Quantitative and qualitative safety analysis of a hemodialysis machine with S

Johannes Leupolz, Axel Habermaier, Wolfgang Reif

Angaben zur Veröffentlichung / Publication details:

Leupolz, Johannes, Axel Habermaier, and Wolfgang Reif. 2018. "Quantitative and qualitative safety analysis of a hemodialysis machine with S#." Journal of Software: Evolution and Process 30 (5): e1942. https://doi.org/10.1002/smr.1942.

Quantitative and qualitative safety analysis of a hemodialysis machine with S#

Johannes Leupolz | **Axel Habermaier** | **Wolfgang Reif**

Institute for Software and Systems Engineering, University of Augsburg, Germany

Correspondence
Johannes Leupolz, Institute for Software and Systems Engineering, University of Augsburg, Germany.
Email: leupolz@isse.de

Abstract

This paper reports on our experiences of applying S# ("safety sharp") to model and analyze the case study "hemodialysis machine." The S# safety analysis approach focuses on the question, what happens if we place a controller with correct software into an unreliable environment. To answer that question, the S# toolchain natively supports the Deductive Cause Consequence Analysis, a fully automatic model checking-based safety analysis technique that determines all sets of component faults with the potential of causing a system hazard. Furthermore, S# can give an approximate estimate of the hazard's probability. To demonstrate our approach, we created a model with a simplified controller of the hemodialysis machine and relevant parts of its environment and performed a safety analysis using Deductive Cause Consequence Analysis.

KEYWORDS

deductive cause consequence analysis, design tools and techniques, embedded domain specific language, executable models, formal methods, hemodialysis, model checking, quantitative analysis, safety analysis, simulation

1 | INTRODUCTION

Classical software verification focuses on answering the question whether the implementation of a piece of software conforms to its specification. This plays an important role in safety-critical domains like railway, automotive, aviation, and medical devices. However, another important aspect is to analyze what happens when a specification conforming controller is embedded into an unreliable environment. Failures of sensors or actuators may lead to situations where the state of the actual environment and the controller's internal data about the environment diverge. Such discrepancies result in degraded situations that could result in a hazard, an undesired situation with consequences like high follow-up costs or even loss of lives.

We have created S#[1-3] to reveal which combinations of unexpected behavior in the environment and which faults at the component level (eg, sensors failing) could result in a hazard. Additionally, S# can give an estimate of the hazard's probability. S# is an embedded Domain Specific Language in C#, specially tailored for the analysis of safety critical systems. These models contain information about the system of interest (controllers, sensors, actuators … everything a system designer has direct influence on) and its environment. Furthermore, S# offers language features for modeling faults and fault effects intuitively. Compared with similar approaches, the models can be highly modularized thanks to the adoption of C#'s object-oriented concepts. Using Visual Studio, modelers can use the standard C# debugging tools to simulate their models.[1] Our S# toolchain supports automatic analysis based on model checking[4] by LTSMIN[5] and MRMC.[6] The formal analysis techniques are integrated into the toolchain and can be used without detailed knowledge of the underlying formalism, which makes it easy for engineers to focus on the models (see also Visser et al[7]). Deductive Cause Consequence Analysis (DCCA), a fully automatic model checking-based safety analysis technique, can reveal detailed scenarios of how faults and unexpected behavior in the environment lead to a hazard.[8] In addition to that, S# can conduct probabilistic analyses to calculate the exact probabilities of hazards in dynamic systems.[2] Furthermore, the S# toolchain can also be used in later phases of the development to validate the behavior of a real controller (implemented in C or C++) in the modeled environment[9] or used for the runtime analysis of self-organizing systems.[10]

To demonstrate our approach, we have modeled a hemodialysis machine in S#, based on a case study presented at the ABZ 2016 conference.[11] This common case study allows the comparison of different verification approaches. The original case study has aimed at creating a formal specification of the controller. Our aim was to analyze how such a controller behaves in a physical environment. To be able to adequately express

the causal dependencies between different physical components, we have also modeled the fluid flows that interconnect them. Therefore, our model contains both the controller of the hemodialysis machine and relevant physical components like pumps and the dialyzer. Furthermore, we have modeled explicitly what happens when selected faults occur. Our model only contains the excerpt of the original controller that is necessary to demonstrate our approach. Our model focuses on the InitiationPhase and EndingPhase described in Mashkoor.[11] We have omitted the PreparationPhase, because our focus is to demonstrate the technical feasibility of an analysis with S#. A more elaborated analysis would require additional interviews with domain experts and more profound modeling of the controller. The emphasis of this paper is on how our approach can be applied to medical devices and not on specific details like timing behavior of the controller. S# and the complete model of the case study are available on http://safetysharp.isse.de.

The main contribution of this paper is to demonstrate how medical devices can be modeled in a way that is amendable to automatic safety analysis. This includes an automatic calculations of qualitative result in the form of minimal cut sets and quantitative results in the form of hazard probabilities. As the second contribution, the paper shows how flow behavior can be modeled in an imperative language. Albeit only being a coarse approximation of the hydrodynamic laws, it is simple enough to allow fast calculations, which is essential for fast model checking. Finally, yet importantly, this paper extends the original case study with non-controller components and their interrelationships. Other approaches for model-based safety analysis can use this extended case study as a foundation for the evaluation of their tools. This paper extends Leupolz et al[12] with details on S#'s step semantics and presents a quantitative analysis.

This paper is organized as follows: In the next section, we introduce the case study. Section 3 shows how we have modeled the control system of the hemodialysis machine in S# using features like state machines. Section 4 shows the step semantics of S#. Section 5 demonstrates how we implemented the physical fluid flows of the case study. Section 6 shows how faults are modeled and how those faults affect the state space, and gives an intuition how S# analyses those models using model checking. In Section 7, we use the DCCA to calculate which combinations of faults can lead to the hazards "dialysis unsuccessful" and "blood entering the vein of the patient is contaminated." In Section 8, we present how hazard probabilities can be approximated from the results of the DCCA and how S# can calculate an even more precise approximation. Section 9 presents related work on model-based safety analysis. Finally, Section 10 summarizes our findings and gives an outlook to future work.

2 | THE HEMODIALYSIS MACHINE CASE STUDY

The human body creates metabolic waste products like urea and minerals. Usually, the kidneys are responsible for the removal of these waste products from the blood. When the kidneys fail, a hemodialysis machine can be used for this removal instead. These machines have a direct influence on the chemical composition of a patient's blood and thus form a safety critical system. This description of a hemodialysis machine here is based on a case study description written to evaluate formal languages and a training handbook for dialysis technicians.[11,13]

A hemodialysis machine (see Figure 1) consists of 3 basic elements. The extracorporeal blood circuit (ECB), the dialyzer, and the dialyzing fluid delivery system (DFDS). Medical staff uses syringes to connect the artery and the vein of the patient to the ECB.

The main purpose of the ECB is to deliver the blood from the patient to the dialyzer and back again to the patient. A blood pump creates a suction to pump the patient's blood through the ECB. The heparin pump adds heparin into the patient's blood to prevent blood clotting. The arterial and venous pressure transducers deliver blood pressure values to allow their monitoring. The venous tubing valve enables another safety measure to prevent bad blood reentering the patient. Whenever the safety detector detects contaminated blood or gas in the blood, the venous tubing valve is closed, and no blood can reenter the patient. The dialyzer itself is part of 2 fluid flows: a blood flow and a dialyzing fluid flow. Inside the dialyzer, a semipermeable membrane separates these 2 flows. At the membrane, small-sized waste products go from the blood side to the dialyzing fluid side. Additionally, big-sized waste products can be removed from the blood side by creating a suction on the dialyzing fluid side (ultrafiltration).

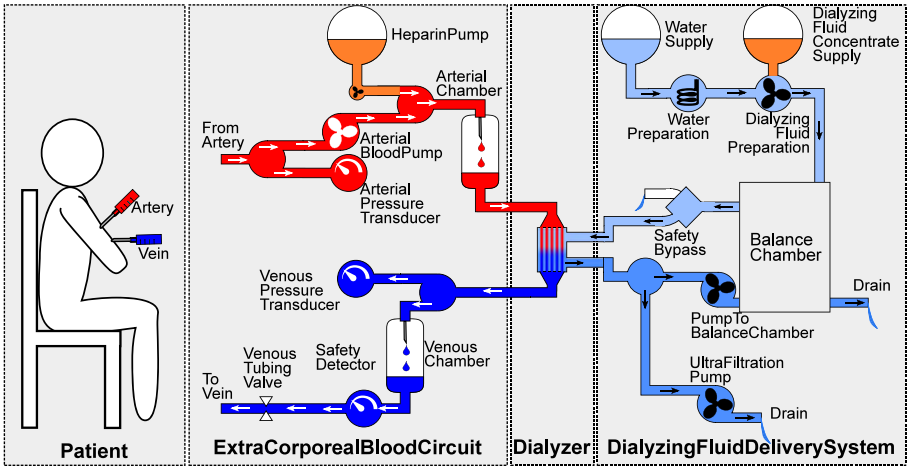

FIGURE 1 Hemodialysis case study

The incoming dialyzing fluid of the dialyzer is produced by the DFDS. The DFDS produces dialyzing fluid in several steps. The balance chamber acts as a buffer for dialyzing fluid. When the dialyzing fluid has the wrong temperature, the saftey bypass pipes the fluid to the drain instead the dialyzer.

Figure 2 shows how the hemodialysis machine can be decomposed using the standard system modeling language SysML.[14] The complete **Specification** contains the **Patient** and the **HdMachine** (hemodialysis machine). It is necessary to include the patient in the model to be able to express hazards that concern the patient. The **HdMachine** itself consists of several parts, namely the **Dialyzer**, the **ControlSystem**, the **DialysingFluidDeliverySystem**, and the **ExtraCorporealBloodCircuit**. **DialyzingFluidDeliverySystem** and **ExtraCorporealBloodCircuit** themselves consist of several parts (eg, **WaterSupply**). The **ControlSystem** itself only contains references to these subparts because they are *physically* not part of **ControlSystem**. Figure 1 shows the interconnection between those parts.

3 | CONTROLLER SPECIFICATION IN S#

There are 2 established ways to model the behavior of controllers, namely by state machines and by sequential code. S# supports both of them. S# even allows nesting state machines with sequential code. Listing 1 shows an excerpt of our model of the control system of the hemodialysis machine.

Listing 1 S# model of the control system's behavior

```
public enum TherapyPhase {.
  InitiationPhase,
  EndingPhase
}
public class ControlSystem: Component {.
  public int TimeStepsLeft = 6; // hard code 6 time steps
  // references to components
  private readonly VenousSafetyDetector VenousSafetyDetector;
  private readonly VenousTubingValve VenousTubingValve;
  /* (other components and constructor left out for brevity) */
  public StateMachine<TherapyPhase> CurrentTherapyPhase = TherapyPhase.InitiationPhase;

  public void StepOfMainTherapy() {/* behavior left out for brevity */}
  public void ShutdownMotors() {
      VenousTubingValve.CloseValve();
      TimeStepsLeft = 0;
      ArterialBloodPump.SpeedOfMotor = 0;
      UltraFiltrationPump.PumpSpeed = 0;
      PumpToBalanceChamber.PumpSpeed = 0;
      DialyzingFluidPreparation.PumpSpeed = 0;
  }
  public override void Update() {
      CurrentTherapyPhase.Transition(
          from: TherapyPhase.InitiationPhase,
          to: TherapyPhase.InitiationPhase,
          guard: TimeStepsLeft>0 &&! VenousSafetyDetector.DetectedGasOrContaminatedBlood,
          action: StepOfMainTherapy
      )
      .Transition(
          from: TherapyPhase.InitiationPhase,
          to: TherapyPhase.EndingPhase,
          guard: TimeStepsLeft <= 0 || VenousSafetyDetector.DetectedGasOrContaminatedBlood,
          action: ShutdownMotors
      );
  }
}
```

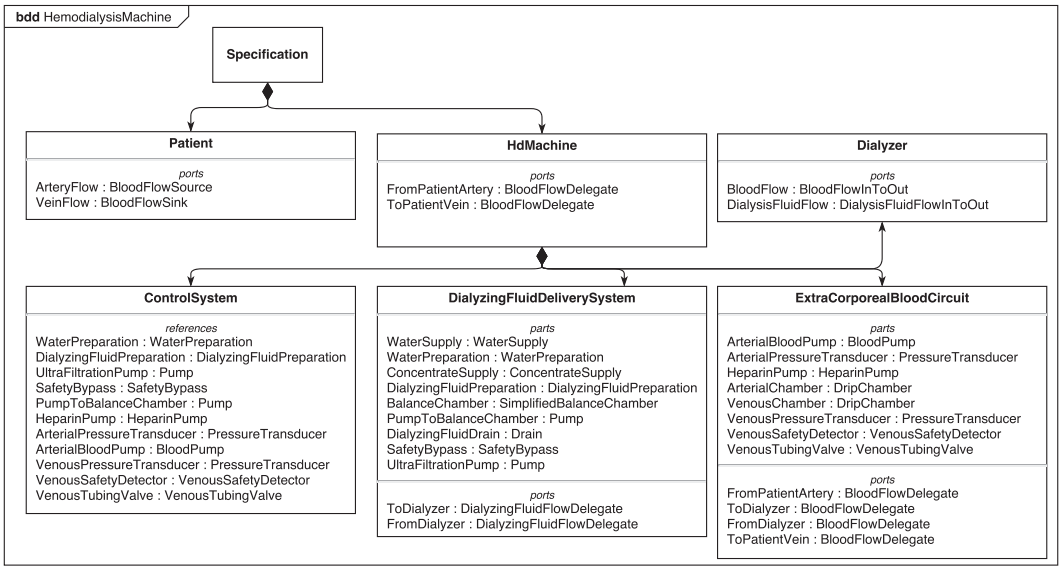

FIGURE 2 Structure of the complete hemodialysis machine as a SysML block definition diagram

First, the possible therapy phases InitiationPhase and EndingPhase are declared in the enumeration. In the following lines, the ControlSystem is declared. To express that the ControlSystem is a component it inherits from the S# class Component. The class contains the field TimeStepsLeft of type int with the initial value 6. The S# Domain Specific Language provides a generic state machine whose states can be determined by instantiating the generic type StateMachine<T > with an enumeration T of the desired states. One such state machine is CurrentTherapyPhase, whose states are determined by the enumeration TherapyPhase. The initial active state is set to InitiationPhase. The ControlSystem contains references to components that are not part of the ControlSystem itself like VenousSafetyDetector of type VenousSafetyDetector.[1] The concrete instances of the references are set in the constructor and are not shown in the excerpt for brevity. The reference is marked as readonly because it cannot be changed during model checking. The class ControlSystem also contains the methods StepOfMainTherapy and ShutdownMotors. The behavior defined in the methods use the previously declared references. Methods may be called from state machines or from other methods. Finally, the Update method of ControlSystem contains 2 transitions of the state machine CurrentTherapyPhase. The first transition is a reflexive transition from the initial state InitiationPhase to itself. A transition is usable when the active state of the state machine is the *from*-state of the transition and the *guard* of the transition evaluates to true. Each time the Update method is called, the state machine selects an arbitrary usable transition, executes its *action*, and sets the *to*-state as the next active state. This *action* can be any method of the containing component.

4 | STEP SEMANTICS OF S#

S# has no direct or implicit notion of time. Instead, time needs to be modeled explicitly. In S#, models are both discrete-time and discrete-state. Figure 3 shows a simplified excerpt, how the dialysis system progresses in the initiation phase of the dialysis in a discrete-time and discrete-state way. Initially, the remaining time of the dialysis is 6 hours. In this state, the membrane of the dialyzer is not defective the safety detector senses blood of good quality and the venous tubing valve is open. After 1 hour, the remaining time decreases to 5 hours, but otherwise nothing changes. Constant time passes between the states.

To be able to model this adequately, S# uses a step semantics based on micro steps and macro steps. The states depicted in Figure 3 show the values of the state variables after the macro steps. A macro step consists of a sequence of micro steps. Figure 4 illustrates how the micro and macro step semantics work in our model of the hemodialysis machine.

Micro steps (depicted by small circles) describe the internal behavior of a macro step. Each basic statement of the C# programming language (eg, incrementing a variable or assigning a value to a variable) can be such a micro step. The controller and all other components have an Update-method, which describes their behavior using C# code.

Algorithm 1 Macro step execution of deterministic model in 2 phases.

function MacroStep(sourceState: State, flowComponents: Component[*], controllers: Component[*])
returns State.
load sourceState;

[1]When the C# compiler is able to distinguish whether an instance or a type is meant, an instance name can be equal to a previously defined type name.

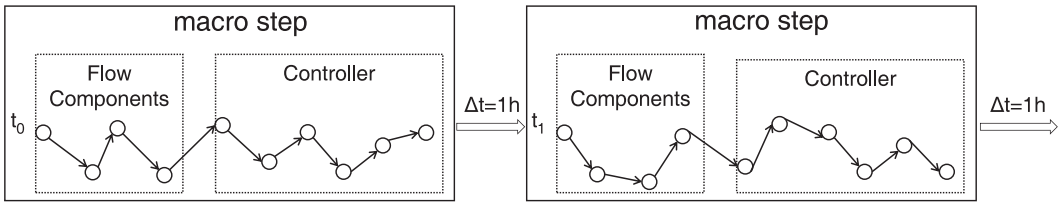

FIGURE 3 Excerpt of the state space, which shows the progress of the dialysis in the initiation phase

```
for f in flowComponent do.
    f.Update();
for c in controllers do.
    c.Update();
return current state;
```

Algorithm 1 shows how S# calculates the successor state of a given state. This assumes that the model is deterministic, ie, every state has exactly 1 successor state. At the beginning of each macro step, S# ensures that the variables inside the modeled components correspond to the variables inside the state (load sourceState). Then, every flow component is updated (the state of the blood pump, the dialyzer, the water preparation, etc.), and then the controller reacts to the changes and sends new inputs for the electronic components inside the system.

The micro steps determine how the state looks after 1 hour. As we assume a treatment time of 6 hours, 6 time steps are necessary to complete the dialysis. Thus, the instant of time t_0 describes the state of the system before the dialysis has been started. After executing the first macro step, the system is in the state t_1 when the dialysis has been running for 1 hour. We update the system state only once each hour to reduce the size of the state space. Although the model is very coarse, the safety analysis with S# reveals some interesting insights about the system's behavior, as shown later.

S# can only check if desired properties of the system are satisfied in the states. It is not possible to check if the desired properties are satisfied between 2 micro steps. We accept this small weakness, because it enables S#'s algorithm to decrease the analysis time by orders of magnitude.

To sum it up, only the behavior of the micro steps needs to be modeled explicitly in our approach. This is a big relief, because modelers can focus on the individual behavior of each component and do not have to know the possibly unpredictable effects of multiple components acting at the same time. How micro steps and macro steps have to be interpreted is model dependent. It is the task of the modeler to ensure that a model is adequate, as with every modeling language. Habermaier et al,[1] Leupolz et al,[2] and Habermaier[3] provide a more formal description of the step semantics of S#.

5 | MODELING OF PHYSICAL FLUID FLOWS IN THE ENVIRONMENT

The model of the control system would be sufficient to verify if the control system fulfills a particular specification. However, we must also include non-controller components into the model to conduct a complete safety analysis. Typically, a controller makes assumptions about its environment. These assumptions may be wrong. The perceived state may be different from the actual state, especially when sensors fail or deliver erroneous values. Furthermore, hazards are typically expressed in terms of a system's environment, eg, contaminated blood is entering the vein of the patient.

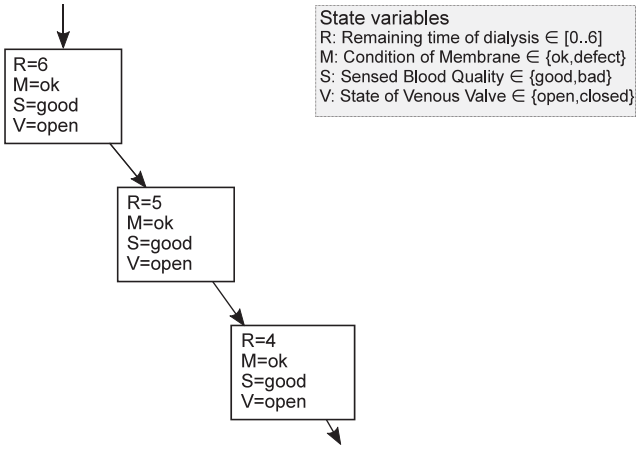

FIGURE 4 Interpretation of micro and macro steps in the case study

To express this hazard in S#, the relevant parts of the controllers' environment have to be included into the model. Another example from the aviation domain affirms this observation. A plane should never retract its landing gear when the plane is on the ground. A faulty sensor may give the controller the wrong impression that the plane is already airborne, leading to a retraction of the landing gear. In this case, it is not enough to proof that the controller never retracts the landing gear when it *senses* that the plane is on the ground. A safety analysis should prove that a fault-tolerant controller never retracts the landing gear when the plane *actually is* on the ground even when the sensing fault occurs. In addition, modeling the environment makes it possible to calculate the probability of a hazard.

We need to express fluid flows to be able to model the non-controller elements of the hemodialysis machine. Fluid flows obey complex physical laws as, eg, the Bernoulli's principle. Fortunately, to create a useful model of many fluid flows, it is not necessary to take account of all details of these laws. It is sufficient to model the basic principles necessary for the case study. It is of greater importance for the subsequent model checking that the calculations can be performed efficiently. Of course, the better the hydrodynamic laws are represented, the better are the results. In this section, we demonstrate how simple fluid flows can be modeled in an efficient way in S#. The implementation is generic and can be reused for any acyclic fluid flow where no backflow is possible, ie, the direction in which the fluids flow is fixed. Finally, we present the model of the pump for dialyzing fluid used in the hemodialysis machine.

5.1 | Flow concept

Our model separates the information of how each flow component works and how all flow components form a common flow. The flow between these components is only declared by connecting these flow components. The separation increases the reuse of any modeled flow component. This corresponds to the design pattern *low coupling*. By contrast, models that encode assumptions about the flow in each component are harder to understand and to revise. Thus, it is more likely they contain bugs.

Imagine that we want to model a fluid flow from a water supply to a drain. A pump that sits in between the water supply and the drain determines the exact amount of water that flows in each step. We want to treat the water supply, the pump, and the drain as separate independent flow components. The challenge is to create an adequate model (where the pump determines the amount of fluid that is emitted by the water supply) and adhere to the low coupling paradigm at the same time.

When a pump adds dynamic pressure to a flow system, a flow with a certain speed is the result of the law of conservation of energy. The basic idea of our simplified model is to see a fluid flow as a bidirectional flow (see Figure 5). In the backward direction, there is a suction representing the added dynamic pressure. This pressure determines the emitted fluid. In the forward direction, there is a flow of a specific amount of fluid, which is the result of the compensation of the added dynamic pressure. In the example, the pump emits a suction on the water supply, which is its predecessor in the flow. Now, based on the incoming suction, the water supply can determine the exact amount of fluid to emit into the direction of the pump. To calculate the amount of fluid that arrives at the drain the following sequence is executed:

1. The **Drain** notifies its predecessor that it is able to receive any amount of fluid (SendBackward = any amount).
2. The **Pump** receives the suction information of the **Drain** (= suction y). It calculates the suction x based on suction y. In this case, y = "any amount of fluid", so x is set to the number of units of liquid the pump moves in a time step. Then, the **Pump** notifies its predecessor that it wants x units of fluid (UpdateBackward = suction x).
3. The **WaterSupply** receives the suction information of the **Pump** (ReceivedBackward = suction x).
4. The **WaterSupply** emits fluid. It knows the amount of fluid to emit from the previously received suction (SendForward = x units fluid).
5. The **Pump** forwards the received fluid to its successor (UpdateForward = x units fluid).
6. The **Drain** receives the fluid (ReceivedForward = x units fluid).

This concept might seem to be too over-engineered for this small example, but adhering to this concept allows implementing a pump in a similar way as the safety valve, drip chambers, or any other fluid component where a flow runs through. On the other hand, it is simple enough to allow fast calculations, which is essential for fast model checking. Nevertheless, adhering more closely to hydrodynamic laws would provide better results but slow down analysis time significantly, because the pressure equilibrium needed to be calculated using iterative methods.

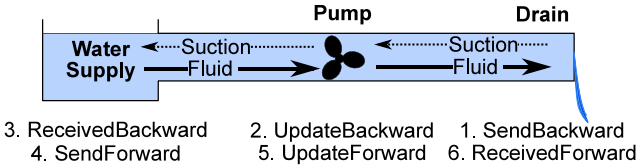

FIGURE 5 Example of a simple fluid flow. Fluid flows from the **WaterSupply** to the **drain**. The **pump** determines the amount of fluid

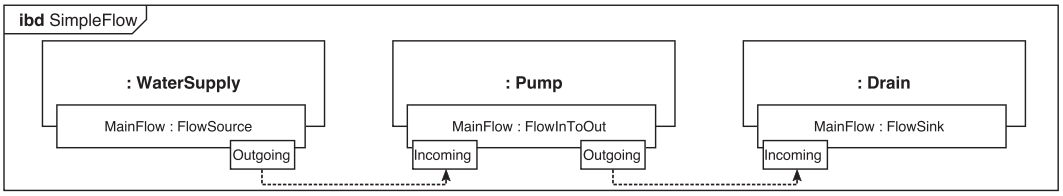

FIGURE 6 Simple fluid flow of Figure 4 as a SysML internal block diagram

5.2 | Modeling fluid flows

The SysML Internal Block Diagram in Figure 6 depicts the structure of the simple fluid flow model of Figure 5 in S#. We created for each flow component a separate component declaration with its behavior. An instance of a component declaration is denoted by InstanceName:DeclarationName where InstanceName is optional. Each component in the example contains a port with the name MainFlow that are of a different type, respectively. WaterSupply has a port of the type FlowSource with the property Outgoing; Drain a port of type FlowSink with the property Incoming; and Pump a port of type FlowInToOut with both properties. A flow is established by connecting the Outgoings with the Incomings. The dashed arrows depict connections.

Listing 2 presents an excerpt of the S# code of the simple fluid flow that creates the simple fluid flow shown in Figure 5.

Listing 2 A simple fluid flow in S#.

```
var supply = new WaterSupply();
var pump = new Pump();
var drain = new Drain();
var combinator = new DialyzingFluidFlowCombinator();
pump.PumpSpeed = 7;
combinator.ConnectOutWithIn(supply.MainFlow, pump.MainFlow);
combinator.ConnectOutWithIn(pump.MainFlow, drain.MainFlow);
combinator.CommitFlow();
```

The classes WaterSupply, Pump, and Drain contain generic "templates" of how water supplies, pumps, and drains work, respectively. These generic templates need to be instantiated to use them in a specific flow. The flow components supply, pump, and drain are instantiated by calling the constructor of their corresponding classes, respectively. A DialyzingFluidFlowCombinator is instantiated, which is used to establish the flow between the instances of the flow components. The speed of the pump is set to 7 units per step. Whether 1 unit of measurement should be interpreted as liters, 250 mL, or even something else is the responsibility of the modeler. We assumed 1 unit of fluid to be 100 mL in the hemodialysis example.

At some places, fluid flows are split into sub flows (eg, before pressure transducers, see Figure 1). Figure 7 shows an example where a flow splits into 2 sub flows. Each sub flow ends in a Drain. These splitting flows are realized with ports of the type FlowSplitter. A FlowSplitter has multiple Outgoing properties, which can be connected to the Incoming properties of different ports in different flow components.

To merge several flows, S# supports FlowMerger. FlowMerger is dual to FlowSplitter and is not described here any further.

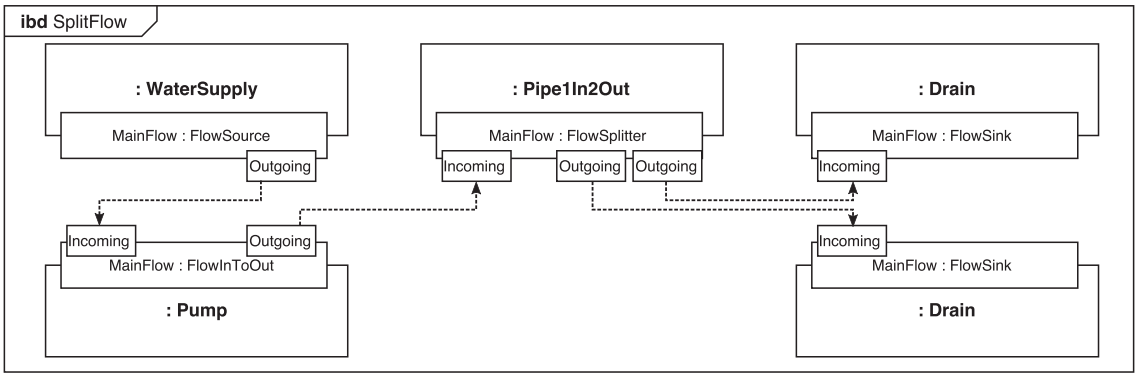

FIGURE 7 Structure of the complete hemodialysis machine as a SysML block definition diagram

5.3 | Modeling a pump

In Listing 3, we present the textual model of the Pump of the hemodialysis machine. This pump is instantiated 2 times in the full model, namely as PumpToBalanceChamber and as UltraFiltrationPump (see Figure 2).

Listing 3 S# model of a pump

```
public class Pump: Component {.
  public readonly FlowInToOut<DialyzingFluid,Suction> MainFlow;
  public int PumpSpeed = 0;
  public Pump() {
      MainFlow = new FlowInToOut<DialyzingFluid,Suction>();
      MainFlow.UpdateBackward = SetMainFlowSuction;
      MainFlow.UpdateForward = SetMainFlow;
  }

  public DialyzingFluid SetMainFlow(DialyzingFluid fromPredecessor) {return fromPredecessor;}
  public virtual Suction SetMainFlowSuction(Suction fromSuccessor) {
      Suction toPredecessor;
      toPredecessor.SuctionType = SuctionType.CustomSuction;
      if (fromSuccessor.SuctionType==SuctionType.SourceDependentSuction)
          toPredecessor.CustomSuctionValue = PumpSpeed;
      else
          toPredecessor.CustomSuctionValue = fromSuccessor.CustomSuctionValue+PumpSpeed;
      toPredecessor.CustomSuctionValue = PumpSpeed;
      return toPredecessor;
  }
}
```

The port MainFlow contains the 2 delegates UpdateBackward and UpdateForward, which determine the methods to call when a suction is received from the successor and a fluid element is received from the predecessor, respectively. In the constructor, the methods SetMainFlowSuction and SetMainFlow are assigned to the 2 delegates. Every time the port receives a suction, the local member SetMainFlowSuction is called. This method creates a suction on its predecessor in the size of PumpSpeed. Incoming fluids are just forwarded to the successor.

6 | SAFETY ANALYSIS OF SYSTEMS WITH EXPLICITLY MODELED FAULTS

As already mentioned earlier, faults may lead to the situation that the perceived state of a controller differs from the actual state. S# needs an explicit model of the fault effects to calculate the effects of those faults automatically. Listing 4 extends Listing 3 with the effect of the fault PumpDefect.

Listing 4 Extended S# model of a pump with explicitly modeled faults

```
public class Pump: Component {.
  /*...code from Listing 3*/
  public readonly Fault PumpDefect = new TransientFault();
  [FaultEffect(Fault = nameof(PumpDefect))]
  public class PumpDefectEffect: Pump {
      public override Suction SetMainFlowSuction(Suction fromSuccessor) {
          Suction toPredecessor;
          toPredecessor.SuctionType = SuctionType.CustomSuction;
          toPredecessor.CustomSuctionValue = 0;
          return toPredecessor;
      }
```

```
  }
}
```

The example shows how to declare faults and fault effects. Every fault might have several fault effects. The single fault effect PumpDefectEffect of the fault PumpDefect where the pump does not emit any suction on its predecessor overrides the original behavior of SetMainFlowSuction. Therefore, the original method SetMainFlowSuction of Pump (declared in Listing 3) is overwritten by the method with the same name in the nested class PumpDefectEffect. A fault always overrides a correct method, and it is even possible for more fault effects to override the same method. A fault may be either transient or permanent. Transient faults may only occur temporarily, ie, they might occur in 1-step and in the next disappear. Permanent faults, by contrast, do not disappear until they are repaired, ie, whenever they become active in 1-step, they also occur in all subsequent steps. More information about faults can be found in our wiki on http://safetysharp.isse.de.

For our evaluation, we integrated 9 faults into the model of the hemodialysis machine:

- BloodPumpDefect: The blood pump of the ECB does not create suction.
- DialyzerMembraneRupturesFault: The membrane of the dialyzer ruptures. The dialyzing fluid inside the dialyzer is contaminated by blood and the chemical composition of the blood inside the dialyzer is abnormal.
- DialyzingFluidPreparationPumpDefect: The pump that pumps the fresh dialyzing fluid to the balance chamber does not create any suction towards the water supply.
- SafetyBypassFault: The safety bypass cannot relay the dialyzing fluid into the drain anymore. Therefore, the bypass forwards all dialyzing fluid into the dialyzer, even if the dialyzing fluid does not meet the temperature constraints.
- WaterHeaterDefect: The water preparation does not heat the incoming water anymore.
- PumpToBalanceChamberDefect: The pump that pumps dialyzing fluid from the dialyzer back to the balance chamber is defect.
- SafetyDetectorDefect: The safety detector signals that the passing blood flow is all right even if it is contaminated.
- ValveDoesNotClose: The venous tubing valve cannot be closed anymore.
- UltrafiltrationPumpDefect: The pump for ultrafiltration is defective.

The S# toolchain offers 2 techniques to analyze the possible behaviors of a model: simulation and exhaustive exploration of the state space by using model checking. The exhaustive exploration is especially useful in case studies with lots of nondeterministic behavior, which is always the case with faults.

An example of how a fault in 1 component can have an impact on the behavior of another component is shown in Figure 8. Each box represents a state consisting of the state variables denoted by capital letters. The lower case letters represent different component faults, which may occur during a transition to a successor state. It is even possible that several faults occur at the same time (eg, {m,v}) or different combinations of faults lead to the same successor state (in these cases, a transition has multiple labels). For instance, when the membrane ruptures ({m}) in the transition leaving the topmost state in Figure 8, the membrane is defective in the upcoming state (M = defect). When otherwise no fault occurs ({}), the membrane in the upcoming state is intact (M = ok). An example of a well-behaved trace is highlighted using double-bordered boxes. In this case, the membrane is not ruptured, and no bad blood enters the vein of the patient. This highlighted trace also represents the trace of the deterministic, fault free model shown in Figure 3. The state space also reveals reachable states where the venous valve was not closed after the membrane ruptured, and thus, contaminated blood enters the patient (states indicated by dashed bordered boxes).

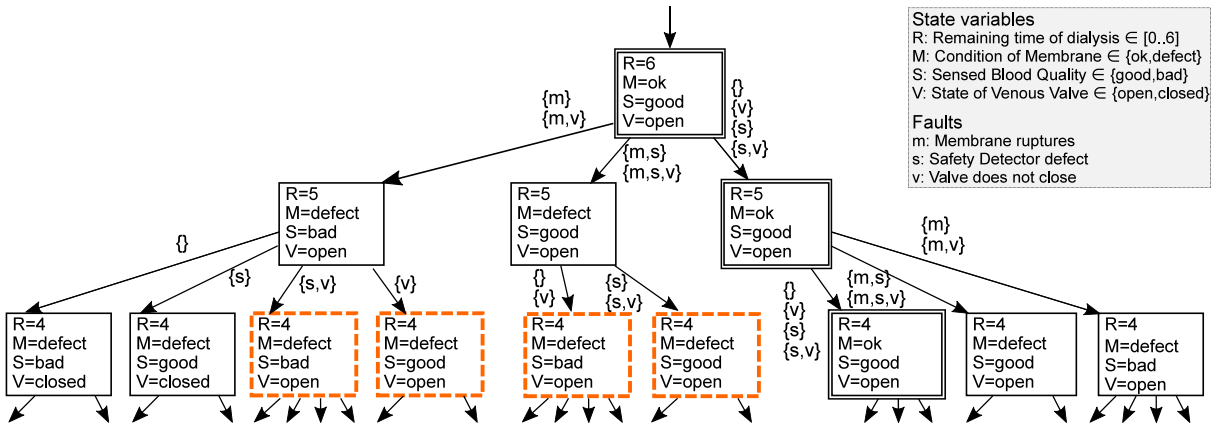

FIGURE 8 Simplified excerpt of the state space in the hemodialysis machine case study

Even in the simplified excerpt, some states have 4 successor states. Because of the vast number of reachable states, there is no way to inspect every possible behavior by hand. Thus, simulations and tests conducted manually can only check a few traces. This fact makes model checking a useful technique to analyze the complete state space of S# models.

Algorithm 5 Macro step execution of non-deterministic model in 2 phases with faults enabled.

```
function MacroStep(sourceState: State, flowComponents: Component[*], controllers: Component[*])
returns State[*].
  successorStates = [];
  for each combination of faults do.
    load sourceState;
  for f in flowComponent do.
      f.Update();
  for c in controllers do.
      c.Update();
    successorStates + = current state;
  return successorStates;
```

To generate the state space, S# executes 5 on every reachable state. It calculates for every sourceState and every combination of faults the successor states. For more details on the used algorithms and a formal semantics, we refer to Leupolz et al.[2]

Simulation complements model checking, because it is a valuable technique to find obvious bugs during the initial development. Model checking can finally be used to find hidden bugs and gain more confidence.

7 | APPLYING QUALITATIVE SAFETY ANALYSIS

A *cut set* for a safety hazard H is a set of faults that in combination leads to this hazard. Cut sets might contain faults that might be removed, while the combination of the other faults still leads to a hazard. A *minimal cut set* is a cut set which cannot be reduced. The interplay of all faults in a minimal cut set is necessary to result in a hazard. This means that being able to prevent the occurrence of only 1 fault in each minimal cut set always prevents the hazard. For instance, the membrane rupturing in combination with the broken valve is a minimal cut set for the hazard "blood entering the vein of the patient is contaminated"', as blood is contaminated in the dialyzer and the valve is unable to prevent the contaminated blood from entering the vein of the patient. This minimal cut set is denoted as {DialyzerMembraneRupturesFault, ValveDoesNotClose}. If only 1 of these 2 faults and no other fault occurs, the system is still safe. Thus, knowing the minimal cut sets also means knowing the weak spots of a system.

In traditional fault tree-based methodologies, first a fault tree is created by hand before minimal cut sets can be derived from such a fault tree.[15] This is costly and labor-intensive. By contrast, S#'s DCCA is a fully automated and model checking-based safety analysis technique. Given a safety hazard*H* H, the DCCA computes all minimal cut sets by individually checking all combinations of component faults Γ, determining whether such a set Γ does or does not have the potential to cause an occurrence of H. Habermaier[3] gives detailed algorithms of the DCCA and formal proofs of their correctness and completeness.

A set of component faults Γ is a cut set for a hazard H if and only if there is the possibility that H occurs and before that, at most the faults in Γ have occurred. The result of the DCCA is a set of minimal cut sets. We denote this resulting set of sets as MCS. DCCA has exponential complexity because all combinations of component faults have to be checked in the worst case. In practice, however, the number of required checks usually is significantly lower, as the cut set property is *monotonic* with respect to set inclusion, that is, once a set of component faults Γ is known to be a cut set, all supersets Γ' with $\Gamma \subseteq \Gamma'$ are cut sets as well.[3]

S#'s qualitative analysis can also reveal temporal order relationships between faults. Informally speaking, an order relationship exists when 1 fault strictly occurs before another fault every time when this combination of faults leads to a hazard.[16]

We analyzed the resulting model with the DCCA for the hazards "dialysis unsuccessful" and "blood entering the vein of the patient is contaminated." We performed S#'s DCCA on the model with the integrated faults regarding both hazards. The results are summarized in Table 1. S# revealed 6 minimal cut sets of size one for the first hazard. Thus, they form single points of failures. For the second hazard S# finds 3 minimal cut sets of size 2 and no single point of failure. In addition, S# detects order relationships for the cut sets of the second hazard. (1) The fault DialyzerMembraneRupturesFault must occur strictly before ValveDoesNotClose. (2) WaterHeaterDefect must occur before or at the same time as SafetyBypassFault. (3) The fault DialyzerMembraneRupturesFault must occur strictly before SafetyDetectorDefect.

The analysis has been executed on a desktop computer with a 4 core, 3.0 GHz Intel i5, and 16 GB of RAM. The state space of the model consists of 84 571 states and 286 057 transitions. Every state has a size of 40 bytes. For every minimal cut set, a trace that leads to the hazard is automatically generated by S#, which can be replayed in a graphical visualization. This facilitates validation together with domain experts.

TABLE 1 Results of the qualitative analysis

Hazard	Minimal Cut Sets	Time
Unsuccessful: Blood is not cleaned and dialysis finished	(1) {DialyzingFluidPreparationPumpDefect} (2) {WaterHeaterDefect} (3) {PumpToBalanceChamberDefect} (4) {UltrafiltrationPumpDefect} (5) {BloodPumpDefect} (6) {DialyzerMembraneRupturesFault}	2 seconds
Contamination: Blood entering the vein of the patient is contaminated	(1) {SafetyBypassFault, WaterHeaterDefect} (2) {DialyzerMembraneRupturesFault, SafetyDetectorDefect} (3) {DialyzerMembraneRupturesFault, ValveDoesNotClose}	6 sec

8 | APPLYING QUANTITATIVE SAFETY ANALYSIS

In traditional fault tree-based methodologies, the hazard probability is deduced from the resulting minimal cut sets of the qualitative analysis.[15] Faults are usually assumed stochastically independent. Thus, the probability of a hazard can be approximated by summing up the probabilities of each minimal cut set leading to a hazard. This is an upper bound for the probability of a hazard. The probability of 1 minimal cut set is approximated by multiplying the probabilities of the faults inside that minimal cut set. This is the so-called "rare event approximation." More formally,

$$\mathrm{P}_{\mathrm{rea}}(\mathit{Hazard}) = \Sigma_{\Gamma \in \mathrm{MCS}} \, \pi_{\delta \in \Gamma} \, \mathrm{P}(\delta), \mathit{where}\ \mathrm{P}(\delta)\ \mathit{denotes\ the\ probability\ of\ fault}\ \delta$$

Table 2 ; we used to analyze the case study. Note that the provided probabilities only serve for illustrating our approach. Of course, for the analysis of a specific system, the manufacturer must provide the precise probabilities of each deployed component. The first column contains the name of the fault. The second column shows the probability P_6 which denotes the probability that the fault occurs at least once in 6 time steps. This probability can be derived easily from the first probability. According to the geometric distribution in statistics, given a probability p of success in 1 trial and k independent trials, the probability that we have at least 1 success within k trials is $1 - (1 - p)^k$.[17] In our case, we want to determine how probable it is to fail within 6 time steps. Ironically, we treat the occurrence of a fault as "success."

Note that component suppliers often publish the mean time to failure (MTTF) of their components. The definition of MTTF is based on the exponential distribution. Due to the discrete nature of S#, the geometric distribution is used, which is the discrete analogue of the exponential distribution. A fault probability of $p = 1 \times 10^{-7}$ means that the chance that the corresponding fault occurs within 1 hour is 10^{-7}. When this fault is permanent, the accumulated probability to fail within 1 year of constant use is $1 - (1 - p)^{365 \times 24} \approx 9 \times 10^{-4}$.

Applying the formula for $\mathrm{P}_{\mathrm{rea}}$ on the minimal cut sets calculated earlier gives us the probability that a hazard occurs in 1 dialysis run:

$$\mathrm{P}_{\mathrm{rea}}(\mathit{Unsuccessful}) = \mathrm{P}_6(\mathit{DialyzingFluidPreparationPumpDefect}) + \ldots \approx 0.066, \mathit{and}$$

$$\mathrm{P}_{\mathrm{rea}}(\mathit{Contamination}) = \mathrm{P}_6(\mathit{SafetyBypassFault}) \times \mathrm{P}_6(\mathit{WaterHeaterDefect}) + \ldots \approx 3.6 \times 10^{-4}.$$

S# does not derive the probability from the minimal cut sets. Instead, S# derives the probability from the traces directly. This produces more accurate estimates and is especially useful when some of the faults are transient or when self-organizing mechanisms are implemented in a system. When a controller detects a fault in a component and decides to switch to a spare component or to go into a degraded mode, this has a direct effect on the traces. The underlying formalism is based on discrete time Markov chains, which supports probabilistic transitions, and model checking is performed by MRMC.[6] Leupolz et al[2] provides details on the used algorithm and the formal semantics of the quantitative analysis with S#.

TABLE 2 Fault probabilities

Fault δ	$\mathrm{P}_1(\delta)$	$\mathrm{P}_6(\delta)$ (approx.)
BloodPumpDefect	1×10^{-5}	6×10^{-5}
DialyzerMembraneRupturesFault	1×10^{-5}	6×10^{-5}
DialyzingFluidPreparationPumpDefect	1×10^{-5}	6×10^{-5}
SafetyBypassFault	1×10^{-3}	6×10^{-3}
WaterHeaterDefect	1×10^{-2}	6×10^{-2}
PumpToBalanceChamberDefect	1×10^{-5}	6×10^{-5}
SafetyDetectorDefect	1×10^{-7}	6×10^{-7}
ValveDoesNotClose	1×10^{-5}	6×10^{-5}
UltrafiltrationPumpDefect	1×10^{-3}	6×10^{-3}

When all faults are specified as permanent faults, then S# deduces the following results from the model:

$$P_{s\#}(\textit{Unsuccessful}) \approx 0.052 \ (\textit{in 2 seconds}), \textit{and}$$

$$P_{s\#}(\textit{Contamination}) \approx 3.5 \times 10^{-4} \ (\textit{in 4 seconds}).$$

These probabilities resemble the results of P_{rea}. The reason they are slightly smaller is that the result of S# is more precise. Some fault combinations may be part of more minimal cut sets. For instance, the fault combination {BloodPumpDefect,WaterHeaterDefect} is part of the minimal cut sets {BloodPumpDefect} and {WaterHeaterDefect} at the same time. Therefore, in the traditional analysis, this fault combination is counted more than once. Due to looking into the traces directly, S# is not affected by this problem. Another source for the difference is that S# does not adhere to the "no miracles rule" of the fault tree analysis.[15] Simply put the "no miracles rule" states that if 1 fault would prevent a more severe situation then assume that this fault does not occur. Thus, when the blood pump is defective, and its defect would prevent that contaminated blood reenters the patient, then assume that the blood pump works correctly. S# does not adhere to this rule, because it looks into the traces directly.

Setting WaterHeaterDefect as the only transient fault and keeping all others as permanent faults gives us the following result:

$P_{S\#}$(Unsuccessful) = 0.014 (in 6 seconds), and

$P_{S\#}$(Contamination) = 2.0×10^{-4} (in 14 seconds).

Clearly, the probability of an unsuccessful dialysis reduces, when the fault is only temporary, and the hemodialysis machine can return to the normal operation when the water heater works again. In addition, the probability of the second hazard is reduced, because as the qualitative analysis showed previously, a defective water heater is only a problem when the safety valve is defective. This example also demonstrates that the precise quantitative analysis does not make the qualitative analysis redundant, because the minimal cut sets can help explain the results of the quantitative analysis.

Additionally, S# can check the influence of different fault probabilities on each hazard. Figure 9 shows the effect of the probability of a defect water heater. Therefore, for each graph, 25 analysis with different fault probabilities in the range from 10^{-3} to 10^{-1} have been conducted for each hazard. It seems that both graphs are almost linear.

The quantitative analysis of S# is not intended to replace traditional analysis but to serve as an additional tool for engineers. Its strength lies in the analysis of systems early in the development process. In early phases, a preliminary analysis of different design variants or different algorithms can be conducted with S#. In this phase, a traditional analysis would be too time-consuming and expensive to be applied on every variant because for each variant a separate fault tree needs to be created manually. The early rigorous analysis of design variants with S# can give differences in hazard probabilities even when traditional analysis would see no difference or expensive prototypes would be needed. Thus, S#'s quantitative analysis can make life easier for engineers. With S#, engineers do not need to know the interrelationship of different faults in these early phases. They just need to know the exact impact of each fault and S# derives the rest automatically.

9 | RELATED WORK

Other approaches also have features in their modeling language to describe faults and design alternatives. The Safety Analysis Modeling Language (SAML) is an extension of the PRISM input language[18,19] to facilitate its application for the analysis of safety critical systems. SAML adds safety-modeling features onto a state machine foundation. The SLIM language is another member of the state machine-based approaches.[20] It is derived from the standardized AADL language and offers native fault modeling. xSAP is an extension of the NuSMV language that allows the explicit modeling of faults in NuSMV and allows the automatic calculation of Minimal Cut Sets.[21]

By contrast, S# augments the industry standard language C# with fault modeling concepts. S# is more flexible than SLIM and SAML in the sense that it allows the behavior of a component to be modeled as either a state machine or as structured sequential code. Additionally, model structuring and composition in S# benefits from well-established object-oriented design principles and language features. The inherent executability of S# models results in a unified approach for model simulation, testing, debugging, visualization as well as rigorous model checking.[1]

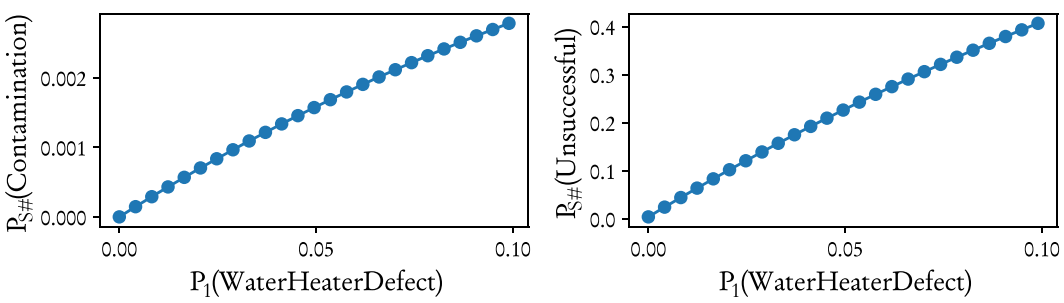

FIGURE 9 Illustration of the impact of the probability P_1(WaterHeaterDefect) Pr(*WaterHeaterDefect*) on each hazard

SYSTEMC extends C++ to make it possible to use C++ both as hardware description and as high-level modeling language.[22] The approach to use a mature programming language is very similar to ours. However, S# was designed with formal analysis, failures, and design exploration in mind and not for comprehensive hardware design.

MODELICA is a formalism that is well suited to model and simulate physical flows.[23] It allows sophisticated expressions of physical laws inside the modeling language. However, it was designed rather for simulation than for model checking. It provides no native means to model faults and execute a sophisticated DCCA analysis.

Furthermore, there exist various low-level tools for model checking, eg, NuSMV, SPIN, and UPPAAL.[24-26] Compared with them, S#'s explicit support for fault modeling facilitates the safety analysis using model checking.

The SCADE Suite has been designed to model safety critical systems. Like most other approaches, it has no support for fault modeling.[27] However, it can generate C and ADA code from the models, which can be compiled and deployed on real hardware. While this should theoretically be also possible with S#, we currently do not have any plans to implement a code transformation.

10 | CONCLUSION

In this paper, we have demonstrated how the S# toolchain could be used to model and analyze the impact of component faults in a hemodialysis machine. The possible faulty behavior in components and their interdependencies has made it essential to include the fluid flows into the model to allow an adequate analysis. It turned out that our approach is suitable for the safety analysis of this case study. The model has been created in approximately 3 person weeks by an expert in S#, who was previously unfamiliar in the medical domain. The current fault probabilities in our model are speculative and only serve the purpose to illustrate the feasibility of our approach. Our analysis is not complete. With the help of domain experts, a future analysis could integrate further models of component faults and more phases of the dialysis. Furthermore, an interactive visualization could also help when a S# model needs to be validated with experts not familiar with formal methods. Our model on http://safetysharp.isse.de already contains a prototype of such a visualization.

Current work on our toolchain includes the integration of the Luster language.[28] Luster is a dataflow language, which is used for creating software for safety critical systems. This integration is a step towards offering modelers a means to create their models visually. Furthermore, we are improving our means to be able to analyze models quantitatively, even when only some but not all fault probabilities are known. Then, S# can at least estimate the minimal and maximal probability of the hazard's occurrence. In addition, we want to add syntax to S# to be able to express the fault probabilities directly as Mean Time To Failure or Failure In Time. Our Wiki on http://safetysharp.isse.de provides more details on S#.

ORCID

Johannes Leupolz http://orcid.org/0000-0003-1710-9910

REFERENCES

1. Habermaier A, Leupolz J, Reif W. Unified simulation, visualization, and formal analysis of safety-critical systems with S#. In: *FMICS-AVoCS 2016*. Springer; 2016 DOI: https://doi.org/10.1007/978-3-319-45943-1_11.
2. Leupolz J, Knapp A, Habermaier A, Reif W. Qualitative and quantitative analysis of safety-critical systems with S#. In: *International Journal on Software Tools for Technology Transfer*. Springer; 2017 DOI: https://doi.org/10.1007/s10009-017-0464-3.
3. Habermaier A. Design time and run time formal safety analysis using executable models. PhD thesis. OPUS Augsburg, 2017. DOI:
4. Baier C, Katoen JP. *Principles of Model Checking*. MIT Press; 2008). ISBN. ISBN:9780262026499.
5. Kant G, Laarman A, Meijer J, van de Pol J, Blom S, van Dijk T. LTSmin: high-performance language-independent model checking. In: Tools and Algorithms for the Construction and Analysis of Systems—21st International Conference, TACAS 2015 2015. DOI: 10.1007/978-3-662-46681-0_61.
6. Katoen JP, Zapreev IS, Hahn EM, Hermanns H, Jansen DN. The ins and outs of the probabilistic model checker MRMC. In: International Conference on Quantitative Evaluation of Systems, QEST 2009. DOI: 10.1109/QEST.2009.11.
7. Visser W, Dwyer M, Whalen M. The hidden models of model checking. *Software & Systems Modeling*. 2012;11(4). https://doi.org/10.1007/s10270-012-0281-9
8. Habermaier A, Güdemann M, Ortmeier F, Reif W, Schellhorn G. The ForMoSA approach to qualitative and quantitative model-based safety analysis. In: Railway Safety, Reliability, and Security IGI Global 2012. DOI: 10.4018/978-1-4666-1643-1.ch004.
9. Habermaier A, Leupolz J, Reif W. Executable specifications of safety-critical systems with S#. *Proc of DCDS IFAC*. 2015;48(7):44-49. https://doi.org/10.1016/j.ifacol.2015.06.471
10. Habermaier A, Eberhardinger B, Seebach H, Leupolz J, Reif W. Runtime model-based safety analysis of self-organizing systems with S#. In: 2015 IEEE 9th Int. Conf. on Self-Adaptive and Self-Organizing Systems Workshops (SASOW 2015) 2015. DOI: 10.1109/SASOW.2015.26.
11. Mashkoor A. The hemodialysis machine case study. In Proceeding ABZ 2016 Proceedings of the 5th International Conference on Abstract State Machines, alloy, B, TLA, VDM, and Z. Springer 2015. DOI: 10.1007/978-3-319-33600-8_29-
12. Leupolz J, Habermaier A, Reif W. Safety analysis of a hemodialysis machine with S#. In: EuroAsiaSPI 2016 Industrial proceedings. Whitebox, 2016. ISBN: 9788799811663.
13. Curtis J, Delaney K, OKane P, Roshto B, Sweeney J. Hemodialysis devices. In: Core curriculum for the dialysis technician: a comprehensive review of hemodialysis, 4th edition. Medical Education Institute

14. Object Management Group: OMG Systems Modeling Language (OMG SysML), Version 1.4 2015. http://www.omg.org/spec/SysML/1.4/ [15 February 2016].
15. Vesely W, Dugan J, Fragola J, Minarick J, Railsback J. Fault tree handbook with aerospace applications. Technical report, NASA, 2002
16. Güdemann M, Ortmeier F, Reif W. Computing ordered minimal critical sets. In: Proceedings of the 7th Symposium on Formal Methods for Automation and Safety in Railway and Automotive Systems, FORMS/FORMAT 08 2008
17. Diez DM, Barr CD, Çetinkaya-Rundel M.OpenIntro statistics. OpenIntro, Incorporated, 2015. ISBN: 9781943450053.
18. Ortmeier F, Struck S, Güdemann M. Efficient optimization of large probabilistic models. *Journal of Systems and Software* 03/78. Elsevier. 2013;86:2488–2501.
19. Kwiatkowska M, Norman G, Parker D. PRISM 4.0: verification of probabilistic real-time systems. In: Proc. 23rd International Conference on Computer Aided Verification (CAV'11). LNCS, vol. 6806. Springer 2011. DOI: 10.1007/978-3-642-22110-1_47.
20. Noll T. Safety, dependability and performance analysis of aerospace systems. In: *Formal Techniques for Safety-Critical Systems: Third International Workshop, FTSCS 2014*. *Communications in Computer and Information Science*. Springer; 2015 DOI: https://doi.org/10.1007/978-3-319-17581-2.
21. Bittner B, Bozzano M, Cavada R, et al. The xSAP safety analysis platform. In Proceedings of TACAS 2016. MA, USA: Kluwer Academic Publishers Norwell.
22. Grötker T, Liao S, Martin G, Swan S. *System Design with SystemC*. 2002. MA, USA: Kluwer Academic Publishers Norwell.
23. Modelica association: modelica—a unified object-oriented language for systems modeling, Language Specification, Version 3.3 2014
24. Cimatti A, Clarke EM, Giunchiglia E, et al. NuSMV 2: an opensource tool for symbolic model checking. In Proceeding of International Conference on Computer-Aided Verification (CAV 2002). Copenhagen, Denmark
25. Holzmann GJ. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, ISBN; 2004. ISBN:0-321-22862-6.
26. Behrmann G, David A, Larsen GK. A tutorial on Uppaal. In proceedings of the 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems. LNCS 3185. Springer. 2004.
27. Vinter J, Bromander L, Raistrick P, Edler H. FISCADE—a fault injection tool for SCADE models. In: 3rd Institution of Engineering and Technology Conference on Automotive Electronics, Warwick, UK, 2007
28. Halbwachs N, Caspi P, Raymond P, Pilaud D. The synchronouse dataflow programming language LUSTRE. *Proceedings of the IEEE*. 79. https://doi.org/10.1109/5.97300