this kind of faults. When one is mainly interested in evaluating the mechanisms on the application level, it is often sufficient to inject faults in memory. It is also useful to be able to control when faults are injected, i.e., how the faults are *triggered*. Fault injections can be related to a certain instruction being executed or a memory location being manipulated, or a fault can be injected after a specified time.

The major weakness of conventional fault injection techniques is their lack of coverage. For example, to evaluate the effect of a fault in a given memory location, typically one bit or a few bits are flipped. But there is no guarantee that these particular faults will actually exhibit any defects present in the fault handling hardware or software. In other words, using fault injection nothing is *proved* about the fault tolerance property of a system. Similar to ordinary testing, fault injection can only show the presence of defects, not their absence.

Conventional fault injection techniques also suffer from other problems. Hardware-implemented techniques require special hardware which is very difficult—sometimes even impossible—to design for modern processors [9]. These techniques are also not easily ported to other platforms or expanded to new classes of faults. In the case of techniques using heavy-ion radiation or electromagnetic interference it is difficult to exactly trigger the time and location of a fault injection [13]. One source of problems with existing SWIFI tools is the target system monitoring for detecting the activation of faults and for investigating the exact effects of the faults [9]. Software solutions for monitoring have an undesired impact on the target system behavior. Moreover, the analysis of the huge amounts of monitor data is both difficult and time-consuming. Another problem with existing SWIFI techniques is that a large proportion of the injected faults are not activated, for example, faults injected into unused memory locations or faults placed in registers before the registers are written to [4].

The approach presented in this paper can be characterized as software-implemented, simulation-based fault injection. In the experiments performed, the simulation consists of the machinery available in the KeY tool for performing symbolic program execution. The fault model so far consists of bit-flips. There are no inherent restrictions on the types of faults we can emulate; if a certain hardware part is explicitly simulated as part of the verification, it is possible to emulate the effects of faults in that part. It would also be quite easy to emulate software faults. Our approach works on the source code level. We emulate transient bit-level faults in the data segment of memory by manipulating the variables in the program, and we relate fault injection to pseudo-statements instrumented into the source code and triggered during symbolic execution.

4 Formal Methods

Formal methods comprise a wide range of techniques including black box approaches such as specification-only or specification-based testcase generation. Here we concentrate on *formal verification* of software. Among the various approaches to formal software verification [1, 10, 11] we single out verification by symbolic program execution [8], because of its compatibility with the analysis of propagation of injected faults through a program.

Our implementation platform is the formal software verification tool KeY [1, 5]. In its current version it can handle most of sequential Java and there is ongoing work to deal with concurrency [18] and for support of the C language. KeY takes as input a Java program (source code) and a formal specification of that program. The combination of the program and the specification is combined into a *proof obligation* expressed in Java Dynamic Logic (JavaDL). JavaDL is a typed first-order logic (FOL) extended with a dynamic part that can handle Java programs.

The idea of verification by symbolic program execution is to use logic in order to represent all possible values of locations in a program and to track their value updates during execution. We illustrate the main ideas by an example.

```
public class C {
  static int a,b;

  public static void swap()
  {
    b = a - b;
    a = a - b;
    b = a + b;
  };
}
```

Fig. 1. The `swap()` method.

The `swap()` method in Fig. 1 exchanges the values of the fields `a` and `b` of class `C` without the need for a temporary variable. Symbolic execution of the method would start by assigning symbolic integer values i and j to fields `C.a` and `C.b`, respectively. Since we want to analyse `swap()` for arbitrary values i and j we quantify universally over them. A total correctness assertion in the program logic used in the KeY system [1] looks then as follows:

$$\forall\,\mathbf{int}\ i;\ \forall\,\mathbf{int}\ j;\ (\{\texttt{C.a} := i\}\{\texttt{C.b} := j\}\\ \langle\texttt{C.swap();}\rangle(\texttt{C.a} \doteq j \,\&\, \texttt{C.b} \doteq i)) \tag{1}$$

The universal quantifiers range over integer variables i and j that are assigned to the fields `C.a` and `C.b` as symbolic initial values. Variables i and j are so-called *rigid* variables whose value cannot be changed during the execution of a program (roughly corresponding to **final** locations in JAVA). For a compilable JAVA program `p`, a formula of the form "$\langle\texttt{p}\rangle$post" expresses that every run of `p` with the current initial values terminates normally and afterwards the postcondition "post" is true. In other words, `p` is *totally correct* with respect to the given postcondition. If one is merely interested in *partial correctness*, the $[\,]$-operator can be used instead: "$[\texttt{p}]$post" expresses that *if* `p` terminates normally *then* "post" will be true in the end state. In formula (1) the postcondition expresses that the initial values of the fields `C.a` and `C.b` have been swapped by stating that the value of `C.a` now is equal to initial value j and `C.b` is equal to i (we use the symbol $\doteq$ to distinguish between equality in formulas and assignment statements). In this way it is possible to formally specify the functionality of a given method.

The translation into a logical framework makes it possible to reason formally about a program. A universally quantified formula such as (1) is valid if and only if the formula

$$\{\texttt{C.a} := i\}\{\texttt{C.b} := j\}\langle\texttt{C.swap();}\rangle(\texttt{C.a} \doteq j \,\&\, \texttt{C.b} \doteq i) \tag{2}$$

is true for any possible interpretation of i and j. The expressions in curly brackets are called state *update*. Let $\mathcal{U} = \{\mathrm{loc} := \mathrm{val}\}$ be such an update, where loc is a location (program variable, field or array access) and val is a side-effect free expression. The semantics of an updated formula $\mathcal{U}\phi$ is to change the environment relative to which ϕ is evaluated in such a way that the value of loc becomes val and everything else is unchanged. Hence, the meaning of formula (2) is: whenever `swap()` is started in an initial state where `C.a` has value i and `C.b` has value j, then `swap()` terminates normally and afterwards the contents of the fields `C.a` and `C.b` is swapped.

The logic JAVADL used in the KeY system provides symbolic execution rules for any formula of the form "$\mathcal{U}\langle\xi;\omega\rangle\mathrm{post}$", where ξ is a single JAVA statement and ω the remaining program. ξ is called the first *active statement* of the program, i.e., the statement the rule operates on. JAVADL rules such as (3) can be seen as an operational semantics of the JAVA language. Application of rules can then be thought of as symbolic code execution. A program is verified by executing its code symbolically and then checking that the FOL conditions after execution is finished are valid. During proof search rules are applied from bottom to top. From the old proof obligation (conclusion), new proof obligations are derived (premisses).

We give some examples of JAVADL symbolic execution rules. Updates are used to record the effect of assignment statements during symbolic execution:

$$\frac{\vdash \{\mathtt{v} := \mathtt{e}\}\langle\omega\rangle\phi}{\vdash \langle\mathtt{v\ =\ e;}\ \omega\rangle\phi} \tag{3}$$

The symbol $\vdash$ stands for derivability. The rule says that in order to derive the formula in the *conclusion* (on bottom) it is sufficient to derive the formula in the single *premiss* (on top). The idea is to simply replace an assignment with a state update. This rule can only be used if `e` is a side-effect free JAVA expression. Otherwise, other rules have to be applied first to evaluate `e` and the resulting state changes must be added to the update.

The effect of an update is not computed until a program has been completely (symbolically) executed. For example, after expanding the method body of `C.swap()` and symbolic execution of the first two statements we obtain the following intermediate result:

$$\vdash \{\mathtt{C.a} := j\}\{\mathtt{C.b} := i - j\}\langle\mathtt{method\text{-}frame(C()):\ b\ =\ a\ +\ b;}\rangle$$
$$(\mathtt{C.a} \doteq j \,\&\, \mathtt{C.b} \doteq i)$$

During method expansion a *method frame*, which records the receiver of the invocation result and marks the boundaries of the inlined implementation, was created. The updates of `C.a` and `C.b` reflect the assignment statements that have been executed already. After executing the last statement and returning from the method call the code has been fully executed. The subgoal reached at this point is similar to $\vdash \{\mathtt{C.a} := j\}\{\mathtt{C.b} := i\}\langle\,\rangle(\mathtt{C.a} \doteq j \,\&\, \mathtt{C.b} \doteq i)$, where the updates are followed by the empty program. Only now updates are applied to the postcondition which results in the trivial subgoal $\vdash j \doteq j \,\&\, i \doteq i$.

Below is another rule example, namely the rule for the **if - else** statement which has two premisses. The rule is slightly simplified.

$$\frac{\mathtt{b} \doteq \mathit{TRUE} \vdash \langle\mathtt{p}\,\omega\rangle\phi \qquad\qquad \mathtt{b} \doteq \mathit{FALSE} \vdash \langle\mathtt{q}\,\omega\rangle\phi}{\vdash \langle\mathbf{if}\ \ \mathtt{(b)\ p}\ \ \mathbf{else}\ \ \mathtt{q;}\ \omega\rangle\phi}$$

This rule shows that in contrast to normal program execution, in symbolic execution even of sequential programs it is sometimes necessary to branch the execution path. This happens whenever it is impossible to determine the value of an expression that has an influence on the control flow. This is the case for conditionals, switch statements, and polymorphic method calls, among others. The rule above is applicable if `b` is an expression without side effects, otherwise other rules need to be applied first.

A problem occurs with loops and recursive method calls. If the loop bound is finite and known, then one can simply unwind the loop a suitable number of times. But in general one needs to apply an induction argument or an invariant rule to prove properties about programs that contain unbounded loops. Both approaches tend to be expensive, because they require human interaction. The automation of induction proofs for imperative programs is an area of active research [26].

5 Symbolic Fault Analysis

5.1 General Idea

Our plan is to extend the approach to formal verification of software sketched in the previous section with the concept of symbolic fault injection. This makes it possible to prove that a program with software-based fault tolerance mechanisms ensures certain properties even in the presence of faults. Alternatively, one may calculate the consequences of the introduced faults in terms of strongest postconditions. The realization is based on the following two ideas:

- The source code is instrumented with pseudo-instructions of the form "`inject( location);`" that are placed where the faults are to be injected. The argument `location` is the name of a memory location (local variable, field access, formal parameter, etc.) visible at this point in the program. This makes it possible to handle (symbolic) fault injection *uniformly* by symbolic code execution.
- Symbolic fault injection is realized by extending the symbolic execution mechanism with suitable rules for the `inject` pseudo-instructions.

The examples given below are in JAVA since the current version of KeY handles JAVA, but the principles given hold for any imperative language.

An injection of a symbolic fault causes a change in the JavaDL representation of the symbolic program state, and this state change corresponds to the consequences of *all* the concrete faults that can appear during program execution and that are instances of the symbolic fault.

Assume that we want to emulate the effect of *all possible* bit-flips in the memory location that corresponds to a given variable. First we need to clarify what is meant by "all possible" bit-flips. Is it the effect of all possible *single* bit-flips or all possible combinations of an arbitrary number of bit-flips (in the same memory location)? Considering a JAVA **int** (represented by 32 bits): there are 32 different possible outcomes in the first case, but 2^{32} in the second. Obviously, when trying to prove properties about algorithms that can detect bit-flips, it is essential to distinguish between single bit-flips (or, perhaps, a fixed, small number) and an arbitrary number of bit-flips. For example, the CRC algorithm discussed in Sect. 6 can detect situations where one or a few bits are flipped. Trying to prove the fault detection capability of such an algorithm using the "arbitrary number of bit-flips" semantics of the `inject` statement will not succeed. However, in other situations it might be desirable and possible to prove properties for an arbitrary number of bit-flips. Our solution is to use two different inject statements: `inject(location)` means that an arbitrary number of bits in the memory location will be flipped, while `inject1(location)` means that a single bit is flipped. To model a situation where a fixed number n of bits in a location is flipped, `inject1` is simply applied at that location n times.

Another important question is whether the "no change" case is included in the meaning of the `inject`/`inject1` statements, i.e. whether the property we are trying to prove should also hold for the case where no bits are flipped. As will become apparent in Sect. 6, sometimes a semantics *not* including the "no change" case is needed. Below we introduce different flavours of rules for handling the `inject` /`inject1` statements covering both cases: one including the "no change" case and the other one excluding it.

5.2 Rules

We need to add new rules to the JAVADL calculus that handle the `inject` pseudo-instructions. The rules for the cases when the `location` argument of `inject` has type **boolean** or **byte** are below. In the case of **boolean** typed variables there is no need to distinguish between single bit-flips and an arbitrary number of bit-flips as they hold only one bit, however, the distinction between inclusion and exclusion of the "no change" case is relevant, and the two rules are presented below (inclusion of the "no change" case is indicated by appending *NC* to the rule name).

$$\text{booleanNC}\ \frac{\vdash \{\mathtt{b} := \textbf{true}\}\langle\omega\rangle\phi \quad \vdash \{\mathtt{b} := \textbf{false}\}\langle\omega\rangle\phi}{\vdash \langle\mathtt{inject(b);}\omega\rangle\phi}$$

$$\text{boolean}\ \frac{\vdash \{\mathtt{b} := \mathtt{!b}\}\langle\omega\rangle\phi}{\vdash \langle\mathtt{inject(b);}\omega\rangle\phi} \tag{4}$$

The first rule splits symbolic execution into two paths, where in exactly one of them `b` is unchanged and in the other it is complemented. The second rule continues symbolic execution with the value of `b` complemented. Next we show the rule for an arbitrary number of bit-flips in a **byte** variable. Only the "no change" version is shown.

$$\text{byteNC}\ \frac{\vdash \forall\, \textbf{byte}\ i; \{\mathtt{b} := i\}\langle\omega\rangle\phi}{\vdash \langle\mathtt{inject(b);}\omega\rangle\phi} \tag{5}$$

In this case the memory location can contain any **byte** value after the injection. This means that whatever program property that should be proved has to be proved for all values of this variable. In logical terms it means that a universal quantification has to be introduced. We do this by quantifying over a new logical variable i followed by an update that assigns the value of i to the location `b`.

Finally, the rules for the `inject1` statement on **byte**s and arrays of **byte**s are presented. Only the rules excluding the "no change" case is shown.

$$\text{byte1}\ \frac{\vdash \forall\, \textbf{int}\ j;\ 0 \leq j \leq 7 \rightarrow \{\mathtt{b} := \mathtt{b}\,\hat{}\,(1 \ll j)\}\langle\omega\rangle\phi}{\vdash \langle\mathtt{inject1(b);}\ \omega\rangle\phi} \tag{6}$$

After injection, the memory location can contain any value resulting from flipping exactly one bit in `b`. The intuition behind the rule is that the variable is xored with the masks, 00000001, 00000010,..., 10000000 respectively. The rule for arrays of **byte**s,

$$\text{byteArr1}\ \frac{\vdash \forall\,\mathsf{int}\,i;\ \forall\,\mathsf{int}\,j;\ 0 \leq i < \mathsf{a.length}\ \&\ 0 \leq j \leq 7 \rightarrow \{\mathsf{a[i]} := \mathsf{a[i]}\,\hat{}\,(1 \ll j)\}\langle\omega\rangle\phi}{\vdash \langle \mathsf{inject1(a)};\, \omega\rangle\phi}$$

Fig. 2. The inject1 rule for byte arrays.

pictured in Fig. 2, is similar but includes universal quantification over the array elements. The rule shown is a bit simplified since the real rule has to take the possibility of a **null** reference into account. We created analogous rules for the other primitive JAVA types, which are not presented here.

5.3 Example: Verification

We proceed to show by example how the rules for the pseudo-instruction `inject` are used in practice. The examples are based on a simple JAVA class shown below.

```
class MyBoolean {
    boolean v;
    boolean myOr(boolean b) {
        boolean t=b;
        inject(t);
        return t||v;
    }
}
```

`MyBoolean` can be viewed as a wrapper for **boolean** primitive valuess. It contains a **boolean** field `v` that holds the value of a `MyBoolean` instance. It also has a method, `myOr`, with obvious meaning. (The temporary variable `t` is unnecessary for the behavior of `myOr`. It is added for the presentations of the proofs below, as it makes it possible to refer to the original value of the parameter `b` in an easy way.) The interesting point is the `inject` statement that injects a fault into the **boolean** argument before the return value is computed. Attempts to prove that `myOr` has certain correctness properties even in the presence of faults are shown below.

Symbolic execution and first-order logic reasoning as implemented in KeY is used in the proof attempts. Note the use of rule (4) for `inject(t)` (marked with an asterisk on the right in Fig. 3 and Fig. 4). In the first example, shown in Fig. 3, an attempt is made to prove that the method still has the semantics expected from logical or (the variable `result` in the proof stands for the return value of `myOr`): the postcondition states that the return value of `myOr` is true if and only if one of the arguments is true. This is impossible to prove due to the injected fault. We get four different branches in the proof, one for each combination of values in field `v` and parameter `b`. All branches must be proven in order to show the property. Due to space restrictions we only show one of the branches that are impossible to prove, indicated by Γ in the antecedent which abbreviates "$\mathtt{v} \doteq \textbf{false}\ \&\ \mathtt{b} \doteq \textbf{true}$". The proof tree is shown in Fig. 3. As expected, we end up with a sequent which is impossible to prove valid.

$$\begin{array}{c}
\Gamma \vdash false \\ \hline
\Gamma \vdash false \leftrightarrow true \\ \hline
\Gamma \vdash \mathsf{false} \doteq \mathsf{true} \leftrightarrow (\mathsf{false} \doteq \mathsf{true} \lor \mathsf{true} \doteq \mathsf{true}) \\ \hline
\Gamma \vdash \mathsf{false} \doteq \mathsf{true} \leftrightarrow (\mathsf{v} \doteq \mathsf{true} \lor \mathsf{b} \doteq \mathsf{true}) \\ \hline
\Gamma \vdash \{\mathrm{result}{:=}\ \mathsf{false}\}\langle\rangle(\mathrm{result} \doteq \mathsf{true} \leftrightarrow (\mathsf{v} \doteq \mathsf{true} \lor \mathsf{b} \doteq \mathsf{true})) \\ \hline
\Gamma \vdash \{\mathrm{result}{:=}\ \mathsf{v}\}\langle\rangle(\mathrm{result} \doteq \mathsf{true} \leftrightarrow (\mathsf{v} \doteq \mathsf{true} \lor \mathsf{b} \doteq \mathsf{true})) \\ \hline
\Gamma \vdash \langle\mathsf{return\ v;}\rangle(\mathrm{result} \doteq \mathsf{true} \leftrightarrow (\mathsf{v} \doteq \mathsf{true} \lor \mathsf{b} \doteq \mathsf{true})) \\ \hline
\Gamma \vdash \{\mathsf{t} := \mathsf{false}\}\langle\mathsf{return\ t\ ?\ true : v;}\rangle(\mathrm{result} \doteq \mathsf{true} \leftrightarrow (\mathsf{v} \doteq \mathsf{true} \lor \mathsf{b} \doteq \mathsf{true})) \\ \hline
\Gamma \vdash \{\mathsf{t} := \mathsf{false}\}\langle\mathsf{return\ t\ \|\ v;}\rangle(\mathrm{result} \doteq \mathsf{true} \leftrightarrow (\mathsf{v} \doteq \mathsf{true} \lor \mathsf{b} \doteq \mathsf{true})) \\ \hline
\Gamma \vdash \{\mathsf{t} := \mathsf{true}\}\langle\mathsf{inject(t);\ return\ t\ \|\ v;}\rangle(\mathrm{result} \doteq \mathsf{true} \leftrightarrow (\mathsf{v} \doteq \mathsf{true} \lor \mathsf{b} \doteq \mathsf{true})) \quad * \\ \hline
\Gamma \vdash \{\mathsf{t} := \mathsf{b}\}\langle\mathsf{inject(t);\ return\ t\ \|\ v;}\rangle(\mathrm{result} \doteq \mathsf{true} \leftrightarrow (\mathsf{v} \doteq \mathsf{true} \lor \mathsf{b} \doteq \mathsf{true})) \\ \hline
\Gamma \vdash \langle\mathsf{boolean\ t{=}b;\ inject(t);\ return\ t\ \|\ v;}\rangle(\mathrm{result} \doteq \mathsf{true} \leftrightarrow (\mathsf{v} \doteq \mathsf{true} \lor \mathsf{b} \doteq \mathsf{true}))
\end{array}$$

Fig. 3. Failed proof attempt: correctness property of MyBoolean::myOr(). One of four branches in the proof: Γ stands for "$\mathsf{v} \doteq \mathsf{false} \,\&\, \mathsf{b} \doteq \mathsf{true}$".

Now consider an attempt to prove something weaker, namely that `myOr()` returns **true** whenever the field v was **true**. Again, only one of the four proof branches is shown, but all are provable. We use Γ in the same way as above. The proof tree is in Fig. 4. We end up with a sequent that is valid indicating provability. We showed the formal proofs in some detail in order to give an impression how symbolic execution of code and injected faults works. All proofs, respectively, proof attempts are created by the KeY prover within fractions of a second and fully automatically.

$$\begin{array}{c}
[\Gamma \vdash true] \\ \hline
\Gamma \vdash false \rightarrow false \\ \hline
\Gamma \vdash \mathsf{false} \doteq \mathsf{true} \rightarrow \mathsf{false} \doteq \mathsf{true} \\ \hline
\Gamma \vdash \mathsf{v} \doteq \mathsf{true} \rightarrow \mathsf{false} \doteq \mathsf{true} \\ \hline
\Gamma \vdash \{\mathrm{result}{:=}\ \mathsf{false}\}\langle\rangle(\mathsf{v} \doteq \mathsf{true} \rightarrow \mathrm{result} \doteq \mathsf{true}) \\ \hline
\Gamma \vdash \{\mathrm{result}{:=}\ \mathsf{v}\}\langle\rangle(\mathsf{v} \doteq \mathsf{true} \rightarrow \mathrm{result} \doteq \mathsf{true}) \\ \hline
\Gamma \vdash \langle\mathsf{return\ v;}\rangle(\mathsf{v} \doteq \mathsf{true} \rightarrow \mathrm{result} \doteq \mathsf{true}) \\ \hline
\Gamma \vdash \{\mathsf{t} := \mathsf{false}\}\langle\mathsf{return\ t\ ?\ true : v;}\rangle(\mathsf{v} \doteq \mathsf{true} \rightarrow \mathrm{result} \doteq \mathsf{true}) \\ \hline
\Gamma \vdash \{\mathsf{t} := \mathsf{false}\}\langle\mathsf{return\ t\ \|\ v;}\rangle(\mathsf{v} \doteq \mathsf{true} \rightarrow \mathrm{result} \doteq \mathsf{true}) \\ \hline
\Gamma \vdash \{\mathsf{t} := \mathsf{true}\}\langle\mathsf{inject(t);\ return\ t\ \|\ v;}\rangle(\mathsf{v} \doteq \mathsf{true} \rightarrow \mathrm{result} \doteq \mathsf{true}) \quad * \\ \hline
\Gamma \vdash \{\mathsf{t} := \mathsf{b}\}\langle\mathsf{inject(t);\ return\ t\ \|\ v;}\rangle(\mathsf{v} \doteq \mathsf{true} \rightarrow \mathrm{result} \doteq \mathsf{true}) \\ \hline
\Gamma \vdash \langle\mathsf{boolean\ t{=}b;\ inject(t);\ return\ t\ \|\ v;}\rangle(\mathsf{v} \doteq \mathsf{true} \rightarrow \mathrm{result} \doteq \mathsf{true})
\end{array}$$

Fig. 4. Successful proof: weakened correctness property of MyBoolean::myOr(). One of four branches in the proof: Γ stands for "$\mathsf{v} \doteq \mathsf{false} \,\&\, \mathsf{b} \doteq \mathsf{true}$".

5.4 Example: Calculating Strongest Postcondition

Besides proving that a program has certain properties in the presence of faults it is possible to calculate the consequences of a fault in terms of strongest postconditions. Below is a simple program containing an `inject` statement for which we calculate the strongest postcondition.

```
int aMethod() {
   int i = 0;
   inject(i);
   return i;
}
```

The calculation of the strongest postcondition of the program is shown below. Note that an `inject` rule for **int** type variables similar to (5) is used.

$$\begin{array}{c}
\vdash \exists\, \mathbf{int}\ k; result \doteq k \\ \hline
\vdash \forall\, \mathbf{int}\ j; \{result{:=}j\}\langle\rangle ? \\ \hline
\vdash \forall\, \mathbf{int}\ j; \{\mathrm{i}{:=}j\}\{result{:=}\mathrm{i}\}\langle\rangle ? \\ \hline
\vdash \forall\, \mathbf{int}\ j; \{\mathrm{i}{:=}j\}\langle \mathbf{return}\ \texttt{i;}\rangle ? \\ \hline
\vdash \forall\, \mathbf{int}\ j; \{\mathrm{i}{:=}0\}\{\mathrm{i}{:=}j\}\langle \mathbf{return}\ \texttt{i;}\rangle ? \\ \hline
\vdash \{\mathrm{i}{:=}0\}\langle \texttt{inject(i);}\ \mathbf{return}\ \texttt{i;}\rangle ? \\ \hline
\vdash \langle \mathbf{int}\ \texttt{i=0; inject(i);}\ \mathbf{return}\ \texttt{i;}\rangle ?
\end{array}$$

The symbolic execution tells us that after a fault injection in variable `i` the return value can be any **int** value. The example is trivial and not very interesting in itself but illustrates the idea: the symbolic execution makes it possible to analyse the consequences of faults for all admissible inputs.

6 Case Study

We illustrate the application of symbolic fault injection to a realistic fault handling mechanism: an implementation of the widely used CRC (Cyclic Redundancy Check) algorithm. CRC is a fault detection algorithm: it calculates a checksum on a block of data. This checksum is typically appended to the data block before it is transmitted and the receiver is then able to determine whether the data has been corrupted. The basic idea behind CRC is to treat the block of data as a binary representation of an integer and then to divide this integer with a predetermined divisor. The remainder of the division becomes the checksum. The kind of division used is not the one found in standard arithmetic but in so-called polynomial arithmetic. The property that makes CRC so useful is that it minimizes the possibility that several bit-flips “even out” with respect to the checksum and therefore go undetected. The algorithm fully utilizes the number of bits used to represent the checksum. By choosing the divisor (also called *poly*) carefully, the algorithm can detect all single bit-flips, all two-bit errors (up to a certain size of the block of data), all errors where an odd number of bits are flipped, and so-called burst errors (where a number of adjacent bits are flipped) up to a certain number of bits depending on the size of the divisor.[1]

We describe briefly the implementation of the algorithm. We cannot use JAVA's built-in division operation, because the block of data, viewed as an integer, in general is far too big to store in a register; also, we need to use polynomial arithmetic. Therefore, the data is fed step by step to a division register while the required operations are applied to its content. In its simplest and least efficient implementation of the algorithm the data is shifted bit by bit, while the most commonly used implementation shifts the data one "register length" at a time and uses a lookup table. Below is an example of a table-driven implementation in JAVA generated by the "CRC generator".[2] The block of data is here represented by an array of **byte**s, which is given as parameter `buf` to the program. The method returns the computed CRC value.

```
static byte compute(byte[] buf) {
    int count = buf.length;
    byte reg = (byte)0x0;
    while (count > 0) {
        byte elem = buf[buf.length-count];
        int t = ((int)(reg^elem)&0xff);
        reg <<= 8;
        reg ^= table[t];
        count--;
    }
    return reg;
}
```

The array `table` in the program above refers to an array of 256 precomputed **byte**s that allows to perform the division, shifting the block of data one **byte** at the time (instead of one *bit* at the time). It would be useful to prove formally that this method has certain properties. Even though the theory behind the CRC algorithm is well known, there is no guarantee that this particular *implementation* of the algorithm is free from errors, in particular, since concrete algorithms are synthesized by a program generator based on several parameters.

In the following we document an attempt to formally prove that the implementation above detects all single bit-flips. Detecting single bit-flips is something we expect even the most simple checksum algorithms to manage, but nevertheless it is valuable to formally prove that a given CRC implementation actually does this. More precisely, the following should be proved. Assume one arbitrary **byte** array of arbitrary length. This array is duplicated, an arbitrary single bit-flip in *one* of the arrays is performed, and then CRC checksums for both arrays are computed. The two checksums should

[1] For the algorithm and possible implementations see http://www.repairfaq.org/filipg/LINK/F_crc_v3.html.

[2] http://members.cox.net/tonedef71/body_jcrcgen.htm#output

differ and the first step to prove this property is to create the test harness below. It is a modified version of the CRC implementation with the following changes.

- All variables (except `count`), including the **byte** array constituting the input to the algorithm, are duplicated. All statements acting on these variables are also duplicated.
- An `inject1` statement (described in Sect. 5.2) is added that injects a bit-flip fault in one of the input arrays.
- Instead of returning the CRC checksum, this modified version returns the comparison of the two computed CRC checksums in form of a **boolean** value. That is, the method returns **true** if the two checksums are equal (the fault is *not* detected) and **false** otherwise.

```
static byte crcTest(byte[] buf1, byte[] buf2) {
    inject1(buf2);
    int count = buf1.length;
    byte reg1 = (byte)0x0;
    byte reg2 = (byte)0x0;
    while (count > 0) {
        byte e1 = buf1[buf1.length-count];
        byte e2 = buf2[buf2.length-count];
        int t1 = ((int)(reg1^e1)&0xff);
        int t2 = ((int)(reg2^e2)&0xff);
        reg1 <<= 8;
        reg2 <<= 8;
        reg1 ^= table[t1];
        reg2 ^= table[t2];
        count--;
    }
    return (reg1 == reg2);;
}
```

The reason for modifying the original program is to facilitate the proving process. It could be argued that this modification might change the behavior of the original program in an unintended way, e.g., that the checksum calculated for the non-faulty array is not equal to the checksum calculated for the same array using the original program. For this program, however, it is fairly easy to see that this is not the case. In case of doubt, it is possible to formally prove this.

The next step is to express formally the property this program should have. The proof obligation expressed in JAVADL is presented below (7). The variable *result* stands for the return value of the method. For sake of clarity some parts dealing with potential `NullPointerException`s and similar are omitted. The variables $msg1lv$ and $msg2lv$ are used to quantify over all possible values of the input message blocks `msg1` and `msg2`. The precondition states that (the reference variables) `msg1` and `msg2` do not refer to the same array, but that the arrays are identical. Note that the [ ]-operator is used, i.e., proving termination is not part of the proof obligation (see Sect. 4). The reason is that this makes it possible to apply a loop invariant rule; see discussion below. Termination has been proven separately.

$$\begin{array}{l}\forall\,\textbf{byte}\texttt{[]}\ msg1lv;\ \forall\,\textbf{byte}\texttt{[]}\ msg2lv;\\ \quad (msg1lv \neq msg2lv\ \&\ msg1lv.length \doteq msg2lv.length\\ \qquad \&\ \forall\,\textbf{int}\ j; (j \geq 0\ \&\ j < msg1lv.length\ \rightarrow msg1lv[j] \doteq msg2lv[j])\\ \quad \rightarrow \{\texttt{msg1} := msg1lv\}\ \{\texttt{msg2} := msg2lv\}\\ \qquad\ [\texttt{Crc.crcTest(msg1,msg2);}](result \doteq \textbf{false}))\end{array} \tag{7}$$

When trying to prove properties about programs containing unbounded loops (like the `crcTest()` method), then either a loop invariant or induction must be used. We chose to use an invariant of which a simplified version is shown below (8). The part of the program preceding the while statement was symbolically executed. Then a loop invariant rule was applied, which includes providing the actual invariant.

$$\begin{array}{l}(inject\_ar\_elem < \texttt{msg1.length} - \texttt{count} \rightarrow \texttt{reg1} \neq \texttt{reg2})\\ \&\ (inject\_ar\_elem \geq \texttt{msg1.length} - \texttt{count} \rightarrow \texttt{reg1} \doteq \texttt{reg2})\end{array} \tag{8}$$

The integer $inject\_ar\_elem$ results from the execution of the `inject1` statement and refers to the element in the `msg2` array where the fault is injected. It is a skolem constant originating from the universal quantification over the array elements used in the rule for `inject1` (see Fig. 2). In other words, the invariant has to hold for all possible values of $inject\_ar\_elem$.

After application of the loop invariant rule, the proof splits into three branches: one where it must be proven that the invariant holds before the while statement starts to execute, one where one needs to prove that (8) is indeed an invariant of the loop body provided that the guard holds, and one where it must be shown that the proof obligation (7) follows from the invariant and the negated loop condition. We proved all three cases using KeY. Here is the summary of the overall proof that (7) holds after execution of `crcTest()`.

1. The part of the program preceding the while statement was symbolically executed. This is automatic.
2. The loop invariant rule was applied and the loop invariant (8) manually provided.
3. KeY's automatic application of rules was restarted which resulted in about 2500 rule applications in less than 8 minutes. The result was 13 open goals, i.e., branches of the proof that could not be proved automatically. In all open goals, the program part of the proof obligation was completely (symbolically) executed, i.e., only program-free FOL formulas remained.

4. The 13 open goals were proved by manual rule application. This is tedious, but straightforward.

In summary, we proved formally that a certain implementation of the CRC fault detection algorithm discovers *all possible* single bit-flips in an arbitrary **byte** array. The proving activity was to a large extent automatic. It is straightforward to apply the same methodology to related algorithms, now that a valid pattern of loop invariants has been established.

7 Related Work

In [19] an approach for evaluating the system reliability with respect to bit-flip errors using model-checking principles is presented. This is applied to a software-implemented mechanism that detect errors corrupting the control flow, a signature analysis technique. A control flow graph of the considered generic target program, which is a representative model over a general class of all possible applications (i.e., it covers all possible fault scenarios with respect to the fault model) is created. The model checker SPIN is applied to the model and the fault detection mechanism in order to investigate whether the detection mechanism detects *all* faults. Since the description of the approach in the paper is highly dependent on the signature analysis technique, it is hard to see to which degree it is possible to generalize it to other kinds of fault tolerance mechanisms. Clearly, a necessary requirement is the ability to construct an abstract model of an imagined target program that covers all possible fault scenarios with respect to a considered fault model.

In several papers one specific fault tolerance mechanism is formally verified. In most cases these are system-level (in contrast to node-level) mechanisms for distributed systems, e.g., the TTP Group Membership Algorithm. Some examples of this line of work follow: in a paper by Bernardeschi et al. [6], a fault tolerance mechanism called “inter-consistency mechanism”, a component of an architecture for embedded safety-critical systems, was formally specified and verified using the model checker JACK. The properties the mechanism should satisfy were expressed as temporal logic formulas and the model of the mechanism was given as a Labelled Transition System (LTS) which included faults that could affect the behavior of the mechanism itself. In [25], a model of a startup algorithm for the Time-Triggered Architecture was proven to have certain safety, liveness, and timeliness properties using model checking (the SAL toolset from SRI). It is claimed that all possible failure modes were examined, an approach the authors call “exhaustive fault simulation”. A fault-tolerant group membership algorithm of TTP was formally specified and verified using a diagrammatic representation of the algorithm. The work is described in [21]. The PVS theorem prover was used for the verification. Clock synchronization algorithms are an important part of distributed dependable real-time systems. The paper [23] describes a formal generic theory of clock synchronization algorithms (that extracts the commonalities of specific algorithms) in the form of parameterized PVS theories. Several concrete algorithms are formally verified with PVS using this framework. In [22], different aspects of formal verification of algorithms for critical systems are discussed. As an example, the Interactive Convergence Algorithm (ICA) is proved to have certain properties using the EHDM system.

What distinguishes our approach from the mentioned papers is that we present *a general framework* for analysis and formal verification of *executable implementations* (in contrast to abstract models) of fault tolerance mechanisms.

Finally, it should be mentioned that symbolic fault injection has been used in a method for calculating the *coverage factor*, i.e., the proportion of faults that are actually handled by a system [17].

8 Discussion and Future Work

Traditional fault injection techniques suffer from a number of drawbacks, notably lack of coverage and failure to activate injected faults. In this paper we presented a new approach called symbolic fault injection which is targeted at validation of SIHFT mechanisms and is based on the concept of symbolic execution of programs.

It is an analytic approach in contrast to experimental evaluation done in conventional fault injection. With symbolic fault injection it becomes possible to emulate the consequences of *all possible* faults in a certain memory location. All injected faults are also activated, which is in general not the case with conventional fault injection. Symbolic fault injection based on formal verification can be expensive and requires some expertise, but this is also the case with conventional fault injection. In particular, to investigate the consequences of an injected fault is difficult and time consuming when using conventional methods.

Our fault model so far consists of single bit-flips in memory locations. This is achieved through pseudo-instructions added to the source code together with rules for handling these pseudo-instructions during symbolic execution. We implemented a prototype of our method based on the formal software verification tool KeY. We showed the viability of the approach by proving that a CRC implementation detects all possible single bit-flips. Clearly, this is only a proof of concept and a proper evaluation with realistic industrial software needs to be done.

An argument that is often raised against the usage of formal methods is that formal specifications of systems are normally not available and are very time consuming to create. Note that our approach is useful even without the availability of a formal specification, because it can be used to compute the symbolic effect of faults in the form of strongest postconditions (Sect. 5.4).

Limitations Our current implementation suffers from a number of limitations: since our fault injection technique is simulation-based, no real-time properties can be evaluated with it. Formal verification of real-time systems is still an area of research. So far we have not considered the injection of faults in pointer- or reference-variables, and we have only looked at faults in the data area of the memory, not the code area. We also inherit a number of limitations from the underlying verification system. The most important are that the program logic of the KeY system at the moment cannot handle multi-threaded programs or floating point data types. Research that overcomes the first of these is under way [18]. A practical limitation is that full automation can only be achieved when bounds on loops and recursion are finite and concrete. Otherwise, induction or invariant rules with expensive user interaction is required. Again, research to improve this situation is under way [20].

Future Work It would be interesting to generalize our approach to different fault models and fault trigger mechanisms. This is principally possible by parameterizing the total correctness modality $\langle \mathtt{p} \rangle \phi$ with additional parameters for a trigger condition t, a symbolic fault `inject`, and a reset expression $\mathcal{R}$, where t is a FOL formula, `inject` is an inject pseudo-statement, and $\mathcal{R}$ is a state update (Sect. 4). The semantics of the formula $\langle \mathtt{p} \,|\, t \,|\, \mathtt{inject} \,|\, \mathcal{R} \rangle \phi$ is the same as $\langle \mathtt{p} \rangle \phi$, but before symbolic execution of the next active statement it is checked whether t holds and `inject` is inserted whenever it does. In addition, after symbolic execution of each statement the update $\mathcal{R}$ is added to the current environment. If $\mathcal{R}$ is something like $\{\mathtt{b} := \textbf{false}\}$, then it easy to emulate a stuck-at-zero fault.

We think that it is attractive to integrate our technique into a framework for design and assessment of dependable software such as Hiller et. al.'s [12]. Part of this framework uses fault injection for error propagation analysis to find the locations in the software where it is most effective to place fault handling mechanisms. We think that our technique could be very useful in the error propagation analysis.

References

1. W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software and Systems Modeling*, 4(1):32–54, 2005.
2. P. Amey. Correctness by construction: Better can also be cheaper. *CrossTalk Magazine, The Journal of Defense Software Engineering*, pages 24–28, March 2002.
3. J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell. Fault injection for dependability validation: A methodology and some applications. *IEEE Trans. Softw. Eng.*, 16(2):166–182, 1990.
4. J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs, and G. H. Leber. Comparison of physical and software-implemented fault injection techniques. *IEEE Trans. Comput.*, 52(9):1115–1133, 2003.
5. B. Beckert, R. Hähnle, and P. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2006.
6. C. Bernardeschi, A. Fantechi, and S. Gnesi. Formal validation of the GUARDS inter-consistency mechanism. In M. Felici, K. Kanoun, and A. Pasquini, editors, *Intl. Conf. on Computer Safety, Security and Reliability (SAFECOMP)*, pages 420–430, 1999.
7. P. Bernardi, L. Bolzani, M. S. Rebaudengo, M. S. Reorda, and M. Violante. An integrated approach for increasing the soft-error detection capabilities in SoCs processors. In *Intl. Symp. on Defect and Fault Tolerance in VLSI Systems (DFT)*, pages 445–453, 2005.
8. R. M. Burstall. Program proving as hand simulation with a little induction. In *Information Processing '74*, pages 308–312. Elsevier/North-Holland, 1974.
9. J. Carreira, H. Madeira, and J. G. Silva. Xception: A technique for the experimental evaluation of dependability in modern computers. *Software Engineering*, 24(2):125–136, 1998.

10. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proc. ACM SIGPLAN Conf. on Progr. Language Design and Implementation, Berlin*, pages 234–245. ACM Press, 2002.
11. K. Havelund and T. Pressburger. Model checking JAVA programs using JAVA pathfinder. *Int. Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.
12. M. Hiller, A. Jhumka, and N. Suri. PROPANE: an environment for examining the propagation of errors in software. In *Proc. ACM SIGSOFT Intl. Symp. on Software Testing and Analysis*, pages 81–85. ACM Press, 2002.
13. M.-C. Hsueh, T. K. Tsai, and R. K. Iyer. Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82, 1997.
14. E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson. Fault injection into VHDL models: The MEFISTO tool. In *Proc. 24th Intl. Symp. on Fault Tolerant Computing, (FTCS-24), IEEE, Austin/TX, USA*, pages 66–75, 1994.
15. J. Karlsson, P. Folkesson, J. Arlat, Y. Crouzet, G. Leber, and J. Reisinger. Application of three physical fault injection techniques to the experimental assessment of the MARS architecture. In *IFIP Working Conf. on Dependable Computing for Critical Applications (DCCA-5)*, pages 267–287, Urbana-Champaign, USA, September 1995. IEEE Computer Society.
16. J. Karlsson, P. Liden, P. Dahlgren, R. Johansson, and U. Gunneflo. Using heavy-ion radiation to validate fault-handling mechanisms. *IEEE Micro*, 14(1):8–23, 1994.
17. L. T. Klauwer. Application of Formal Methods to Fault Injection and Coverage Factor Calculation. Master's thesis, Chalmers University of Technology, Department of Computer Science and Engineering, Göteborg, Sweden, 2006.
18. V. Klebanov, P. Rümmer, S. Schlager, and P. H. Schmitt. Verification of JCSP programs. *Concurrent Systems Engineering*, 63:203–218, 2005.
19. B. Nicolescu, Y. Savaria, E. Aboulhamid, and R. Velazco. On the use of model checking for the verification of a dynamic signature monitoring approach. *IEEE Transactions on Nuclear Science*, 52:1555–1561, Oct. 2005.
20. O. Olsson and A. Wallenburg. Customised induction rules for proving correctness of imperative programs. In B. Beckert and B. Aichernig, editors, *Proc. Software Engineering and Formal Methods, Koblenz, Germany*, pages 180–189. IEEE Press, 2005.
21. H. Pfeifer. Formal verification of the TTP Group Membership algorithm. In *Proc. FIP TC6 WG6.1 Joint Intl. Conf. on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XIII) and Protocol Specification, Testing and Verification (PSTV XX)*, pages 3–18. Kluwer, 2000.
22. J. M. Rushby and F. von Henke. Formal verification of algorithms for critical systems. *IEEE Trans. Softw. Eng.*, 19(1):13–23, 1993.
23. D. Schwier and F. W. von Henke. Mechanical verification of clock synchronization algorithms. In *Proc. 5th Intl. Symp. on Formal Techniques in Real-Time and Fault-Tolerant Systems*, LNCS, pages 262–271. Springer-Verlag, 1998.
24. A. E. K. Sobel and M. R. Clarkson. Formal methods application: An empirical tale of software development. *IEEE Transactions on Software Engineering*, 28(3):308–320, 2002.
25. W. Steiner, J. Rushby, M. Sorea, and H. Pfeifer. Model checking a fault-tolerant startup algorithm: From design exploration to exhaustive fault simulation. In *The Intl. Conf. on Dependable Systems and Networks*, pages 189–198, Florence, Italy, June 2004. IEEE Computer Society.
26. A. Wallenburg. Proving by induction. In B. Beckert, R. Hähnle, and P. Schmitt, editors, *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*, pages 453–480. Springer-Verlag, 2006.