Toward Developing a Methodology for Building Secure Software Systems

Michael Gelfond
Texas Tech University

September 27, 2015
The Goal of Research:

Develop a methodology for design and implementation of secure software systems.

The methodology should guide us in:

- Defining a mathematical model of the system.
- Defining what it means for the system to be secure.
- Finding algorithms for checking security.

The design should be elaboration tolerant – small changes in specifications should correspond to small changes in the system.
The Approach:

Model the system and its past behavior by:

- A state-action-state transition diagram describing all possible trajectories of the system. Describe the diagram in an action language.
- A recorded history defining past trajectories believed to be possible by the agent.
- A policy specification characterizing system trajectories which are preferred or discouraged.

Reduce reasoning about security to computing answer sets of logic programs in ASP based language(s).
Consider a simple domain in which different commanders authorize and/or assume command of various military missions.

Even though both actions can be actually performed by commanders the process is guided by a collection of authorization policies:

1. A military officer is not allowed to command a mission he authorized.
2. A colonel is allowed to command a mission he authorized.
3. A military observer shall never authorize a mission.

*Build a system* to reason about effects of commanders’ actions and check their compliance with the policies.
The Basic Terms

sorts:
   #missions   #commanders

actions:
   authorize(#commanders, #missions)
   assume_command(#commanders, #missions)

inertial fluents:
   authorized(#commanders, #missions)
   commands(#commanders, #missions)

defined fluents:
   authorized(#missions)

statics:
   colonel(#commanders)
   observer(#commanders)
Dynamic Causal Law:

authorize(C, M) causes authorized(C, M)

Definition:

authorized(M) if authorized(C, M)

Executability Condition:

impossible authorize(C, M) if authorized(M)

Dynamic Causal Law:

assume_command(C, M) causes commands(C, M)
Histories

Sorts and Statics:

#commanders = \{c_1, c_2, c_3\}
#missions = \{m_1, m_2\}
observer(c_1)
colonel(c_2)

History 1:

hpd(authorize(c_3, m_1), 0)
hpd(assume_command(c_2, m_1), 1)
complies with the military policy while

History 2:

hpd(authorize(c_1, m_1), 0)
hpd(assume_command(c_2, m_1), 1)
does not.
In “Authorization and Obligation Policies in Dynamic Systems” (ICLP 2008) M. Gelfond and J. Lobo introduced a policy description language $\mathcal{APL}$.

Authorization policies of $\mathcal{APL}$ are of the form:

\[
\text{permitted}(e) \text{ if cond} \quad (1)
\]
\[
\neg\text{permitted}(e) \text{ if cond} \quad (2)
\]
\[
d : \text{normally permitted}(e) \text{ if cond} \quad (3)
\]
\[
d : \text{normally } \neg\text{permitted}(e) \text{ if cond} \quad (4)
\]
\[
\text{prefer}(d_1, d_2). \quad (5)
\]
Military Policies in \textit{ALP}

1. \(d_1(C, M)\): A military officer is not allowed to command a mission he authorized.

\[
\text{normally } \neg \text{perm}(\text{assume\_com}(C, M)) \text{ if auth}(C, M)
\]

2. \(d_2(C, M)\): A colonel is allowed to command a mission he authorized.

\[
\text{normally perm}(\text{assume\_com}(C, M)) \text{ if col}(C), \text{auth}(C, M)
\]

3. A military observer shall never authorize a mission.

\[
\neg \text{permitted}(\text{authorize}(C, M)) \text{ if observer}(C)
\]

To prefer more specific policy we add:

\[
\text{prefer}(d_2(C, M), d_1(C, M))
\]
Semantics of $\mathcal{APL}$ defines a mapping $\mathcal{P}$ from states of the transition diagram $\tau$ into the collection of permissions and prohibitions defined by the policy. This is done by translating the domain axioms and policies into a logic program $lp$.

For a state $\sigma$, $\mathcal{P}(\sigma)$ returns the set of all statements of the form permitted$(a)$ and $\neg$permitted$(a)$ which belong to all the answer sets of $lp \cup \sigma$. 
The translation of axioms into a logic program becomes possible thanks to extensive theoretical work on knowledge representation in ASP and the relationship between ASP and reasoning about actions.

It contains solutions of the Frame and Ramification problems, allows recursive relations between fluents, non-determinism of actions, etc.
The translation of policies utilizes the ASP ability to represent defaults.

For instance, the translation of a policy $d_1$:

$$\text{permitted}(e) \text{ if } \text{cond}_1$$

has a form:

$$\text{permitted}(e) \leftarrow \text{cond}_1,$$
$$\text{not } \text{ab}(d_1),$$
$$\text{not } \neg \text{permitted}(e).$$

Preference of policy $d_1$ over $d_2$ is given by

$$\text{ab}(d_2) \leftarrow \text{cond}_1.$$
Policy $\mathcal{P}$ is called

- **consistent** if for every state $\sigma$ program $\mathcal{L} \cup \sigma$ has an answer set.
- **unambiguous** if for every state $\sigma$ program $\mathcal{L} \cup \sigma$ has exactly one answer set.
- **complete** if for every state $\sigma$ and every action $a$, $\text{permitted}(a) \in \mathcal{P}(\sigma)$ or $\neg \text{permitted}(a) \in \mathcal{P}(\sigma)$.

These are important properties which often can be proven or even checked automatically.
Let $P$ be an authorization policy. An event $\langle \sigma, a \rangle$ where $a$ is a set of elementary actions is called

- **strongly compliant** with $P$ if for every $e \in a$ we have that $\text{permitted}(e) \in P(\sigma)$.
- **weakly compliant** with $P$ if for every $e \in a$ we have that $\neg \text{permitted}(e) \notin P(\sigma)$.
- **not compliant** with $P$ if for some $e \in a$ we have that $\neg \text{permitted}(e) \in P(\sigma)$.

A path $\langle \sigma_0, a_0, \sigma_1, \ldots, \sigma_{n-1}, a_{n-1}, \sigma_n \rangle$ is said to be **strongly (weakly) compliant** with $P$ if for every $0 \leq i < n$ the event $\langle \sigma_i, a_i \rangle$ is strongly (weakly) compliant with $P$. 
Event $\langle \sigma, e \rangle$ is strongly compliant with consistent policy $\mathcal{P}$ iff a logic program

\[ \text{lp} \cup \sigma \cup \{ \leftarrow \text{permitted}(e) \} \]

is inconsistent.

Event $\langle \sigma, e \rangle$ is weakly compliant with $\mathcal{P}$ iff a logic program

\[ \text{lp} \cup \sigma \cup \{ \leftarrow \lnot \text{permitted}(e) \} \]

is consistent.

Event $\langle \sigma, e \rangle$ is not compliant with policy $\mathcal{P}$ iff a logic program

\[ \text{lp} \cup \sigma \cup \{ \leftarrow \lnot \text{permitted}(e) \} \]

is inconsistent.
Let $s$ be a set of fluent literals, $\delta(s)$ be a set of states compatible with $s$, and $D = \{f \text{ or } \neg f : f \text{ is a fluent}\}$. 

- Event $\langle \sigma, e \rangle$ is strongly compliant with $P$ for every $\sigma \in \delta(s)$ iff program $lp \cup s \cup D \cup \{\leftarrow \text{ permitted}(e)\}$ is inconsistent.

- If $P$ is categorical then an event $\langle \sigma, e \rangle$ is weakly compliant with $P$ for every $\sigma \in \delta(s)$ iff program $lp \cup s \cup D \cup \{\leftarrow \neg \text{ permitted}(e)\}$ is inconsistent.

- Event $\langle \sigma, e \rangle$ is not compliant with $P$ for every $\sigma \in \delta(s)$ iff program $lp(P, s) \cup D \cup lp(SL) \cup \{\leftarrow \neg \text{ permitted}(e)\}$ is inconsistent.
These and other similar results can be used to check compliance of actions and sequences of actions using ASP solvers.

The checking can be done by an agent associated with the system or by an outside monitor.

Policies can be easily modified and updated with proper consistency checks.

The corresponding ASP programs can be constructed automatically from descriptions of the system, its history, policies, and the type of compliance one is interested in.
We were reasonably satisfied with the proposed methodology.

It guided us in constructing a model of the system, in precisely describing policies and the meaning of security, and provided algorithms for checking quality of the policy and the system’s compliance.

_The approach allowed to put security concerns at the center of the system design instead of viewing it as a necessary but unpleasant addition to the already designed system._

More traditional role based approaches to policy specifications (e.g. RBAC) were shown to be easily expressible in our language.
Unfortunately, the work did not continue.

Partly it is related to non-scientific reasons, like Jorge leaving the IBM and my failure to find a useful medium size system to build using our methodology.

But it was also related to a scientific problem – a failure of 2008 solvers to efficiently reason with numerical constraints which were frequently needed in practice.

With advent of various types of Constraint ASP the problem may have already disappeared.

It maybe a perfect time to systematize existing work on policies, action languages, and ASP and see if it can give us more insights and be useful in practice.