Runtime Monitoring, Verification, Enforcement and Control of C Programs (From Tool to Semantics)

Zhe Chen

Nanjing University of Aeronautics and Astronautics, China

(an extension of TASE'15 paper)
5 December, 2015

Outline

Outline

Terminologies

- Software systems are usually constrained by a set of properties, e.g., correctness requirements, safety and security policies.

Terminologies

- Software systems are usually constrained by a set of properties, e.g., correctness requirements, safety and security policies.
- Runtime monitoring is an infrastructural method that uses *monitors* to observe the dynamic execution of a target program at runtime.

Terminologies

- Software systems are usually constrained by a set of properties, e.g., correctness requirements, safety and security policies.
- Runtime monitoring is an infrastructural method that uses *monitors* to observe the dynamic execution of a target program at runtime.
- Runtime verification uses runtime monitoring for verification purpose, i.e., analyzing the dynamic execution at runtime to detect property violations.

Terminologies

- Software systems are usually constrained by a set of properties, e.g., correctness requirements, safety and security policies.
- Runtime monitoring is an infrastructural method that uses *monitors* to observe the dynamic execution of a target program at runtime.
- Runtime verification uses runtime monitoring for verification purpose, i.e., analyzing the dynamic execution at runtime to detect property violations.
- Runtime enforcement uses runtime monitoring for enforcement purpose, i.e., halting a system if it does not respect desired properties.

Terminologies

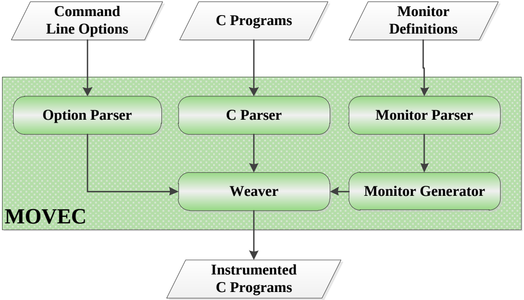
- Software systems are usually constrained by a set of properties, e.g., correctness requirements, safety and security policies.
- Runtime monitoring is an infrastructural method that uses *monitors* to observe the dynamic execution of a target program at runtime.
- Runtime verification uses runtime monitoring for verification purpose, i.e., analyzing the dynamic execution at runtime to detect property violations.
- Runtime enforcement uses runtime monitoring for enforcement purpose, i.e., halting a system if it does not respect desired properties.
- Runtime control uses runtime monitoring to actively control and correct the execution of the target system at runtime by calling some predefined controlling actions.

The MOVEC Tool

- MOVEC: an automated tool for
  MOnitoring, VErification and Control of C Programs

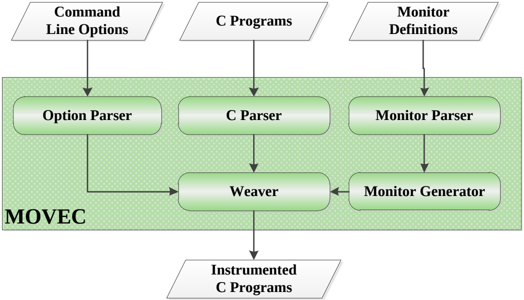
The MOVEC Tool

- MOVEC: an automated tool for
  MOnitoring, VErification and Control of C Programs
- Principle:



The MOVEC Tool

- MOVEC: an automated tool for
  MOnitoring, VErification and Control of C Programs
- Principle:



- Outperforms many monitoring tools for C programs, according to our preliminary experimental results.

Tool Demo

TOOL DEMO

- target program ⇒ instrumented controlled program
- specification ⇒ controlling program
- weave the two by compiling

Motivations

Existing problems:

- The state-of-the-art study of these topics lacks an appropriate formal program semantics of runtime monitoring, in contrast to the relatively abundant implementations.
- The existing works on semantics are too general to express the semantics of key implementation techniques, such as program instrumentation and synthesis of controlling programs from specifications.

Contributions

- We will propose a theory of runtime control at an appropriate level of formalization to provide a formal program semantics for MOVEC.

Contributions

- We will propose a theory of runtime control at an appropriate level of formalization to provide a formal program semantics for MOVEC.
- The semantics contains:
    - target programs, to be controlled.
    - controlling programs, which can perform
        - passive actions for monitoring, i.e., to observe the execution of a target program at runtime.
        - active actions for controlling, i.e., to control and correct its execution via active controlling actions.
    - transition system semantics of instrumented target programs under the control of controlling programs.

Contributions

- We will propose a theory of runtime control at an appropriate level of formalization to provide a formal program semantics for MOVEC.
- The semantics contains:
  - target programs, to be controlled.
  - controlling programs, which can perform
    - passive actions for monitoring, i.e., to observe the execution of a target program at runtime.
    - active actions for controlling, i.e., to control and correct its execution via active controlling actions.
  - transition system semantics of instrumented target programs under the control of controlling programs.
- Objective:
  - provides a complete formal semantics for real implementations of runtime monitoring and control.
  - retains a good balance between implementation and generality.

Outline

1. Introduction
2. Preliminaries
3. Semantics of Runtime Control
4. Semantics of Synthesis of Controlling Programs
5. Expressiveness of Controlling Programs
6. Conclusion

Semantics

program graphs ⇒ transition systems

Programs as Program Graphs (PG)

Definition (Program Graphs (PG))

A program graph PG over set Var of typed variables is a tuple $(Loc, Act, Eff, Tr, Loc_0, g_0)$ where
- Loc is a set of locations,
- Act is a set of actions,
- $Eff : Act \times Eval(Var) \rightarrow Eval(Var)$ is the effect function,
- $Tr \subseteq Loc \times Cond(Var) \times Act \times Loc$ is the conditional transition relation,
- $Loc_0 \subseteq Loc$ is a set of initial locations, and
- $g_0 \in Cond(Var)$ is the initial condition.

For example, let $l \overset{g:\alpha}{\hookrightarrow} l' \in Tr$, where g denotes a guard, α denotes the action $x = y + 1$, and η is the evaluation with $\eta(x, y) = (1, 1)$, then $Eff(\alpha, \eta)(x, y) = (2, 1)$.

Transition Systems (TS)

A transition system is basically a directed graph where nodes represent *states*, and edges model *transitions*.

Definition (Transition Systems (TS))

A transition system TS is a tuple $(S, Act, \delta, I, AP, L)$ where
- S is a set of states,
- Act is a set of actions,
- $\delta \subseteq S \times Act \times S$ is a transition relation,
- $I \subseteq S$ is a set of initial states,
- AP is a set of atomic propositions, and
- $L : S \rightarrow 2^{AP}$ is a labeling function.

Transition System Semantics of a Program Graph

Each program graph can be interpreted as a transition system by unfolding the program graph.

Definition (Transition System Semantics of a Program Graph)

The transition system TS(PG) of program graph PG is the tuple $(S, Act, \delta, I, AP, L)$ where

- $S = Loc \times Eval(Var)$
- $\delta \subseteq S \times Act \times S$ is defined by the following rule:

$$\frac{l \overset{g:\alpha}{\hookrightarrow} l' \wedge \eta \models g}{\langle l, \eta \rangle \xrightarrow{\alpha} \langle l', Eff(\alpha, \eta) \rangle}$$

- $I = \{\langle l, \eta \rangle \mid l \in Loc_0, \eta \models g_0\}$
- $AP = Loc \cup Cond(Var)$
- $L(\langle l, \eta \rangle) = \{l\} \cup \{g \in Cond(Var) \mid \eta \models g\}$.

Outline

Semantics of Runtime Control

PG ⇒ instrumented PG (IPG)

IPG + controlling PG ⇒ TS

Controlling Programs

A controlling program is a program that implements desired properties and controls the execution of a target program to fulfill the properties.

Controlling Programs

A controlling program is a program that implements desired properties and controls the execution of a target program to fulfill the properties.

It is a program with action partitioning:

- Passive actions are used to "passively" observe and monitor the actions of the controlled program graph. (side-effect free) They are further partitioned into:
  - pre-actions are monitored before each invocation of the interested action,
  - post-actions are monitored after each invocation.
- Active actions are "actively" performed to modify its state as well as the state of the controlled program graph.

Controlling Program Graphs (CPG)

Formally,

Definition (Controlling Program Graphs (CPG))

A controlling program graph CPG over set $\widehat{Var}$ of typed variables, which controls a program graph PG, is a tuple $(\widehat{Loc}, \widehat{Act}, \widehat{Eff}, \widehat{Tr}, \widehat{Loc}_0, \widehat{g}_0)$ where

- $\widehat{Loc}$ is a set of locations, including passive locations $\widehat{Loc}^{pas}$ and active locations $\widehat{Loc}^{act}$ which can perform passive actions and active actions respectively, i.e., $\widehat{Loc} = \widehat{Loc}^{pas} \cup \widehat{Loc}^{act}$,
- $\widehat{Act}$ is a set of actions, including passive actions $\widehat{Act}^{pas}$ and active actions $\widehat{Act}^{act}$, i.e., $\widehat{Act} = \widehat{Act}^{pas} \cup \widehat{Act}^{act}$, and the set of passive actions further includes *pre-actions* $\widehat{Act}^{pre}$ and *post-actions* $\widehat{Act}^{post}$, i.e., $\widehat{Act}^{pas} = \widehat{Act}^{pre} \cup \widehat{Act}^{post}$,

Controlling Program Graphs (CPG)

Definition (cont'd)

- $\widehat{Eff} : \widehat{Act} \times Eval(PC \cup Var \cup \widehat{Var}) \rightarrow Eval(PC \cup Var \cup \widehat{Var})$ is the effect function, satisfying that, if $\alpha \in \widehat{Act}^{pas}$, then $\widehat{Eff}(\alpha, \langle l, \eta, \widehat{\eta} \rangle) = \langle l, \eta, \widehat{\eta} \rangle$ (passive actions are side-effect free), where PC is a program counter with a value from Loc indicating the current location of the controlled program graph, i.e., $dom(PC) = Loc$,

Note that the effect function of an action indicates how an evaluation $\langle l, \eta, \widehat{\eta} \rangle$ of variables is modified, including not only the variables $\widehat{Var}$ of the CPG, but also the program counter PC and the variables Var of the controlled PG.

Controlling Program Graphs (CPG)

Definition (cont'd)

- $\widehat{Tr} \subseteq \widehat{Loc} \times Cond(\widehat{Var}) \times \widehat{Act} \times \widehat{Loc}$ is the conditional transition relation, satisfying
  (1) If $(l, g, \alpha, l') \in \widehat{Tr} \wedge \alpha \in \widehat{Act}^{pas}$, then $g = \top$ (unconditional monitoring of passive actions), $l \in \widehat{Loc}^{pas}$ and $\forall \beta \in \widehat{Act}^{act}, \forall g'', \forall l'', (l, g'', \beta, l'') \notin \widehat{Tr}$. (consistency of passive actions and passive locations, and separation of passive and active actions)
  (2) If $(l, g, \alpha, l') \in \widehat{Tr} \wedge \alpha \in \widehat{Act}^{act}$, then $l \in \widehat{Loc}^{act}$ and $\forall \beta \in \widehat{Act}^{pas}, \forall g'', \forall l'', (l, g'', \beta, l'') \notin \widehat{Tr}$. (consistency of active actions and active locations, and separation of passive and active actions)
- $\widehat{Loc}_0 \subseteq \widehat{Loc}$ is a set of initial locations, and
- $\widehat{g}_0 \in Cond(\widehat{Var})$ is the initial condition.

Semantics of Runtime Control

PG $\Rightarrow$ instrumented PG (IPG)

IPG $+$ controlling PG $\Rightarrow$ TS

Instrumenting Controlled Programs

- CPGs should be notified before or after the invocations of the monitored actions, i.e., to implement the couplings between PGs and CPGs.

Instrumenting Controlled Programs

- CPGs should be notified before or after the invocations of the monitored actions, i.e., to implement the couplings between PGs and CPGs.
- We rewrite the original PG by using automated program instrumentation of pre-locations or/and post-locations.

  For example, assume that the transition $l \overset{g:\alpha}{\hookrightarrow} l'$ is in PG, and α is monitored both pre- and post- its invocations, then the transition is split into three transitions:

$$l \overset{g:\alpha^{pre}}{\hookrightarrow} l^{\alpha^{pre}} \overset{\alpha}{\hookrightarrow} l^{\alpha^{post}} \overset{\alpha^{post}}{\hookrightarrow} l'$$

Instrumenting Controlled Programs

- CPGs should be notified before or after the invocations of the monitored actions, i.e., to implement the couplings between PGs and CPGs.
- We rewrite the original PG by using automated program instrumentation of pre-locations or/and post-locations.

  For example, assume that the transition $l \overset{g:\alpha}{\hookrightarrow} l'$ is in PG, and α is monitored both pre- and post- its invocations, then the transition is split into three transitions:

$$l \overset{g:\alpha^{pre}}{\hookrightarrow} l^{\alpha^{pre}} \overset{\alpha}{\hookrightarrow} l^{\alpha^{post}} \overset{\alpha^{post}}{\hookrightarrow} l'$$

- After instrumentation, the invocations of the passive actions of PG can be observed by CPG via the synchronization of PG and CPG on passive actions, e.g., function calls.

Instrumenting Controlled Programs

Formally,

Definition (Instrumented Program Graphs)

The instrumented program graph of PG is the program graph $IPG = (Loc', Act', Eff', Tr', Loc_0, g_0)$ over Var, where

- $Loc' = Loc \cup Loc^{pre} \cup Loc^{post}$, where $Loc^{pre} = \{l^{\alpha^{pre}} \mid l \overset{g:\alpha}{\hookrightarrow} l' \in Tr \wedge \alpha^{pre} \in \widehat{Act}^{pre}\}$ and $Loc^{post} = \{l^{\alpha^{post}} \mid l \overset{g:\alpha}{\hookrightarrow} l' \in Tr \wedge \alpha^{post} \in \widehat{Act}^{post}\}$
- $Act' = Act \cup \widehat{Act}^{pre} \cup \widehat{Act}^{post}$
- $Eff' = \{Eff'(\alpha, \eta) = \eta' \mid Eff(\alpha, \eta) = \eta'\}$ $\cup \{Eff'(\alpha^{pre}, \eta) = \eta \mid \alpha^{pre} \in \widehat{Act}^{pre}\}$ $\cup \{Eff'(\alpha^{post}, \eta) = \eta \mid \alpha^{post} \in \widehat{Act}^{post}\}$

Instrumenting Controlled Programs

Definition (cont'd)

- $$\begin{aligned} Tr' = \{ & l \overset{g:\alpha}{\hookrightarrow} l' \\ & \mid l \overset{g:\alpha}{\hookrightarrow} l' \in Tr \wedge \alpha^{pre} \notin \widehat{Act}^{pre} \wedge \alpha^{post} \notin \widehat{Act}^{post} \} \\ \cup \{ & l \overset{g:\alpha}{\hookrightarrow} l^{\alpha^{post}} \overset{\alpha^{post}}{\hookrightarrow} l' \\ & \mid l \overset{g:\alpha}{\hookrightarrow} l' \in Tr \wedge \alpha^{pre} \notin \widehat{Act}^{pre} \wedge \alpha^{post} \in \widehat{Act}^{post} \} \\ \cup \{ & l \overset{g:\alpha^{pre}}{\hookrightarrow} l^{\alpha^{pre}} \overset{\alpha}{\hookrightarrow} l' \\ & \mid l \overset{g:\alpha}{\hookrightarrow} l' \in Tr \wedge \alpha^{pre} \in \widehat{Act}^{pre} \wedge \alpha^{post} \notin \widehat{Act}^{post} \} \\ \cup \{ & l \overset{g:\alpha^{pre}}{\hookrightarrow} l^{\alpha^{pre}} \overset{\alpha}{\hookrightarrow} l^{\alpha^{post}} \overset{\alpha^{post}}{\hookrightarrow} l' \\ & \mid l \overset{g:\alpha}{\hookrightarrow} l' \in Tr \wedge \alpha^{pre} \in \widehat{Act}^{pre} \wedge \alpha^{post} \in \widehat{Act}^{post} \} \end{aligned}$$

Semantics of Runtime Control

PG ⇒ instrumented PG (IPG)

IPG + controlling PG ⇒ TS

Semantics of Runtime Control

Definition

The transition system $TS(PG \lhd CPG)$ of program graph PG controlled by a controlling program graph CPG, is the tuple $(S, Act \cup \widehat{Act}, \delta, I, AP, L)$ where

- $S = (Loc \cup Loc^{pre} \cup Loc^{post}) \times Eval(Var) \times \widehat{Loc} \times Eval(\widehat{Var})$, where $Loc^{pre} = \{l^{\alpha^{pre}} \mid l \overset{g:\alpha}{\hookrightarrow} l' \in Tr \wedge \alpha^{pre} \in \widehat{Act}^{pre}\}$ and $Loc^{post} = \{l^{\alpha^{post}} \mid l \overset{g:\alpha}{\hookrightarrow} l' \in Tr \wedge \alpha^{post} \in \widehat{Act}^{post}\}$
- $\delta \subseteq S \times (Act \cup \widehat{Act}) \times S$ is defined by the rules in the next two slides
- $I = \{\langle l, \eta, \widehat{l}, \widehat{\eta} \rangle \mid l \in Loc_0, \eta \models g_0, \widehat{l} \in \widehat{Loc}_0, \widehat{\eta} \models \widehat{g}_0\}$
- $AP = Loc \cup Cond(Var) \cup Cond(\widehat{Var})$
- $L(\langle l, \eta, \widehat{l}, \widehat{\eta} \rangle) = \{l\} \cup \{g \in Cond(Var) \mid \eta \models g\} \cup \{f \in Cond(\widehat{Var}) \mid \widehat{\eta} \models f\}$.

The Transition Rules

slicing rule

$$\frac{l \overset{g:\alpha}{\hookrightarrow} l' \wedge \eta \models g \qquad \widehat{l} \in \widehat{Loc}^{pas} \wedge \alpha^{pre} \notin \widehat{Act}^{pre} \wedge \alpha^{post} \notin \widehat{Act}^{post}}{\langle l, \eta, \widehat{l}, \widehat{\eta} \rangle \xrightarrow{\alpha} \langle l', Eff(\alpha, \eta), \widehat{l}, \widehat{\eta} \rangle}$$

pre-action rule

$$\frac{l \overset{g:\alpha}{\hookrightarrow} l' \wedge \eta \models g \qquad \widehat{l} \in \widehat{Loc}^{pas} \wedge \alpha^{pre} \in \widehat{Act}^{pre} \wedge \widehat{l} \overset{\alpha^{pre}}{\hookrightarrow} \widehat{l'}}{\langle l, \eta, \widehat{l}, \widehat{\eta} \rangle \xrightarrow{\alpha^{pre}} \langle l^{\alpha^{pre}}, \eta, \widehat{l'}, \widehat{\eta} \rangle}$$

transition rules

$$\frac{l \overset{g:\alpha}{\hookrightarrow} l' \wedge \eta \models g \qquad \widehat{l} \in \widehat{Loc}^{pas} \wedge \alpha^{pre} \notin \widehat{Act}^{pre} \wedge \alpha^{post} \in \widehat{Act}^{post}}{\langle l, \eta, \widehat{l}, \widehat{\eta} \rangle \xrightarrow{\alpha} \langle l^{\alpha^{post}}, Eff(\alpha, \eta), \widehat{l}, \widehat{\eta} \rangle}$$

$$\frac{l \overset{g:\alpha}{\hookrightarrow} l' \qquad \widehat{l} \in \widehat{Loc}^{pas} \wedge \alpha^{pre} \in \widehat{Act}^{pre} \wedge \alpha^{post} \in \widehat{Act}^{post}}{\langle l^{\alpha^{pre}}, \eta, \widehat{l}, \widehat{\eta} \rangle \xrightarrow{\alpha} \langle l^{\alpha^{post}}, Eff(\alpha, \eta), \widehat{l}, \widehat{\eta} \rangle}$$

The Transition Rules (cont'd)

transition rules (cont'd)

$$\frac{l \overset{g:\alpha}{\hookrightarrow} l' \qquad \widehat{l} \in \widehat{Loc}^{pas} \wedge \alpha^{pre} \in \widehat{Act}^{pre} \wedge \alpha^{post} \notin \widehat{Act}^{post}}{\langle l^{\alpha^{pre}}, \eta, \widehat{l}, \widehat{\eta} \rangle \xrightarrow{\alpha} \langle l', Eff(\alpha, \eta), \widehat{l}, \widehat{\eta} \rangle}$$

post-action rule

$$\frac{l \overset{g:\alpha}{\hookrightarrow} l' \qquad \widehat{l} \in \widehat{Loc}^{pas} \wedge \alpha^{post} \in \widehat{Act}^{post} \wedge \widehat{l} \overset{\alpha^{post}}{\hookrightarrow} \widehat{l'}}{\langle l^{\alpha^{post}}, \eta, \widehat{l}, \widehat{\eta} \rangle \xrightarrow{\alpha^{post}} \langle l', \eta, \widehat{l'}, \widehat{\eta} \rangle}$$

active-action rule

$$\frac{\top \qquad \widehat{l} \in \widehat{Loc}^{act} \wedge \widehat{l} \overset{\widehat{g}:\beta}{\hookrightarrow} \widehat{l'} \wedge \widehat{\eta} \models \widehat{g}}{\langle l, \eta, \widehat{l}, \widehat{\eta} \rangle \xrightarrow{\beta} \langle \widehat{Eff}(\beta, l), \widehat{Eff}(\beta, \eta), \widehat{l'}, \widehat{Eff}(\beta, \widehat{\eta}) \rangle}$$

Outline

Synthesis of Controlling Programs

- In the previous part, we assumed that the controlling program already exists. But where it comes?

Synthesis of Controlling Programs

- In the previous part, we assumed that the controlling program already exists. But where it comes?
- Directly and manually writing controlling programs is time-consuming and error-prone.

Synthesis of Controlling Programs

- In the previous part, we assumed that the controlling program already exists. But where it comes?
- Directly and manually writing controlling programs is time-consuming and error-prone.
- Instead, we can write a specification for a controlling program using a high level description. Then the controlling program can be automatically synthesized from the specification.

Synthesis of Controlling Programs

- In the previous part, we assumed that the controlling program already exists. But where it comes?
- Directly and manually writing controlling programs is time-consuming and error-prone.
- Instead, we can write a specification for a controlling program using a high level description. Then the controlling program can be automatically synthesized from the specification.
- We will propose the semantics for the specification and synthesis of controlling programs.

$$\text{Specification} \overset{\textit{synthesize}}{\Longrightarrow} \text{CPG} \overset{\textit{generate}}{\Longrightarrow} \text{Controlling Program}$$

Specifications

- A high level specification of controlling programs should consist of variables, passive actions, active actions and a property.

Specifications

- A high level specification of controlling programs should consist of variables, passive actions, active actions and a property.
- A property over passive actions is written in some formalism such as regular expressions, finite automata and LTL formulae.

Definition (Deterministic Finite Automata)

A *deterministic finite automaton* (DFA) is a tuple $A = (Q, \Sigma, \delta, q_0, C, \mathcal{C})$, where Q is a finite set of *states*, Σ is a finite set of *actions*, δ is a *transition function* mapping $Q \times \Sigma \mapsto Q$, $q_0 \in Q$ is the *initial state*, C is a finite set of *categories*, e.g., match and violation, and $\mathcal{C} : Q \times C$ is a classification relation.

Specifications

Formally,

Definition (Specifications)

A specification is a tuple $Spec = (\widehat{Var}, \widehat{g}_0, \widehat{Act}^{pas}, \widehat{Act}^{act}, A, R)$, where
- $\widehat{Var}$ is a set of variables,
- $\widehat{g}_0 \in Cond(\widehat{Var})$ is the initial condition,
- $\widehat{Act}^{pas}$ is a set of passive actions,
- $\widehat{Act}^{act}$ is a set of active actions,
- A is a DFA including a set of categories $A.C$, and
- R is an *association partial function* $(\widehat{Act}^{pas} \cup A.C) \rightharpoonup \widehat{Act}^{act}$.

Synthesis of Controlling Programs

$$\text{Specification} \overset{\textit{synthesize}}{\Longrightarrow} \text{CPG}$$

For DFA, we add some active locations to the finite automaton, at which active actions are executed.

- If a passive action α is associated with an active action $R(\alpha)$, then:
  $q \overset{\alpha}{\hookrightarrow} q' \Rightarrow q \overset{\alpha}{\hookrightarrow} q^{\alpha} \overset{R(\alpha)}{\hookrightarrow} q'$
- If q' is in the category c which is associated with an active action $R(c)$, then:
  $q \overset{\alpha}{\hookrightarrow} q' \Rightarrow q \overset{\alpha}{\hookrightarrow} q'^{c} \overset{R(c)}{\hookrightarrow} q'$
- If q' is in the categories $c_1, \ldots, c_n$ which are associated with active actions $R(c_1), \ldots, R(c_n)$ respectively, then:
  $q \overset{\alpha}{\hookrightarrow} q' \Rightarrow q \overset{\alpha}{\hookrightarrow} q'^{c_1} \overset{R(c_1)}{\hookrightarrow} q'^{c_2} \overset{R(c_2)}{\hookrightarrow} \ldots \overset{R(c_n)}{\hookrightarrow} q'$

Synthesis of Controlling Programs

Formally,

Definition (Synthesized Controlling Program Graph)

Let $Spec = (\widehat{Var}, \widehat{g}_0, \widehat{Act}^{pas}, \widehat{Act}^{act}, A, R)$ be a specification where $A = (Q, \widehat{Act}^{pas}, \delta, q_0, C, \mathcal{C})$ be a DFA. A controlling program graph CPG can be synthesized from the specification as a tuple $(\widehat{Loc}, \widehat{Act}, \widehat{Eff}, \widehat{Tr}, \widehat{Loc}_0, \widehat{g}_0)$ where

- $\widehat{Loc} = \widehat{Loc}^{pas} \cup \widehat{Loc}^{act}$, where $\widehat{Loc}^{pas} = Q$ and $\widehat{Loc}^{act} = \{q^{\alpha} \mid q \in Q, \alpha \in \widehat{Act}^{pas}$ and $R(\alpha)$ is defined$\} \cup \{q^{c} \mid q \in Q, c \in \mathcal{C}(q)$ and $R(c)$ is defined$\}$,
- $\widehat{Act} = \widehat{Act}^{pas} \cup \widehat{Act}^{act}$,
- $\widehat{Eff}$ is the effect function, which is defined by the host programming language,

Synthesis of Controlling Programs

Definition (cont'd)

- $\widehat{Tr}$ is defined as follows: for each transition $q \xrightarrow{\alpha} q' \in \delta$,
    - if $R(\alpha)$ is undefined and $R(\mathcal{C}(q'))$ is undefined, then $q \overset{\alpha}{\hookrightarrow} q' \in \widehat{Tr}$.
    - if $R(\alpha)$ is defined and $R(\mathcal{C}(q'))$ is undefined, then $q \overset{\alpha}{\hookrightarrow} q^{\alpha} \overset{R(\alpha)}{\hookrightarrow} q' \in \widehat{Tr}$.
    - if $R(\alpha)$ is undefined and $R(\mathcal{C}(q'))$ is defined, then $q \overset{\alpha}{\hookrightarrow} q'^{c_1} \overset{R(c_1)}{\hookrightarrow} q'^{c_2} \overset{R(c_2)}{\hookrightarrow} \cdots \overset{R(c_n)}{\hookrightarrow} q' \in \widehat{Tr}$ where $c_1, ..., c_n \in \mathcal{C}(q')$ and $R(c_1), ..., R(c_n)$ are defined.
    - if $R(\alpha)$ is defined and $R(\mathcal{C}(q'))$ is defined, then $q \overset{\alpha}{\hookrightarrow} q^{\alpha} \overset{R(\alpha)}{\hookrightarrow} q'^{c_1} \overset{R(c_1)}{\hookrightarrow} q'^{c_2} \overset{R(c_2)}{\hookrightarrow} \cdots \overset{R(c_n)}{\hookrightarrow} q' \in \widehat{Tr}$ where $c_1, ..., c_n \in \mathcal{C}(q')$ and $R(c_1), ..., R(c_n)$ are defined.
- $\widehat{Loc}_0 = \{q_0\}$ is a set of initial locations.

Outline

Strong Expressiveness

Typical existing formalisms for monitoring can be translated into equivalent controlling programs, e.g.,

- enforcement monitors
- security automata
- edit automata

Outline

Conclusion

Theoretical Contributions:

- Our theory provides a complete formal semantics for real implementations of runtime monitoring and control.
- Our theory retains a better balance between implementation and generality than existing formalisms.
- Many existing formalisms about runtime monitoring can be considered as special cases of our theory.

Conclusion

Theoretical Contributions:

- Our theory provides a complete formal semantics for real implementations of runtime monitoring and control.
- Our theory retains a better balance between implementation and generality than existing formalisms.
- Many existing formalisms about runtime monitoring can be considered as special cases of our theory.

Applications:

- The semantics helps to accurately understand the principle of our tool.
- The semantics can be used for model checking the correctness of target programs under control, i.e., checking whether a controlling program can really make a target program satisfy desired requirements at runtime.

THE END

Thank you!

Questions?