The Observer-Based Technique for Requirements Validation in Embedded Real-Time Systems

Jiale Zhou, Yue Lu, Kristina Lundqvist
School of Innovation, Design and Engineering, Mälardalen University, Västerås, Sweden
{zhou.jiale, yue.lu, kristina.lundqvist}@mdh.se

***Abstract*—Model-based requirements validation is an increasingly attractive approach to discovering hidden flaws in requirements in the early phases of systems development life cycle. The application of using traditional methods such as model checking for the validation purpose is limited by the growing complexity of embedded real-time systems (ERTS). The observer-based technique is a lightweight validation technique, which has shown its potential as a means of validating the correctness of model behaviors. In this paper, the novelty of our contributions is three-fold: 1) we formally define the observer constructs for our formal specification language namely the Timed Abstract State Machine (TASM) language and, 2) we propose the Events Monitoring Logic (EvML) to facilitate the observer specification and, 3) we show how to execute observers to validate the requirements describing the functional behaviors and non-functional properties (such as timing) of ERTS. We also illustrate the applicability of the extended TASM language through an industrial application of a Vehicle Locking-Unlocking system.**

***Index Terms*—model-based requirements validation; TASM; systems functional behaviors and non-functional properties; runtime monitoring; observer technique**

I. Introduction

Studies [1] [2] have revealed that most of the anomalies discovered in late phases of systems development life cycle can be traced back to hidden flaws in the requirements specification. These deficiencies in the requirements specification will usually lead to extensive rework costs and sometimes even unrecoverable failures. For this reason, requirements validation techniques play a pivotal role in increasing the confidence that the requirements are correct in the sense of consistent and complete. This is particularly true for embedded real-time systems (ERTS) which require a set of stringent requirements to describe their functional behaviors and non-functional properties.

With the growing maturity of the model-based development paradigm, executable requirements specifications (i.e., requirements models) become increasingly attractive to cope with the boosting complexity of modern ERTS as well as to reduce the underlying anomalies. In this scenario, requirements models with well-defined semantics can capture the intended behavior of the system and thus are used as the source of information for validation purpose. Model checking [3] is a rigorous approach to assuring that correctness properties hold for the system under development. In this technique, the system design derived from requirements is specified in terms of analyzable models at a certain level of abstraction. Further, requirements are formalized into verifiable queries and then fed into the models to be checked. In this way, requirements are reasoned about to resolve contradictions, and it is also verified that they are neither so strict to forbid desired behaviors, nor so weak to allow undesired behaviors. However, such validation technique suffers from the state explosion problem.

The need for a lightweight validation technique to avoid the aforementioned problem has motivated the development of our requirements validation technique via observers [4]. To be specific, we choose the Timed Abstract State Machine (TASM) language [5] as the requirements modeling language for ERTS, based upon its distinctive features in terms of the ability to specify systems' functional behaviors and non-functional properties, the low learning costs, and a toolset that supports model execution. Additionally, we extend the TASM language with the *Event* and *Observer* constructs to specify the corresponding requirements. When the TASM models are executed, they will generate a number of events reflecting the functional behaviors and non-functional properties of the system under consideration, which can be abstracted as a linear trace of events. An observer monitors the event trace and determines whether a given correctness property is satisfied by the system under consideration. In our previous work, we assume that the observer representing the property of interests can be specified by following the logic of regular expressions [6], and the entire event trace generated by the TASM model is available before the observer starts to monitor. The monitoring algorithm is implemented in the same way as searching for the sub-traces that match the observer regular expression. However, the main drawbacks of these assumptions are two-fold: 1) the expressiveness power of regular expressions falls short of expressing unordered fixed-count events where the occurrence multiplicities of these events are pre-defined but the corresponding order is random and, 2) the monitoring algorithm used in [4] can not be applied at runtime because of the assumption that the entire event trace is pre-achieved. Therefore, improving the logic used to specify observers is of paramount importance for our validation technique to achieve success in practice.

In this paper, we enhance our observer-based requirements validation technique via proposing a new observer specification logic that originates from the Extended Regular Expressions (ERE) and introducing a rewriting-based monitoring algorithm. Especially, the novelty of our contributions are three-fold: 1) we formally define the observer constructs for our formal specification language namely the Timed Abstract

978-1-4799-6334-8/14 © 2014 IEEE

RET 2014, Karlskrona, Sweden

Accepted for publication by IEEE. © 2014 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

State Machine (TASM) language and, 2) we propose the Events Monitoring Logic (EvML) to facilitate the observer specification and, 3) we show how to execute observers to validate the requirements describing functional behaviors and non-functional properties of ERTS. We also illustrate the applicability of our technique by using a Vehicle Locking-Unlocking (VLU) system.

The remainder of this paper is organized as follows: An introduction to the background knowledge is presented in Section II. The improved observer-based technique is described in Section III. Section IV illustrates the applicability of the extended TASM through an industrial application of the VLU system. Section V discusses the related work, and finally concluding remarks and future work are drawn in Section VI.

II. Background

In this section, we briefly introduce the formal specification language TASM used in our validation approach and ERE for better understanding of our work.

A. Timed Abstract State Machine

TASM [5] is a formal language for the specification of ERTS, which extends the Abstract State Machine (ASM) [7] with the capability of modeling timing properties and resource consumption of the target system. TASM inherits the easy-to-use feature from ASM, which is a literate specification language understandable and usable without extensive mathematical training [8]. A TASM model consists of two parts – an environment and a set of main machines. The environment defines the set and the type of variables, and the set of named resources which machines can consume. The main machine is made up of a set of monitored variables which can affect the machine execution, a set of controlled variables which can be modified by machines, and a set of machine rules. The set of rules specify the machine execution logic in the form of "if *condition* then *action*", where *condition* is an expression depending on the monitored variables, and *action* is a set of updates of the controlled variables. We can also use the rule "else then *action*" which is enabled merely when no other rules are enabled. A rule can specify the annotation of the time duration and resource consumption of its execution. The duration of a rule execution can be the keyword *next* that essentially states the fact that time should elapse until one of the other rules is enabled. TASM describes the basic execution semantics as the computing steps with time and resource annotations: In one step, it reads the monitored variables, selects a rule of which *condition* is satisfied, consumes the specified resources, and after waiting for the duration of the execution, it applies the update set instantaneously. If more than one rules are enabled at the same time, it non-deterministically selects one to execute. As a specification language, TASM supports the concepts of parallelism which stipulates TASM machines are executed in parallel, and hierarchical composition which is achieved by means of auxiliary machines which can be used in other machines. There are two kinds of auxiliary machines - *function* machines which can take environment variables as parameters and return execution result, and *sub* machines which can encapsulate machine rules for reuse purpose [5]. Communication between machines, including main machines and auxiliary machines, can be achieved by defining corresponding environment variables.

B. The Extended Regular Expressions

The Extended Regular Expressions (ERE) [6] represent a succinct and useful technique to specify patterns in strings by inductively utilizing the *union* $(+)$, *concatenation* $(\cdot)$, *repetition* $(*)$ and *complementation* $(\wedge)$ operators. There are programming and/or scripting languages, such as Perl and Java, which are mostly based on efficient implementations of pattern matching via ERE. Because of their convenience in specifying patterns, ERE have many applications including but not limited to text searching. The observer-based technique (a.k.a. runtime monitoring or runtime verification) is one of the application areas of interests for ERE. Since the running behaviors of computer programs or executable models can usually be abstracted as a linear trace of events or system states, the main idea behind the observer-based technique is to specify a set of observers (i.e., the extended regular expressions) that monitor the received events or system states and report abnormalities. Then, the monitoring process can be regarded as solving the membership problem for an extended regular expression R and a given word $\omega = a_1 a_2 \ldots a_n$ (a_n represents an event or a system state), which is to decide whether ω is in the regular language generated by R.

The observer-based technique usually assumes that the events or system states are received incrementally, i.e., each event is supposed to be processed as it arrives. An efficient implementation of the incremental membership problem are of critical importance to this application. A rewriting-based algorithm has been proposed in [9] for monitoring system events. The intuition is that in order to incrementally check the membership of an incoming trace of events to a given extended regular expression, the algorithm can process the events as soon as they are available by rewriting the extended regular expression contingently. Since the event is consumed incrementally in this way, the event consumption idea is more suitable for runtime monitoring, comparing the monitoring algorithm used in [4].

III. The Extension of TASM

In this section, we introduce the extension of TASM in terms of the fundamental concepts, the Events Monitoring Logic, and the observer execution process.

A. The Fundamental Concepts

The extended constructs comprise two main parts, i.e., *TASM Event* and *TASM Observer* as shown in Figure 1, which defines the meta-model of the extended TASM language.

Definition 1: TASM Event. An event e is a tuple $<E, t, r_1, r_2 ...>$, where E defines the type of an e in the sense of *ResourceUsedUpEvent* (*ReUUE*), *ChangeValueEvent* (*ChVE*), *RuleEnableEvent* (*RuEE*), and *RuleDisableEvent*

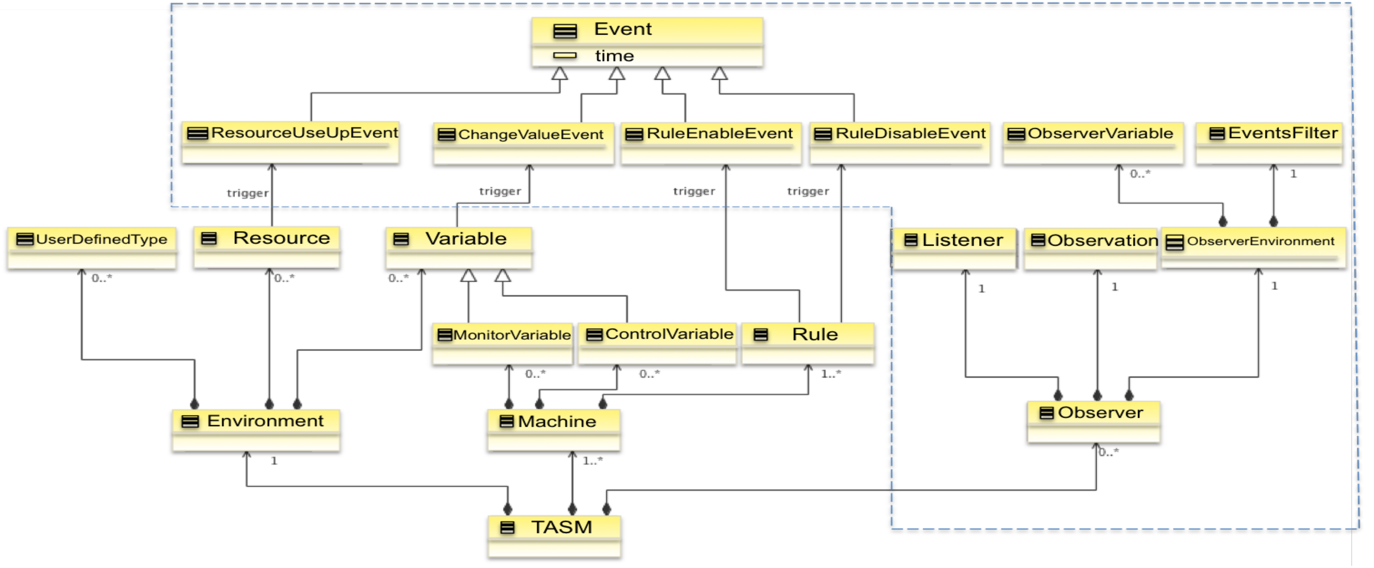

Fig. 1. The Meta-model of the extended TASM language.

(*RuDE*), t records the time stamp when the event occurs, r_1, r_2, *etc.* denotes the possible consumed resources by the event.

In our extension, an event e which is triggered by the corresponding TASM construct c, can be referenced in the form of $c{\to}e$. To be specific, the event of $ChangeValueEvent$ type triggered by a specific TASM environment variable whenever its value is updated, is referenced in the form of *VariableName→ChVE*. The $ResourceUsedUpEvent$ type triggered by the case whenever the resource of the application is consumed up, is referenced in the form of *ResourceName→ReUUE*. The $RuleEnableEvent$ (resp. $RuleDisableEvent$) type triggered whenever a specific TASM rule is enabled (resp. disabled), is referenced in the form of *MachineName→RuleName→RuEE* (resp. *MachineName→RuleName→RuDE*). Similarly, the time stamp t of an e is referenced in the form of $e{\to}t$ and the consumed resource r in the form of $e{\to}r$. Examples illustrating how a certain event is referenced can be found in Figure 10.

Here are some useful definitions which are related to *TASM Event*:

Definition 2: Event trace. An event trace is a finite sequence of events, denoted by $\omega = e_1 e_2 \dots e_n$.

Definition 3: Event pattern. An event pattern is an expression following a certain logic to describe a set of event traces of interests in a compact and succinct way, denoted by E. The set of the event traces of interests (i.e., matching the EvML expression E), are denoted by $\mathcal{L}(E)$.

The TASM Observer monitoring events is defined as comprising:

Definition 4: TASM Observer. An observer *ob* is a tuple $<OE, L, Obv>$, where:

- OE denotes the $ObserverEnvironment$, which is defined as a tuple $<OV, TU, EF>$, where
  - OV denotes the $ObserverVariables$, which defines a set of local typed variables that can only be used by the L and Obv defined in this observer,
  - TU denotes the $TypeUniverse$, which is a set of types that include the TASM primitive types (i.e., $Reals$, $Integers$, $Boolean$ and *User-defined*) and the TASM extended types in terms of $Time$ and $Resource$,
  - EF denotes the $EventsFilter$, which defines a set of events considered to be relevant or irrelevant to the observer.
- L denotes the $Listener$, which is in the form of "**listening** *keyword*: *condition* **then** *action*", where the *keyword* can be either *compulsory* or *optional* which will be further explained in Section III-C, the *condition* specifies the event pattern following the Events Monitoring Logic (EvML) which will be defined in Section III-B, and the *action* is a set of actions updating the value of observer variables when the *condition* evaluates to be true.
- Obv denotes the $Observation$, which is a predicate representing the properties needed to validate. An observation can evaluate to be either true or false, depending on the value of corresponding observer variables.

B. The Events Monitoring Logic

The Events Monitoring Logic (EvML) is inspired by the extended regular expressions (ERE) that have been widely applied to solve the pattern matching problem. Although ERE can provide a compact and powerful implementation of pattern matching, we encounter an issue when we use it in our case. When specifying an event pattern by using ERE, the occurrence order of events is implied in the expression, e.g., $E = e_1 e_2$ implies that the event trace, where the occurrence of event e_1 is immediately followed by the occurrence of

event e_2, will match the pattern. However, in some cases, the occurrence order of events is trivial. For instance, when we monitor the synchronization of two events, we are merely concerned about whether both events do occur in the event trace, rather than which event comes earlier. To describe an event trace like this (i.e., unordered event trace, hereafter), ERE have to list all the possibilities of the occurrence order, which is a clumsy and error-prone task.

Therefore, we formally define a logic, namely the Events Monitoring Logic, which can specify an unordered event trace in a more elegant way. EvML inherits the basic syntax and semantics from ERE, which defines the set of interested events by inductively applying *union* $(+)$, *concatenation* $(\cdot)$, *repetition* $(*)$, and *complementation* $(^\wedge)$ operators. To solve the aforementioned issue, we introduce a new delimiter *parallel* and the *event multi-set expression* into EvML, denoted as $\|M\|$, in order to specify unordered event traces.

1) The EvML Syntax: For an alphabet Σ whose elements are the possible events, an EvML expression E over Σ is defined as follows:

$$E ::= \emptyset|\epsilon|e|E+E|E\cdot E|E^*|^\wedge E|\|M\|,$$

where $\emptyset$ denoting the empty set, ϵ denoting an empty event, and $e\in\Sigma$ denoting a regular event. M denotes the *event multi-set expression* over Σ, which is defined as a tuple $<\mathcal{A}, m>$:

- $\mathcal{A}$ denotes the underlying set of events composing the unordered event trace,
- $m : \mathcal{A}\rightarrow Mul_{\geq 0}$ is a function indicating the multiplicity of the occurrences of the event $e_{\mathcal{A}}\in\mathcal{A}$ in the event trace, denoted as $m(e_{\mathcal{A}}) \in Mul_{\geq 0} = \{0, 1, 2, 3, \cdots\}\cup\{*\}$. The repetition operator $(*)$ denotes that the number of an event $e_{\mathcal{A}}\in\mathcal{A}$ is not explicitly-defined, which can be any number $n\in Mul_{\geq 0}$.

We define a set of additional rules to further facilitate the specification of event pattern:

- In order to keep the expression succinct, we define a meta-character (.) to represent any event in the alphabet Σ.
- The concatenation operators between EvML expressions can be omitted for simplicity (i.e., $E_1 \cdot E_2$ can be denoted as E_1E_2),
- The operators $(^\wedge)$ for complementation, $(*)$ for repetition, $(\cdot)$ for concatenation, and $(+)$ for union in the EvML expression are defined in decreasing order of precedence,
- The delimiter "()" for parentheses can increase the precedence of the braced operators,
- The multi-set expression $\|M\|$ can be written in the form of $\|\{e_1, e_2, \ldots\}, \{m(e_1), m(e_2), \ldots\}\|$

2) The EvML Semantics: Some new notions and notations are needed before we can define the EvML semantics. For any given event trace ω, we assume that it is easy to calculate the underlying set of events composing the trace (denoted as $\mathcal{C}_\omega$) and the number of occurrences of a given event $e\in\omega$ (denoted as $n_\omega(e)$). The semantics of EvML is defined as shown in Figure 2.

$$\begin{aligned}
\mathcal{L}(\emptyset) &= \emptyset \\
\mathcal{L}(\epsilon) &= \{\epsilon\} \\
\mathcal{L}(e) &= \{e\} \\
\mathcal{L}(E_1+E_2) &= \mathcal{L}(E_1)\cup\mathcal{L}(E_2) \\
\mathcal{L}(E_1\cdot E_2) &= \{\omega_1\cdot\omega_2|\omega_1\in\mathcal{L}(E_1)\ and\ \omega_2\in\mathcal{L}(E_2)\} \\
\mathcal{L}(E^*) &= (\mathcal{L}(E))^* \\
\mathcal{L}(^\wedge E) &= \Sigma^*\backslash\mathcal{L}(E) \\
\mathcal{L}(\|M\|) &= \{\omega|\mathcal{C}_\omega=\mathcal{A}_M\ and\ \forall e\in\omega, n_\omega(e)=m_M(e)\}
\end{aligned}$$

Fig. 2. The semantics of EvML

Note that the operator $(+)$ is associative and commutative, and the operator $(\cdot)$ is associative.

$$\begin{aligned}
(E_1+E_2)+E_3 &\equiv E_1+(E_2+E_3) \\
E_1+E_2 &\equiv E_2+E_1 \\
(E_1\cdot E_2)\cdot E_3 &\equiv E_1\cdot(E_2\cdot E_3)
\end{aligned}$$

According to the EvML semantics, several simplification equations are defined, including the following:

$$\begin{aligned}
E+\emptyset &\equiv E \\
E+E &\equiv E \\
E_1\cdot E_3+E_2\cdot E_3 &\equiv (E_1+E_2)\cdot E_3 \\
\epsilon\cdot E &\equiv E
\end{aligned}$$

Figure 3 shows some examples to illustrate the EvML semantics.

Examples:
Assume that $\Sigma=\{\epsilon, e_1, e_2, e_3\}$.

$"e_1+e_2*"$ **denotes** $\{\epsilon, "e_1", "e_2", "e_2e_2", "e_2e_2e_2" \ldots\}$

$"(e_1+e_2)*"$ **denotes**
↪ $\{\epsilon, "e_1", "e_2", "e_1e_1", "e_1e_2", "e_2e_2", "e_2e_1", "e_1e_1e_1" \ldots\}$

$"\|\{e_1, e_2, e_3\}, \{1, 1, 1\}\|"$ **denotes**
↪ $\{"e_1e_2e_3", "e_1e_3e_2", "e_2e_1e_3", "e_2e_3e_1", "e_3e_1e_2", "e_3e_2e_1"\}$

$".e_1"$ **denotes** $\{"e_1", "e_1e_1", "e_2e_1", "e_3e_1"\}$

$"^\wedge e_1"$ **denotes** $\{"e_2", "e_3"\}$

Fig. 3. The examples of the Events Monitoring Logic

3) The EvML Operational Semantics: In this work, the event pattern matching algorithm is based on the event consumption idea as well, in the sense that the EvML expression E can consume an event e in the trace and produces another EvML expression denoted as $E\{e\}$, with the property that for any trace ω, $e\cdot\omega\in\mathcal{L}(E)$ if and only if $\omega\in\mathcal{L}(E\{e\})$. Roşu et al. [9] presented a set of rewriting rules and a rewriting-based algorithm to implement the event consumption idea for ERE. In our case, since EvML is a further extended version of ERE, we can easily adapt their work to the Events Monitoring Logic. We give the rewriting rules which define the EvML

$$(E_1 + E_2)\{e\} \rightarrow E_1\{e\} + E_2\{e\} \quad (1)$$
$$(E_1 \cdot E_2)\{e\} \rightarrow (E_1\{e\}) \cdot E_2 + \; if \; (\epsilon \in \mathcal{L}(E_1)) \; then \; E_2\{e\} \; else \; \emptyset \; fi \quad (2)$$
$$(E^*)\{e\} \rightarrow (E\{e\}) \cdot E^* \quad (3)$$
$$(^\wedge E)\{e\} \rightarrow {}^\wedge(E\{e\}) \quad (4)$$
$$e_1\{e\} \rightarrow if \; (e_1 = e) \; then \; \epsilon \; else \; \emptyset \; fi \quad (5)$$
$$\epsilon\{e\} \rightarrow \emptyset \quad (6)$$
$$\emptyset\{e\} \rightarrow \emptyset \quad (7)$$
$$\|M\|\{e\} \rightarrow if \; (e \in \mathcal{A}_M \text{ and } m_M(e) \neq 0) \; then \; m_M(e) = m'_M(e) \; else \; \emptyset \; fi \quad (8)$$

Fig. 4. The rewriting rules for EvML

operational semantics recursively, using the structure of the EvML expression, as shown in Figure 4. In particular, the Rules 1 to 7 are defined for the inherited ERE operators. The Rule 8 defines that when the available event is found in the specified event multi-set, if the occurrence number of the event is explicitly-defined, the number is decreased by one; otherwise, the number remains not explicitly-defined, where:

$$m'_M(e) = \begin{cases} n-1 & , m_M(e) = n \text{ and } n > 0 \\ * & , m_M(e) = * \end{cases}$$

and once all of the explicitly-defined occurrence numbers decrease to zero, we have the rewriting rule:

$$\|M\| \rightarrow if \; (\forall e \in \mathcal{A}_M, m_M(e) = 0 \text{ or } *) \; then \; \epsilon \; else \; \|M\| \; fi \quad (9)$$

The structure "*if then else*" taking a boolean term and two EvML expressions is defined by two rewriting rules:

$$if \; (true) \; then \; E_1 \; else \; E_2 \; fi \rightarrow E_1 \quad (10)$$
$$if \; (false) \; then \; E_1 \; else \; E_2 \; fi \rightarrow E_2 \quad (11)$$

For the evaluation of the boolean expression $\epsilon \in \mathcal{L}(E)$, we define the following rules:

$$\epsilon \in (\mathcal{L}(E_1) + \mathcal{L}(E_2)) \rightarrow \epsilon \in \mathcal{L}(E_1) \vee \epsilon \in \mathcal{L}(E_2) \quad (12)$$
$$\epsilon \in (\mathcal{L}(E_1) \cdot \mathcal{L}(E_2)) \rightarrow \epsilon \in \mathcal{L}(E_1) \wedge \epsilon \in \mathcal{L}(E_2) \quad (13)$$
$$\epsilon \in (\mathcal{L}(E^*)) \rightarrow true \quad (14)$$
$$\epsilon \in (\mathcal{L}(^\wedge E)) \rightarrow not \; (\epsilon \in \mathcal{L}(E)) \quad (15)$$
$$\epsilon \in \mathcal{L}(e) \rightarrow false \quad (16)$$
$$\epsilon \in \mathcal{L}(\epsilon) \rightarrow true \quad (17)$$
$$\epsilon \in \mathcal{L}(\emptyset) \rightarrow false \quad (18)$$
$$\epsilon \in \mathcal{L}(\|M\|) \rightarrow false \quad (19)$$

Since we also use the meta-character (.) to specify EvML expressions for simplicity, we have the rewriting rule for it:

$$.\{e\} \rightarrow \epsilon \quad (20)$$

Additionally, the Rule 20 for the meta-character (.) is a special case of the Rule 5, where $e_1 \equiv e$.

These rewriting rules are natural and intuitive. We omit the proof of the *terminating* and *Church-Rosser* property of the rewriting system, and leave it as our future work.

4) The Event Pattern Matching Algorithm: After introducing the operational semantics of EvML, we present the Algorithm 1 that describes the event pattern matching algorithm. The algorithm will be used in the observer execution process (as stated in Section III-C) to determine that the ***first m-events*** trace ($\omega_m = e_1 e_2 \ldots e_m$ where $m \leq n$) of an input trace ($\omega = e_1 e_2 \ldots e_n$) matches the event pattern:

Algorithm 1 $EventPatternMatching(E, \omega)$

```
INPUT:  An EvML expression E and an event trace
        ω = e1 e2 ... en .
OUTPUT: match when ωm = e1 e2 ... em (m≤n) matches E ;
        nomatch when ωm does not match E

let E' be E;          % start the matching process
let m be 1;
while m≤ n do
  wait until em is available;
  let E' be E'{em};              % consume one event
  if ϵ ∈ L(E') then
    return match;      % the first m-events trace matches E
  if E' = ∅ then       % the current event does not match E
    return nomatch;      % the input trace does not match E
  let m be m+1;        % consume the next event
return nomatch;        % the input trace does not match E
```

C. The Observer Execution Process

In this section, we introduce the observer execution process operated to enable an observer working with a given event trace. Briefly speaking, with the execution of the TASM model at runtime, different TASM constructs will generate massive events which can be abstracted as a linear sequence of events. The observer can spawn one or more child observers, together to determine the satisfaction of the upcoming events with the event pattern and to evaluate corresponding observations.

In particular, the events in the trace are consumed one by one. When an event e is available, the $EventsFilter$ is applied to filter out irrelevant events to the desired property. If the available event is relevant, the expressions of the observer E_o and its child observers E_c (if any exists) will be transformed to the corresponding new expressions E'_o and E'_c by applying the rewriting rules. Regarding the parent observer, if the new expression can match ϵ (i.e., $\epsilon \in \mathcal{L}(E'_o)$), which means the $Listener$'s condition is satisfied, then the action will be

executed and the observation will be concluded. If the new expression is the empty set (i.e., $E'_o = \emptyset$), which means the current event can not satisfy the $Listener$'s condition, then the event is dropped and the observer waits for the next available event. If the new expression neither is the empty set, nor matches ϵ, which means the current event is probably the first event of one of the event traces that can satisfy the $Listener$'s $condition$, then the observer will spawn a child observer that inherits the new expression (i.e., $E_c = E'_o$) and the child observer starts to wait for the next available event, as depicted in Figure 5.

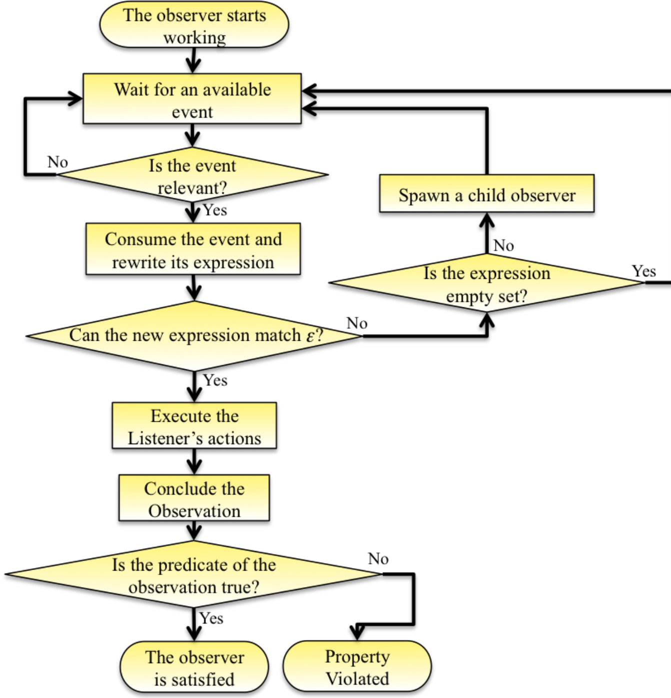

Fig. 5. The observer execution process

Regarding the child observers, they will take over monitoring whether the subsequent events match the event expression E_c by applying the event pattern matching algorithm, as illustrated in Figure 6. When a new relevant event is available, the child observer event expression will be rewritten into E'_c:

- If the *condition* is satisfied by the subsequent trace (i.e., $\epsilon \in \mathcal{L}(E'_c)$), then the *action* will be immediately executed to update corresponding variables. The observation predicate will be concluded based on the updated observer variables. In this situation, if the value of the predicate is evaluated to be false, which means the represented property is deemed to be violated, its parent observer and all the other child observers will stop monitoring. On the contrary, if the predicate is evaluated to be true, the child observer is deemed to be satisfied. Then, its parent observer and all the other child observers will continue monitoring.
- If the subsequent events violate the event pattern (i.e., $E'_c = \emptyset$), one of the two possible consequences will take place, which depends on the *keyword* of the $Listener$. If the *keyword* is specified as $optional$, then the child observer will be destroyed and its parent observer continues monitoring. If the *keyword* is $compulsory$, then the property represented by its parent observer will directly evaluate to be violated and the parent observer and all the other child observers will stop monitoring.

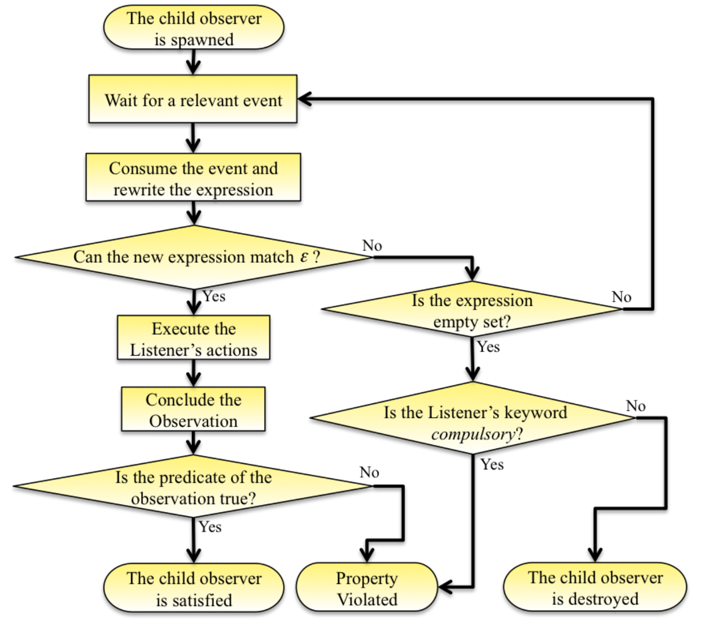

Fig. 6. The child observer execution process

Note that a running TASM model can be observed by several observers at the same time. Meanwhile, an observer can have many active instances (i.e., child observers) simultaneously before the end of monitoring an event trace.

IV. Illustration Application

In this section, we describe a simplified Vehicle Locking-Unlocking (VLU) system. This is used to illustrate how to specify the observer according to the requirement for validation purpose.

A. Vehicle Locking-Unlocking

The proposed VLU system aims at replacing the mechanical key, as a control access to a vehicle, and it follows a common pattern in feature-oriented requirements specification [10]: The basic functionality is encapsulated as an individual feature, and additional/optional enhancements are specified as features that provide increments in functionality. Specifically, such features are *Central Locking* (CL), *Auto-lockout* (AUL) and *Anti-lockout* (ANL), where:

- **Central Locking (a basic feature)** locks and unlocks all the doors of the vehicle upon receipt of a command from the user key fob.
- **Auto-lockout (an optional feature)** locks all the doors of the vehicle when a timeout expires. It provides theft

protection in case that the driver forgets to manually lock the doors.
- **Anti-lockout (an optional feature)** enables unlocking of the doors while a key is in ignition. The purpose of this feature is to prevent the driver from being locked out of the vehicle.

Figure 7 shows the features of the VLU system in the form of technical feature model tree presented in the EAST-ADL language [11].

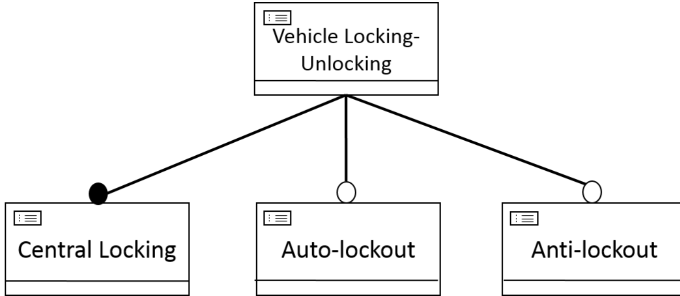

Fig. 7. The technical feature model tree of the VLU system.

B. Observer Specification

Assume that we are interested in monitoring the satisfaction of the AUL feature requirement which states that "The system shall lock all the doors of the vehicle when the vehicle is still and a timeout ($timer = 20$ in this case) expires." This feature can show how to use the TASM language and the extended observer-based technique to specify functional behaviors as well as non-functional properties in terms of timing property in this case. In addition, assume that we have two TASM machines in terms of AUL (as shown in Figure 8) and $DOOR$ (as shown in Figure 9), modeling the behaviors of the AUL feature and the doors, respectively. Recall that each event is time-stamped during the model execution, and the time stamp of the event can be obtained by referencing the time property t of the event.

The observer is specified as shown in Figure 10. The $EventsFilter$ will filter out the events of the ReUUE, ChVE, and RuDE types. Since the auto-locking process can be interrupted by either moving the vehicle or manually locking the doors, the keyword of the $Listener$ is $optional$. The event pattern of the $Listener$ consists of three parts, as shown in Line 9, 10, and 11 respectively. The first part "$||\{AUL{\rightarrow}Timer{\rightarrow}RuEE$, $\hat{}AUL{\rightarrow}TimerReset{\rightarrow}RuEE\}$,{20, $*$}||" models the behavior that the timer starts and then expires, where the event $AUL{\rightarrow}Timer{\rightarrow}RuEE$ is supposed to be triggered 20 times and there could be some other events but $TimerReset{\rightarrow}RuEE$ events that will be triggered during the expiring process. The second part ".*" represents an arbitrary number of arbitrary events that could be triggered by other TASM machines after the timer expires but before the doors are locked. The last part "$DOOR{\rightarrow}Lock{\rightarrow}RuEE$" models the behavior that the doors

```
R1:Timeout{   % the name of the rule
  if aul_state = idle and door_state = close and
     vehicle_state = still and timer = 20 then
    aul_state := timeout;
    timer   := 0;
}
R2:Autolock{
  if aul_state = timeout then
    door_action := lock;
    aul_state := idle;
}
R3:Timer{
  t := 1;     % the time duration of the rule
  if aul_state = idle and door_state = close and
     vehicle_state = still and timer < 20  then
    timer := timer + 1;
}
R4:TimerReset{
  t := next;
  else  then
    timer := 0;
}
```

Fig. 8. The TASM machine models the behavior of the Auto-lockout feature.

```
R1:Close{
  t:=closing_time;   % the time duration of the rule
  if door_action = close then
    door_state := closed;
}
R2:Lock{
  t:=locking_time;   % the time duration of the rule
  if door_action = lock then
    door_state := locked;
}
R3:UnLock{
  t:=unlocking_time; % the time duration of the rule
  if door_action = unlock then
    door_state := closed;
}
R4:Open{
  t := opening_time; % the time duration of the rule
  if door_action = open then
    door_state := opened;
}
```

Fig. 9. The TASM machine models the behavior of the doors.

are locked. If the event pattern is matched, the observation "obt2$-$obt1==20+$DOOR{\rightarrow}Lock{\rightarrow}locking_time$" will be evaluated accordingly, which indicates whether the doors are locked properly when the timer expires.

If the observer is violated, it means there must exist some inconsistencies in the requirements. Those inconsistencies cause the doors not being locked properly when the timer expires.

V. Related Work

A. The Monitoring Logic

Giannakopoulou et al. [12] present an approach to checking a running program against Linear Temporal Logic (LTL) specifications. In particular, the LTL formulae representing the properties of interests are translated into finite-state automata, which are used as observers monitoring the program behaviors.

Roşu et al. [9] present lower bounds and rewriting algorithms for testing membership of a word in a regular language described by an extended regular expression. The algorithms are based on an event consumption idea: a just arrived event is

```
1 ObserverVariables:{
2  Time obt1 := 0; Time obt2:=0;
3 }
4 EventsFilter:{
5  irrelevant event types: ReUUE, ChVE, RuDE;
6 }
7 Listener:{
8  listening optional:
9  ||{AUL→Timer→RuEE,^AUL→TimerReset→RuEE},{20,*}||
10  .*
11  DOOR→Lock→RuEE then
12
13  obt1:=AUL→Timer→RuEE(1)→t; %the stamped time of the
14                              first event of the
15                             AUL→Timer→RuEE type
16
17  obt2:=DOOR→Lock→RuEE→t;
18 }
19 Observation:{
20  obt2-obt1 == 20+DOOR→Lock→locking_time;
21 }
```

Fig. 10. The Observer for the AUL feature requirement

consumed by the regular expression, i.e., the extended regular expression modifies itself into another expression dropping the event.

Barringer et al. [13] present a compact and powerful logic, namely Eagle, which is based on recursive parameterized rule definitions over the standard propositional logic operators together with three primitive temporal operators in the sense of a past-state operator, a next-state operator, and a concatenation-state operator.

Basin et al. [14] extend the metric first-order temporal logic (MFOTL) with aggregation operators in order to specify observers that represent the compliance policies on aggregated data. Compliance policies represent normative regulations, which specify permissive and obligatory actions for system users. The authors provide a monitoring algorithm for the enriched observer specification language as well.

Comparing the aforementioned observer-based techniques, EvML has a more succinct way to express unordered fixed-count events sequence. Moreover, the observers defined in the aforementioned techniques merely use logical expressions to specify the property of interests, but we use the combination of the logical expression (i.e., $Listener's condition$) and other constructs (i.e., $Listener's action$, $ObserverEnvironment$ and $Observation$). By using the combination, more expressiveness power is possible to achieve, which will be discussed in detail as our future work.

B. Other Related Work

Bauer et al. [15] discuss a three-value semantics (false, true, inconclusive) for LTL and TLTL observers on finite traces, where an observer outputs *false* when a finite prefix is impossible to be the prefix of any accepting trace and, *true* when a finite prefix can be accepted by any infinite extension of the trace and, *inconclusive* in other cases. Additionally, Falcone et al. [16] give an related and interesting discussion about the monitorability of properties in the safety-progress classification. Leucker et al. [17] present a brief account of the field of runtime monitoring. They give a definition of runtime monitoring and make a comparison to well-known verification techniques in terms of model checking and testing.

VI. Conclusion

In this paper, we have enhanced our observer-based requirements validation technique presented in [4] via a proposed new observer specification logic (namely EvML), as well as a newly introduced rewriting-based monitoring algorithm for EvML. EvML originates from the extended regular expressions, and can help to specify the situation in which the occurrence number of the events of interests is predefined and the occurrence order is trivial. The rewriting-based monitoring algorithm implements the incremental event consumption idea which enables runtime monitoring. Our illustration application using a Vehicle Locking-Unlocking system has shown that EvML is capable to specify observers for validation purpose. As a part of our future work, we are interested in having more extensive industrial cooperations for validating our observer-based technique, as well as improving the current implementation of our TASM Toolset.

References

[1] A. Ellis, "Achieving safety in complex control systems," in *Proceedings of SCSC'95*. Springer London, 1995, pp. 1–14.
[2] N. G. Leveson, *Safeware: System Safety and Computers*. NY, USA: ACM, 1995.
[3] C. Baier and J.-P. Katoen, *Principles of Model Checking*. The MIT Press, 2008.
[4] J. Zhou, Y. Lu, and K. Lundqvist, "A tasm-based requirements validation approach for safety-critical embedded systems," in *Proceedings of Ada-Europe'14*, June 2014.
[5] M. Ouimet, "A formal framework for specification-based embedded real-time system engineering," Ph.D. dissertation, Department of Aeronautics and Astronautics, MIT, 2008.
[6] J. E. F. Friedl, *Mastering Regular Expressions*, 2nd ed., A. Oram, Ed. O'Reilly & Associates, Inc., 2002.
[7] E. Börger and R. F. Stärk, *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.
[8] E. M. Clarke and J. M. Wing, "Formal methods: state of the art and future directions," *ACM Comput. Surv.*, vol. 28, no. 4, pp. 626–643, Dec. 1996.
[9] G. Roşu and M. Viswanathan, "Testing extended regular language membership incrementally by rewriting," in *Proceedings of RTA03*. Springer-Verlag, 2003, pp. 499–514.
[10] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (foda) feasibility study," CMU/SEI-90-TR-21, ESD-90-TR-222, Tech. Rep., November 1990.
[11] H. Blom, H. Lönn, F. Hagl, Y. Papadopoulos, M.-O. Reiser, C.-J. Sjöstedt, D.-J. Chen, and R. T. Kolagari, "EAST-ADL - An Architecture Description Language for Automotive Software-Intensive Systems," The EAST-ADL 2 Consortium, Tech. Rep., 2012.
[12] D. Giannakopoulou and K. Havelund, "Automata-based verification of temporal properties on running programs," in *Proceedings of ASE'01*, Nov 2001, pp. 412–416.
[13] H. Barringer, A. Goldberg, K. Havelund, and K. Sen, "Rule-based runtime verification," in *Proceedings of VMCAI'04*, 2004, pp. 44–57.
[14] D. Basin, F. Klaedtke, S. Marinovic, and E. Zălinescu, "Monitoring of temporal first-order properties with aggregations," in *Proceedings of RV'13*, 2013, pp. 40–58.
[15] A. Bauer, M. Leucker, and C. Schallhart, "Monitoring of real-time properties," in *Proceedings of FSTTCS'06*. Springer, 2006, pp. 260–272.
[16] Y. Falcone, J.-C. Fernandez, and L. Mounier, "Runtime verification of safety-progress properties," in *Runtime Verification*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 40–59.
[17] M. Leucker and C. Schallhart, "A brief account of runtime verification," *The Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 293–303, 2009.