Towards Dynamic Dependable Systems through Evidence-Based Continuous Certification

Rasha Faqeh, Christof Fetzer, Holger Herrmanns, Jörg Hoffmann, Michaela Klauck, Maximilian Koehl, Marcel Steinmetz, Christoph Weidenbach

Towards Dynamic Dependable Systems through Evidence-Based Continuous Certification*

Rasha Faqeh[1], Christof Fetzer[1], Holger Hermanns[2,3], Jörg Hoffmann[2], Michaela Klauck[2], Maximilian A. Köhl[2], Marcel Steinmetz[2], and Christoph Weidenbach[4]

[1]Technische Universität Dresden, Dresden, Germany
[2]Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
[3]Institute of Intelligent Software, Guangzhou, China
[4]Max Planck Institute for Informatics, Saarland Informatics Campus, Saarbrücken, Germany

Abstract. Future cyber-physical systems are expected to be dynamic, evolving while already being deployed. Frequent updates of software components are likely to become the norm even for safety-critical systems. In this setting, a full re-certification before each software update might delay important updates that fix previous bugs, or security or safety issues. Here we propose a vision addressing this challenge, namely through the evidence-based continuous supervision and certification of software variants in the field. The idea is to run both old and new variants of component software inside the same system, together with a supervising instance that monitors their behavior. Updated variants are phased into operation after sufficient evidence for correct behavior has been collected. The variants are required to explicate their decisions in a logical language, enabling the supervisor to reason about these decisions and to identify inconsistencies. To resolve contradictory information, the supervisor can run a component analysis to identify potentially faulty components on the basis of previously observed behavior, and can trigger micro-experiments which plan and execute system behavior specifically aimed at reducing uncertainty. We spell out our overall vision, and provide a first formalization of the different components and their interplay. In order to provide efficient supervisor reasoning as well as automatic verification of supervisor properties we introduce SupERLog, a logic specifically designed to this end.

Keywords: Certification · Dependability · Model Checking · Planning · Supervision.

* Authors are listed alphabetically. This work was partially supported by the ERC Advanced Investigators Grant 695614 (POWVER), by DFG Grant 389792660 as part of TRR 248 (see `https://perspicuous-computing.science`), and by the Key-Area Research and Development Program Grant 2018B010107004 of Guangdong Province.

1 Introduction

The complexity of constructing dependable systems is increasing dramatically, as future cyber-physical systems – like those used in the context of autonomous vehicles – are expected to change *dynamically*: they need to evolve not only to changing needs, but also according to lessons learned in the field. Systems where *software components* are constantly updated in the field are likely to become the norm, together with a general acceptance of the fact that it seems outright impossible to upfront guarantee safety-by-design. Over the last years, Tesla has been pioneering this approach with apparent success [34,24] and other companies will follow.

Safety certification of such systems requires a process combining formal guarantees, statistical methods, in-field testing, simulation and proven-in-practice arguments. This is doomed to be a lengthy process. The update of any component requires, however, a new certification so as to assure absence of safety violations. Such a re-certification is costly and might delay important updates that are meant to fix previous bugs. Rapid deployment of fixes is imperative especially for security-relevant bugs since adversaries notoriously attempt to attack systems that are not yet patched, and because security loopholes make any safety guarantee void.

In this paper, we are thus exploring how to enable the immediate deployment of new software variants without time-intensive re-certification. Software updates might take the form of device drivers and microcode updates relating to hardware components, so we are factually facing variability of software as well as hardware(-induced) behavior.

In this setting, the primary goal is to ensure that new variants for system components do not cause safety violations, e.g., by the introduction of new bugs. Our vision is the *evidence-based continuous supervision and certification of software variants* in the field. At its core is the idea to run multiple variants of the component software inside the same system, together with a *supervising instance* that monitors and compares the variants' behaviors. The supervising instance itself is trusted, i.e., it is formally verified. Ideally, the verification can be done fully automatically. Furthermore, the supervisor reasoning needs to be effective and efficient to fulfill the monitoring and decision task of the supervisor. The logic SupERLog, (Sup)ervisor (E)ffective (R)easoning (Log)ics, supports all these requirements: it is decidable, so properties of the supervisor can be automatically verified. SupERLog enables fast reasoning on ground facts, the typical situation when the supervisor is run on the evidence (ground facts) provided by the different components. The supervisor itself is expected to not evolve regularly during the lifetime of the system. If it were to evolve, this would require a re-certification. The supervisor can be designed *fail-operational*, i.e., it tolerates failures and hence is considered not to be a single point of failure in the system.

Note that the supervisor can not use the traditional majority voting to handle the variants because the variants are expected to be non-deterministic in behavior. Therefore, there might not be any majority despite all variants being correct. And even if a majority exists, it is not guaranteed that the majority is

correct. In our approach, each variant has to deliver evidence for its decisions and a majority that does not provide sufficient evidence will lose to a minority that provides sufficient evidence.

To explain why variants need to be treated as non-deterministic, consider that a new variant might be putting different emphasis on the different sensor types like video vs. LIDAR signals. Moreover, a new variant might have added new features or have fixed some bugs or deficiencies in the older variants. However, a new variant might also introduce new bugs that could violate the safety of the system.

Our approach is meant to enable the continuous certification of variants of component software, by collecting in-field evidence demonstrating the safety of new variants while at the same time using the older variants to safeguard critical activities. Of course, the older variant of a component software is not always the one to trust. In order to determine the ground truth despite component misbehavior, we envision an approach to resolve contradictory information arriving from different components and their variants: a *component analysis* and *micro-experiments*. *Component analysis* identifies potentially faulty components based on previously observed behavior; and, in case previous observations are insufficient to disambiguate between faulty vs. correct variants, *micro-experiments* generate behavior specifically aimed to achieve such disambiguation.

We will first introduce our generic architecture and then give a more formal and detailed description of our approach.

2 Approach

A component is *dependable* if and only if reliance can justifiably be placed on the service it delivers [30,7]. Intuitively, we are justified in placing reliance on a component, provided the body of available evidence suggests it to be dependable. Stronger evidence enables us to place more reliance on a component or system. We apply a staged approach to continuously certify variants of component software, centered on the collection of evidence to show the safety of a new variant.[1] New component variants are introduced after bug fixes or after the addition of new features. Updates are executed in *shadow mode*, to test their behavior while older variants are still in charge. In this way, the system can fail-over to an updated but not yet certified variant in case the behavior of an older variant is considered to be unsafe. Updates are phased into operation after sufficient evidence for correct behavior was collected.

We envision this process to be orchestrated by a central *supervisor* component. All component variants $V_1, V_2, \ldots$ of a component are assumed to produce outputs, but only the output of one variant is forwarded to other components of the system. The supervisor is responsible for the decision on which output to forward. It monitors the outputs received, and selects one. So a component variant

[1] The idea is not limited to software components but expands to hardware components as well.

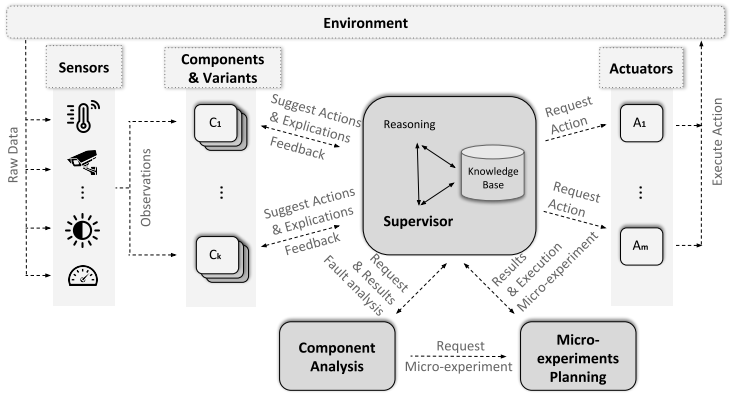

Fig. 1. Architecture overview.

can never expect to be *in charge*, i.e., that its outputs are the ones forwarded. Note that such arrangements are already being used in practice:

Example 1 (Component not in Charge). Modern cars have software components to assist in the steering of the car, but the human driver can always overwrite the decision of the software. So the outputs are controlled by the human driver and the software components are robust enough to adjust their future actions to the behavior that is imposed by the driver.

Figure 1 sketches our architectural vision. The supervisor component is the central decision entity. It monitors the different variants of each component, and decides which variant is in charge. It can at any point switch from one variant to another. To support these decisions, each component variant must be able to *explicate* its decision for a given output to the supervisor, which cross-checks the explications. We envision that these explications will be based on predicate logics extended with theories. The supervisor maintains a knowledge base, and reasons about the explications using standard as well as sophisticated reasoning mechanisms tailored to the efficacy needs in a running system.

The supervisor reasoning connects and compares outputs produced by different variants of individual components, and across components. Leveraging the explications in combination with the supervisor's knowledge base, the supervisor thus attempts to resolve any contradictions that may exist between variants.

If the supervisor reasoning can resolve the relevant contradictions, a unique decision is obtained. But that will not always be the case:

Example 2 (Conflicting Sensors). Assume we have updated the software component responsible for lane changing maneuvers in an autonomous vehicle. The new variant of the component proposes to change to the left lane and explains its decision to the supervisor by a LIDAR sensor reading indicating that there is an obstacle ahead but the left lane is free. The older variant of the component instead proposes to stay on the current lane and explains its decision by a reading of a different sensor, a RADAR sensor, indicating the lane is free but the

left lane is blocked. To resolve this conflict, the supervisor needs to know which sensor to trust - unless there is an alternative on which all variants agree upon like changing to the right lane instead.

To allow the supervisor to determine the ground truth despite component misbehavior, we envision machinery to resolve contradictory information and decisions arriving from different component variants. The supervisor can trigger this machinery when needed.

The first step in the machinery is *component analysis*, which identifies faulty components based on previously observed behavior. Component analysis compares models of component behavior to system-trace observations collected in the past, and based on this evidence reduces the set of component models to those that are indeed compatible with the observations, including possible fault models. The latter information can be qualitative ("which faults are possible?") or quantitative ("which faults are most likely?"). This feeds back to the supervisor, fostering its ability to decide which actions are safe to be carried out and which components to trust.

However, previous observations may not be enough to sufficiently disambiguate between faulty vs. correct components. For such cases, we envision the use of *micro-experiments* as a second, more radical, step. Based on the possible model information as derived by component analysis, micro-experiment planning then identifies behavior – system actions – optimized to minimize the uncertainty about the actual model employed by the components, and thus about which components are faulty in which way. The supervisor executes the micro-experiment (in an execution-and-replanning loop) and draws conclusions from the observed behaviors. Micro-experiments are limited to "small" activities (hence the name) which do not endanger the mission performance, nor, of course, the safety of the system.

Example 3 (Micro-experiment). In the above situation, we keep a model that among others estimates the distance to obstacles with two different sensors, LIDAR and RADAR. Assuming that the alleged obstacle is fixed, e.g., a traffic cone, deceleration and acceleration has a very specific effect on the sensor reading. If we now make the vehicle decelerate slightly, we can predict the distance change if the respective sensors were behaving correctly. So we carry out the deceleration (which is not a risk maneuver) while observing the sensor changes, so as to draw conclusions about the sensor believed to be faulty.

Notably, component analysis and micro-experiments can be performed only under particular circumstances. Component analysis is a complex time-consuming process. The same is true for micro-experiment planning, and actually executing a micro-experiment can of course be done only when circumstances allow, i.e., when the system is safe. Our vision therefore is for component analysis and micro-experiment planning to be triggered and executed alongside system execution whenever computational resources are available (locally or via communication with an external server), and for the supervisor to execute safe fallback actions (like pulling over to the right-hand side of the road) if conflicts cannot

be resolved on the spot. Supervisor reasoning about explications, in this context, serves as a quick means to detect and resolve simple conflicts online, without the need to embark on more complex analyses.

The role of the supervisor in a dynamic software deployment process is analogous to the role of a certifier for a sophisticated algorithm. Verification of a sophisticated algorithm is beyond the scope of any automatic verification technology. Certifiers are rather small, far less complicated components that check the correctness of the output of the sophisticated algorithm. They can often be addressed by automatic verification technology. For example, while the verification of modern SAT solving technology is still at the level of effort of a PhD thesis using an interactive theorem prover [17], the implementation and verification of a proof certifier for SAT in an interactive theorem prover is a standard exercise and even leads to code that outperforms hand-written certifiers [29]. In the same vein, the software deployed into a cyber-physical system is beyond the scope of automatic verification, but the safety of its suggested actions can be controlled by reasoning on its explications, effectuated by a far simpler supervisor, for which we envision a logic, called *SupERLogs* for (Sup)ervisor (E)ffective (R)easoning (Log)ics, as a standard knowledge representation framework maintained by the supervisor. Reasoning about the explications will use standard reasoning mechanisms, with satisfiability of the relevant SupERLogs fragments being decidable, and their Horn fragments serving as an efficient rule-based programming language to ensure efficacy in the running system.

In the sequel, we will shed more light on the challenges and intricacies of this vision. We will first discuss related literature, then detail the concepts needed for the three major system components, i.e., supervisor, component analysis, and micro-experiments.

3 Related Work

Dependable systems. Dependable systems must detect and tolerate failures at different levels of abstraction such as hardware, architecture, systems software and application level [41]. Traditional dependable systems are usually static [6], i.e., there will be at most minor changes after deployment. Their correctness needs to be certified prior to the use in operations, e.g., [37,3]. This certification is mostly process-based in the sense that one ensures the quality of a newly developed system by assessing the quality of the development processes [13]. Recent software systems tend to be more dynamic and require continuous updates, fast and cost effective deployment [1] which is hard to achieve with such traditional certification process [12]. Therefore, an alternative approach for the certification of dependable systems is an evidence-based approach [9]: one collects sufficient evidence to show that a system is safe in a given environment and safety cases attest to the safety of the system based on the given evidence.

Online updates of dependable software in the field is a double-edged sword as it can increase safety, fixing critical vulnerabilities, but can also decrease safety through the introduction of new bugs. Dynamic software updates require

mechanisms for robust software management [15]. System failures can differ substantially in impact [21]. Avoiding system failures that violate safety, i.e., put human lives at risk, has the highest priority. System failures that reduce system availability have lower priority, but higher priority than failures that merely result in inconveniences. This can be achieved with multiple mechanisms like proactive fault management [40] and failure transformation [44].

Ensuring system integrity online. Our approach focuses on collecting evidence during run-time from different variants using explications. We depend on a supervisor component that cross-checks explications and conducts online experiments to obtain additional information to identify faulty components when required. Related work investigates the use of agents testing each other [20] requiring a predefined number of agents to confirm an agent faulty. Agents check the information collected from the environment in addition to generating test events and evaluating the reaction of their peers. Similarly, we test multiple, diverse components at run-time using explications and using micro-experiments but we rely on the centralized supervisor component. This centralized unit allows to overcome the problem of having multiple agents failing by which the predefined number of agents may no longer be in reach.

The supervisor is required to be fail-operational, otherwise, it would potentially be a single point of failure. In addition to the traditional approaches to ensure that the supervisor is correctly designed, we also need to ensure that it is correctly executed. Specifically, related work investigates the correct execution of the supervisor when executed on potentially unreliable hardware [25], under security attacks [28,5,27] or even protect its integrity despite the existence of CPU design faults [26].

Supervisory control. Supervisory control theory, rooted in the work of Ramadge and Wonham [38] is a method for automatically synthesizing controllers based on formal models of hardware and control requirements. The supervisory controller observes machine behavior by receiving signals from ongoing activities, upon which it sends back control signals about allowed activities [10]. Assuming that the controller reacts sufficiently fast on machine input, this feedback loop is modeled as a pair of synchronizing processes. The model of the machine, referred to as plant, is restricted by synchronization with the model of the controller, referred to as supervisor. This theory has been the nucleus for a plethora of foundational and applied work, reaching out to the SCADA (supervisory control and data acquisition) framework in process industry, but also linking to the context of reactive synthesis, where cyber-physical systems are abstracted as discrete transition systems subject to specifications expressed in temporal logic. Notably, while our setup has a conceptual similarity with that setting, we address supervision at a higher architectural level. Our supervisors are software artifacts (of considerable complexity) that are manually crafted in a SupERLog, not synthesized; they supervise several other software artifacts each of which is meant to play the role of a partial plant controller; our machinery of explications, reasoning about these, and formal methods for component analysis and micro-experiment planning are used to achieve supervision at this level.

4 Formal Underpinning

To set the stage for the discussion that follows, we now provide details on the the formal models needed to conceptualize supervisor, component analysis, and micro-experiments.

4.1 SupERLog

In order to allow for a sketch of our supervisor architecture in a generic form (Section 5.2), we introduce the basics of a SupERLog based on a function-free predicate logic with integer arithmetic. The logic supports the modeling of supervisor behavior via rules. Reasoning in SupERLog is effective in the sense that verification of supervisor properties is decidable, and that consequence finding, can be efficiently done in a bottom-up reasoning style. Consequence finding is the main task of the supervisor.

Predicates of SupERLog range over abstract objects (constants) as well as numbers with respect to an integer arithmetic theory. Constant symbols identify the objects of interest, function symbols are not required. Abstract objects identify particles of the real world such as technical components, a sensor, or a car. Integer numbers represent sensor input or calibration data of the supervisor. The resulting logic can be viewed (i) as an extension of basic datalog with integer arithmetic, potentially non-Horn rules and unsafe variables [11], (ii) as an extension of SMT (Satisfiability Modulo Theory) by universally quantified variables [36], and (iii) as an instance of function-free first-order logic extended with integer arithmetic [22,43]. Satisfiability in this logic is undecidable in general [23]. However, it is decidable if integer variables are of bounded range in all rules. We assume this as a typical property in a technical environment. The prerequisite of finitely bounded variables also enables SupERLog support for rich arithmetic operations, as we will demonstrate in Example 5.

Definition 1 (SupERLog Signature). *A* SupERLog *signature* $\Sigma = (\Omega, \Pi, IA)$ *consists of a finite set* Ω *of predicate symbols, a finite set* Π *of constant symbols, and the symbols* IA *from integer arithmetic.*

Each $P \in \Omega$ *is associated with its* arity k*. We also write* $P(x_0, \ldots, x_{k-1})$ *to indicate the predicate's arguments explicitly. Variables may either range over finite domains generated by constants from* Π *or over integer numbers.*

Definition 2 (SupERLog Rules). *Given a SupERLog signature* Σ*, a rule*

$$H_1, \ldots, H_n \leftarrow B_1, \ldots, B_m \parallel \Lambda$$

for $H_i = P_i(\vec{x_i}, \vec{c_j})$*,* $B_i = Q_i(\vec{y_i}, \vec{d_i})$*,* $Q_i, P_i \in \Omega$*,* $c_j, d_j \in \Pi$*, and* Λ *is an arithmetic* IA *constraint over variables* $\bigcup_i (\vec{x_i}, \vec{y_i})$*.*

Definition 3 (Facts). *Given a SupERLog signature* $\Sigma = (\Omega, \Pi, IA)$ *and an atom* $P(x_0, \ldots, x_{k-1})$*,* $P \in \Omega$*, for any tuple* $\vec{c} = (c_0, \ldots, c_{k-1}) \in (\Pi \cup \text{const}(IA))^k$ *we say that the instantiation of* $P(x_0, \ldots, x_{k-1})$ *with* $\vec{c}$*, written* $P(c_0, \ldots, c_{k-1})$*, is a* (ground) fact*. We denote by* $F[\Sigma]$ *the set of all facts in* Σ*.*

In case $n = 1$ a SupERLog rule becomes a Horn rule. Such rules are used to define supervisor behavior, because of their unique minimal model semantics. In this case reasoning is complete in a datalog-style, hyper-resolution fashion: the premises B_i are matched against ground facts A_i by a ground substitution σ and if the respective constraint $\Lambda\sigma$ is satisfied, the head H_1 is inferred. Given a finite set R of SupERLog Horn rules, we say that a fact H is derivable from R if there exists an iteratively applicable sequence of (grounded) rules from R whose outcome contains H. The derivation $F[[R]] \subseteq F[\Sigma]$ from F with R is the set of all such H. Obviously, because Π is finite and integer variables appear only bounded, the above bottom-up reasoning terminates and $F[[R]]$ is finite.

The case of general SupERLog rules ($n > 1$) results from verification. For example, a component model typically includes non-deterministic behavior expressible by disjunction. Then, bottom-up reasoning is no longer complete and we stick to model-driven reasoning [16] which is also terminating for SupERLog non-Horn clause sets.

4.2 System Model

We will be working in a setting with components interacting through input/output synchronization. We use a very natural extension of the I/O-automata formalism [33] to a setting with probabilistic transition effects, known as probabilistic I/O systems[19]. Probabilistic I/O automata give us the opportunity to model in a very natural manner typical cyber-physical systems (CPS) which are built up of components that interact and exchange information. In addition, CPS often occur in only partially controllable or known environments and also the modelling is bounded by technical and physical challenges which can be represented in the design of probabilistic automata.

Definition 4 (PIOS [19]). *A* probabilistic I/O atom *is a tuple* $(S, Act, G, R, \bar{s})$*, where* S *is a finite set of states,* Act *is a finite set of action labels,* $G \subseteq S \times \mathcal{D}(Act \times S)$ *is a generative output transition relation,* $R\colon S \times Act \to \mathcal{D}(S)$ *is a reactive transition function, and* $\bar{s} \in S$ *is an initial state.*

A probabilistic I/O system (PIOS) *is a finite vector* $\mathcal{P} = (\alpha_1, \ldots, \alpha_n)$ *of probabilistic I/O atoms* $\alpha_i = (S_i, Act_i, G_i, R_i, \bar{s}_i)$ *for* $i \in \{1, \ldots, n\}$*. The set of states of the system is the product of the component states* $S(\mathcal{P}) := \times_i S_i$ *and* $\bar{s}(\mathcal{P}) := (\bar{s}_1, \ldots, \bar{s}_n)$ *is the system's initial state. Let* $\mathcal{A} := \mathcal{D}(\bigcup_i Act_i)$ *be the set of* transition labels*. We define a transition relation* $\to \subseteq S(\mathcal{P}) \times \mathcal{A} \times \mathcal{D}(S(\mathcal{P}))$ *such that* $((s_1, \ldots, s_n), \kappa, \mu) \in \to$ *if and only if there exists an* $i \in \{1, \ldots, n\}$ *and* $\kappa_i \in \mathcal{D}(Act_i \times S_i)$ *such that* $(s_i, \kappa_i) \in G_i$*, for all* $a \in Act_i$*,* $\kappa(a) = \sum_{s \in S_i} \kappa_i(a, s)$*, and for all* $(s'_1, \ldots, s'_n) \in S(\mathcal{P})$*:*

$$\mu(s'_1, \ldots, s'_n) = \sum_{a \in Act_i} \kappa(a, s'_i) \prod_{j \neq i} \begin{cases} R_j(s_j, a)(s'_j) & a \in Act_j \\ \delta(s_j)(s'_j) & a \notin Act_j \end{cases}$$

Executions of CPS modelled as probabilistic I/O systems can be described and observed by traces which include the actions taken in the system and paths

which in addition contain the system states which occurred during the execution. Having these information one can reconstruct what happened during the execution and what was the reason.

Definition 5 (Paths and Traces). *For a given PIOS* $\mathcal{P} = (\alpha_1, \ldots, \alpha_n)$*, a finite* path *is an alternating sequence of states and transitions* $s_0\, t_0\, s_1\, t_1\, \ldots\, t_{k-1}\, s_k$ *where* $s_i \in S(\mathcal{P})$ *for* $0 \leqslant i \leqslant k$ *and for each index* $j \in \{0, 1, \ldots, k-1\}$*,* $t_j = (s_j, \kappa_j, \mu_j) \in \rightarrow$ *such that* $\mu(s_{j+1}) > 0$*. For such a path, the sequence of actions* $\kappa_1\, \kappa_2\, \ldots\, \kappa_k$ *is called its* trace*. Each such trace is a word* $\hat{\rho} \in \mathcal{A}^*$*. Let* $\mathbb{T}[\mathcal{P}]$ *denote the set of all traces of PIOS* $\mathcal{P}$*.*

The supervisor is a special component of the complete PIOS defining the CPS under investigation. This component contains only one state and repeatedly executes control actions and thereby fully determines the system's behavior. This is later needed for conducting the micro-experiments

Definition 6 (Supervisor Model). *We model the supervisor as a component* $\alpha_s = (S_s, Act_s, G_s, R_s, \bar{s}_s)$ *with* $S_s := \{\bar{s}_s\}$*,* $G_s := \{(\bar{s}_s, \delta((a, \bar{s}_s))) \mid a \in Act_C\}$ *for some set of* control actions $Act_C \subseteq Act_s$*, and* $R_s(\bar{s}_s, a) := \delta(\bar{s}_s)$ *.*

The set Act_C will correspond to a certain set of facts under the control of the supervisor.

4.3 Observers and Boolean Monitors

We assume that a system perceives its environment through sets of facts which are provided by an *observer* based on an execution trace as defined in Def. 5. The action sequences given by execution traces contain information such as sensor readings which the observer translates into facts:

Definition 7. *An* observer *is a function mapping traces to sets of facts:*

$$O : \mathcal{A}^* \rightarrow 2^{F[\Sigma]}$$

To formally specify observers, we harvest results from the area of runtime verification. Runtime verification techniques allow to check whether a trace of a system under scrutiny satisfies or violates a given property [32] usually specified in a formal specification language e.g., [8,14]. Work in the area also expands to the computation of quantitative properties e.g., [2,31]. We abstract from the concrete specification language and introduce the following framework-agnostic notion of boolean monitors:

Definition 8. *A* boolean monitor *is a function mapping traces to booleans:*

$$M_{\mathbb{B}} : \mathcal{A}^* \rightarrow \mathbb{B}$$

The property observed by a boolean monitor then provides a verdict regarding a particular ground fact. For instance, there may be a boolean monitor that determines whether a lane is free based on information provided by a LIDAR sensor extracted from the current execution trace. We capture this correspondence between boolean monitors and ground facts formally as follows.

Definition 9 ($\mathcal{M}$-Observer). *Let $\mathcal{M}$ be a set of pairs $\langle M_{\mathbb{B}}, p(\vec{c})\rangle$ of boolean monitors and grounded facts. For each $\mathcal{M}$ we define an observer:*

$$O[\mathcal{M}_{\mathbb{B}}](\hat{\rho}) := \{\, p(\vec{c}) \mid \langle M_{\mathbb{B}}, p(\vec{c})\rangle \in \mathcal{M} \text{ s.t. } M_{\mathbb{B}}(\hat{\rho}) = \top \,\}$$

The boolean monitors making up $\mathcal{M}$ can be based on any work in the area of runtime verification that is suitable for computing boolean properties over traces as defined in Def. 5. This includes specification languages for quantitative properties as long as they *also* allow the computation of boolean properties. While this setup is very general, there are some practical constraints. In particular, the set $\mathcal{M}$ must be finite or at least finitely representable such that we can actually compute the set of facts for a given trace.

5 The Supervisor

With its central role in our envisioned architecture, the supervisor has to fulfill a variety of tasks. Most prominent is the reasoning about component explications, but the coordination role requires also other activities. We first give an overview of the supervisor role as a whole, then delve into the details of reasoning. Component analysis and micro-experiments are tackled by separate components, and will be addressed in Section 6 and Section 7 respectively.

The supervisor itself is assumed to be a reusable, dependable component. It is designed using state-of-the-art dependability approaches like formal proofs and fault-tolerance mechanisms, like [42], to *prevent* that the supervisor becomes a single point of failure.

5.1 Overall Role and Tasks

Consider again the architecture overview in Figure 1. The supervisor is the entity communicating with all other components and emitting the action decisions to be executed in the system's environment. It makes use of a knowledge base and a reasoning engine for reasoning about component explications. From this central position and design, the following tasks arise:

(i) *Knowledge base maintenance.* The supervisor needs to update its knowledge of the environment and its behavior, adjusting, e.g., for environment changes and low-probability events that were not observed before deployment. This knowledge base might be shared with the supervisors of other systems.

(ii) *Reasoning about explications.* As previously outlined, the supervisor needs to check component outputs, and in particular suggested action decisions, for contradictions given its knowledge.

(iii) *Synchronization with component variants.* The supervisor needs to continuously inform the component variants about the state of affairs, i.e., which action decisions were executed, which variants are in charge, which outputs have been forwarded to other components.

(iv) *Observations statistics maintenance.* The supervisor must collect and maintain the system execution data relevant for component analysis and, indirectly, micro-experiments.
(v) *Taking action decisions.* The supervisor is responsible for deciding whether the outcome of reasoning is sufficient to take an action decision, whether a safe fallback action should be executed, or whether further investigation through component analysis or micro experiments should be triggered.
(vi) *Executing micro-experiments.* Micro-experiments are used to identify action strategies that minimize uncertainty about faulty components. The supervisor is responsible for executing these strategies.
(vii) *Taking analysis results into account.* The supervisor must be able to incorporate the results from component analysis and micro-experiments into its decisions as per (v).

While items (i), (iii), and (iv) can be based on well-understood principles (e.g., [35,39]), items (ii), (v) and (vi) need more discussion. For reasoning about explications (ii) we employ SupERLog reasoning, as outlined below. This comes with a trade-off between expressivity and efficiency paving a controlled way to the use of online, real-time decision making in dynamic systems.

For the core of item (v), a straightforward solution consists of hardcoded rules like "execute an action only if proved safe" or "trigger component analysis if uncertainty greater than threshold". A more advanced and robust solution is to formulate the entire decision-making process – encompassing regular actions, fallback actions and whether to trigger component analysis – as a single overall reasoning process. Our SupERLog reasoning mechanism is suited for this purpose.

Micro-experiment execution in item (vi) takes the form of re-planning [18], based on an action policy suggested by the micro-experiment planner (see Section 7). In each execution step, this policy suggests an action A to execute to the supervisor. The supervisor decides whether to execute A or another action (like a fallback action, or a decision to remain idle and just observe the environment behavior). The supervisor communicates its decision back to the micro-experiment planner, which re-plans an adapted policy if needed.

For taking analysis results into account (vii), Micro-experiment execution generates new observations (feeding into the supervisor like all observations, via an $\mathcal{M}$-Observer cf. next sub-section), while component analysis results need to be directly fed into the supervisor knowledge base. A canonical instance of the latter is to feed back assessments classifying component variants as "correct" vs. "potentially faulty" vs. "definitely faulty" (which we shall specify formally in Section 6). If required, more detailed properties can be communicated by including corresponding predicates and knowledge about fault-model specifics.

5.2 Reasoning about Component Explications

A central issue in logical reasoning is the trade-off between expressivity and efficiency: more powerful logics allow to express more complex phenomena, but

are more complex to reason about and quickly become undecidable. Reasoning in Horn SupERLog terminates and can be efficiently implemented making it a perfect choice for online decision making in dynamic systems. We next specify a possible form of a supervisor model based on SupERLog and introduce different modes of reasoning along these lines, using SupERLog deduction over ground facts. We first address component variants, then the supervisor itself. For the implementation of the supervisor we employ a Horn SupERLog rule base.

Component variants are components in system control, working together to choose actions (from the action set Act_C in our system model). At the abstraction level in our supervisor model here, they are collections of functions on facts over a signature, implemented by Horn rules of the supervisor, where the target signature contains an encoding of system control information:

Definition 10 (Variant). *Given the set of actions Act_C, a (component)* variant *V is a finite set $\{V_1, \ldots, V_k\}$ of functions $V_i : 2^{F[\Sigma^I]} \mapsto 2^{F[\Sigma_i^O]}$, where Σ^I and Σ_i^O are SupERLog signatures. We require that $\Pi^I \supseteq Act_C$ and that, for all i, $\Pi_i^O \supseteq \Pi^I \cup \{V\}$ and $\Omega_i^O \supseteq \Omega^I \cup \{p_i^{\text{ctrl}}\}$. We will refer to p_i^{ctrl} as V_i's* control predicate.

Note that the set Act_C comprises the actions in control of the supervisor, cf. Definition 6. The input $2^{F[\Sigma^I]}$ common to all functions V_i here connects to observations on the trace of the system, i.e., the facts in $F[\Sigma^I]$ are associated with boolean monitors in an $\mathcal{M}$-Observer as per Definition 9. That $\mathcal{M}$-Observer is permanently associated with the supervisor, serving to connect its representation of facts to the system trace observations in the model underlying component analysis.

A component variant is a collection of functions, rather than a single function, to allow to distinguish its outputs with respect to individual control predicates. Such predicates encode outputs that form part of the system control, i.e., that provide information about which actions Act_C should be chosen. This includes direct control information through predicates like $doAction(V, a)$ and $illegalAction(V, a)$ where variant V decides to execute a or deems that action to be illegal in the current situation. It also includes intermediate information like $emergencyBreakNeeded(V)$ indicating that one part of the machinery suggests an action (like an emergency-break unit which raises a need to break whenever an obstacle is detected ahead) which will be combined with other information in the supervisor before taking an actual decision (like changing the lane instead). The remaining (non-control-predicate) output of each individual function V_i is an *explication* in terms of the subset of relevant *input facts* responsible for the control-predicate decision made by V_i.

Naturally, for every input fact set F, the output $V_i(F)$ of each function should be a subset of F (the explication) plus exactly one control fact, and the control facts should not contradict each other across different $V_i(F)$. Our definition does not make these restrictions to permit exceptions, and to allow the model to capture faulty components where implementation bugs may disvalidate these properties. It is the supervisor's task to reason about inconsistencies in

component variants' outputs. Furthermore, our defined predicate logic enables effective computation of guarantees such as the existence of single control facts.

The functions V_i are arbitrary in our definition to keep our concepts generic and as the internal working of the component variant is not of interest in the supervisor specification (it is, instead, the subject of faulty component analysis and micro-experiments). In practice, implementations of the component variants must, for use in our framework, be extended with an implementation of the functions V_i describing their behavior in terms of predicates as specified above to support supervisor reasoning. As an example, the emergency-break unit as above, reacting to obstacles detected in front-camera pictures, requires to have simple adaptors that translate the relevant signals ("obstacle detected", "control: break") into suitable ground facts.

The supervisor now simply takes as input the component variants' outputs, and processes this information with its SupERLog rules set.

Definition 11 (Supervisor). *Given a set of actions Act_C and a set $\mathcal{V}$ of variants, a supervisor is defined by a SupERLog signature Σ_s where $\bigcup_{V\in\mathcal{V},V_i\in V} \Omega_i^O \subseteq \Omega_s$, as well as a finite set R_s of SupERLog Horn rules for Σ_s.*

Given a set of facts $F \subseteq F[\Sigma^I]$ for each variant V, the supervisor computes the SupERLog derivation $F^O[[R_s]]$ from the union of outputs $F^O := \bigcup_{V\in\mathcal{V},V_i\in V} V_i(F)$. It takes decisions based on that derivation.

For example, a simple situation is that where the supervisor checks for contradictions when dealing with a variant V that is already certified, and another variant V' with recent security updates that we want to certify. To this end, we include a predicate $\mathit{contradiction}(x, y)$ into P_s, and the rule $\mathit{contradiction}(x, y) \leftarrow \mathit{doAction}(x, a), \mathit{illegalAction}(y, a)$ into R_s. We then check for ground instances $\mathit{contradiction}(x, y) \in F^O[[R_s]]$. The absence of such a contradiction between V and V' for a long period of time increases confidence in the safety of V'. If there is a contradiction however, there are two possible cases. First, $\mathit{contradiction}(V', V)$ where V forbids an action suggested by V', indicating that V' is unsafe. Second, vice versa $\mathit{contradiction}(V, V')$, which results in an ambiguous situation as V' is not yet certified yet may have an important security update. In both cases, the supervisor may decide to take an emergency action, like a handover to the human operator. Or, given sufficient time is available, it may invoke component analysis, and transitively micro-experiments, to gain more confidence in which of V and V' is correct.

In the latter case, the supervisor uses the explications delivered by V and V' as part of their output. Namely, say the reason for V's decision is the input fact $p \in F$ and that for V''s decision is $q \in F$. We assume that the integrity of the inputs can be verified by the supervisor, i.e., a variant can neither generate inputs nor can it modify inputs. This can be achieved, for example, by digitally signing all inputs. Each of p and q are associated with boolean monitors in the supervisor's $\mathcal{M}$-Observer, $\langle M^p_{\mathbb{B}}, p(\vec{c})\rangle$ and $\langle M^q_{\mathbb{B}}, q(\vec{c})\rangle$. The supervisor communicates $M^p_{\mathbb{B}}$ and $M^q_{\mathbb{B}}$ to component analysis, which makes use of this information about system traces to narrow down the possible fault models as we shall specify in the next section.

Example 4 (Conflicting Sensors: Supervisor Reasoning). Consider, as in Example 2, an update to the software component responsible for lane changing maneuvers in an autonomous vehicle. Say the vehicle is currently driving in the right lane. The old variant V outputs a control predicate $doAction(V, gostraight) \in V_i(F)$ while the new variant V' outputs $illegalAction(V', gostraight) \in V'_j(F)$. With the above simple reasoning, the supervisor concludes $contradiction(V, V')$, indicating a conflict. Say the supervisor decides to handover to the human operator and, simultaneously, to trigger component analysis. The explications provided by V_i and V'_j are $p = free(rightlane)$ and $q = blocked(rightlane)$ respectively.[2] The supervisor maps these to boolean monitors $M^p_{\mathbb{B}}$ and $M^q_{\mathbb{B}}$ in its $\mathcal{M}$-Observer.

From a SupERLog reasoning perspective, action-contradiction checking as above is extremely simple. More complex reasoning arises, for example, in meta-reasoning about which decision to take given mixed evidence (go ahead? bail out? trigger component analysis?); and when components do not output actions directly, but pieces of information that will need to be assembled by the supervisor reasoning to arrive at an action decision. The latter makes sense, for example, in the context of a lane change scenario, where what is a safe distance depends on road conditions:

Example 5 (Lane Change). The supervisor calculates a safe distance ahead of car depending on wet or dry road conditions using the explications provided by the variants ($speed(S)$, $distance(D)$). For example, the supervisor decides based on following rules if the the distance between the car x and a car y in front of it is safe ($Safe\_Distance\_Ahead(SDA)$). Specifically, the distance should be large enough ($Base\_Safe\_Distance\_Ahead(BSDA)$) in addition to extra space required if the road is wet ($Extra\_Distance(ED)$). Note that $BSDA$ rules varies depending on the speed of the car x compared to the speed of car y.

$$\frac{\mathrm{SDA}(x,y,sx,sy,z) \leftarrow \mathrm{BSDA}(x,y,sx,sy,v), \mathrm{ED}(x,sx,w), \parallel z > v + w}{\begin{array}{c}\mathrm{BSDA}(x,y,sx,sy,v) \leftarrow \mathrm{S}(x,sx), \mathrm{S}(y,sy), \mathrm{D}(z,x,y) \parallel sx > sy, z > sx * 10 \\ \mathrm{BSDA}(x,y,sx,sy,v) \leftarrow \mathrm{S}(x,sx), \mathrm{S}(y,sy), \mathrm{D}(z,x,y) \parallel sx \leqslant sy, z > sx\end{array}}$$

$$\mathrm{ED}(x,sx,w) \leftarrow \mathrm{S}(x,sx), \mathrm{Wet}(u) \parallel w = \mathrm{div}(sx * u, 10)$$

6 Component Analysis

Given a formal and componentwise model of the system, the component analysis identifies potentially faulty components in the system.

Definition 12 (System Configuration). *Let $\langle C_1, \ldots, C_n \rangle$ be a finite vector of* components *and $\mathcal{M}_i := \{\alpha^i_1, \ldots, \alpha^i_{k_i}\}$ be a set of PIOS atoms for each component C_i. A* system configuration *is a PIOS $c = \langle \alpha^1, \ldots, \alpha^n \rangle$ such that $\alpha^i \in \mathcal{M}_i$. Let $\mathcal{C}$ be the set of all system configurations. Then $\mathbb{T}[\![\mathcal{C}]\!] := \bigcup_{c \in \mathcal{C}} \mathbb{T}[\![c]\!]$ is the set of all finite traces over all system configurations.*

[2] Note that, for the sake of efficiency, SupERLog reasoning may not explicitly handle negation. Instead, the relevant contradictory fact combinations can be identified via appropriate extra rules.

Definition 13 (Observation Function). *Let Obs be a potentially infinite set of* observables*. An* observation function $\mathcal{O} : \mathbb{T}[\mathcal{C}] \to Obs$ *maps traces to observables $o \in Obs$. For each system configuration $c \in \mathcal{C}$ the set $Obs(c) = \{\mathcal{O}(\hat{\pi}) \mid \hat{\pi} \in \mathbb{T}[c]\} \subseteq Obs$ is the set of observables* consistent *with c.*

Given an observation of an observable $o \in Obs$ we seek to explain o in virtue of a system configuration which is consistent with o, i.e., a system configuration that may result in an observation of o. The different models α_x^i for each component C_i enable us to ask and answer model-based *what-if* questions.

Example 6. Returning to Example 2, imagine a distance sensor component C_i measuring the distance to obstacles in the present lane based on which a boolean monitor concluded that there is an obstacle ahead. Now, given an observation o and different models α_N^i and α_F^i for sensor C_i where α_N^i describes the behavior of a perfectly functioning and α_F^i the behavior of a faulty sensor, we may ask: what if the sensor were faulty, would that explain the observation o? To answer this question, we check whether o is consistent with a configuration where the sensor behaves normally versus a configuration where the sensor is faulty. The answer allows the supervisor to determine which components to trust.

Definition 14 (Faulty Components). *With respect to an observable o, we call a component* potentially faulty *if and only if there exists a configuration c consistent with o such that the component's model in c describes faulty behavior. We call a component* definitely faulty *if and only if for all configurations c consistent with o, the component's model in c describes faulty behavior.*

Note that if there is a failure mode of the component that may still produce the correct behavior in some execution, e.g., a sensor which non-deterministically provides values, then Definition 14 always considers the sensor potentially faulty. This matches our intuition that in such a case we can never be certain that the sensor is not already faulty until it actually misbehaves.

In case of probabilistic systems one may further assign a (minimal/maximal) probability to each configuration c with respect to a given observable o capturing how probable it is to observe o if the system were configured according to c. Such probabilities provide the supervisor with insights about the likelihood of certain components being faulty and can be a further basis for its decision.

With regard to the architecture displayed in Figure 1, the environment, each sensor, and the actuators are represented as components. Treating them as components allows us to capture interactions between actuator commands and expected sensor readings.

For instance, if the obstacle in Example 2 is fixed then acceleration and deceleration should have a specific effect on the slope of the measured distance.

Example 7 (Component Analysis). Recapitulating Example 2 and Example 3, we model the system as four components, the LIDAR sensor C_L, the RADAR sensor C_R, the environment C_E, and the supervisor C_S. For the environment, we define a single probabilistic I/O atom with digital clock semantics describing

how the distance of the car changes in response to (discrete) speed changes of the supervisor. The supervisor is modeled as per Definition 6 which results in non-determinism in the model regarding the acceleration and deceleration decisions. Those acceleration and deceleration actions are, however, provided by the observation and thereby resolved by the actual system. For both sensors, we have a nominal model capturing the measured distance with respect to changes in the environment. In addition, we have the following failure models: (1) the measured distance is non-deterministically stuck and does not change and (2) the measured distance is multiplied by 5. Now, if the car decelerates slightly, the correct measurements are described by the nominal models. Our model predicts that given an observation of a deceleration action this should be followed by a specific change in distance. If the observed value for a sensor does not change at all, then this is in line with what the failure model (1) would predict. If the value changes over-proportionally by a factor of 5, then this is in line with what the failure model (2) would predict.

For the observation function, we assume that a subset $\mathcal{A}_O \subseteq \mathcal{A}$ of the transition labels in the PIOS are directly observable, e.g., sensor readings and actuator commands. The set Obs of observables is the set of finite words over $\mathcal{A}_O$, i.e., $Obs = \mathcal{A}_O^*$. For a finite trace $\hat{\rho} = \kappa_0\kappa_1 \dots \kappa_n \in \mathcal{A}^*$ of a PIOS $(\alpha_1, \dots, \alpha_n)$ we define the observation function O as follows:

$$\mathcal{O}(\kappa_0\kappa_1 \dots \kappa_n) = \mathcal{O}(\kappa_0)\mathcal{O}(\kappa_1) \dots \mathcal{O}(\kappa_n) \quad \text{with } \mathcal{O}(\kappa) := \begin{cases} \kappa & \text{if } \kappa \in \mathcal{A}_O \\ \epsilon & \text{otherwise} \end{cases}$$

The supervisor provides the component analysis with the boolean monitors responsible for the contradiction. We assume that some of the boolean monitors respond to specific transition labels generated by particular components. For example, the I/O atoms modeling a LIDAR sensor have generative transitions for specific *LIDAR-sensor-reading* actions which are the basis for the monitor's verdict regarding the distance to obstacles on the lanes. Based on this information, we prioritize different sets of system configurations we consider for the analysis. Note, however, that a contradiction of facts may not only be caused by malfunctioning of the components which are directly connected via monitors to those facts. Hence, the information provided by the supervisor is merely used as a hint to quickly identify potential faults by assuming that all other components except those involved in the contradiction are functioning nominally.

If the component analysis is not able to determine which of the components is faulty because there are multiple configurations consistent with a given observation, then the component analysis invokes micro-experiment planning. To this end, the micro-experiment planner is provided with a set of *possible* configurations $\mathcal{C}_\Diamond \subseteq \mathcal{C}$ and the given observation o.

7 Micro-Experiments

The purpose of a micro-experiment is to provide the supervisor with specific instructions that, when followed, will allow to distinguish between the system models $\mathcal{C}_\Diamond$ still considered possible after component analysis. Ultimately, upon the completion of a micro-experiment, a unique possible model is identified. We next define what a micro-experiment is (in terms of an action selection function), and outline the use within the supervisor framework.

As previously discussed, the supervisor resolves action choices, as captured by actions in our model (cf. Section 4.2). For instance, in our lane change example, a component could request a lane change in either left or right direction in order to evade an obstacle on the current lane. We formalize micro-experiments based on these action choices, as a function of observations.

Concretely, in the system model formalization, Definition 4, the supervisor is represented abstractly via atom α_s. The supervisor's control actions are Act_C. For ease of presentation, we assume that the control actions available in a specific situation can be observed by the supervisor directly, i.e., $Act_C \subseteq \mathcal{A}_O$. Intuitively, components must explicitly request the supervisor to make a decision. For that request to make sense, the supervisor must be able to know (and thus to observe) the available options. Given this, micro-experiments are defined as follows:

Definition 15 (Micro-Experiment). *The* observation histories $\mathcal{H}$ *denote the set of all finite sequences of pairs* $\mathcal{A}_O \times \{\top, \bot\}$. *A* micro-experiment *is a (partial) function* $\pi : \mathcal{H} \mapsto \mathcal{A}_O \cup \{\square\}$.

Observation histories allow to track and to accordingly react to observations made since the beginning of the micro-experiment execution. Micro-experiments are hence decision strategies that inform the supervisor what to do next depending on what has been observed so far.

This is non-trivial in two aspects. First, the uncertainty about system models entails uncertainty which actions are enabled. Indeed, applicability of observable actions is the information at our disposal to distinguish between different system models. Definition 15 reflects this with the annotation symbols $\top$ and $\bot$, which encode applicability and inapplicability respectively. Second, the supervisor can control only some, not all, of the observable actions. Non-controlled observable actions must be taken into account as signals for the execution of a micro-experiment, i.e., they must be added to the observation history. Definition 15 hence permits arbitrary observable actions as the micro-experiment output.

The micro-experiment execution is structured accordingly. It consists of a loop, where in each iteration the supervisor queries the micro-experiment with the current observation history h. The history is initially empty, and is extended by one more element in each step. Each loop iteration has an active or passive nature depending on whether the suggested action $a := \pi(h)$ is (a) a control action $a \in Act_C$ or (b) a non-control action $a \in \mathcal{A}_O \backslash Act_C$. In case (a), the supervisor checks whether a is enabled. If yes, the supervisor executes a and updates the observation history to $h \circ \langle \pi(h), \top \rangle$; if no, the supervisor updates the observation history to $h \circ \langle \pi(h), \bot \rangle$. In case (b), the supervisor takes a passive role,

merely monitoring the system behavior and updating the history accordingly. The micro-experiment at such a point essentially asks the supervisor whether the next action executed by the system is a. The supervisor updates h to $h \circ \langle \pi(h), \top \rangle$ if the observed action indeed matches the requested action a, and to $h \circ \langle \pi(h), \bot \rangle$ otherwise. The micro-experiment terminates when $\pi(h) = \square$. The observation history h collected until this point allows to remove those models from $\mathcal{C}_\Diamond$ that are inconsistent with h.

Example 8. In continuation of Examples 2, 3, and 7, assume that our system contains a potentially faulty distance sensor C_S. Let α_N (nominal), α_V (errors at higher speeds), and α_A (errors at certain view angles to the obstacle) be our models of C_S that are consistent with the current observations. The supervisor now wishes to obtain additional information about the potential faultiness of C_S. Figure 2 illustrates a possible micro-experiment π for that purpose. The control actions here are those actions that actively change the state of the car. The non-control actions represent sensor outputs. The distinguish between the two, the latter are shown in *italic*. The gray sets reflect how the possible models of C_S change according to the observations made during micro-experiment execution.

Assume that the car is currently moving on a straight road with a speed of $14\frac{m}{s}$. Let there be a static obstacle detected by C_S with a distance of $100m$. π immediately requests the supervisor to slightly decelerate (step i). If this is not possible, i.e., the supervisor answers with "$\bot$", the execution of π is stopped without obtaining additional information about the models. If it is possible "$\top$", π requests the supervisor to observe how the distance measurement of C_S changes when decelerating (step ii). The amount of deceleration is chosen large enough to assure that even under fault model α_V, the sensor now measures the correct distance. Consequently, if the new measurement does not agree with what would be expected given the initial distance estimate, the "$\bot$" case, then the initial distance estimate must have been wrong. Thus, the chosen control action implies that α_V has to be the model of C_S. Vice versa, for "$\top$" we know that α_V cannot be underlying C_S since the change in speed was indeed adequately represented in the distance measurement. The execution of π then continues from the reduced model set α_N and α_A as further depicted in Figure 2.

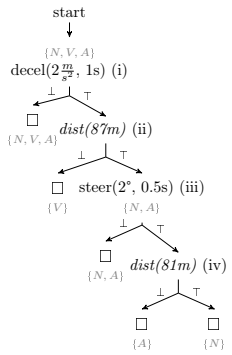

Fig. 2. An example micro-experiment π.

The question of course is how to construct a suitable micro-experiment in the first place. We conjecture that this can be done via standard tools based on an encoding into partially observable Markov decision processes (POMDP) [45]. Specifically, the uncertainty about system models can be encoded into POMDP state uncertainty by making the true system model an unobservable part of the state, and defining the transition behavior according to the true model. Observable actions in such a POMDP then decrease state uncertainty by distinguishing

states where the action is enabled from ones where it is not. The objective of maximally decreasing uncertainty can be encoded as a reward dependent on the POMDP's belief state, as in so-called ρPOMDPs [4]. It remains a topic for future work to spell out this approach.

8 Discussion and Outlook

Cyber-physical systems that evolve dynamically through in-field component updates are clearly a grand challenge to dependability, especially when facing insufficient time for full re-certification of the updates. Our approach brings together five key ideas to design an architecture addressing that challenge:

(i) Not a single but several component variants are maintained, and a *supervisor* monitors and arbitrates their behavior.
(ii) Component variants must provide *explications* for their decisions.
(iii) Effective and verifiable *reasoning* is used by the supervisor to identify inconsistencies across components and component variants.
(iv) In-field *component analysis* identifies possible faults from past system observations.
(v) In-field *micro-experiments* disambiguate between faults through injection of dedicated system behavior.

In short, our proposed architecture safeguards uncertified component variants through a certifiable control layer, and offers the system the possibility to self-diagnose (autonomously, if necessary) in the field. Each of the ideas (i) – (v) draws on established concepts in different areas of computer science. Our contribution lies in bringing them together, and providing a first formalization of the constituents and their interplay.

At this stage, our contribution is a vision. It remains to implement and evaluate the envisioned architecture.

Observe that, to this end, for (i), (iii), (iv), and (v) one can draw on established techniques in dependability, logics, verification, and artificial intelligence respectively. The present paper brings together these areas and outlines a possible architecture instance together with its formalization. Many other instantiations are conceivable given the wealth of concepts available in each area.

For (ii), the computation of explications, matters are somewhat different as the process of explicating decisions remains in itself a grand challenge for classical software engineering as well as for machine learning (ML) systems. For ML this is commonly known as the Explainable AI challenge. While for now we assume a relatively benign form of explications, namely identifying the relevant subset of the input which led to the output, we expect that our architecture vision will profit from progress in that highly active area.

Our next steps will be to spell out different aspects of our architecture, to be implemented and evaluated in individual use cases, paving the way to the vision of a full integrated system. We believe that this opens a new sub-area of dependable systems research, which we hope will inspire other researchers as well.

References

1. Aizpurua, J., Muxika, E., Papadopoulos, Y., Chiacchio, F., Manno, G.: Application of the d3h2 methodology for the cost-effective design of dependable systems. Safety **2**(2), 9 (Mar 2016). https://doi.org/10.3390/safety2020009, http://dx.doi.org/10.3390/safety2020009
2. Alur, R., Fisman, D., Raghothaman, M.: Regular programming for quantitative properties of data streams. In: Thiemann, P. (ed.) Programming Languages and Systems. pp. 15–40. Springer Berlin Heidelberg, Berlin, Heidelberg (2016)
3. Alvaro, A., de Almeida, E.S., de Lemos Meira, S.R.: Software component certification: a survey. In: 31st EUROMICRO Conference on Software Engineering and Advanced Applications. pp. 106–113 (2005)
4. Araya, M., Buffet, O., Thomas, V., Charpillet, F.: A pomdp extension with belief-dependent rewards. In: Lafferty, J.D., Williams, C.K.I., Shawe-Taylor, J., Zemel, R.S., Culotta, A. (eds.) Advances in Neural Information Processing Systems 23, pp. 64–72. Curran Associates, Inc. (2010)
5. Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'Keeffe, D., Stillwell, M.L., Goltzsche, D., Eyers, D., Kapitza, R., Pietzuch, P., Fetzer, C.: SCONE: Secure linux containers with intel SGX. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). pp. 689–703. USENIX Association, Savannah, GA (2016), https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov
6. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Secur. Comput. **1**(1), 11–33 (Jan 2004). https://doi.org/10.1109/TDSC.2004.2, https://doi.org/10.1109/TDSC.2004.2
7. Avizienis, A., Laprie, J.C., Randell, B., et al.: Fundamental concepts of dependability. University of Newcastle upon Tyne, Computing Science (2001)
8. Bauer, A., Leucker, M., Schallhart, C.: Monitoring of real-time properties. In: Arun-Kumar, S., Garg, N. (eds.) FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science. pp. 260–272. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
9. Bishop, P., Bloomfield, R.: A methodology for safety case development (02 1998). https://doi.org/10.1007/978-1-4471-1534-2_14
10. Cassandras, C.G., Lafortune, S.: Introduction to Discrete Event Systems. Springer Publishing Company, Incorporated, 2nd edn. (2010)
11. Ceri, S., Gottlob, G., Tanca, L.: What you always wanted to know about datalog (and never dared to ask). IEEE Trans. Knowl. Data Eng. **1**(1), 146–166 (March 1989)
12. Council, N.R.: Software for Dependable Systems: Sufficient Evidence? The National Academies Press, Washington, DC (2007). https://doi.org/10.17226/11923, https://www.nap.edu/catalog/11923/software-for-dependable-systems-sufficient-evidence
13. Currit, P.A., Dyer, M., Mills, H.D.: Certifying the reliability of software. IEEE Transactions on Software Engineering **SE-12**(1), 3–11 (1986)
14. D'Angelo, B., Sankaranarayanan, S., Sanchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: Lola: runtime monitoring of synchronous systems. In: 12th International Symposium on Temporal Representation and Reasoning (TIME'05). pp. 166–174 (2005)

15. Felser, M., Kapitza, R., Kleinöder, J., Schröder-Preikschat, W.: Dynamic software update of resource-constrained distributed embedded systems. In: Rettberg, A., Zanella, M.C., Dömer, R., Gerstlauer, A., Rammig, F.J. (eds.) Embedded System Design: Topics, Techniques and Trends. pp. 387–400. Springer US, Boston, MA (2007)
16. Fiori, A., Weidenbach, C.: SCL with theory constraints. CoRR **abs/2003.04627** (2020), https://arxiv.org/abs/2003.04627
17. Fleury, M.: Formalization of logical calculi in Isabelle/HOL. Ph.D. thesis, Saarland University, Saarbrücken, Germany (2020)
18. Ghallab, M., Nau, D., Traverso, P.: Automated Planning and Acting. Cambridge University Press (2016)
19. Giro, S., D'Argenio, P.R., Fioriti], L.M.F.: Distributed probabilistic input/output automata: Expressiveness, (un)decidability and algorithms. Theoretical Computer Science **538**, 84 – 102 (2014). https://doi.org/https://doi.org/10.1016/j.tcs.2013.07.017, http://www.sciencedirect.com/science/article/pii/S0304397513005203, quantitative Aspects of Programming Languages and Systems (2011-12)
20. Heck, H., Rudolph, S., Gruhl, C., Wacker, A., Hähner, J., Sick, B., Tomforde, S.: Towards autonomous self-tests at runtime. In: 2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W). pp. 98–99 (2016)
21. Heimerdinger, W., Weinstock, C.: A conceptual framework for system fault tolerance. Tech. Rep. CMU/SEI-92-TR-033, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (1992), http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=11747
22. Horbach, M., Voigt, M., Weidenbach, C.: On the combination of the bernays-schönfinkel-ramsey fragment with simple linear integer arithmetic. In: de Moura, L. (ed.) Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10395, pp. 77–94. Springer (2017)
23. Horbach, M., Voigt, M., Weidenbach, C.: The universal fragment of presburger arithmetic with unary uninterpreted predicates is undecidable. CoRR **abs/1703.01212** (2017)
24. Kessler, A.M.: Elon musk says self-driving tesla cars will be in the us by summer. The New York Times **19** (2015)
25. Kuvaiskii, D., Faqeh, R., Bhatotia, P., Felber, P., Fetzer, C.: Haft: Hardware-assisted fault tolerance. In: Proceedings of the Eleventh European Conference on Computer Systems. EuroSys '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2901318.2901339, https://doi.org/10.1145/2901318.2901339
26. Kuvaiskii, D., Fetzer, C.: δ-encoding : Practical encoded processing (2015)
27. Kuvaiskii, D., Oleksenko, O., Arnautov, S., Trach, B., Bhatotia, P., Felber, P., Fetzer, C.: Sgxbounds: Memory safety for shielded execution. In: Proceedings of the Twelfth European Conference on Computer Systems. p. 205–221. EuroSys '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3064176.3064192, https://doi.org/10.1145/3064176.3064192
28. Kuvaiskii, D., Oleksenko, O., Bhatotia, P., Felber, P., Fetzer, C.: Elzar: Triple modular redundancy using intel avx (practical experience report). 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks

(DSN) (Jun 2016). https://doi.org/10.1109/dsn.2016.65, http://dx.doi.org/10.1109/DSN.2016.65
29. Lammich, P.: Efficient verified (UN)SAT certificate checking. Journal of Automated Reasoning **64**(3), 513–532 (2020)
30. Laprie, J.C.: Dependability: Basic concepts and terminology. In: Dependability: Basic Concepts and Terminology, pp. 3–245. Springer (1992)
31. Leucker, M., Sánchez, C., Scheffel, T., Schmitz, M., Schramm, A.: Tessla: Runtime verification of non-synchronized real-time streams. In: ACM Symposium on Applied Computing (SAC). ACM, ACM, France (04/2018 2018)
32. Leucker, M., Schallhart, C.: A brief account of runtime verification. The Journal of Logic and Algebraic Programming **78**(5), 293 – 303 (2009), the 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07)
33. Lynch, N.A.: Input/output automata: Basic, timed, hybrid, probabilistic, dynamic, .. In: Amadio, R.M., Lugiez, D. (eds.) CONCUR 2003 - Concurrency Theory, 14th International Conference, Marseille, France, September 3-5, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2761, pp. 187–188. Springer (2003). https://doi.org/10.1007/978-3-540-45187-7_12, https://doi.org/10.1007/978-3-540-45187-7_12
34. Lyyra, A.K., Koskinen, K.M.: With software updates, tesla upends product life cycle in the car industry. LSE Business Review (2017)
35. Moore, E.F., et al.: Gedanken-experiments on sequential machines. Automata studies **34**, 129–153 (1956)
36. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(t). Journal of the ACM **53**, 937–977 (November 2006)
37. Palin, R., Ward, D., Habli, I., Rivett, R.: Iso 26262 safety cases: Compliance and assurance. vol. 2011 (09 2011). https://doi.org/10.1049/cp.2011.0251
38. Ramadge, P., Wonham, W.: Supervisory Control of a Class of Discrete Event Processes, vol. 25, pp. 206–230 (01 1987). https://doi.org/10.1007/BFb0006306
39. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach (Third Edition) (2010)
40. Salfner, F., Malek, M.: Architecting Dependable Systems with Proactive Fault Management, p. 171–200. Springer-Verlag, Berlin, Heidelberg (2010)
41. Saltzer, J.H., Reed, D.P., Clark, D.D.: End-to-end arguments in system design. ACM Trans. Comput. Syst. **2**(4), 277–288 (Nov 1984). https://doi.org/10.1145/357401.357402, https://doi.org/10.1145/357401.357402
42. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Computing Surveys (CSUR) **22**(4), 299–319 (1990)
43. Voigt, M.: The bernays-schönfinkel-ramsey fragment with bounded difference constraints over the reals is decidable. In: Dixon, C., Finger, M. (eds.) Frontiers of Combining Systems - 11th International Symposium, FroCoS 2017, Brasília, Brazil, September 27-29, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10483, pp. 244–261. Springer (2017)
44. Weihang Wu, Kelly, T.: Safety tactics for software architecture design. In: Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004. pp. 368–375 vol.1 (2004)
45. Åström, K.: Optimal control of markov processes with incomplete state information. Journal of Mathematical Analysis and Applications **10**(1), 174 – 205 (1965). https://doi.org/https://doi.org/10.1016/0022-247X(65)90154-X