electronics

MDPI

Article

A Formally Reliable Cognitive Middleware for the Security of Industrial Control Systems

Muhammad Taimoor Khan [1,*], Dimitrios Serpanos [2] and Howard Shrobe [3]

[1] Institute of Informatics, Alpen-Adria University, A-9020 Klagenfurt, Austria
[2] Industrial Systems Institute/RC-Athena & ECE, University of Patras, GR 26504 Patras, Greece; serpanos@ece.upatras.gr
[3] MIT CSAIL, Cambridge, MA 02139, USA; hes@csail.mit.edu
* Correspondence: muhammad.khan@aau.at; Tel.: +43-463-2700-3529

Received: 31 May 2017; Accepted: 8 August 2017; Published: 11 August 2017

Abstract: In this paper, we present our results on the formal reliability analysis of the behavioral correctness of our cognitive middleware ARMET. The formally assured behavioral correctness of a software system is a fundamental prerequisite for the system's security. Therefore, the goal of this study is to, first, formalize the behavioral semantics of the middleware and, second, to prove its behavioral correctness. In this study, we focus only on the core and critical component of the middleware: the execution monitor. The execution monitor identifies inconsistencies between runtime observations of an industrial control system (ICS) application and predictions of the specification of the application. As a starting point, we have defined the formal (denotational) semantics of the observations (produced by the application at run-time), and predictions (produced by the executable specification of the application). Then, based on the formal semantices, we have formalized the behavior of the execution monitor. Finally, based on the semantics, we have proved soundness (absence of false alarms) and completeness (detection of arbitrary attacks) to assure the behavioral correctness of the monitor.

Keywords: run-time monitoring; security monitor; absence of false alarms; ICS; CPS

1. Introduction

Defending industrial control systems (ICS) against cyber-attack requires us to be able to rapidly and accurately detect that an attack has occurred in order to, on one hand, assure the continuous operation of ICS and, on the other, to meet ICS real-time requirements. Today's detection systems are woefully inadequate, suffering from both high false positive and false negative rates. There are two key reasons for this. First, the systems do not understand the complete behavior of the system they are protecting. The second is that they do not understand what an attacker is trying to achieve. Most systems that exhibit this behavior, in fact, are retrospective, that is they understand some surface signatures of previous attacks and attempt to recognize the same signature in current traffic. Furthermore, they are passive in character, they sit back and wait for something similar to what has already happened to reoccur. Attackers, of course, respond by varying their attacks, so as to avoid detection.

ARMET [1] is a representative of a new class of protection systems that employ a different, active form of perception, one that is informed both by knowledge of what the protected application is trying to do and by knowledge of how attackers think. It employs both bottom-up reasoning (going from sensors data to conclusions about what attacks might be in progress) and top-down reasoning (given a set of hypotheses about what attacks might be in progress, it focuses its attention to those events most likely to significantly help in discerning the ground truth).

Based on AWDRAT [2], ARMET is a general purpose middleware system that provides survivability to any kind of new and legacy software system and to ICS in particular. As shown in Figure 1, the run-time monitor (RSM) of ARMET checks consistency between the run-time behavior of the application implementation (AppImpl) and the specified behavior (AppSpec) of the system. If there is an attack, then the diagnostic engine identifies an attack (an illegal behavioral pattern) and the corresponding set of resources that were compromised during the attack. After identifying an attack, a larger system (e.g., AWDRAT) attempts to repair and then regenerate the compromised system into a safer state, to only allow fail-safe, if possible. The task of regeneration is based on the dependency-directed reasoning [3] engine of the system that contributes to self-organization and self-awareness. It does so by recording execution steps intrinsically, the states of the system and their corresponding justification (reason). The details on diagnosis and recovery are beyond the scope of this paper. Based on the execution monitor and the reasoning engine of ARMET, not only is the detection of known attacks possible but also the detection of unknown attacks and potential bugs in the application implementation is possible.

The RSM has been developed as prototype implementations in a general-purpose, computing environment (laptop) using a general-purpose functional programming environment (Lisp). In addition to the software's ability to be easily ported to a wide range of systems, the software can be directly developed in any embedded OS and RTOS middleware environment, such as RTLinux, Windows CE, LynxOS, VxWorks, etc. The current trend toward sophisticated PLC and SCADA systems with advanced OS and middleware capabilities provides an appropriate environment for developing highly advanced software systems for control and management [4].

The rest of this paper is organized as follows: Section 2 presents the syntax and semantics of the specification language of ARMET, while Section 3 explains the syntax and semantics of the monitor. The proof of behavioral correctness (i.e., soundness and completeness) of the monitor is discussed in Section 4. Finally, we conclude in Section 5.

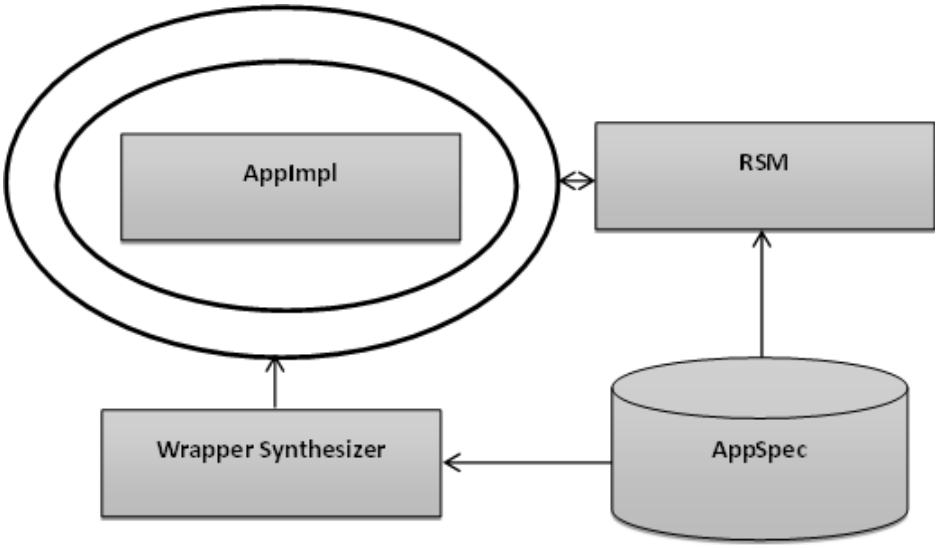

Figure 1. ARMET—Run-time security monitor.

2. A Specification Language of ARMET

A specification language of ARMET allows the description of the behavior of ICS application implementation (AppImpl) based on a fairly high-level description written in a language of "Plan Calculus" [3], which is a decomposition of pre- and post-conditions and invariant for each computing component (module) of the system. The description can be considered as an executable specification of the system. The specification is a hierarchical nesting of a system's components such that the input and output ports of each component are connected by data and control flow links of

respective specifications. Furthermore, each component is specified with corresponding pre- and post-conditions. However, the specification also includes a variety of event specifications.

In detail, the behavioral specification ("AppSpec"—as shown in Figure 2) of an application implementation ("AppImpl"—as shown in Figure 3) is described at the following two logical levels:

1. The *control level* describes the control structure of each of the component (e.g., sub-components, control flow and data flow links), which is:
   - Defined by the syntactic domain "StrModSeq", while the control flow can further be elaborated with syntactic domain "SplModSeq"
2. The *behavior level* describes the actual method's behavioral specification of each of component, which is defined by the syntactic domain "BehModSeq".

Furthermore, the registration of the *observations* is given by the syntactic domain "RegModSeq", at the top of the above domains. All four of the aforementioned domains are the top-level syntactic domains of the specification. Our specification is hierarchical, i.e., it specifies the components of the implementations as hierarchical modules. In the following, we discuss the syntax of *control* and *behavioral* elements of the specification using the specification of a temperature control as an example, shown in Figure 2.

1. The description of each component type consists of
   (a) Its interface, which is comprised of:
      - a list of inputs
      - a list of its outputs
      - a list of the resources it uses (e.g., files it reads, the code in memory that represents this component)
      - a list of sub-components required for the execution of the subject component
      - a list of events that represent entry into the component
      - a list of events that represent exit from the component
      - a list of events that are allowed to occur during any execution of this component
      - a set of conditional probabilities between the possible modes of the resources and the possible modes of the whole component
      - a list of known vulnerabilities occurred to the component
   (b) and a structural model that is a list of sub-components, some of which might be splits or joins of:
      - data-flows between linking ports of the sub-components (outputs of one to inputs of another)
      - control-flow links between cases of a branch and a component that will be enabled if that branch is taken.

   The description of the component type is represented by the syntactical domain "StrMod", which is defined as follows:

StrMod ::= **define-ensemble** CompName
:entry-events :auto | (EvntSeq)
:exit-events (EvntSeq)
:allowable-events (EvntSeq)
:inputs (ObjNameSeq)
:outputs (ObjNameSeq)
:components (CompSeq)
:controlflows (CtrlFlowSeq)
:splits (SpltCFSeq)
:joins (JoinCFSeq)
:dataflows (DataFlowSeq)
:resources (ResSeq)

:resource-mapping (ResMapSeq)
:model-mappings (ModMapSeq)
:vulnerabilities (VulnrabltySeq)

Example 1. *For instance, a room temperature controller:* `control`*, periodically receives the current temperature value (*`sens-temp`*) from a sensor. The controller may also receive a user command (*`set-temp`*) to set (either increase or decrease) the current temperature of the room. Based on the received user command, the controller either raises the temperature through the sub-component* `temp-up` *or reduces the temperature through the sub-component* `temp-down`*, after computing the error* `compute-epsilon` *of the given command as shown in Figure 2. Furthermore, the controller issues a command as an output (*`com`*), which contains an updated temperature value. Figure 3 reflects the corresponding implementation parts of the controller.*

2. The behavioral specification of a component (a component type may have one normal behavioral specification and many abnormal behavioral specifications, each one representing some failure mode) consists of:

- inputs and outputs
- preconditions on the inputs (logical expressions involving one or more of the inputs)
- postconditions (logical expressions involving one or more of the outputs and the inputs)
- allowable events during the execution in this mode

The behavioral specification of a component is represented by a corresponding syntactical domain "BehMod", as follows:

BehMod ::= **defbehavior-model** (CompName **normal** | **compromised**)
:inputs (ObjNameSeq_1)
:outputs (ObjNameSeq_2)
:allowable-events (EvntSeq)
:prerequisites (BehCondSeq_1)
:post-conditions (BehCondSeq_2)

Example 2. *For instance, in our temperature control example, the* `normal` *and* `compromised` *behavior of a controller component* `temp-up` *is modeled in Figure 2. The normal behavior of a temperature raising component* `temp-up` *describes the: (i) input condition* `prerequisites`*, i.e., the component receives a valid input* `new-temp`*; and (ii) output conditions* `post-conditions`*, i.e., the new computed temperature* `new-temp` *is equal to the sum of the current temperature* `old-temp` *and the computed* `delta` *(error). The computed temperature respects the temperature range (1-40). Similarly, the* `compromised` *behavior of the component illustrates the corresponding input and output conditions. Figure 3 reflects the corresponding implementation parts of the* `temp-up` *component.*

The complete syntactic details of the specification language are discussed in [5].

Based on the core idea of Lamport [6], we have defined the semantics of the specification as a state relationship to achieve the desired insight of the program's behavior. This does so by relating pre- and post-states [7]. For simplicity, we chose to discuss semantics of the behavioral domain "BehMod". The denotational semantics of the specification language is based on denotational algebras [8]. We define the result of semantic valuation function as a predicate. The behavioral relation (BehRel) is defined as a predicate over an environment, a pre-state, and a post-state. The corresponding relation is defined as:

$$\text{BehRel} := \mathbb{P}(\text{Environment} \times \text{State} \times \text{State}_{\perp})$$

The valuation function for the abstract syntax domain "BehMod" values is defined as:

$$[\![\text{BehMod}]\!]: \text{Environment} \rightarrow \text{BehRel}$$

```
(define-component-type control
 :entry-events (control)
 :exit-events (control)
  :allowable-events (update-temp compute-epsilon)
  :inputs (set-temp sens-temp)
  :outputs (com)

   :components
 ((temp-up :type temp-up :models (normal))
 (temp-down :type temp-down :models (normal)) ... )

   :dataflows
 ((set-temp control set-temp temp-up)
 (epsilon temp-up epsilon temp-down)...))


  (define-component-type compute-delta
    :Primitive t
    :entry-events (compute-delta)
    :exit-events (clip-delta)
    :inputs (proportional integral derivative)
    :outputs (delta)
    :behavior-modes (normal)
    )

  (defbehavior-model (compute-delta normal)
    :inputs (proportional integral derivative)
    :outputs (delta)
    :prerequisites ([data-type-of ?proportional number]
     [data-type-of ?integral number]
     [data-type-of ?derivative number])
    :post-conditions ([data-type-of ?delta number])
    )

 (define-component-type temp-up
  :entry-events (temp-up)
  :exit-events (temp-up)
  :inputs (delta old-temp time-step)
  :outputs (new-temp)
  :behavior-modes (normal compromised) )

 (defbehavior-model (temp-up normal)
  :inputs (delta old-temp time-step)
  :outputs (new-temp)
  :prerequisites ([data-type-of ?delta number])
  :post-conditions ([and [and [data-type-of ?delta number]
    [equal new-temp (+ ?old-temp ?delta)]]
    [and [(<= new-temp 40)] [(>= new-temp 1)]]]))

  (defbehavior-model (temp-up compromised)
  :inputs (delta old-temp time-step)
  :outputs (new-temp)
  :prerequisites ()
  :post-conditions (not [and [and [data-type-of ?delta number]
    [equal new-temp (+ ?old-temp ?delta)]]
    [and [(<= new-temp 40)] [(>= new-temp 1)]]]))


 (define-component-type temp-down
  :entry-events (temp-down)
  :exit-events (temp-down)
  :inputs (delta old-temp time-step)
  :outputs (new-temp)
  :behavior-modes (normal compromised) )

 (defbehavior-model (temp-down normal)
  :inputs (delta old-temp time-step)
  :outputs (new-temp)
  :prerequisites ([data-type-of ?delta number])
   :post-conditions ([and [and [data-type-of ?delta number]
    [equal new-temp (- ?old-temp ?delta)]]
    [and [(<= new-temp 40)] [(>= new-temp 1)]]]))

  (defbehavior-model (temp-down compromised)
  :inputs (delta old-temp time-step)
  :outputs (new-temp)
  :prerequisites ()
  :post-conditions (not [and [and [data-type-of ?delta number]
    [equal new-temp (- ?old-temp ?delta)]]
    [and [(<= new-temp 40)] [(>= new-temp 1)]]]))
...
```

Figure 2. An example specification (AppSpec) of a temperature control.

Semantically, normal and compromised behavioral models result in modifying the corresponding elements of the environment value "Component" as defined below:

⟦BehMod⟧(e)(e', s, s') ⇔
∀ e_1 ∈ Environment, nseq ∈ EvntNameSeq, eseq ∈ ObsEvent*, inseq, outseq ∈ Value*:
⟦$ObjNameSeq_1$⟧(e)(inState$_\perp$(s), inseq) ∧ ⟦$BehCondSeq_1$⟧(e) (inState$_\perp$(s)) ∧
⟦EvntSeq⟧(e) (e_1, s, s', nseq, eseq)
⟦$ObjNameSeq_2$⟧(e_1)(s', outseq) ∧ ⟦$BehCondSeq_2$⟧(e_1) (s') ∧
∃ c ∈ Component: ⟦CompName⟧(e_1)(inValue(c)) ∧
IF eqMode(inState$_\perp$(s'), "normal") THEN
LET sbeh = c[1], nbeh = <inseq, outseq, s, s'>, cbeh = c[3] IN
e' = push(e_1, store(inState(s'))(⟦CompName⟧(e_1)), c(sbeh, nbeh, cbeh, s, s'))
END
ELSE
LET sbeh = c[1], nbeh = c[2], cbeh = <inseq, outseq, s, s'> IN
e' = push(e_1, store(inState(s'))(⟦CompName⟧(e_1)), c(sbeh, nbeh, cbeh, s, s'))
END
END

In detail, if the semantics of the syntactic domain "BehMod" holds in a given environment e, resulting in environment e' and transforming a pre-state s into a corresponding post-state s', then:

- The inputs "$ObjNameSeq_1$" evaluate to a sequence of values *inseq* in a given environment e and a given state s, which satisfies the corresponding pre-conditions "$BehCondSeq_1$" in the same e and s.
- The allowable events happen and their evaluation results in new environment e_1 and a given post-state s' with some auxiliary sequences *nseq* and *eseq*.
- The outputs "$ObjNameSeq_2$" evaluates to a sequence of values *outseq* in an environment e_1 and given post-state s', which satisfies the corresponding post-conditions "$BehCondSeq_2$" in the same environment e_1. State s' and the given environment e' may be constructed such that:
  - If the post-state is "normal" then e' is an update to the normal behavior "nbeh" of the component "CompName" in environment e_1, otherwise
  - e' is an update to the compromised behavior "cbeh" of the component.

In the construction of the environment e', the rest of the semantics of the component do not change as represented in the corresponding LET-IN constructs.

The complete definitions of the auxiliary functions, predicates, and semantics are presented in [5].

3. An Execution Monitor of ARMET

In principle, an execution monitor interprets the event stream (traces of the execution of the target system, i.e., *observations*) against the system specification (the execution of the specification is also called *predictions*) by detecting inconsistencies between *observations* and the *predictions*, if there are any.

When the system implementation "AppImpl" (as shown in Figure 3) starts execution, an initial "startup" event is generated and dispatched to the top level component (module) of the system that transforms the execution state of the component into "running" mode. If there is a subnetwork of components, the component instantiates it and propagates the data along its data links by enabling the corresponding control links, if involved. When the data arrives on the input port of the component, the execution monitor checks if it is complete. If so, the execution monitor checks the preconditions of the component for the data and, if they succeed, it transforms the state of the component into "ready" mode. In the case that any of the preconditions fail, it enables the diagnosis engine.

After the startup of the implementation, described above, the execution monitor starts monitoring the arrival of every *observation* (runtime event) as follows:

1. If the event is a "method entry", the execution monitor checks if this is one of the "entry events" of the corresponding component in the "ready" state. If so, after receiving the data and when the respective preconditions are checked, if they succeed, the data is applied on the input port of the component and the mode of the execution state is changed to "running".
2. If the event is a "method exit", the execution monitor checks if this one of the "exit events" of the component is in the "running" state. If so, it changes its state into "completed" mode, collects the data from the output port of the component, and checks for corresponding postconditions. Should the checks fail, the execution monitor enables the diagnosis engine.
3. If the event is one of the "allowable events" of the component, it continues execution.
4. If the event is an unexpected event (i.e., it is neither an "entry event", an "exit event", nor in the "allowable events"), the execution monitor starts its diagnosis.

```
...

public class TemperatureControl{

    int old-temp = 0;
    int new-temp;

    public TemperatureControl(...){
        ...
    }

    public void set-temp(int temp){
        ...
        float delta = compute-delta(...);
        if(temp < old-temp)
            new-temp = temp-down(delta);
        else if(temp > old-temp)
            new-temp = temp-up(delta);
    }

    public double compute-delta(double proportional, double integral, double derivative){
        return proportional + integral + derivative;
    }

    public int temp-up(double delta){
        return old-temp + Math.round(delta);
    }

    public int temp-down(double delta){
        return old-temp - Math.round(delta);
    }

    ....

}
```

Figure 3. An example implementation (AppImpl) of a temperature control.

Based on the above behavioral description of the execution monitor, we have formalized the corresponding semantics of the execution monitor as follows:

$$
\begin{aligned}
&\forall\, app \in AppImpl, sam \in AppSpec, c \in Component,\\
&\quad s, s' \in State, t, t' \in State_s, d, d' \in Environment_s, e, e' \in Environment, rte \in RTEvent:\\
&\quad [\![sam]\!](d)(d', t, t') \wedge [\![app]\!](e)(e', s, s') \wedge startup(s, app) \wedge isTop(c, [\![app]\!](e)(e', s, s')) \wedge\\
&\quad setMode(s, \text{“running”}) \wedge arrives(rte, s) \wedge equals(t, s) \wedge equals(d, e)\\
&\quad \Rightarrow\\
&\qquad \forall\, p, p' \in Environment^*, m, n \in State^*_\perp : equals(m(0), s) \wedge equals(p(0), e)\\
&\qquad\quad \Rightarrow\\
&\qquad\qquad \exists\, k \in \mathbb{N}, p, p' \in Environment^*, m, n \in State^*_\perp :\\
&\qquad\qquad\quad \forall\, i \in \mathbb{N}_k : monitors(i, rte, c, p, p', m, n) \wedge\\
&\qquad\qquad\qquad ((eqMode(n(k), \text{“completed”}) \wedge eqFlag(n(k), \text{“normal”}) \wedge equals(s', n(k)))\\
&\qquad\qquad\qquad \vee\\
&\qquad\qquad\qquad eqFlag(n(k), \text{“compromised”}))
\end{aligned}
$$

$\Rightarrow$

enableDiagnosis(p'(k))(n(k), inBool(`true`)) $\wedge$ equals(s', n(k)))

The semantics of recursive monitoring is determined by two sequences of states: pre and post, constructed from the pre-state of the monitor. Any *ith* iteration of the monitor transforms the $pre(i)$ state into the $post(i+1)$ state, from which the $pre(i+1)$ state is constructed. No event can be accepted in an *Error* state and the corresponding monitoring terminates when either the application has terminated with "normal" mode or when some misbehavior is detected, as indicated by the respective "compromised" state. This recursive idea of monitoring is formalized as a "monitors" predicate, as follows:

monitors $\subset \mathbb{N} \times$ **RTEvent** $\times$ **Component** $\times$ **Environment**$^* \times$ **Environment**$^* \times$ **State**$^* \times$ **State**$^*_\perp$
monitors(i, ⟦rte⟧, ⟦c⟧, e, e', s, s') $\Leftrightarrow$
(eqMode(s(i), "running") $\vee$ eqMode(s(i), "ready")) $\wedge$ ⟦c⟧(e(i))(e'(i), s(i), s'(i)) $\wedge$
$\exists$ oe $\in$ ObEvent: equals(rte, store(⟦name(rte)⟧)(e(i))) $\wedge$
IF entryEvent(oe, c) THEN
data(c, s(i), s'(i)) $\wedge$
(preconditions(c, e(i), e'(i), s(i), s'(i), "compromised") $\Rightarrow$ equals(s(i+1), s(i)) $\wedge$ equals(s'(i+1), s(i+1))
$\wedge$ setFlag(inState(s'(i+1)), "compromised")) $\vee$ (preconditions(c, e(i), e'(i), s(i), s'(i), "normal")
$\Rightarrow$ setMode(s(i), "running") $\wedge$
LET cseq = components(c) IN
equals(s(i+1), s'(i)) $\wedge$ equals(e(i+1), e'(i)) $\wedge$
$\forall$ $c_1 \in$ cseq, $rte_1 \in$ RTEvent:
arrives(rte_1, s(i+1)) $\wedge$ monitor(i+1, rte_1, c_1, e(i+1), e'(i+1), s(i+1), s'(i+1))
END)
ELSE IF exitEvent(oe, c) THEN
data(c, s(i), s'(i)) $\wedge$ eqMode(inState(s'(i)), "completed") $\wedge$
(postconditions(c, e(i), e'(i), s(i), s'(i), "compromised") $\Rightarrow$ equals(s(i+1), s(i)) $\wedge$ equals(s'(i+1), s(i+1))
$\wedge$ setFlag(inState(s'(i+1)), "compromised")) $\vee$
(postconditions(c, e(i), e'(i), s(i), s'(i), "normal") $\Rightarrow$ equals(s(i+1), s'(i)) $\wedge$ equals(e(i+1), e'(i)) $\wedge$
setMode(inState(s'(i+1), "completed"))
ELSE IF allowableEvent(oe, c) THEN equals(s(i+1), s'(i)) $\wedge$ equals(e(i+1), e'(i))
ELSE equals(s(i+1), s(i)) $\wedge$ equals(s'(i+1), s(i+1)) $\wedge$ setFlag(inState(s'(i+1)), "compromised")
END

The predicate "monitors" are defined as a relation on

- the number of observation i, with respect to the iteration of a component
- an observation (runtime event) *rte*
- the corresponding component *c* under observation
- a sequence of pre-environments *e*
- a sequence of post-environments e'
- a sequence of pre-states *s*
- a sequence of post-states s'

The predicate "monitors" are defined such that when an any arbitrary observation is made, if the current execution state $s(i)$ of component *c* is "ready" or "running", the behavior of component *c* has been evaluated, and there is a *prediction oe* that is semantically equal to an *observation rte*, any of the following can happen:

- The *prediction* or *observation* is an entry event of the component *c* and it waits until the complete data for the component *c* arrives. If this occurs then either:

  - Preconditions of "normal" behavior of the component hold.If so, the subnetwork of the component is initiated and the components in the subnetwork are monitored iteratively with the corresponding arrival of the *observation*, or
  - Preconditions of "compromised" behavior of the component hold. In this case, the state is marked as "compromised" and returns.
- The *observation* is an exit event and, after the completion of the data arrival, the postconditions hold and the resulting state is marked as "completed".
- The *observation* is an allowable event and just continues the execution.
- The *observation* is an unexpected event (or any of the above does not hold) and the state is marked as "compromised", and returns.

4. Proof of Behavioral Correctness

Based on the formalization of the denotational semantics of the specification language and the monitor, we have proved that the monitor is sound and complete, i.e., if the application implementation (AppImpl) is consistent with its specification (AppSpec), the security monitor will produce no false alarms (soundness) and the monitor will detect any deviation of the application execution from the behavior sanctioned by the specification language (completeness). In the following subsections, we articulate soundness and completeness statements and sketch their corresponding proofs.

4.1. Soundness

The intent of the soundness statement is to articulate whether the system's behavior is consistent with behavioral specification. Essentially, the goal is to show the absence of a false negative alarm such that whenever the security monitor alarms, there is a semantic inconsistency between the post-state of the program execution and the post-state of the specification execution. The soundness theorem is stated as follows:

Theorem 1 (**Soundness**). *The result of the security monitor is sound for any execution of the target system and its specification, iff, the specification is consistent with the program and the program executes in a safe pre-state and in an environment that is consistent with the environment of the specification, then*

- *for the pre-state of the program, there is an equivalent safe pre-state for which the specification can be executed and the monitor can be observed and*
- *if we execute the specification in an equivalent safe pre-state and observe the monitor at any arbitrary (combined) post-state, then*
  - *either there is no alarm, and then the post-state is safe and the program execution (post-state) is semantically consistent with the specification execution (post-state)*
  - *or there is an alarm, and then the post-state is compromised and the program execution (post-state) and the specification execution (post-state) are semantically inconsistent.*

Formally, the soundness theorem has the following signatures and definition.

$$\begin{aligned}
&\text{Soundness} \subseteq \mathbb{P}(\text{AppImpl} \times \text{AppSpec} \times \text{Bool})\\
&\text{Soundness}(\kappa, \omega, b) \Leftrightarrow\\
&\forall\, e_s \in \text{Environment}_s, e_r, e_r' \in \text{Environment}_r, s, s' \in \text{State}_r\text{: consistent}(e_s, e_r) \wedge \text{consistent}(\kappa, \omega) \wedge\\
&\quad [\![\kappa]\!](e_r)(e_r', s, s') \wedge \text{eqMode}(s, \text{"normal"})\\
&\Rightarrow\\
&\quad \exists\, t, t' \in \text{State}_s, e_s' \in \text{Environment}_s\text{: equals}(s, t) \wedge [\![\omega]\!](e_s)(e_s', t, t') \wedge \text{monitor}(\kappa, \omega)(e_r;e_s)(s;t, s';t') \wedge\\
&\quad \forall\, t, t' \in \text{State}_s, e_s' \in \text{Environment}_s\text{: equals}(s, t) \wedge [\![\omega]\!](e_s)(e_s', t, t') \wedge \text{monitor}(\kappa, \omega)(e_r;e_s)(s;t, s';t')\\
&\quad \Rightarrow\\
&\quad \text{LET } b = \text{eqMode}(s', \text{"normal"}) \text{ IN}\\
&\qquad \text{IF } b = \texttt{True} \text{ THEN equals}(s', t') \text{ ELSE } \neg\, \text{equals}(s', t')
\end{aligned}$$

In detail, the soundness statement says that, if the following are satisfied:

1. a specification environment (e_s) is *consistent* with a run-time environment (e_r), and
2. a target system (κ) is *consistent* with its specification (ω), and
3. in a given run-time environment (e_r), execution of the system (κ) transforms the pre-state (s) into a post-state (s'), and
4. the pre-state (s) is safe, i.e., the state is in "normal" mode,

Then the following occurs:

- there exist the pre- and post-states (t and t', respectively) and the environment (e_s') of the specification execution such that in a given specification environment (e_s), the execution of the specification (ω) transforms the pre-state (t) into a post-state (t')
- the pre-states s and t are *equal* and *monitor*ing of the system (κ) transforms the combined pre-state (s; t) into a combined post-state (s'; t')
- if both the following occur: in a given specification environment (e_s), execution of the specification (ω) transforms pre-state (t) into a post-state (t'); and the pre-states s and t are *equal* and *monitor*ing of the system (κ) transforms the pre-state (s) into a post-state (s'), then either:
  - there is no alarm (b is `True`), the post-state s' of a program execution is safe, and the resulting states s' and t' are semantically *equal*, or
  - the security monitor alarms (b is `False`), the post-state s' of program execution is compromised, and the resulting states s' and t' are semantically not *equal*.

In the following section we present proof of the soundness statement.

Proof. The proof is essentially a structural induction on the elements of the specification (ω) of the system (κ). We have proved only the interesting case β of the specification to show that the proof works in principle. However, the proof of the remaining parts can easily be rehearsed following a similar approach.

The proof is based on certain lemmas, which are mainly about the relationships between different elements of the system and its specification (being at different levels of abstraction). These lemmas and relations can be proved based on the defined auxiliary functions and predicates that are based on the method suggested by Hoare [9]. The complete proof is presented in [10]. □

4.2. Completeness

The goal of the completeness statement is to show the absence of false positive alarms such that whenever there is a semantic inconsistency between the post-state of the program execution and the post-state of the specification execution, the security monitor alarms. The completeness theorem is stated as follows:

Theorem 2 (**Completeness**). *The result of the security monitor is complete for a given execution of the target system and its specification, iff, the specification is consistent with the program and the program executes in a safe pre-state and in an environment that is consistent with the environment of the specification, then*

- *for the pre-state of the program, there is an equivalent safe pre-state for which the specification can be executed and the monitor can be observed and*
- *if we execute the specification in an equivalent safe pre-state and observe the monitor at any arbitrary (combined) post-state, then*
  - *either the program execution (post-state) is semantically consistent with the specification execution (post-state), then there is no alarm and the program execution is safe*
  - *or the program execution (post-state) and the specification execution (post-state) are semantically inconsistent, then there is an alarm and the program execution has been compromised.*

Formally, the completeness theorem has the following signatures and definition.

Completeness $\subseteq \mathbb{P}$(AppImpl $\times$ AppSpec $\times$ Bool)
Completeness(κ, ω, b) $\Leftrightarrow$
$\forall$ $e_s \in$ Environment$_s$, e_r, $e_r' \in$ Environment$_r$, s, s' $\in$ State$_r$: consistent(e_s, e_r) $\wedge$ consistent(κ, ω) $\wedge$
$[\![\kappa]\!](e_r)(e_r', s, s') \wedge$ eqMode(s, "normal")
$\Rightarrow$
$\exists$ t, t' $\in$ State$_s$, $e_s' \in$ Environment$_s$: equals(s, t) $\wedge$ $[\![\omega]\!](e_s)(e_s', t, t') \wedge$ monitor(κ, ω)(e_r;e_s)(s;t, s';t') $\wedge$
$\forall$ t, t' $\in$ State$_s$, $e_s' \in$ Environment$_s$: equals(s, t) $\wedge$ $[\![\omega]\!](e_s)(e_s', t, t') \wedge$ monitor(κ, ω)(e_r;e_s)(s;t, s';t')
$\Rightarrow$
IF equals(s', t') THEN b = `True` $\wedge$ b = eqMode(s', "normal")
ELSE b = `False` $\wedge$ b = eqMode(s', "normal")

In detail, the completeness statement says that, if the following are satisfied:

1. a specification environment (e_s) is *consistent* with a run-time environment (e_r), and
2. a target system (κ) is *consistent* with its specification (ω), and
3. in a given run-time environment (e_r), execution of the system (κ) transforms the pre-state (s) into a post-state (s'), and
4. the pre-state (s) is safe, i.e., the state is in "normal" mode,

Then the following occurs:

- there exist the pre- and post-states (t and t', respectively) and the environment (e_s') of specification execution such that, in a given specification environment (e_s), execution of the specification (ω) transforms the pre-state (t) into a post-state (t')
- the pre-states s and t are *equal* and *monitor*ing of the system (κ) transforms the combined pre-state (s; t) into a combined post-state (s'; t')
- if both: in a given specification environment (e_s), the execution of the specification (ω) transforms the pre-state (t) into a post-state (t'); and the pre-states s and t are *equal* and *monitor*ing of the system (κ) transforms the pre-state (s) into a post-state (s'), then either
  - the resulting two post-states s' and t' are semantically *equal* and there is no alarm, or
  - the resulting two post-states s' and t' are semantically not *equal* and the security monitor alarms.

Proof. The proof of completeness is very similar to the proof of soundness. The complete proof is presented in [10]. □

5. Conclusions

We have presented a formalization of the semantics of the specification language and monitor of the cognitive middleware ARMET. In order to assure the continuous operation of ICS applications and to meet the real-time requirements of ICS, we have proved that our run-time security monitor produces no false alarm and always detects behavioral deviation of the ICS application. We plan to integrate our run-time security monitor with a security-by-design component to ensure comprehensive security solution for ICS applications.

Acknowledgments: The authors thank the anonymous reviewers on the earlier version of this work.

Author Contributions: All authors of the paper have contributed to the presented results. The ARMET prototype is based on the AWDRAT software developed by Howard Shrobe.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Khan, M.T.; Serpanos, D.; Shrobe, H. A Rigorous and Efficient Run-time Security Monitor for Real-time Critical Embedded System Applications. In Proceedings of the 2016 IEEE 3rd World Forum on Internet of Things (WF-IoT), Reston, VA, USA, 12–14 December 2016; pp. 100–105.
2. Shrobe, H.; Laddaga, R.; Balzer, B.; Goldman, N.; Wile, D.; Tallis, M.; Hollebeek, T.; Egyed, A. AWDRAT: A Cognitive Middleware System for Information Survivability. In Proceedings of the IAAI'06, 18th Conference on Innovative Applications of Artificial Intelligence, Boston, MA, USA, 16–20 July 2006; AAAI Press: San Francisco, CA, USA, 2006; pp. 1836–1843.
3. Shrobe, H.E. *Dependency Directed Reasoning for Complex Program Understanding*; Technical Report; Massachusetts Institute of Technology: Cambridge, MA, USA, 1979.
4. Lynx Software Technologies. LynxOS. Available online: http://www.lynx.com/industry-solutions/industrial-control/ (accessed on 20 July 2017).
5. Khan, M.T.; Serpanos, D.; Shrobe, H. *On the Formal Semantics of the Cognitive Middleware AWDRAT*; Technical Report MIT-CSAIL-TR-2015-007; Computer Science and Artificial Intelligence Laboratory, MIT: Cambridge, MA, USA, 2015.
6. Lamport, L. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.* **1994**, *16*, 872–923.
7. Khan, M.T.; Schreiner, W. Towards the Formal Specification and Verification of Maple Programs. In *Intelligent Computer Mathematics*; Jeuring, J., Campbell, J.A., Carette, J., Reis, G.D., Sojka, P., Wenzel, M., Sorge, V., Eds.; Springer: Berlin, Germany, 2012; pp. 231–247.
8. Schmidt, D.A. *Denotational Semantics: A Methodology for Language Development*; William, C., Ed.; Brown Publishers: Dubuque, IA, USA, 1986.
9. Hoare, C.A.R. Proof of correctness of data representations. *Acta Inform.* **1972**, *1*, 271–281.
10. Khan, M.T.; Serpanos, D.; Shrobe, H. *Sound and Complete Runtime Security Monitor for Application Software*; Technical Report MIT-CSAIL-TR-2016-017; Computer Science and Artificial Intelligence Laboratory, MIT: Cambridge, MA, USA, 2016.