Defining requirements on domain-specific languages in model-driven software engineering of safety-critical systems

Michael Wasilewski[1], Wilhelm Hasselbring[2], Dirk Nowotka[3]

[1]Vossloh Locomotives GmbH, 24152 Kiel
http://www.vossloh-locomotives.com/
[2]Kiel University, Dept. Computer Science, Software Engineering Group, 24118 Kiel
http://se.uni-kiel.de/
[3]Kiel University, Dept. Computer Science, Dependable Systems Group, 24118 Kiel
http://zs.uni-kiel.de/

Abstract: Domain-specific languages are designed and used to assist software development in various domains. Safety-critical systems such as aviation systems, railway control systems and nuclear power plants require certified software by law. This paper focuses on domain-specific languages that are used to represent a physical reality and to describe the behavior of a control software as a finite state machine. Furthermore we focus on domain-specific languages that are able to generate source code for sensor/actor systems from a specified finite state machine model. The source code is intended to be compiled and operated in a fixed time slot of a real-time operating system of a safety-critical controlling hardware.

We give an example of a model that is expressed using a functional tree, a method that is based on input and state space partitioning. We show that models expressed by a functional tree are equivalent to deterministic and complete finite state machines. To formally prove the equivalence we analyze a model in terms of automata theory. We will furthermore show that omitting the properties of determinism and completeness violates normative requirements when a model is used to generate software for safety-critical systems.

The major contribution of this paper is the definition of formal requirements on domain-specific languages employing formalisms of automata theory. The requirements are easily verifiable criteria for domain-specific languages to assess the suitability in an engineering process of a safety-critical system. We analyze two example modeling languages for their suitability to create a source code for safety-critical applications.

1 Introduction

Model-driven software development uses a formal description of a model and code generators to obtain an executable software for specific purposes. The base for a formal description of a model can be either a domain-specific language (DSL) or a universal modeling language such as the UML. The design goal for a DSL is to cover specific problems and properties of its domain such as track topologies and signaling layouts for railroad infrastructures [GHH+12]. Increased attention has to be payed if DSLs are used to create

source-code to be used in safety-critical applications. Design properties of a DSL can lead to a risks for legal assets like life, healthy and property.

Our goal is to define requirements for domain-specific languages from which source code is generated for safety-critical applications. In Section 2 we present the scope of this paper. We will illustrate the use of a Functional Tree (FT) and Finite State Machine (FSM) as two possible ways to express a model of a signal processing system. In Section 3 we will give a brief introduction into expressing a model using FTs with reference to [WH11].

We will compare the expression of a model as FT and show the weaknesses of the equivalent expression as FSM. With an example model we show a simple method to obtain an alphabet from physical states and inputs in order to transform a model expressed by FTs into an automaton. We analyze in Section 4 which properties of a automaton are needed to represent a model as a FSM with the same formal rigor as a representation with a FT provides.

A formal definition of an automaton is introduced in Section 5 and an automaton type is selected for our goal to define formal requirements for a DSL. The analysis in Section 4 is the base for defining requirements on DSLs in Section 6. We will use the formalisms of automata theory to define requirements on a domain-specific language as the contribution of this paper. We will compare the defined requirements to legal requirements such as normative regulations for the certification of software in safety-critical systems and show potential violations.

Finally in Section 7 we analyze two languages based on our defined requirements. As examples we use the DSL MENGES [GHH+12] as a representative of a highly specialized DSL. As representative for a general-purpose language we analyze state charts in the UML. We show the weaknesses and which additional effort is needed for a suitable use of these languages for modeling of safety-critical systems.

2 Scope

The scope of this paper is the development process for the software of a safety-critical system as in Figure 1.

We assume that functional requirements for the software are defined and a DSL is used for their formal expression. We furthermore assume that a selected DSL's output model can be expressed as a FSM or a FT. For both options a specific code generator can create a source code that is passed to a target specific compiler. The finally generated software can be operated in a safety-critical system. Our goal to define formal requirements on a DSL for safety-critical systems is achieved with the following steps

1. We give an example and analyze (see 1 in Figure 1) the properties of a model that is expressed using a FT
2. We transfer the analyzed properties to a model that is expressed as a FSM and show that models without the analyzed properties violate legal requirements for software

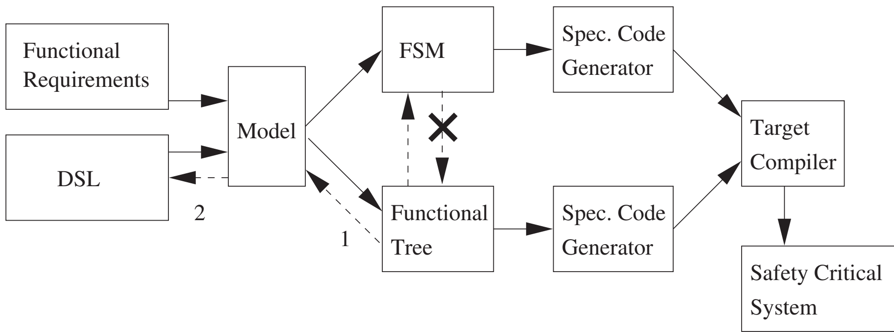

Figure 1: Scope of analysis

in safety-critical system

3. We define formal requirements on a DSL for model-driven software engineering of safety-critical systems to create a model such that legal requirements are not violated,(see 2 in Figure 1)

The reason for the choice of FTs is the method's strength that formal uniqueness and completeness of a designed model is implicitly given if it can be expressed by a FT. This property is given by the mapping of physical values as intervals on defined input and state space partitions of a model. Formal uniqueness and completeness of expressing functions or models by FTs are proven with the set theory. The formalism of this method was first time introduced in the context of a rail vehicle project in [WH11]. We will show that a model expressed by a FT has an equivalent expression by a FSM, but only a limited class of FSMs can be expressed by a FT (see Figure 1). We will furthermore show that the classes of FSM's that cannot be expressed by FTs violate normative requirements for safety-critical software.

3 Background and formal basis

In a first step we design an example model of a function for the control of a diesel engine as in Figure 2. We use a FT as expression method to explain the background of this paper. We will present an equivalent expression of our designed model as FSM to show the weaknesses of a FSM expression. Our goal is to find the differences of both expression methods by an analysis of our model as an automaton. We will introduce the formal definitions needed for the analysis of the designed model in terms of automaton type and an alphabet. Finally we will analyze the properties of our model considering the following aspects

- What kind of automaton is created if it is expressible by a FT ?

Variable	Value	Meaning
S (discr.)	S_{halt} S_{start} S_{run} S_{stop} S_{Ω}	Engine halted Engine starting Engine running Engine stopping Unknown state
AS (bool)	TRUE FALSE	Switch ON Switch OFF
ST (bool)	TRUE FALSE	Start No Start
n_{rpm} (cont.)	R_{LO} : $n_{rpm} < 1$ R_0 : $1 \geq n_{rpm} < 400$ R_{HI} : $n_{rpm} \geq 400$	Engine halted Engine turning Engine running

Table 1: Variable partitions

- Which is the alphabet of the automaton ?
- Which language is accepted by the automaton ?
- Which are the differences of a model expression by a FSM and by a FT ?

3.1 Example model design and functional trees expression

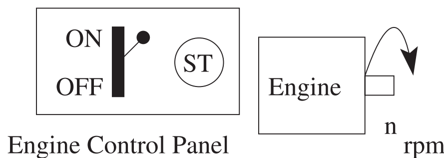

Figure 2: Example of an Engine Control

The system in our example in Figure 2 consists of an Engine Control Panel and the engine. The input values for our model are the activation switch (AS) and the starter button (ST) on the Engine Control Panel and the engine speed (n_{rpm}). Our design starts with the decisions which input and state values we use in our model and how the input and state values are represented and divided into partitions. For an expression of the model by a FT the partitioning is the basic step.

We will design our model such that the starter button (ST) and activation switch (AS) are represented by boolean values with the partitioning into TRUE/FALSE. The engine speed (n_{rpm}) is represented by a continuous value with a partitioning into intervals. We will furthermore define that our engine can be in the states S_{halt} if the engine is halted, S_{start} if the engine is commanded to start, S_{run} if the engine is running and S_{stop} if the engine

is commanded to stop. For any other state we define S_Ω as the 'unknown state'. The input and state variables and their partitions for our model are shown in Table 1. To design the model of our engine control function we have to define the conditions for the changes of the state variable S from one of it's partitions into another one. This step is equivalent to the design of a FSM and the definitions of the transitions from one state to another. To show the capability of using FTs as an expression for our model we use an example as in Figure 3.

This FT shows an example of a model which starts by processing state variable S with all of it's partitions as in Table 1. The partitions are represented by the edges of the tree. Depending on the partitions and the values of subsequent variables the edges are taken to the subsequent nodes (circle) until a leaf (square) is reached. The leafs (squares) of the tree contain the states into which our model changes after one computation step. So every path in this FT is equivalent to a transition of a FSM.

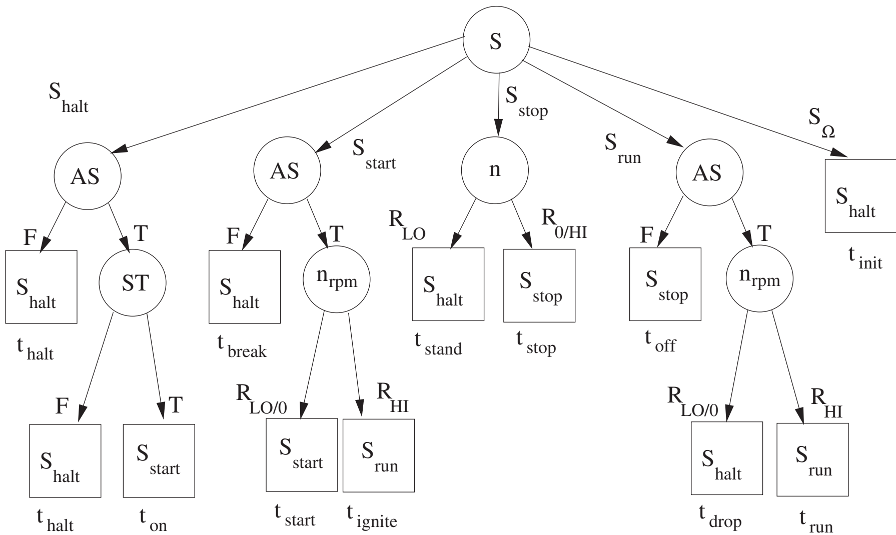

Figure 3: Functional tree for Engine Control

We can see that the different paths in our FT depend on different input variables and different processing orders. This is equivalent to different conditions for a transition of a FSM. For example the transitions t_{stand} and t_{stop} are valid for a stopping engine (S_{stop}) and only depend on the speed (n_{rpm}) to determine if the engine is already stopped ($n_{rpm} \in R_{LO}$) or still stopping($n_{rpm} \in (R_0 \cup R_{HI})$). As second example the transitions t_{halt} and t_{on} are valid for a halted engine (S_{halt}). It is started (t_{on}) or remaining in standstill (t_{halt}) depends on the inputs of the activation switch (AS) and the start button (ST). The conditions for all transitions of this model can be found in Table 3 when we analyze our model.

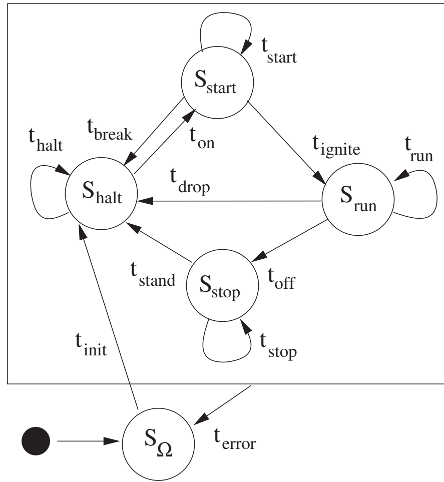

Figure 4: Finite State Machine for Engine Control

Our example shows that if a variable is considered in a FT to determine a result then it is considered with all it's partitions (completeness). If a variable is not considered in a FT then the result is the same for all of it's partitions (independence). The characteristic property of a FT is that every value of a variable has to be and can only be element of exactly one of the variable's partitions. For the expression of a model with a FT it means that every combination of input and state space corresponds to exactly one path in the FT. The consequence is that only models can have an expression as a FT which correspond to a deterministic and complete automaton. We will formally prove this property in this paper.

As shown in Figure 1 we assumed that a model expressed as FT can be also expressed as FSM. For our example, the FT in Figure 3 is equivalent to the FSM in Figure 4. Note that the expression as FSM shows the transition t_{error} to represent an unknown error that can corrupt the state variable of the model (e.g. bad memory access, hardware error etc.). This transition is not shown in the FT. The reason is that both expression FT and FSM cannot specify a defined condition for this transition. Important is that both expressions show the same reaction of the model if the state variable is in the 'unknown state' S_{Ω}

The expression as FSM shows the states and the transitions of the model as shown in Table 3. The conditions for every transition can be defined, but there is no automatic mechanism to ensure that all combinations of input signal are be processed by the FSM in every state. There is also no automatic mechanism that ensures that determinism for the transition is specified. To ensure uniqueness and completeness of a FSM additional verification effort is required in the design process while using a FT implies these properties. If a DSL is involved in the model design for a safety-critical system we demand that the DSL's output is a formally complete and unique model.

With FTs a designer has to start the modeling process with a correct partitioning of input and state variables for a model. Extensions, refinements and changes of the model design are expressed as changes in the paths of the FT for the model. The properties of FTs as expression method for a model ensure that for all combinations of input and state variables the designed model is formally unique and complete. The method gives a designer the choice in which context physical values are used to compute a result or not. This makes the method easily scalable for higher numbers of physical values. Verification efforts and finally development costs are only limited to verify the correctness of the designed model to meet the functional requirements, as shown in in Figure 1.

The verification of correctness of a model expressed as FSM is often limited to only cover the functional requirements. A verification of formal uniqueness and completeness of all input and state value combinations is often not done. The reason is that no selection mechanisms for relevant combinations of variables and states exists comparable to the partitioning mechanism of a FT. A non-deterministic FSM can be in theory transformed into a deterministic FSM by using it's power set. Completeness and determinism can be verified then, but this will increase verification effort and development costs significantly.

Our model is intended to be used safety-critical applications. Mistakes such as incomplete or contra-dictionary definitions during the model design can result in severe hazards for human life or material assets. Normative regulations such as [CEN01b](see 8.4.2), [IEC11](see 7.2.2 and 7.4.5) require a unique and complete functional specification of software for safety-critical applications. We have demonstrated that expressing a model with a FT is limited to models that correspond to a deterministic and complete automaton, but this is exactly the type of model we need to meet the requirements of software in safety-critical systems. Our goal will be therefore to define formal requirements for a DSL such that the output model will have a defined behavior for every combination of inputs and states and be expressible as a FT.

3.2 Functional tree expressed model as automaton

For the statement of formal requirements on DSLs we will analyze a designed model that we expressed using a FT as an automaton. The automata theory uses sets of states and symbols to describe alphabets and languages that are accepted by an automaton. The basic elements of FTs are disjoint sets to describe the processed signals. We will connect both concepts.

In a first step we will describe the relation between the disjoint sets of value partitions as used for a FT and the sets of input symbols of an automaton, as in Table 2. We have listed all input space partition combinations of our example model in Figure 4. Every input space partition combination is assigned to one symbol s_1 to s_{12}. We obtain a bijective function between our model's input space partition combinations and the symbols of an formal language. This approach can be extended to any number of input variables such that we obtain a partitioning of the input space of a model and one corresponding language symbol for every partition. Similarly the partitions of one or more state variables provide

a partitioning of the states space of a model. Every partition of the state space of the model will correspond to one state of an automaton.

The analysis of a model that is expressed as a FT is based on the input space $\mathbb{I}$, the state space $\mathbb{S}$ of the model and a definition of functional independence. The input and the state spaces are defined based on the signals that are processed in the computation (note that variables over continuous domains are discretized by considering intervals):

$$\mathbb{I} \subseteq (\mathbb{R} \times \mathbb{R})^{i} \times \mathbb{Z}^{j} \times \mathbb{B}^{k}$$
$$\mathbb{S} \subseteq (\mathbb{R} \times \mathbb{R})^{i'} \times \mathbb{Z}^{j'} \times \mathbb{B}^{k'}$$

where $\mathbb{I}$ and $\mathbb{S}$ are finite and i, i', j, j', k, k' are naturals. By common notation, $\mathbb{R}$ denotes the set of reals, $\mathbb{Z}$ denotes the set of integers, and $\mathbb{B}$ denotes a binary set. Our example in Table 1 uses just one state variable, namely S. We use three signals in our example, namely AS, ST and n_{rpm}.

AS	ST	n_{rpm}	Symbol
OFF	T	R_{HI}	s_1
OFF	T	R_0	s_2
OFF	T	R_{LO}	s_3
OFF	F	R_{HI}	s_4
OFF	F	R_0	s_5
OFF	F	R_{LO}	s_6
ON	T	R_{HI}	s_7
ON	T	R_0	s_8
ON	T	R_{LO}	s_9
ON	F	R_{HI}	s_{10}
ON	F	R_0	s_{11}
ON	F	R_{LO}	s_{12}

Table 2: Symbol table

For the specification of a function in our context we use a set of n variables x_0 to x_n of continuous, discrete or binary type. A function of an automaton (e.g. transition, output) will be defined as $f(x_0 \dots x_n)$. We define functional independence of a function $f(x_0 \dots x_k \dots x_n)$ from the variable x_k if the following holds:

$$\forall x\colon f(x_0 \dots x_k \dots x_n) = f(x_0 \dots x \dots x_n)$$

This definition of functional independence is required for a formally correct split of the input and state space $\mathbb{S} \times \mathbb{I}$ into partitions. The connection of a FT as in Figure 3 and the symbols, states and transitions of an automaton is shown in Table 3. Each row of the table represents one path of the FT. We can also see that each row represents one of the partition P_1 to P_{12} of our model's input variable (AS, St, n_{rpm}) and state variable (S) space.

Considering the functional table 3 as state and transition table of an automaton we can find the automaton states as partitions of the state variable S of the functions tree. For comparison see Figure 4. The automaton transitions correspond to paths of the FT. Note that the transition t_{halt} is mentioned two times in Table 3 and two times in the FT, see Figure 3 while it is only mentioned once on the FSM expression of the model, see Figure 4. Referring to our example model the technical meaning is that the start of the engine can be blocked by two conditions, namely the activation switch turned off (AS=OFF) or the starter button not pushed (ST=F).

Not all transitions from one state to another depend on all input variables while all input variables have to be considered for a complete and unique design of the model. The $\forall$ symbol in a row means that an input variable has not to be considered, as the final result is the same for any of it's values. In the FT expression of a model the corresponding variable

Partition	State S	Inp. AS	Inp. ST	Inp. n_{rpm}	Transition	Symbol
P_1	S_{halt}	OFF	$\forall$	$\forall$	t_{halt}	s_1,s_2,s_3 s_4,s_5,s_6
P_2	S_{halt}	ON	F	$\forall$	t_{halt}	s_7,s_8,s_9
P_3	S_{halt}	ON	T	$\forall$	t_{on}	s_{10},s_{11},s_{12}
P_4	S_{start}	OFF	$\forall$	$\forall$	t_{break}	s_1,s_2,s_3 s_4,s_5,s_6
P_5	S_{start}	ON	$\forall$	R_{HI}	t_{ignite}	s_7,s_{10}
P_6	S_{start}	ON	$\forall$	$R_{LO/0}$	t_{start}	s_8,s_9,s_{11},s_{12}
P_7	S_{stop}	$\forall$	$\forall$	$R_{0/HI}$	t_{stop}	s_1,s_2,s_4,s_5 s_7,s_8,s_{10},s_{11}
P_8	S_{stop}	$\forall$	$\forall$	R_{LO}	t_{stand}	s_3,s_6,s_9,s_{12}
P_9	S_{run}	OFF	$\forall$	$\forall$	t_{off}	s_1,s_2,s_3 s_4,s_5,s_6
P_{10}	S_{run}	ON	$\forall$	R_{HI}	t_{run}	s_7,s_{10}
P_{11}	S_{run}	ON	$\forall$	$R_{LO/0}$	t_{drop}	s_8,s_9,s_{11},s_{12}
P_{12}	S_Ω	$\forall$	$\forall$	$\forall$	t_{init}	s_7,s_8,s_9 s_{10},s_{11},s_{12}

Table 3: Functional table

is not in the path that represents the transition. For example the transition t_{halt} of partition P_2 is independent of the variable n_{rpm}. In the FT of the model Figure 3 the variable n_{rpm} is not in the corresponding path. Considering the model as an automaton we need to select the input symbols for transition t_{halt} such that we cover the corresponding input space partitions as in Table 2. In our example the symbols s_7,s_8,s_9 have all in common the input variables AS=ON and ST=F while all together cover all partition of n_{rpm}.

With the definition of functional independence we can use the symbols as in Table 2 for a formally correct transformation of a model expression as FT into a model expression as an automaton. Our assumption is proved that every model expressed as FT can be expressed as a FSM. For the partitions P_1 to P_n of the FT and the input and state space of our model $\mathbb{S} \times \mathbb{I}$ holds:

$$\bigcup_{k=1}^{n} P_k = \mathbb{S} \times \mathbb{I}$$

Furthermore for every partition of the state variable S we can find all input symbols of the automaton as in the Symbol column in Table 3 exactly once. The formal definition of functional independence says that not considering a variable in a computed result is equivalent to a result that hold for all values of the variable. This proves that an automaton that corresponds to model expressed as FT is deterministic and complete.

4 Properties of an automaton obtained via functional trees

Our analysis starts on the base of a transition system (TS) as defined in [BK08], Chapter 2. We focus on the states and transitions of an automaton so the mentioned TS is an appropriate base. We will consider the processing of a new input symbol as a transition from one state to another state.

The analysis is based on the assumption that an automaton is transformed into source code that is finally compiled and executed in a safety-critical system. The execution of the source code is steadily repeated in fixed time slots and potentially never terminates. We will analyze the properties of an automaton that is obtained by designing a model using a FT. The goal is to express the found properties in terms of automata theory [Sak09].

A FT model creates an automaton processing a finite length input
: This property results from the fact that the automaton created by a FT runs for a potentially infinite time. However in every processing cycle a new symbol of the input space is processed which corresponds one transition of the automaton. The finally processed sequence of input symbols keeps a finite length in every processing cycle.

The automaton is deterministic
: The determinism of the automaton is given by the uniqueness property of the FT. For *each* input symbol and state there is at most one transition. The automaton has therefore at most one transition for each input element at each state.

The automaton is complete
: The completeness of the automaton is given by the completeness property of the FT. For *every* input symbol and state combination there is exactly one transition. A preserved state is expressed by a reflexive transition. The automaton has therefore exactly one transition for each input symbol. In safety-critical applications the complete behavior of an automaton has to be specified, verified and tested. That includes implicit reflexive transitions.

The automaton provides a specified result for every processed input sequence
: The FT formal base ensures that an automaton provides a specified transition for every partition of the input and state space. The output of the automaton is a specified, verified and tested function for every computation cycle. This property is required for operation in a safety-critical system.

Note that, a notion of automaton is not yet given here, but will follow in the next section. The property to provide a specified result for every partition of the input and state space of the automaton is the key point for the definition. Our automaton can, in principle, be operated with any sequence of input symbols. However it is required that the automaton's behavior is always within the specified scope. For the further analysis we therefore assume that our automaton processes any sequences of input symbols. This assumption is the base for our selection of an automaton model.

5 Selection of automaton type

In Section 4 we have obtained the properties for an automaton is the output of a model design using a FT. The analysis showed that we require a complete, deterministic automaton with output to describe the automaton investigated. A complete, deterministic automaton A with output is defined as follows (see also [AS03]):

$$\mathcal{A} = (\Sigma, X, Z, z_0, \delta, f) \tag{1}$$

where Σ denotes the set of inputs, X denotes the set of possible outputs, Z denotes the set of states, $z_0 \in Z$ denotes the start state, $\delta\colon \Sigma \times Z \to Z$ denotes the transition function and $f\colon Z \to X$ denotes the output function of the automaton. As usual, we extend δ on the set of all finite sequences over Σ as follows. Let $\delta'\colon \Sigma^* \times Z \to Z$ be such that $\delta'(\sigma' a, z) = \delta(a, \delta'(\sigma', z))$, where $\sigma' \in \Sigma^*$ and $a \in \Sigma$ and $z \in Z$. The output of the automaton is taken by f from the state the automaton ends in after reading a sequence σ, formally, $f(\delta'(\sigma, z_0))$. This type of automaton meets our key requirement to provide a specified result for every combination of input symbol and automaton state. This rigorous requirement result from normative regulations for software in safety-critical applications. The behavior of software has to be completely specified and evaluated for potential safety hazards. The use of the usual semantics of Mealy automata to ignore unspecified input would provide the same result, but violate the requirement for a complete specification. It is therefore not suitable for evaluation purposes of safety-critical software.

6 Requirements on DSLs for safety-critical systems

For the definition of the formal requirements on a source code generating DSL we refer to an automation as in Equation 1 of Section 5. Based on our analysis of automaton properties we define the following formal requirements for the use of DSLs and their generated models in a safety-critical system. We refer to the output of a DSL as a model as we have seen that a model can be expressed in different ways, namely FSM or FT.

A bijective function has to exist between the physical value and state space partitions and a model's input and state space partitions: A DSL has to provide a formal mechanism for the transformation of physical input and state partitions (such as intervals of sensor values, switch states, etc.) and a model's input and state space partitions. It has to be ensured that every partition of the input and state space of a model has a physical correspondence – even if the correspondence is identified as invalid.

The generated model has to be a non-terminating transducer: The generated model has to process inputs of potentially infinite length. However every received new input symbol since the start of the processing keeps the already processed word's length finite.

Partition	Physical Reality
$R_{LO} : T < -273°C$	Valid Temperature
$R_{HI} : T \geq -273°C$	Sensor fail

Table 4: Example of bijective function between partitions and physical reality

The generated model has to be a deterministic automaton: For application in a safety-critical systems the behavior of software has to have a unique description for the assessment of potential risks. For any symbol of Σ there has to be exactly one transition.

The generated model has to be complete automaton: For application in a safety-critical system the behavior of software has to have a complete description for the assessment of potential risks. For every symbol of Σ there has to be an element in $\delta : \Sigma \times Z \rightarrow Z$. The state machine is in continuous computation in a running system. We also require the the model for all states and inputs to be provide a specified result to be suitable for the use in safety-critical systems.

These requirements are based on the established formalism of automata as theoretical foundation and can be easily verified in practical use. Our defined requirements can be well justified by normative regulations that are demanded by law for software in safety-critical systems [CEN01b, IEC11]. The bijective function between physical input and state space partitions and automaton symbols and states is a formal criterion for the transformation of a physical reality into a model.

An example for such a transformation is the representation of a temperature in °C as signed integer as in Table 4. A value $T \geq -273°C$ represents a valid temperature measured by a sensor as physical reality while a value $T < -273°C$ is the physical reality of a sensor fail. We want to evaluate for every partition of our model whether the input and state values have a physical sense. We want furthermore specify a reaction of the software for every partition we identify as physically invalid, inconsistent or erroneous.

If a DSL's output is an automaton that ignores an input (such as a Mealy automaton can do) it violates the legal requirement for a complete specification. If the DSL's output is a non-deterministic automaton it violates the legal requirement for a unique specification that can be evaluated for safety hazards. The safety of software should not rely on potentially incorrect expectations about a physical reality and erroneous model design. Our approach automatically covers effects of component failures (such as invalid sensor data, unexpected data memory changes, etc.) that have to be evaluated during the system development process [CEN01a]. The computation of a model itself has to be specified, verified and tested with the required rigor for safety-critical systems.

The requirement for a non-terminating transducer results from the potentially infinite runtime of an embedded system. However, in practice the time between a system start and any processing cycle will be finite. This justifies to considers a signal processing system

Listing 1: MENGES example

```
process SwitchChange{ start check_lock;
    condition check_lock{
        case this.locked: release_lock()
        default: start_SwitchMotion() }
    action start_SwitchMotion() {
        ...
        continue SwitchCheckPosition}}
```

as an automaton processing words of a finite length avoiding problems with processing infinite formal objects.

7 Property analysis of existing modeling languages

We analyze two representatives of modeling languages based on our formal requirements for DSLs being used in safety-critical applications. One modeling language (MENGES project) is designed highly specific for the domain of rail signaling. Another example is UML which serves as a general-purpose language to describe models.

7.1 DSL of the MENGES project

The DSL of the MENGES project [GHH^{+}12] was designed to describe rail signaling infrastructures. This DSL is able to describe the topology of a specific installation based on object instances of "interlocking elements" such as switches. The communication between interlocking elements is based on a role model and defined interaction protocols. The logic of interlocking elements is specified using finite state machines or processes. The elements to describe a logic are actions and conditions.

For our analysis we take a closer look at a process at the example in Listing 1. This example describes the process to change the position of a switch. It starts with a check if the switch is locked. A locked switch triggers an action to release the switch otherwise the motion of the switch is started. The motion is not mentioned in detail here. The process terminates and a following process to check the switch position is started.

The presentation paper of the DSL of the MENGES project [GHH^{+}12] mentioned that a process can block during evaluating a condition. To realize a non-blocking process the definition a **default**-action is required. We have considered a process as a FSM. The evaluation of conditions is equivalent to the processing of different input symbols. Our analysis showed that the possibility of a process to block is equivalent to the processing of an unspecified input. The requirement for a total finite state machine was violated. Our analysis resulted in a change of the DSL design such that a **default**-action is mandatory.

Based on our formal requirements for DSLs in safety-critical systems, we have identified the model design as the critical point if the DSL of the MENGES project is used in a software development process as in Figure 1. The challenge is a suitable choice of states and input to represent rail infrastructure elements. The requirement for a bijective function between the real rail infrastructure elements and their representation in a software is a possible point for future work. Requirements for determinism, a total transitions function and specified processing of all input and state space partitions can be fulfilled, but the verification of these properties is not withing the scope of the language. Also this is a challenge for future work.

7.2 State charts in the UML

The Unified Modeling Language [Oes12] is able to specify hierarchical state machines as State charts. State charts contain states and transitions and represent an object life cycle. Actions can be defined for state entry, exit and persistence in a state. Transitions are only labels and can be described as external, implicit transitions or as so called "guards" to detect a defined conditions.

UML was designed to be processed by other tools such as code-generators to transform the models into other structures. This language was not explicitly designed to assist software development for safety-critical applications, but is however used for this purpose. Additional effort such as definitions of subsets of the language and verification techniques are needed to achieve a suitable usability of UML to assist the development of safety-critical software. Our formal requirements can be criteria for a rigorous check of language subsets or designed models. Future work can show how the formal requirements are realized with UML.

8 Related Work

We have defined our requirements on a domain-specific language for safety-critical systems without using a formal language. The problems of obtaining complete models for control systems and the limits of language parsing finite state automata are discussed in [WWW04]. This work identifies the problems without formalization, but in the context of the tool StateWorks. Our work uses a formal definition of completeness based on the set theory [WH11].

Different to [Lat03] and [KL06] we do not express "safety properties" as properties of a formal language. We use the automata theory as in [Sak09] with focus on computation mechanisms and the ability of a FSM to provide a specified result for any input. For the suitability in safety-critical systems we require a FSM to generate an output that is considered safe for every computation step. Our approach to achieve a safe behavior in a project context is based on a formally complete specification of a model providing a specified output.

Similar to purposes of runtime verification [FFM09, BGHS04] and model-checking [BBB+04, BK08] our work is focused on software with a potentially infinite runtime. We however use different different types of automata for the design of a FSM model than for the verification. Runtime verification and model-checking require decisions whether formally expressed properties are fulfilled or not. Such decision making processes and algorithms are based on infinite formal expressions and Büchi automata [Büc62]. The focus of our work is the design of FSMs. Our intention is to introduce well-researched formalisms, properties and methods for domain-specific languages to support the design of FSMs for safety-critical systems in industrial settings.

9 Conclusions and Outlook

The base of our investigations was the design of model with the formalized method of functional trees, that is already applied in industry [WH11]. We shown that a designed model with functional trees has an equivalent expression as FSM, but is more suitable for the use in safety-critical applications. The rigorous formalism of functional trees ensures that for every combinations of input and state values the model provides a specified, verified and tested result.

We have demonstrated a method to express a functional tree designed model as an automaton with an alphabet and states. The properties of the automaton are the base to define formal requirements on domain-specific languages for model-driven software engineering safety-critical systems. Our analysis showed that a model has to be able to provide a specified result for every element of it's input and state space. If a domain-specific language's model output is a FSM it has meet the following formal requirements:

- There has to be a bijective function from physical input intervals and states to model inputs and states – a simple realization is possible by space partitioning.
- The FSM has to be non-terminating, deterministic and complete. A rigorously specified, verifiable and testable reaction on invalid, erroneous or inconsistent input is required.

Our defined requirements are based on well-understood properties and an established formalism: automatons and their expression as FSMs. The key point is that these requirements are easily verifiable. With our defined requirements, existing DSLs and modeling languages can be evaluated whether they are suitable for the use in the development process of safety-critical systems. For the design of future DSLs these requirements can be a base and existing DSLs can be evaluated for their suitability using our criteria. Future work furthermore addresses analyzing the impact of our work on model checking [BBB+04] and source code instrumentation [FHRS07, vHKGH11] for runtime verification techniques.

References

[AS03] Jean-Paul Allouche and Jeffrey O. Shallit. *Automatic Sequences - Theory, Applications, Generalizations*. Cambridge University Press, 2003.

[BBB+04] R. Buschermöhle, M. Brörkens, I. Brückner, W. Damm, W. Hasselbring, B. Josko, C. Schulte, and T. Wolf. Model Checking – Grundlagen und Praxiserfahrungen. *Informatik-Spektrum*, 27(2):146–158, April 2004.

[BGHS04] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-Based Runtime Verification. In Bernhard Steffen and Giorgio Levi, editors, *VMCAI*, volume 2937 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2004.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[Büc62] J. R. Büchi. Proc. Internat. Congr. on Logic, Methodology and Philosophy of Science 1-11. In Nagel et al., editor, *On a decision method in restricted second-order arithmetic*. Stanford Univ. Press, 1962.

[CEN01a] CENELEC. *EN50126 - Railway applications: the specification and demonstation of Reliability, Availabity, Maintainability and Safety*. CENELEC, 2001.

[CEN01b] CENELEC. *EN50128 - Railway Applications: Software for Railway Control and Protection Systems*. CENELEC, 2001.

[FFM09] Ylies Falcone, Jean-Claude Fernandez, and Laurent Mounier. Runtime Verification of Safety Progress Properties. In *Runtime Verification 2009*, Lecture Notes in Computer Science, Grenoble, France, June 2009.

[FHRS07] Thilo Focke, Wilhelm Hasselbring, Matthias Rohr, and Johannes-Gerhard Schute. Instrumentierung zum Monitoring mittels Aspekt-orientierter Programmierung. In *Tagungsband Software Engineering 2007*, volume 105 of *LNI*, pages 55–58, 2007.

[GHH+12] Wolfgang Goerigk, Wilhelm Hasselbring, Gregor Hennings, Reiner Jung, Holger Neustock, Heiko Schaefer, Christian Schneider, Elferik Schultz, Thomas Stahl, Reinhard von Hanxleden, Steffen Weik, and Stefan Zeug. Entwurf einer domänenspezifischen Sprache für elektronische Stellwerke. In Stefan Jähnichen, Axel Küpper, and Sahin Albayrak, editors, *Software Engineering*, volume 198 of *LNI*, pages 119–130. GI, 2012.

[IEC11] IEC. *IEC 61508 - Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems (E/E/PE, or E/E/PES)*. IEC, 2011.

[KL06] Orna Kupferman and Robby Lampert. On the construction of fine automata for safety properties. In *In Proc. 4th ATVA, LNCS 4218*, pages 110–124, 2006.

[Lat03] Timo Latvala. Efficient Model Checking of Safety Properties. In *In Model Checking Software. 10th International SPIN Workshop*, pages 74–88. Springer, 2003.

[Oes12] Bernd Oestereich. *Analyse und Design mit UML 2.5*. Oldenbourg Verlag, 10th edition, 2012.

[Sak09] Jacques Sakarovitch. *Elements of Automata Theory*. Cambridge University Press, 2009.

[vHKGH11] André van Hoorn, Holger Knoche, Wolfgang Goerigk, and Wilhelm Hasselbring. Model-Driven Instrumentation for Dynamic Analysis of Legacy Software Systems. In *Proceedings of the 13th Workshop Software-Reengineering (WSR 2011)*, pages 26–27, May 2011. (Softwaretechnik-Trends 31(2) (May 2011) 18–19).

[WH11] Michael Wasilewski and Wilhelm Hasselbring. A Formal and Pragmatic Approach to Engineering Safety-critical Rail Vehicle Control Software. In *Software Engineering*, volume 183 of *LNI*, pages 99–110. GI, 2011.

[WWW04] Ferdinand Wagner, T. Wagner, and Peter Wolstenholme. Closing the Gap Between Software Modelling and Code. In *ECBS*, pages 52–60. IEEE Computer Society, 2004.