Towards Certified Runtime Verification

Jan Olaf Blech[1], Yliès Falcone[2], and Klaus Becker[1]

[1] fortiss GmbH, Munich, Germany {blech,becker}@fortiss.org
[2] Université Grenoble I, Grenoble, France ylies.falcone@ujf-grenoble.fr

Abstract. Runtime verification (RV) is a successful technique to monitor system behavior at runtime and potentially take compensating actions in case of deviation from a specification. For the usage in safety critical systems the question of reliability of RV components arises since in existing approaches RV components are not verified and may themselves be erroneous.
In this paper, we present work towards a framework for certified RV components. We present a solution for implementations of transition functions of RV monitors and prove them correct using the Coq proof assistant. We extract certified executable OCaml code and use it inside RV monitors. We investigate an application scenario in the domain of automotive embedded systems and present performance evaluation for some monitored properties.

1 Introduction

Behavioral guarantees are an important prerequisite for using embedded systems in safety critical environments. Runtime verification [HG05,PZ06,FFM09,BH11] (RV) has become an important technique to monitor a system's behavior at runtime and take compensating actions in case of deviation from a specification. In RV, a system is typically extended with instrumentation code that communicates with a monitor. The monitor may be realized as an external program, the monitor is then referred as an outlined monitor. Once an abnormal behavior is detected, the monitor tries to bring the system into a fail-safe state using some feedback loop. This increases the confidence to handle system errors appropriately when the system is running, and, helps discovering them during testing.

Going one step beyond classical RV: for achieving an even higher level of confidence the question of whether an RV system itself has been implemented correctly arises. We address this question in this paper. In particular we guarantee that runtime-monitors do indeed monitor the desired specification and show the practicability of these runtime monitors for regular properties expressed with regular expressions.

The described approach targets OCaml based runtime monitors and their verification using the Coq proof assistant [Coq12]. Coq based runtime monitors can be extracted as OCaml code and verified in the Coq environment. In the context of this paper, the code that is verified and extracted out of Coq is said to be *certified*.

Our approach is suitable for *regular expression based properties* in the embedded systems domain. As an analyzed and implemented example we are discussing properties that can be monitored while analyzing the traffic on a bus structure. This paper aims to demonstrate that *certified runtime verification* is feasible and can be used in safety-critical application domains. More precisely, this paper features the following contributions:

- Verified runtime monitors automatically generated out of Coq and a method to verify regular expression based monitors.
- An OCaml based framework for runtime monitoring and its evaluation.
- An experimental application for monitoring properties inspired by needs from the automotive embedded systems domain. Furthermore, we present a description of the ingredients of a certified RV system and dependencies between different RV components.

The main focus of the experimental application shall demonstrate the possibility to integrate the approach for real applications in the embedded systems domain especially in the automotive area. We are not at the stage of deploying a concrete application in a car. Many parameters are not fixed yet, e.g., the embedded devices and the exact type of bus. For this reason we evaluate some aspects of the approach using standard hardware in this paper, but mention the constraints for real applications in the automotive area.

Paper Organization This paper is structured as follows. Related approaches are discussed in Section 2. Section 3 introduces guiding examples from the automotive domain that are regarded throughout this paper. Prerequisites including definitions on RV correctness are described in Section 4. Our OCaml based runtime monitors are introduced in Section 5. Section 6 describes runtime monitor formalization, and correctness proofs. An evaluation including a study on the integration of our Coq based code is presented in Section 7 and Section 8 features a conclusion and ideas for future work.

2 Related Approaches

Runtime Verification principles Over more than a decade, the field of *runtime verification* has produced many frameworks dedicated to the verification of the behavior of monolithic programs w.r.t. user-defined specifications. From an abstract point of view, most of these approaches proceed as follows. Two inputs are needed: a user-defined specification characterizing some desired or proscribed behavior, and an implementation to be runtime-checked. The (abstract) events of this specification are related to the (concrete) events of the program. An instrumentation technique is then used to observe the execution of these events when the program is executed. The so-called monitor is generated from the specification. A monitor is a decision procedure for the specification: it is fed by events coming from the program and indicates specification fulfillment or violation.

RV implementations Many tools have been proposed as implementations of the existing runtime verification frameworks. A large effort is the Java-MOP line of work conducted by Rosu et al. (see [MJG+11] for an overview). Java-MOP uses as input specifications which can be written in different formalisms (e.g., LTL, regular expression, context-free grammars). Java-MOP allows to generate an AspectJ aspect that instruments the underlying program (using weaving) and embeds the (automatically generated) monitor. Besides its genericity, Java-MOP is also efficient as demonstrated by experimentation.

On the other hand, a series of tools and approaches are based on the (less efficient) paradigm of rewriting. These approaches focus on expressiveness of the specification formalism (rather than the runtime efficiency). A major effort in this regard is the endeavor conducted by Barringer and Havelund with the tools Eagle [BGHS04], RuleR [BGHS04,BRH10], LogScope [BGHS10], and TraceContract [BH11]. Eagle handles LTL formulae as input and uses the techniques of progression that was proposed earlier in planning. RuleR is a more general system where specifications are directly encoded as a set of rewrite rules. This confers RuleR the ability to handle very expressive specifications. From an abstract point of view, LogScope can be seen as a variant of RuleR internally using state-machines. TraceContract is an embedding of LogScope in the Scala programming language (as an internal domain-specific language).

In addition to these two major research endeavors there are the tools TraceMatches [AAC+05], JLO [SB06], and LARVA [CPS09]. TraceMatches is an extension of AspectJ allowing to write regular expressions over pointcuts. JLO allows to generate monitors from LTL formulae where events are AspectJ pointcuts. Finally, LARVA allows to monitor different specification formalisms such as Lustre and duration calculus. LARVA translates specifications into the so called dynamic event timed automata and then uses AspectJ to weave the monitor.

Formal Treatment As for the formal verification part of the work, a summary of usages of RV for certification has been proposed by Rushby [Rus08]. Additional ideas for monitoring systems in the context of certification are stated in [SH11]. Up to our knowledge, we are the first who have actually realized and evaluated certified RV monitors. Moreover, none of the previous tools or framework contains a certified subset. The presented verification technique, explicitely establishing a simulation relation that captures characteristics between property and states of a monitor in Coq, is similar to one of the authors work on compiler verification [BG11].

3 Guiding Examples

Figure 1 presents an abstract view on a guiding example from the embedded automotive system domain. There are sensors, a control unit and an actuator connected to a bus with with some timing guarantees[3]. The control unit receives

[3] cf. existing bus systems used in the automotive area with timing guarantees in the embedded domain: FlexRay [FRay05], TTEthernet [SKS10] or TTP [Kop93].

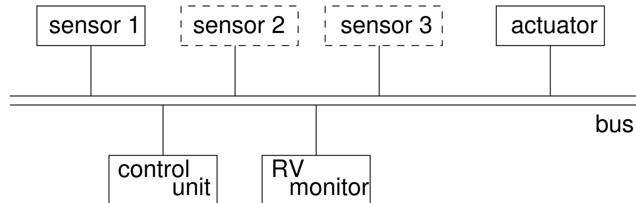

Fig. 1. An example bus

data from the sensors and processes them. Based on this, the actuator receives messages from the control unit. Sensor 1 is mandatory, while the other sensors are optional and can be added later.

The monitor observes the bus. In particular, we are interested whether the communication with actuator and sensor conforms to a given specified protocol. If the protocol is violated, the monitor notices this and can trigger a handling for this problem. However, the error-handling itself is not part of the monitor. Different kinds of errors due to hardware failures (e.g., bit-shifts, packet loss) or software errors can occur and need to be detected.

The bus features a time signal every 10 ms. The fact that reasonable data is sent on the bus by either a sensor, the actuator, or the control unit is abstracted into events.

In a real system, the first sensor might be a manual control providing a user interface for a functionality that is realized as an actor, while the two other sensors might analyze the environment to improve the service quality. With the advent of real-time operating systems that provide some timing and non-interference guarantees for parallel execution (e.g., PikeOS [KW]) it becomes possible to execute the control unit and the monitor in parallel even on the same processor core. The rate of arriving events is typically in the lower ms range, while an operating system's context switch can be typically done in a few μs.

A special feature of the given example is the ability to add and remove components to the IT system of a car during runtime, including connecting and disconnecting them from a bus. It is especially crucial that some core system behavior is preserved – and runtime monitored in this process. Two example applications and properties are regarded in the scope of this paper

3.1 A Rain-Sensor Application

A Rain-Sensor senses whether it is raining and determines the rain intensity. It sends the calculated intensity value to a Wiper-Controller, which analyses the value and sends control values to a Wiper-Actuator. Communication is done using the bus. The communication between system components is shown in Fig. 2. For the given example we have realized the communication between sensor and controller in the following way: The Rain-Sensor encodes the rain intensity in 10 single event bits, followed by a parity event bit P. Each event

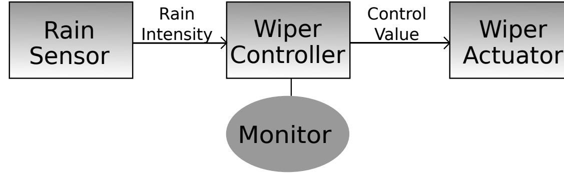

Fig. 2. Example application

can be either 0 or 1. The more 1s are sent, the higher is the rain intensity. After this sequence of 11 events has been sent, the next sequence is sent, transmitting a new value of rain intensity. The corresponding regular expression is: $1(1)^n(0)^m0P \quad$ such that $n + m = 8$ and $P \in \{0, 1\}$.

The regular expression is checked by a monitor. In addition to this, the monitor checks the parity bit P. If the number of previous 1s was odd (n was even) P shall be 1. In addition, the monitor can check another consistency condition that the rain-intensity (number of 1s) in two succeeding sequences should only change by an amount of 2. This consistency condition reflects that the rain amount is unlikely to change much in the very short time interval of 10 ms. For instance, the sequence 11100000001 can be followed by 11111000000, but not by 11111110001.

Regarding the communication between controller and actuator: The controller counts the number of 1s (denoted #1) received from the Sensor per sequence (without counting the parity) and sends a control value $C \in \{0, 1, 2\}$ to the actuator after each sequence. If $0 \leq \#1 \leq 2 : C := 0$; if $3 \leq \#1 \leq 6 : C := 1$; otherwise $C := 2$. The value of C denotes the speed of the wiper.

Due to the constraint that #1 change at most by 2 in two succeeding sequences, the sequence of sent control values C is also constrained. After C was 0, C cannot become 2, but only 0 or 1. Assuming that $C = 0$ is the start and end state, the possible sequences of C can by expressed by $0^+(1^+(2^+1^+)^*0^+)^*$. This can be verified during runtime by a second monitor.

The monitors observe if both the sequence of received values by the controller and the sequence of sent control values are valid.

In addition to properties specified by regular expressions, here we also observe behavior that is not specified in the regular expression itself: we check the "maximal rain-intensity variation by two" property between two signals and the parity bit. Such properties are checked by our monitors and can be specified in Coq and generated out of the Coq specification. However, only the regular expression part is verified. It is important for us, that our framework supports monitors that are able to observe additional constraints – thus, these monitors are more restrictive than the regular expression. This enlarges the monitor but since these aspects may be less critical they do not have to be verified. In the given example, however, it would be possible to specify the extra constraints by large regular expressions.

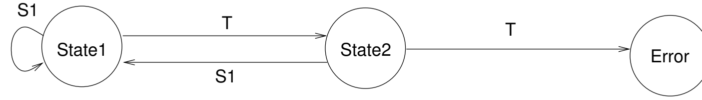

Fig. 3. Automaton for property 1

3.2 Timing-Properties

Our bus systems typically features some time events. This is used to evaluate properties requiring that certain signals arrive in certain time intervals. The bus systems features the following events:

$$\Sigma = \{tick(T), actuator(A), sensor_1(S_1), sensor_2(S_2), sensor_3(S_3)\}$$

We assume the requirement that between two time ticks (10 ms interval) there is at least one $sensor_1$ event and the *actuator* status should be updated. Furthermore, either $sensor_2$ or $sensor_3$ should send a message event between two ticks.

The formalization of the previous requirement is ensured by the conjunction of the three following properties expressed with regular expressions:

1. $\phi_1 = (S_1 + T.S_1)^*.(T + \epsilon)$
2. $\phi_2 = (A + T.A)^*.(T + \epsilon)$
3. $\phi_3 = ((S_2 + S_3) + T.(S_2 + S_3))^*.(T + \epsilon)$

When abstracting from the other events, the first property can be realized by an automaton as depicted in Fig. 3.

4 Prerequisites

We now introduce prerequisites for our RV framework: a formalization of the notions of correctness of RV and the minimal concepts of regular expressions used to state properties.

4.1 A Formalization of RV Basics

In our RV scenario we distinguish a syntactical representation (e.g., the source code) of a system s, its instrumentation s^I and the syntactical representation of a monitor m. The instrumentation may modify the source code of a system.

– Assuming that an operational semantics can be assigned to s and that a *system state* of type Σ_s and *concrete events* of type Σ_c can be observed during runs associated with this semantics. The semantics of s is given by a function σ returning a set of traces in $(\Sigma_s \times \Sigma_c)^*$. The Σ_c events abstract system states. The motivation for distinguishing system states and concrete events is that system states represent all important information to determine the control flow and important actions of the system. Concrete events explicitly specify observable system behavior.

- The semantics of s^I is given by another function – for simplicity it is also denoted σ returning a set of traces. Each trace has the type:

$$(\Sigma_s \times \Sigma_c \times (\Sigma_a \cup \{ignore\}))^*$$

  Each trace element comprises a tuple of a system state, two corresponding events of system events in Σ_c and their instrumented counterparts in $\Sigma_a \cup \{ignore\}$. In the case of *ignore* no abstract corresponds to a concrete one. In the deployed RV system this is used to reduce communication overhead between the instrumented system and the monitor.
- Monitors are defined as state transition systems comprising monitor states m_{States} and a transition function

$$m_{Step} : (m_{States} \times \Sigma_a) \rightarrow (m_{States} \times bool)$$

  taking a monitor state and an abstract event and returning a new (updated) monitor state and a verdict.
  In addition to this, the monitor comprises code for communicating with the instrumented system and calling m_{Step}.
- The semantics of m with s^I running in parallel is denoted: $s^I||m$.
  The semantics is given by another function – for simplicity it is also denoted σ as a set of traces, each trace comprising tuples of four components. A trace has the type: $(\Sigma_s \times \Sigma_c \times (\Sigma_a \cup \{ignore\}) \times bool)^*$
  Traces comprise system states Σ_s, concrete events Σ_c, their instrumented counterparts $\Sigma_a \cup \{ignore\}$ and truth values returned by the monitor.

Traces of systems Our correctness definitions use the projection of sets of traces (e.g., given by one of the σ functions) to the Σ_s^* parts (denoted T_s for a set of traces of tuples T). Furthermore, we need projections for the Σ_c^* and Σ_a^* (denoted T_c and T_a for a set of traces of tuples T). The event *ignore* is omitted in this projection. A projection for sets of the $bool^*$ parts of traces of tuples (denoted T_b for a set of tuples of traces T) is also needed.

The fact that a system s can generate a trace t ($t \in \sigma(s)$) is denoted $s(t)$. Likewise we introduce the notations $s^I(t)$ and $(s^I||m)(t)$ and projections $(\sigma(s^I))_c$ and $(\sigma(s^I||m))_c$ to restrict sets of traces to the Σ_c^* parts. The fact that a monitor m accepts a trace t_a is denoted $m(t_a)$. In the case of safety properties this means that its output trace contains only *true*.

Correctness of instrumentation It is possible that the instrumentation or running the monitor in parallel with the instrumented system does change the semantics of the original system due to side effects. Thus, we require a definition of correct instrumentation and monitor integration with respect to the uninstrumented system. This correctness definition is done by checking equality of sets of (projected) sets of traces.

Abscence of side-effect of instrumentation is defined as:

$$\sigma(s)_s = \sigma(s^I)_s \text{ and } \sigma(s)_c = \sigma(s^I)_c$$

Furthermore, the instrumentation is responsible for abstracting concrete events. Abstract and concrete events shall correspond to each other using an abstraction function $\psi : \Sigma_c \rightarrow (\Sigma_a \cup \{ignore\})$. Thus, we require that applying ψ to the Σ_c component in a tuple in each trace is equal to the $\Sigma_a \cup \{ignore\}$ component.

Correctness of instrumentation is defined as the conjunction between absence of side-effects and the correspondence of concrete and abstract events.

Correctness of monitor integration Correctness of monitor integration requires a correct instrumentation and the following conditions:

- The monitor does not pose any side-effects on the instrumented system: $\sigma(s^I)_s = \sigma(s^I||m)_s \wedge \sigma(s^I)_c = \sigma(s^I||m)_c \wedge \sigma(s^I)_a = \sigma(s^I||m)_a$
- The monitor is not effected by side-effects: $\forall\ t \ \in (\Sigma_s \times \Sigma_c \times (\Sigma_a \cup \{ignore\}) \times bool)^* \ .\ \sigma(s^I||m)_b(t) \quad \text{iff} \quad m(t_a)$

Monitor correctness Correctness of a monitor m_φ is defined with respect to a property φ. When a trace t fulfills a property φ, we note it $\varphi(t)$. Thus, monitor correctness is defined as: $\forall\ t_a \ \in \Sigma_a^* \ .\ \varphi(t_a) = m_\varphi(t_a)$ Correctness of monitor implementation correctness is defined as:

Combined correctness properties An RV system $s^I||m_\varphi$ is considered correct with respect to a property φ iff:

- the instrumentation is done correctly,
- the monitor has been integrated correctly, and
- the monitor is correct with respect to φ.

In addition, it has to be ensured that the system is a correct deployment and compilation of $s^I||m_\varphi$ which is not in the scope of this paper.

4.2 Regular Expressions

Regular expressions are defined as an inductive datatype Φ_{Σ_a} parameterized with a set of (abstract) events Σ_a. In our Coq formalization they comprise constructors for atoms, concatenations, disjunctions, the star and plus operators, and the epsilon (corresponds to an empty list of events) and empty expressions (corresponds to nothing). Furthermore, we have defined abbreviations to ease the specification like concatenating n-times the same (sub-)regular expression to each other.

Semantics of regular expressions The semantics of a regular expression is defined in the usual way, by associating a regular expression ϕ with a function that indicates whether a list of events el is in the language described by ϕ. We note $\phi(el)$ when this is the case. We define an equivalence relation $\simeq$ over regular expressions ϕ, ϕ' that describe languages over the same vocabulary Σ:

$$\forall\ el \in \Sigma^*.\ \phi(el) = \phi'(el) \text{ iff } \phi \simeq \phi'$$

This partitions syntactical representations of regular expressions into semantic equivalence classes.

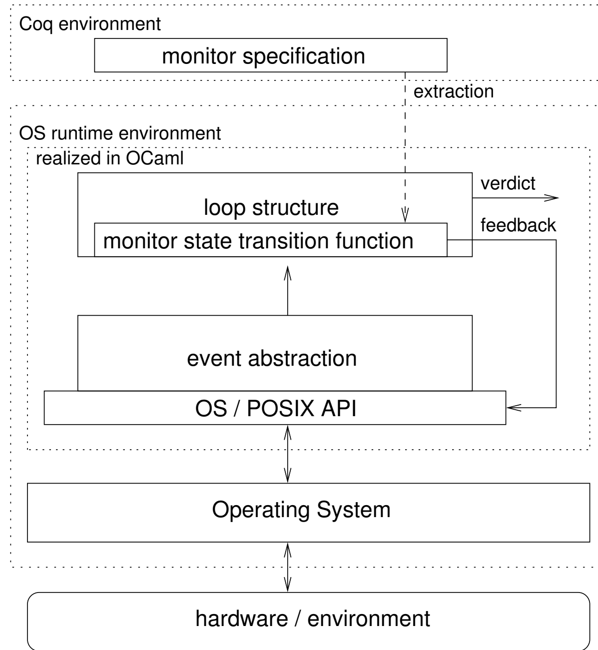

Fig. 4. Our RV monitoring environment

Accepting an event When comparing a regular expression ϕ to a list of events, it is helpful to define an operator that returns a modified regular expression that captures the effect of having consummed an event e. For this reason, we define the / operator that has the following property:

$$\forall\ e\ el\ .\ \phi(e :: el) \quad \text{iff} \quad \phi/e(el)$$

where $e :: el$ is the OCaml notation for the concatenation of an event e to a list of events el. Note that, for space reasons, we do not give the full formal definition specifying how this is achieved for regular expressions, but this is part of our Coq formalization.

5 OCaml Based RV Monitors

Our RV framework distinguishes between an instrumented system and monitors. Outlined monitors observe the system behavior by using information provided by the instrumented system.

Figure 4 shows the environment in which our RV monitors are deployed. RV monitors are relatively small pieces of software and are in this work realized as a state-transition function which performs transitions of the internal monitor state. This state-transition function is called from a loop structure over and over

again, thereby consuming behavioral events from the system and emitting an overall system status value to indicate possible deviations from an acceptable system behavior. The events received by the RV monitor are originating from abstraction functions which observe, e.g., a stream from a Posix socket. This stream can contain concrete events which are originating from the instrumented system. Here, the monitor functionality is realized in OCaml. Communication with the rest of the system and the environment is done using the operating system API.

While we generate the state transition functions for our RV monitors out of Coq specifications which are verified in the Coq environment, the state transition functions have to be embedded into a monitor environment which takes care of the monitor's communication with the system, as shown in Fig. 4.

The OCaml part of the monitor is divided into three parts:

- A generic part, that takes care of the Posix based communication with the environment and is the same for all of the monitors targeted in this paper.
- A verified state transition generated out of Coq and all depending definitions.
- A file that realizes glue code for the interaction between the generic part and the generated state transition function. Naturally this file has to be adapted for each individual monitor. It contains abstractions (cf. Section 4.1), e.g., of network packets to events. The abstractions can be verified optionally.

6 Certified Monitors with Coq

We describe how we obtain certified monitors using Coq: the formalization of a monitor, of regular expressions, monitor extraction and the correctness proofs.

6.1 Formalization of Monitor Correctness in Coq

We have adapted an existing library (definitions, morphisms and lemma) for handling regular expressions in Coq in order to work with our definitions of events [4]. Small adaptations were necessary since the original library only handles regular expressions of a particular String type.

We define the following artifacts in Coq:

- Possible events as abstract data types.
- States of monitors as data types and state transition functions written in a functional style.
- The regular expression that specifies the correctness property.
- The statement that the monitor corresponds to the regular expression, using simulation.

[4] Available at http://coq.inria.fr/pylons/contribs/view/RegExp/v8.4 by Takashi Miyamoto.

OCaml program extraction Due to the functional nature of our monitors, they can be extracted using the Coq command **Recursive Extraction**. The resulting extraction provides a state-transition function and definitions of the used datatypes and possible auxiliary functions. Extraction follows the Coq definitions. This means that our choice of datatypes can influence the performance of monitors. If non-performant definitions are used (e.g., a definition of natural numbers as abstract datatype using 0 and successor constructors) non-performant implementations will be generated.

6.2 Verification of a Monitor with Respect to a Regular Expression

We describe the verification of a monitor with respect to a regular expression in Coq. We establish monitor correctness and assume a state transition function of the internal state of the monitor (cf. Section 4):

$$m_{Step} : (m_{States} \times \Sigma_a) \rightarrow (m_{States} \times \mathit{bool}).$$

To verify monitor correctness for a property ϕ, we first establish a simulation relation R between states (m_{States}) and regular expressions of type Φ – each alphabet of events instantiates a parameterized type of regular expressions; logically this results in an distinguishable types for each alphabet – with $\phi \in \Phi$:

$$R : m_{States} \times \Phi \rightarrow \mathit{bool}$$

The relation performs a check on the semantic correspondence of states using the $\simeq$ relation on regular expressions (see Section 4.2). Finding regular expressions corresponding to a state is done using Arden's Lemma [Ard61]. The proof is done using an induction. The initial case is resolved by using (1). The induction step is proven using the property (2) described below. The following items have to be proven:

1. The first property that is proven states that the initial state of m_{Step} and ϕ are in the simulation relation. This is the basis for proving that m accepts the same lists of events as specified by ϕ.
2. We prove the following property (step-relation correspondence) using the implication $\longrightarrow$:

   $$\begin{array}{l} \forall\ m\ m'\ :\ m_{States}\quad \phi\ \phi'\ :\ \Phi\quad e\ :\Sigma_a\quad b\ :\ \mathit{bool}\ . \\ \ m_{Step}(m,e) = (m',b) \longrightarrow \phi' = \phi/e \longrightarrow R(m,\phi) \longrightarrow \\ \ R(m',\phi') \end{array}$$

   It states: for a regular expression ϕ and a state m in the simulation relation R, the succeeding state m' after processing one event e and a succeeding regular expression ϕ' are in the simulation relation again. ϕ' corresponds to accepting (using the / operator) the same event e on ϕ. A Coq formalization of this property is shown in Fig. 5. We use `s`, `s'` to indicate states of type `StatesAutomaton1`. The message alphabet has type `MAa1`. The `=R=` is a relation denoting equivalent regular expressions. `derive` realizes the / operator.

```
Lemma step_correspondence :
 forall (e : MAa1) (s s' : StatesAutomaton1) (r r' : RegExp MAa1),
   s' = (fst(A1Step s e)) ->
   r' =R= derive MAa1 dec_MAa1 e r ->
   regexp_states_rel_a1 s r ->
   regexp_states_rel_a1  s' r'.
```

Fig. 5. Coq formalization of step-relation correspondence

Our correctness criterion for safety invariants states that each finite prefix of event streams (our regular expressions typically work on potentially infinite streams) will only result in non-error states iff the regular expression gets non empty. We prove a stronger property first which implies the correctness criterion and requires that all encountered states and the acceptance state of the regular expressions are in the relation.

Non-safety properties are characterized by the fact that finite prefixes of a trace do not have to fulfill the property even if the entire trace does. It remains possible to prove an adequate simulation relation, and, based on this derive that at least in distinct states a property holds using the method described above.

Example relation An example relation for the property checked by the automaton from Fig. 3 associates states to their corresponding regular expressions:

$$\begin{array}{rcl} \text{S1} & \simeq & (S_1 + T.S_1)^*.(T + \epsilon) \\ \text{S2} & \simeq & (S_1.(S_1 + T.S_1)^*.(T + \epsilon)) + \epsilon \\ \text{ERROR} & \simeq & \mathit{Empty} \end{array}$$

Figure 6 shows the same simulation relation in Coq. The events and states have slightly different names to make them usable together with other automata in the same file. In addition, we also have defined some abbreviations to make the look of constructors Star, Atom, Eps closer to the mathematical notations during the interactive proofs. It can be seen that regular expressions used inside the relation can become larger than the original property. In case of wrong relations, however, we will not be able to establish an overall correctness proof. Thus, the size of the relation does not enlarge the trusted computing base of our approach.

6.3 Verification of Abstractions

It is convenient to deliver only certain events to a monitor. For this reason, we have introduced abstraction functions in Section 4. We verify abstractions by specifying them in Coq and proving the required properties, e.g., for an abstraction function *abs* and an abstract event e_a and a set of possible concrete events Σ_c:

$$\forall\ e_c\ :\ \Sigma_c\ .\ \ e_c = \text{specification of concrete events} \longrightarrow e_a = abs(e_c)$$

Proofs are straightforward. The extraction of OCaml code from Coq has to take into account that datatypes are sometimes defined in a different way in Coq and

```
Definition regexp_states_rel_a1
   (s : StatesAutomaton1) (r : RegExp MAa1) : Prop :=
match s , r with
| A1S1 , x => ((Star MAa1
      ((Atom MAa1 S_a1) || ((Atom MAa1 T_a1)  ++ (Atom MAa1 S_a1) ) ))
    ++ ((Atom MAa1 T_a1) || Eps MAa1)
    ) =R= x
| A1S2 , x => (((Atom MAa1 S_a1 ++
    Star MAa1
      (Atom MAa1 S_a1
       || (Atom MAa1 T_a1 ++ Atom MAa1 S_a1)))
   ++ (Atom MAa1 T_a1 || Eps MAa1)) || Eps MAa1)=R= x
| A1SError , x => Empty MAa1 =R= x
end.
```

Fig. 6. Simulation relation in Coq

OCaml. For instance, integers are defined using native processor arithmetics in OCaml, but are realized as an inductive datatype in Coq.

7 Evaluation

We evaluate our approach with respect to three criteria that are used in a first step aiming to assess the feasibility of certified RV in an industrial context: proving effort, integration into a bus simulator and performance of monitors.

Proving effort for regular expression based monitors For proving a new regular expression based monitor correct, one has to establish a relation comprising states and corresponding regular expressions. The main effort is the proof of step-relation correspondence which requires a case distinction on possible states and events ($|states| \times |events|$ different cases). Each case essentially requires some rewriting of equivalent regular expressions to prove the correspondence. This can require several lines of proof code for each event. Automation using tactics might be possible, but require some clever rewriting strategies which we have not developed currently. The other parts of the proofs are either relatively easy or can be reused (are generic) with some adaptations.

Generation of monitors from Coq specifications We have generated monitor state-transition functions out of our verified Coq formalizations. Coq allows the extraction of executable state-transition functions. Extraction works recursively, so all required types and depending functions are also extracted from their Coq specifications.

Integration of monitors into a bus simulator We have demonstrated the applicability of our approach by an implementation of a bus simulator for the

Rain-Sensor scenario from Section 3.1. The Rain-Sensor senses the rain intensity, sends the intensity value to a Wiper-Controller, which analyses the value and sends control values to a Wiper-Actuator. The example application is implemented in C++ using BSD Sockets for communication over UDP/IP. Only the Wiper-Controller is monitored. The communication between the controller and its monitor is based on Unix pipes. This solution can be used with any Posix compatible protocol. For example ethernet based bus implementations that fulfill real-time constraints (e.g., [SKS10]).

Performance evaluation of OCaml based monitors We have built an experimental setup to evaluate the performance of certified monitoring code. A trace generator creates multiple traces and send them to a certified monitor for analysis. Since the performance of the embedded hardware we are aiming at is still subject to change, we have conducted the experiments on different standard machines: Machine 1 is a Macbook Pro i7 at 2GHz, Machine 2 is a Pentium D at 3GHz, Machine 3 is Machine 2 down-clocked at 850MHz. Results are given in Table 1. In each cell, the indicated result (in seconds) is obtained by taking the mean value after a hundred executions. The table shows five properties. The first column shows the property under consideration. The entry **|tr.|** denotes the length of the traces sent to the monitor. The entry **no mon** (resp. **mon**) denotes the execution time in seconds when the trace is not monitored (resp. is monitored) by the certified monitors. The entry **ovhd** indicates the overhead induced by the monitor on the original system: $\frac{\textbf{mon - no mon}}{\textbf{no mon}}$. The entry **kevt/s** indicates the throughput of the monitor, i.e., how many thousands of events it can handle in a second. The timing properties are taken from Section 3.2. The monitors ϕ_4 and ϕ_5 monitor the rain-intensity as explained in Section 3.1.

Timings in Table 1 clearly substantiate our claim that the performance of certified runtime monitors is good. The overhead induced by the monitoring code on the initial system is negligible. This is due to the performance of the optimized code generated by the OCaml compiler. The throughput of the monitors is also very satisfactory. Actually, the similar performance results observed on Machine 2 and Machine 3 made us notice that the throughput of the monitor is actually not limited by the monitoring code but by the performance of the OS primitives used to establish communication between the system and the monitors. Note that, for traces of length 10^6, some erratic measures were taken on Machine 3, probably because of down-clocking. Thus, we could not report reliable numbers.

8 Conclusion and Future Work

We presented work towards certified RV by means of higher-order theorem provers. In particular, we have demonstrated the feasibility to generate OCaml based runtime monitors out of verified Coq formalizations. We have demonstrated the deployment in a bus simulator. Furthermore, we have presented a performance evaluation of OCaml based monitors in general. Our properties can

Table 1. Performance evaluation of certified monitors

ϕ	|tr.|	Machine 1				Machine 2				Machine 3			
		no mon	**mon**	**ovhd**	**kevt/s**	**no mon**	**mon**	**ovhd**	**kevt/s**	**no mon**	**mon**	**ovhd**	**kevt/s**
ϕ_1	10^4	.7055	.7056	.00022	438.94	.7119	.7119	0	75.98	2.3357	2.3357	0	81.69
	10^5	1.9884	1.9920	.0019		5.6453	5.6453	0		18.585	18.585	0	
	10^6	14.599	14.599	0		55.061	54.451	.0027					
ϕ_2	10^4	.7047	.7047	0	449.76	.7093	.7138	0	73.28	2.1986	2.2474	.03077	75.52
	10^5	2.0095	2.0095	0		5.6589	5.6469	.0027		17.5597	17.898	.0318	
	10^6	14.387	14.436	.003588		54.48	54.690	0.00405					
ϕ_3	10^4	.7050	.7051	.00022	445.79	.712	.7167	.00816	77.88	2.2309	2.2882	.03402	74.79
	10^5	2.0981	2.1032	.00267		5.6476	5.6738	.00539		17.465	18.207	.057	
	10^6	14.507	14.517	.00125		54.48	54.789	0					
ϕ_4	10^4	.7054	.7054	0	441.73	.7085	.7136	.00895	77.16	2.234	2.3109	.04426	73.85
	10^5	2.1334	2.1338	.00036		5.6778	5.6778	0		17.524	18.077	.041	
	10^6	14.343	14.343	0		54.974	54.974	0					
ϕ_5	10^4	.7047	.7051	.0064	450.32	.7093	.7127	.00652	74.07	2.1978	2.2261	.02127	78.92
	10^5	2.0754	2.0762	.00057		5.6273	5.6273	0		17.724	17.827	.01723	
	10^6	14.502	14.502	0		54.729	54.809	0.00204					

be checked in an acceptable time and it seems feasible to deploy the demonstrated solutions in industrial domains. The shown aspects are an important prerequisite for demonstrating the feasibility for large scale applications.

Despite being sufficient for the sketched property monitoring of bus messages, the properties regarded in this paper are relatively small and simple. As an academic goal future work should extend the expressiveness and also regard more complex properties. More particularly, it would be of great interest to be able to extract monitoring code for parametric properties, i.e., properties featuring parameterized events taking values at runtime. Recent advances [RC12] in runtime verification that give a semantics to these properties will certainly help. Furthermore, as a goal with a strong engineering focus we want to deploy monitors on real embedded hardware and industrial demonstrators as a next step. Dynamic aspects like adding or removing components during runtime and the impact on monitors are another aspects.

References

[RC12] G. Rosu, F. Chen Semantics and Algorithms for Parametric Monitoring Logical Methods in Computer Science vol. 8

[PZ06] A. Pnueli, A. Zaks PSL Model Checking and Run-Time Verification Via Testers FM 2006: Formal Methods, 14th International Symposium on Formal Methods LNCS vol. 4085

[HG05] K. Havelund and A. Goldberg Verify Your Runs Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005

[AAC+05] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding

trace matching with free variables to AspectJ. SIGPLAN Notices, ACM, October 2005.

[Ard61] D.N. Arden. Delayed-logic and finite-state machines. pages 133–151. IEEE, 1961.

[BGHS04] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. Verification, Model Checking, and Abstract Interpretation, LNCS vol. 2937, Springer, 2004. (VMCAI'04)

[BGHS10] H. Barringer, A. Groce, K. Havelund, and M. Smith. Formal analysis of log files. Journal of Aerospace Computing, Information, and Communication. The American Institute of Aeronautics and Astronautics, 2010.

[FFM09] Y. Falcone, J-C. Fernandez, L. Mounier. Runtime Verification of Safety-Progress Properties. Proc. of the 9th international workshop on Runtime Verification, LNCS vol. 5779, Springer, 2009.

[BH11] H. Barringer and K. Havelund. Tracecontract: a Scala DSL for trace analysis. Proc. of the 17th international conference on Formal methods, LNCS vol. 6664, Springer, 2011.

[BRH10] H. Barringer, D. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: from EAGLE to RuleR. Journal of Logic and Computation, Oxford University Press, June 2010.

[BG11] J. O. Blech und B. Grégoire Certifying compilers using higher-order theorem provers as certificate checkers. Formal Methods in System Design, Springer, 2011.

[CPS09] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. Larva — safer monitoring of real-time java programs (tool paper). Software Engineering and Formal Methods, pages 33–37. IEEE Computer Society, 2009. (SEFM'09)

[Coq12] The Coq development team. The coq proof assistant reference manual v8.4. 2012. Available at http://coq.inria.fr.

[FRay05] FlexRay communications system protocol specification version 2.1 FlexRay Consortium - May, 2005.

[KW] R. Kaiser and S. Wagner. The PikeOS Concept: History and Design. SysGO AG White Paper. Available: http://www. sysgo. com

[Kop93] H. Kopetz. TTP - A time-triggered protocol for fault-tolerant real-time systems. Fault-Tolerant Computing, IEEE, 1993.

[MJG+11] P. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu. An overview of the mop runtime verification framework. International Journal on Software Tools for Technology Transfer, 10.1007/s10009-011-0198-6.

[RC12] G. Rosu and F. Chen. Semantics and Algorithms for Parametric Monitoring. Logical Methods in Computer Science, 2007.

[Rus08] J. Rushby. Runtime Certification. Invited paper, Runtime Verification, LNCS vol. 5289, Springer Verlag, 2008. (RV'08)

[SKS10] T. Steinbach, F. Korf, T. C. Schmidt. Comparing time-triggered Ethernet with FlexRay: An evaluation of competing approaches to real-time for in-vehicle networks. 8th IEEE International Workshop on Factory Communication Systems, 2010.

[SB06] V. Stolz and E. Bodden. Temporal assertions using AspectJ. Proc. of the 5th Int. Workshop on Runtime Verification , volume 144(4) of *ENTCS*, Elsevier, 2006. (RV'05)

[SH11] M. Sridhar and K. W. Hamlen. Flexible in-lined reference monitor certification: challenges and future directions. Programming Languages meets Program Verification. ACM, 2011.