On the Behavioral Formalization of the Cognitive Middleware AWDRAT

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

[1] QCRI, Qatar
{mtkhan, dserpanos}@qf.org.qa
[2] CSAIL, MIT, U.S.A.
hes@csail.mit.edu

We present our ongoing work and initial results towards the (behavioral) correctness analysis of the cognitive middleware AWDRAT [5]. Since, the (provable) behavioral correctness of a software system is a fundamental pre-requisite of the system's security. Therefore, the goal of the work is to first formalize the behavioral semantics of the middleware as a pre-requisite for our proof of the behavioral correctness. However, in this paper, we focus only on the core and critical component of the middleware, i.e. Execution Monitor which is a part of the module "Architectural Differencer" of AWDRAT. The role of the execution monitor is to identify inconsistencies between runtime *observations* of the target system and *predictions* of the specification System Architectural Model of the system. As a starting point we have defined the formal (denotational) semantics of the *observations* (runtime events) and *predictions* (executable specifications as of System Architectural Model); then based on the aforementioned formal semantices, we have formalized the behavior of the "Execution Monitor" of the middleware. The material of the parts of this paper is based on [3].

AWDRAT is a general purpose middleware system that provides survivability to any kind of new and legacy software system. In detail, the middleware checks for consistency between the target system's actual (runtime) behavior and the expected (system specification) behavior of the system, if there is the one then the diagnostic engine identifies an attack (illegal behavioral pattern) and the corresponding set of resources which were compromised during the attack. After identifying an attack, AWDRAT attempts to repair respectively regenerate the compromised system into a safer state, if possible. The task of regeneration is based on the dependency-directed reasoning [6] engine of the system that contributes to the self-organization and self-awareness of the system by recording execution steps intrinsically states of the system and their corresponding justification (reason). Based on the Execution Monitor and the reasoning engine of AWDRAT not only the detection of known attacks is possible but also detection (resp. recovery from) the unknown attacks is also possible.

A Specification Language of AWDRAT: A specification language "System Architectural Model" of AWDRAT supports to specify the target system behavior based on a fairly high-level description written in a language of "Plan Calculus" [6] which is a decomposition of pre- and post- and invariant conditions for each computing component (module) of the target system. The description can be considered as an executable specification of the system. The specification is a hierarchical nesting of system's components such that input and output ports of each component are connected by data and control flow links 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 specification (System Architectural Model) of target system is described at the following two logical levels:

1. *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. *behavior level* describes the actual method’s behavioral specification of each of the 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 System Architectural Model. For syntactic details, please see [3].

Based on the core idea of Lamport [4], we have defined the semantics of the specification as a state relationship to achieve the desired insight of the program’s behavior by relating pre- and post-states. Semantically, the System Architectural Model *SAM* holds in a given environment e resulting in an environment e' by transforming a pre-state s into post-state s' as defined below.

$$
\begin{array}{l}
\llbracket \text{SAM} \rrbracket(\text{e})(\text{e'}, \text{s}, \text{s'}) \Leftrightarrow \\
\ \forall\, e_1, e_2, e_3 \in \text{Environment}, s_1, s_2, s_3 \in \text{State:} \\
\quad \llbracket \text{RegModSeq} \rrbracket(\text{e})(\text{e}_1, \text{s}, \text{inState}_\perp(\text{s}_1)) \wedge \llbracket \text{StrModSeq} \rrbracket(\text{e}_1)(\text{e}_2, \text{s}_1, \text{inState}_\perp(\text{s}_2)) \wedge \\
\quad \llbracket \text{BehModSeq} \rrbracket(\text{e}_2)\ (\text{e}_3, \text{s}_2, \text{inState}_\perp(\text{s}_3)) \wedge \llbracket \text{SpltModSeq} \rrbracket(\text{e}_3)(\text{e'}, \text{s}_3, \text{s'})
\end{array}
$$

For further details on the semantics, please see [3].

An Execution Monitor of AWDRAT: In principle, an execution monitor interprets the event stream (traces of the execution of the target system aka *observations*) against the system specification (the execution of the specification is also called *predictions*) by detecting inconsistencies between *observations* and the *predictions*, if there is any.

When the target system starts execution, an initial “startup” event is generated and dispatched to the top level component (module) of the system which transforms the execution state of the component into “running” mode. The component instantiates its subnetwork (of components, if there is one) and also 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 transform the state of the component into “ready” mode. In case, any of the preconditions fails, it enables diagnosis engine.

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

1. If the event is a “method entry”, then the execution monitor checks if this is one of the “entry events” of the corresponding component in the “ready” state; if so, then after receiving the data and the respective preconditions are checked; if they succeed, then 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”, then the execution monitor checks if this one of the “exit events” of the component in the “running” state; if so, it changes its state into “completed” mode and collects the data from the output port of the component and checks for the 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 and finally

4. if the event is an unexpected event, i.e. it is neither an “entry event”, nor an “exit event” and also not in “allowable events”, then the execution monitor starts diagnosis.

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

$$
\begin{array}{l}
\forall \text{ app} \in \text{Target\_System, sam} \in \text{System\_Architectural\_Model, c} \in \text{Component,}\\
\quad \text{s, s'} \in \text{State, t, t'} \in \text{State}_s\text{, d, d'} \in \text{Environment}_s\text{, e, e'} \in \text{Environment, rte} \in \text{RTEvent:}\\
\quad [\![\text{sam}]\!]\text{(d)(d', t, t')} \wedge [\![\text{app}]\!]\text{(e)(e', s, s')} \wedge \text{startup(s, app)} \wedge \text{isTop(c, }[\![\text{app}]\!]\text{(e)(e', s, s'))} \wedge\\
\quad \text{setMode(s, ``running'')} \wedge \text{arrives(rte, s)} \wedge \text{equals(t, s)} \wedge \text{equals(d, e)}\\
\quad \Rightarrow\\
\qquad \forall \text{ p, p'} \in \text{Environment}^*\text{, m, n} \in \text{State}^*_\perp\text{:}\\
\qquad\quad \text{equals(m(0), s)} \wedge \text{equals(p(0), e)}\\
\qquad\quad \Rightarrow\\
\qquad\qquad \exists \text{ k} \in \mathbb{N}\text{, p, p'} \in \text{Environment}^*\text{, m, n} \in \text{State}^*_\perp\text{:}\\
\qquad\qquad\quad \forall \text{ i} \in \mathbb{N}_k : \text{monitors(i, rte, c, p, p', m, n)} \wedge\\
\qquad\qquad\qquad \text{(eqMode(n(k), ``completed'')} \wedge \text{eqFlag(n(k), ``normal'')} \wedge\\
\qquad\qquad\qquad\qquad \text{equals(s', n(k))}\\
\qquad\qquad\qquad \vee\\
\qquad\qquad\qquad \text{eqFlag(n(k), ``compromised'')}\\
\qquad\qquad\qquad\quad \Rightarrow\\
\qquad\qquad\qquad\qquad \text{enableDiagnosis(p'(k))(n(k), inBool(}\texttt{true}\text{))} \wedge \text{equals(s', n(k)))}
\end{array}
$$

The semantics of recursive monitoring is determined by two sequences of states pre and post that are constructed from the pre-state of the monitor. Any *ith* recursion of the monitor transforms $pre(i)$ state into $post(i+1)$ state from which the $pre(i+1)$ is constructed. No event can be accepted in an *Error* state and the corresponding monitoring terminates either when the application has terminated with "normal" mode or when some misbehavior is detected as indicated by the respective "compromised" state. The corresponding "monitors" predicate formalizes the aforementioned semantics as discussed in [3].

The formalization gives deep insight of the internal behavior of AWDRAT increasing its usability on the one hand and developing basis for its correctness (to be proved by automated tools) on the other hand. Based on this formalism, we are currently working on the proof of the soundness of the Execution Monitor. The proof is essentially a structural induction proof, however, the non-trivial part is the recursive definition of the semantics of the monitor that is to be proved by the principle of rule induction [7]. We also plan to extend AWDRAT such that a target system's behavior is specified using Abstract State Machine [1] based formalism which then will automatically translate into a semantically equivalent System Architectural Model allowing to already check the inconsistencies in the intra system behavior with various ASM automated tools, e.g. DKAL [2].

References

[1] E. Borger and Robert F. Stark. *Abstract State Machines: A Method for High-Level System Design and Analysis.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[2] Jean-Baptiste Jeannin, Guido de Caso, Juan Chen, Yuri Gurevich, Prasad Naldurg, and Nikhil Swamy. DKAL*: Constructing Executable Specifications of Authorization Protocols. Technical Report MSR-TR-2013-19, March 2013.

[3] Muhammad Taimoor Khan, Dimitrios Serpanos, and Howard Shrobe. On the Formal Semantics of the Cognitive Middleware AWDRAT. Technical Report CSAIL, MIT (to appear), September 2014.

[4] Leslie Lamport. The Temporal Logic of Actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, May 1994.

[5] Howard Shrobe, Robert Laddaga, Bob Balzer, Neil Goldman, Dave Wile, Marcelo Tallis, Tim Hollebeek, and Alexander Egyed. AWDRAT: A Cognitive Middleware System for Information Survivability. In *Proceedings of the 18th Conference on Innovative Applications of Artificial Intelligence - Volume 2*, IAAI'06, pages 1836–1843. AAAI Press, 2006.

[6] Shrobe, Howard E. Dependency Directed Reasoning for Complex Program Understanding. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1979.

[7] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction.* MIT Press, Cambridge, MA, USA, 1993.