Finite State Transducers Framework for Monitors Conflict Detection and Resolution

Soha Hussein

University of Illinois at Urbana-Champaign
soha@illinois.edu

Abstract. Runtime monitoring and verification systems monitor target's events and verify them against specifications during program execution. For such systems the same event might trigger different monitors remedial actions, which can be contradictory in behavior or complementary (with a specific order). This urges the need to have a method to detect and resolve potential conflict between monitors.

In this paper, we present a formal model for modeling monitors based on *Finite State Transducers*. Monitors in the model are transducers with events as their input and output alphabet. Monitors composition is used for those monitors in conflict, where each monitor can add to the output set of events, but it can never remove an event. The output set of events is later evaluated using 2 rewrite rules and resulting in non-conflicting behavior.

1 Introduction

Runtime monitoring and verification has become a widely used methodology for testing and verification of systems. The idea is to encode system's specifications into monitors and observe them during system's operation, not only guaranteeing that the specifications of interests are monitored but issuing a remedial action upon violation that might as well resolve/prevent the problem before actually occurring.

JavaMOP [15,4] is a generic runtime and monitoring verification systems, whose output is an AspectJ file that can later be instrumented on the target program.

JavaMOP, by default, allows multiple specifications to coexist within a given target program. However it makes no guarantees on how they will operate together if they are triggered by the same monitored event. By default, which specification will be triggered first when events interfere is decidable only by the order in which AspectJ [10] files are weaved into the program or by using an aspect precedence declaration, which is part of the AspectJ standard. These are, at best, an incomplete way to allow for policy composition. To ensure proper composition, some sort of coordination and management among different specifications is necessary, in order to allow precedence as well as handlers conflict resolution.

The work in this paper was motivated by resolving conflicts among JavaMOP specifications, although it is not restricted to it. We provide a model that

N. Meghanathan et al. (Eds.): CCSIT 2012, Part III, LNICST 86, pp. 112–123, 2012.

models monitors as finite state transducers, we define what does it mean to have two monitors in conflict and we provide a methodology using the model to resolve potential conflicts and/or provide little management of monitors over other monitors.

In our model, a monitor receives a set of events and output a set of events. There are no restrictions on the input set of event, but for the set of output events a monitor can output either the same received set of events *or* append either {skip, proceed} events (these are two especial types of events see section 3.1) with possibly some statements to execute. Thus the output set of events keeps growing by visiting each monitor in conflict, or for which coordination is required.

The monitored program (usually we refer to it as just target) is modeled as a special finite state transducer that propagates its events to other transducers in the network. For each target step, an event the network of transducers operate resulting in a set of output events which is later evaluated, using 2 rewrite rules, into an action to be executed by the program.

The paper is organized as follows: the rest of this sections discusses some related work and some properties about conflicting specifications, then in section 2 general definitions used in the paper for finite state transducers are given. Section 3 represents the model, and finally conclusion and future work are in Sections 4.

1.1 Related Work

Policies composition with detecting and resolving potential conflicts are widely addressed in security language specification languages, however they have not received considerable attention in prior research in runtime monitoring systems or execution monitoring systems. For instance, JavaMOP [14] and Tracematches [1] are two runtime monitoring systems that do not specify means to detect and resolve potential conflicts when multiple specifications co-exist together right at the same time, thus causing non-determinism in the final output.

On the other hand, execution monitor systems such as SpoX [8,9], Naccio [7,6] and PoET [5] does not define a means to detect or resolve conflicts between potential specifications, whitelist Polymer [3,2,12,11,13], which defines a number of *Policy Combinators* to enable different compositions of policies. However, there is no formal definition for such combinators.

1.2 Properties about Monitors Conflicts Resolutions

Monitors Conflicts Definition: We say that two monitors are in conflict if they monitor the same event yet each fires a different remedial action.

Types of Conflict: One can define two types of conflicts among monitors:

- *Remedial Action Conflict:* for instance for a file access event that is observed by two monitors; one of the monitors might allow it with a warning while the other prohibit the access. This is a flaw in the design of the policies, and one needs to resolve such conflicts by allowing a single remedial action to take place.

– *Remedial Action Execution Order:* this happens when there is a dependency between the remedial action of both specifications. For example if one specifications writes to a file a certain value while the other reads the old value and outputs a warning message. In that case executing both remedial actions is what one wants but it is the order of execution that should be coordinated.

Typical Procedure: A typical procedure to address such conflicts is usually done in two steps:

1. *Detect Potential Conflicts:* this is the first step that should take place, one needs to detect a conflict. It is better off automating this part, so as to ensure that there are no conflict in the pool of policies that are going to co-exist in the target.
2. *Resolve Potential Conflicts:* once a conflict is detected, resolving it is the next step. There are cases resolving of conflicts can be automated but not always. For instance, the remedial action of a a higher priority can be taken instead of the lower conflicting remedial action (usually the lower bound of the remedial action lattice defined earlier), like if one says Halt, and the other prints out the warning, if one wants to be conservative then Halt should be the dominated remedial action. However if both have the same priority then a different procedure of conflict resolution should be taken, like a precise specification language that manages coordination between specifications based on the user's definition.

2 Finite State Transducers

Finite State transducers are widely used machines in natural language processing. They are also a good candidate to represent monitors, the reason being, one can think of monitors as that machine that, upon receiving an event, moves from one state to another while possibly outputting another event. It thus abstracts what a monitor is doing as a series of transitions on states when receiving events and outputting other events.

In this section we first start by listing some definitions that we are going to use in the model then we will represent out model and show how it can be used to detect potential conflicts and then resolve them.

2.1 Definition of Finite State Transducers (FST)

Finite State Transducers FST [16] can be seen as a Finite State Automata, in which each transition is labeled by a pair of symbols rather than by a single symbol. A Finite State Transducer FST is a 6-tuple $(\Sigma_1, \Sigma_2, Q, i, F, E)$ such that:

– Σ_1: is a finite input alphabet.
– Σ_2: is a finite output alphabet.
– Q is the set of states.

- $i \in Q$ is the initial state.
- $F \subset Q$ is the set of final states.
- $E : Q \times \Sigma_1^* \times \Sigma_2^* \times Q$, is the set of edges.

An alternative definition consists in replacing the set of edges E by the transition function d a mapping from $Q \times \Sigma_1^*$ to 2^Q and an emission function δ a mapping from $Q \times \Sigma_1^* \times Q$ to $2^{\Sigma_2^*}$.

2.2 Sequential Transducers

A *sequential transducer*, is a six-tuple$(\Sigma_1, \Sigma_2, Q, i, \otimes, *)$ such that,

- Σ_1 and Σ_2 are two finite alphabets.
- Q is a finite set of states.
- $i \in Q$ is the initial states.
- $\otimes$ is the partial deterministic transition function mapping $Q \times \Sigma_1$ on Q noted $q \otimes a = q'$.
- $*$ is the partial emission function mapping $Q \times \Sigma_1$ on Σ_2^*, noted $q * a = w$.

Sequential transducers can be seen as a subclass of finite state transducers without final states and with deterministic transition function.

2.3 More about FST

Finite State Transducers are powerful because of the various closure and algorithmic properties. We start by defining *letter transducers* then we represent three closure properties for FST.

Extended Edges: The extended set of edges $\hat{E}$, is the least subset of $Q \times \Sigma_1^* \times \Sigma_2^* \times Q$ such that:

- $\forall q \in Q, (q, \epsilon, \epsilon, q) \in \hat{E}$
- $\forall w_1 \in \Sigma_1^*, \forall w_2 \in \Sigma_2^*$ if $(q_1, w_2, w_2, q) \in \hat{E}$ and $(q_2, a, b, q_3) \in E$ then $(q_1, w_1 a, w_2 b, q_3) \in \hat{E}$.

Transducer Language: The above definition of extended edges us allows us to define a relation $L(T)$ on $\Sigma_1^* \times \Sigma_2^*$ for FST T as:

$$L(T) = \{(w_1, w_2) \in \Sigma_1^* \times \Sigma_2^* \mid \exists (i, w_1, w_2, q) \in \hat{E}\} \text{ with } q \in F$$

$$|T|(u) = \{v \in \Sigma_2^* \mid (u, v) \in L(T)\}$$

Letter Transducers: If $T_1 = (\Sigma_1, \Sigma_2, Q, i.F, E_1)$ is a transducer such that $\epsilon \notin |T_1|(\epsilon)$ then there is a letter transducer $T_2 = (\Sigma_1, \Sigma_2, Q_2, i_2, F_2, E_2)$ such that:

- $| T_1 | = | T_2 |$
- $E_2 \subseteq (Q_1 \times (\Sigma_1 \cup \{\epsilon\}) \times (\Sigma_2 \cup \{\epsilon\}) \times Q_2)$
- $E_2 \cap (Q_1 \times \{\epsilon\} \times \{\epsilon\} \times Q_2) = \phi$

2.4 Closure Properties of FST

Closure under Union: if T_1 and T_2 are two FST, there exists and FST $T_1 \cup T_2$ such that $|T_1 \cup T_2| = |T_1| \cup |T_2|$, i.e., s.t. $\forall u \in \Sigma^*, |T_1 \cup T_2|(u) = |T_1|(u) \cup |T_2|(u)$.

Closure under Composition: Given two letter transducers FSTs $T_1 = (\Sigma_1, \Sigma_2, Q, i, F, E_1)$ and $T_2 = (\Sigma_2, \Sigma_3, Q_2, i_2, F_2, E_2)$, there exists an FST $T_2 \odot T_1$ such that for each $u \in \Sigma_1^*$, $\mid T_2 \odot T_1 \mid (u) = |T_2(|T_1|(u))$. Furthermore the transducer

$T_3 = (\Sigma_1, \Sigma_3, Q_1 \times Q_2, (i_1, i_2), F_1 \times F_2, E_3)$ such that

$$E_3 = ((x_1, x_2), a, b, (y_1, y_2)) | \exists c \in \Sigma_2 \cup \{\epsilon\} \text{ s.t. } (x_1, a, c, y_1) \in E_1, (x_2, c, b, y_2) \in E_2$$

satisfies

$$\mid T_3 \mid = \mid T_2 \odot T_1 \mid (u) = \mid T_2(\mid T_1 \mid (u)), \forall u \in \Sigma_1^*$$

3 Modeling of Monitors as FST

3.1 Events Definition

Let ξ be the set of all possible events that can be exhibited by the program or propagated by the transducers. Each member in ξ is defined by $e_{name} < a >$ (we sometimes refer to it as just e_{name}), where $a \in A$ such that A is a set of all system actions and where a is defined by the pair (r, s) such that r is the range or scope of interest, i.e., before or after, and s is the statement(s) enclosed inside the action.

For example, if we want to have a monitor event that monitors just before the creation of files then the monitored event will be in the form of $createFile < before, s >$, where *createFile* is the monitoring event for the method of file creation, s is the enclosed statement(s) inside the file creation method, and r is the range of the event that can take one of three values: *"before"* (marking the position before the execution of the action), *"after"* (marking the position after execution of the action) and *"."* (this is only used for *result events R*, see Section 3.1, since their intended meaning does not carry position on their own instead they depends on the action on which they refer to (i.e., follow).)

Events Types: In this model we distinguish between two types of events that can exist mainly:

- **Specification Events:** These are the events that are the monitors want to observe in the target, like the *createFile* event example above. We refer to this type of events as E_{spec} such that $E_{spec} \subset \xi$.
- **Result Events:** These are exactly two events, that each monitor output to indicate its recommended remedial action for the seen events. Precisely we define the set of result events $R \subset \xi$ as follows:

$$R = \{proceed < (., s_{\rho_m}) >, skip < (., s_{\rho_m}) >\} \quad (1)$$

with the meaning to "*proceed*" or "*skip*" the execution of the previous events while executing statements "s" in the environment ρ_m of their monitor "m".

Events Matching: We define matching of events and matching of sets of events as follows:

- Two events $e < (r, s) >$ and $e' < (r', s') >$ match *iff* $e = e'$.
- Two ordered sets of event τ_a and τ_b, s.t $\tau_a, \tau_b \in \xi^*$, match *iff* $e_{a_i} = e_{b_i}$, where i is the index of the event in each set.

3.2 Monitors Modeling Architecture

The model is composed of two main layers of transducers, the first is a simple transducer (we call it TM_{pgm}) that models the targets' events and is responsible for propagating them to the next layer of transducers (we call the TM_{net}) which in turn is responsible for enforcing different specifications.

We define a composition function between the two layers of monitor transducers as:

$$comp = TM_{pgm}; TM_{net}; TM_{pgm} \tag{2}$$

The above function specifies how the composition between monitor transducers should take place, such that the output of the TM_{pgm} is read by TM_{net} and the output of the TM_{net} is read by the TM_{pgm}. The TM_{pgm} should only react to the output of final TM in the TM_{net} transducers, thus it should not be able to see any internal events that are used for internal communication between monitors.

We next define the model of TM_{pgm} and TM_{net}.

3.3 Modeling of a Program Monitor TM_{pgm}

We define the program monitor TM_{pgm} $(\Sigma_1, \Sigma_2, Q, i, F, \otimes, *)$ as a letter transducer as follows:

- Q: three states, *Read*, *Wait* and *End*.
- Σ_1: is the set of input events $\{e_{in} : e_{in} \in \xi^*\}$. That is $\Sigma_1 \subseteq \{R \cup E_{spec}\}$.
- Σ_2: is the set of output events $\{e_{out} : e_{out} \in \{\xi \cup \epsilon\}^*\}$. That is $\Sigma_2 \subseteq \{R \cup E_{spec} \cup \epsilon\}$
- i: *Read*
- F: *End*
- $\otimes$ is a deterministic transition function for letter transducers from $Q \times \Sigma_1^*$ to $Q : q \otimes \tau = q'$.
- $*$ is the partial emission function for letter transducers mapping from $Q \times \Sigma_1^*$ to $\Sigma_2 : q * \tau = \tau'$.

Figure 1 shows the Transducer of the program monitor TM_{pgm}, where we write the notation x/y to refer to the input and output events respectively.

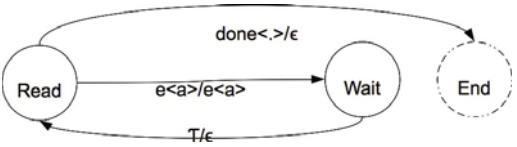

Fig. 1. TM_{pgm}

Semantics for TM_{pgm}:

– *Reading Program Events-Step:*

$$\text{if } q = \text{"Read", then} \quad q \otimes e < a >= \text{"Wait"} \ , \ q * e < a >= e < a > \tag{3}$$

– *Reading TM_{pgm} Monitor Result-Step:*

$$\text{if } q = \text{"Wait" and } \tau' \in \Sigma_1^*, \text{ then} \quad q \otimes \tau = \text{"Read"} \ , \ q * \tau = \epsilon \tag{4}$$

– *Stop-Step:*

$$\text{if } q = \text{"Read" } \& \ e < a >= done < . >, \text{then} \quad q \otimes e < a >= \text{"End"}, q * e < a >= \epsilon \tag{5}$$

3.4 Modeling Monitor Transducers in TM_{net} Monitor

Construction of TM monitors (Transducer Monitor) in TM_{net} is simple, it follows the definition of finite state transducer monitors in Section 2.1.

A **TM Monitor** is defined as a variation of a finite state transducer TM, as the tuple $(\Sigma_1, \Sigma_2, Q, 1, \otimes, *)$ such that $\epsilon \notin |TM|(\epsilon)$ and:

– Σ_1: is the the set of input events such that $\Sigma_1 \subseteq \{\xi \cup \epsilon\}$, such that $\Sigma_1 = \{e_{in} : e_{in} \in \{\xi \ \cup \epsilon\}\}$.
– Σ_2: contains the same alphabet of Σ_1, that is, it is the set of output events, such that $\Sigma_2 \subseteq \xi$, such that $\Sigma_2 = \{e_{out} : e_{out} \in \xi\}$.
– Q is the set of states.
– 1 is the initial state.
– $\otimes$ is a deterministic transition function from $Q \times \Sigma_1^*$ to $Q : q \otimes \tau = q'$, where q and q' are the current and next states of the monitor and $\tau \in \Sigma_1^*$ is the input set of events.
– $*$ is the partial emission function mapping from $Q \times \Sigma_1^*$ to $\Sigma_2^* : q * \tau = \tau'$, where $\tau \in \Sigma_1^*$ and $\tau' \in \Sigma_2^*$ are the set of input and output events respectively.

Informally speaking, each monitor is defined in our model as a variation of finite state transducer which upon receiving a set of events it moves from one state to another (could be the same state) while outputting either the same set of events

(thus propagating them)or else appending an element of R to the received set of events, the following specify the restrictions on the emission function:

$$q * \tau = \tau, \text{ where } \tau \in \Sigma_2^* \tag{6}$$

OR

$$q * \tau = \tau; e < a >, \text{ where } e < a > \in R \tag{7}$$

3.5 Identifying Potential Conflicts

Informally speaking, conflicts among monitors happens when the same program event invokes more than one monitor with possibly different remedial actions. Thus conflicts between TM monitors, before composition (i.e., the design of the TM monitor instrumented as a single specification on the program), can simply be identified by using intersection of the input alphabet (excluding R) of the n existing transducers in the system: $\forall_i \bigcap \{\Sigma_{1i} - R\} \neq \emptyset$ for $i = 1..n$.

Or in other words we define conflict between two monitor transducers $TM_x = (\Sigma_{x1}, \Sigma_{x2}, Q_x, 1, Q_x, E_x)$ and $TM_y(\Sigma_{y1}, \Sigma_{y2}, Q_y, 1, Q_y, E_y)$ as:

$$\exists TM_{x_i} = (q_{x_i}, e < a >, e' < a' >, q'_{x_i}) \text{ and } \exists TM_{y_j} = (q_{y_j}, e < a >, e'' < a' >, q'_{y_j}) \tag{8}$$

where $e < a > \notin R$ and $e < a > \neq \epsilon$

3.6 Conflict Resolution with TM Composition

For those monitor transducers that have conflicts as show in Section 3.5, one can distinguish between three types of scenarios to resolve conflicts:

- Allow only a single remedial action.
- Allow both remedial action but in a precise order.
- Allow one monitor to delay its remedial action and base it on the remedial action taken place by the other monitor.

Our model of TM monitors support all the above scenarios, which scenario to be used however should be provided by a monitor specification language that would express which way to go.

Our solution to resolve conflicts among TM monitors is by composing them. The idea behind composition of two or more transducers is that each TM monitor transducer actually gives a distinct output for events that it cares about, while acting as identity function on all other inputs, i.e., propagating other irrelevant events to the next transducer in the chain of conflicting TM monitors. This provides a natural way of composition that does not change almost anything in the original transducers since the identity transition can be easily added as follows:

$$q \otimes \tau = q \quad and \quad q * \tau = \tau, \text{ where } \tau \text{ is a set of input events} \tag{9}$$

Transducer Composition: Since our TM monitors are such transducer where $\epsilon \notin |TM|(\epsilon)$, a *letter* conflicting TM monitors TM_x and TM_y can be resolved by finding their composed letter TM monitor. Thus, if

$$TM_x = (\Sigma_{x1}, \Sigma_{x2}, Q_x, 1, Q_x, E_x) \text{ and } TM_y = (\Sigma_{y1}, \Sigma_{y2}, Q_y, 1, Q_y, E_y)$$

Then there exists an $TM_{comp} = TM_y \odot TM_x$ such that for each $\tau \in \Sigma_1^*$,

$$|TM_y \odot TM_x|(\tau) = |TM_y|(|TM_x|(\tau))$$

3.7 Modeling Monitors Not in Conflict

Monitors that do not lay in conflict with others do not need to be in composition, since they potentially have different interests in event. Thus an equivalent union TM can represent those with disjoint interests, even though their composition to the rest of the network is also correct and thus can always be an option.

3.8 Other Composition Usages

As mentioned in Section 3.6 allowing composition of TM monitors can be used for more than just for resolving conflicts. TM monitors up in the chain of composition can be used to alter received events from other transducers (instead of acting as identity function) and thus changing the resulting output for the set of transducers. This change requires user's intervention and it is based on user's definition and the privileges each TM monitor is granted.

With such allowances, the output for the set of transducers will require a different interpretation, since it is going to be the decomposed output for each transition. We now define how to interpret the resulting output for a series of TM monitors.

Interpreting Transducers Results: The output of a composed set of TM monitors is a set of events that keeps growing (element in ξ^*). Suppose we have composition between n TM monitors in the form $TM_1; TM_2; ...; TM_n$, and if $e < a >$ is a the current monitored event generated by the program, and if q_1 to q_n represent the current state for $TM_1, ..TM_n$ respectively then, the generated set of output O of the last TM_n in a composed chain would be:

$$O = \{q_{pgm} * e < a >, q_1 * (q_{pgm} * e < a >), q_2 * (q_1 * (q_{pgm} * e < a >))...\} =$$

$$O = \{e < a >, e' < a' >, e'' < a'' >, ..\}$$

where $\{e < a >, e' < a' >, e'' < a'' >, ..\}$ represents the set of events results from each transition.

The output of the transducers can be evaluated (from left to right) using these two rewrite rules for *skip* and *proceed*. We use the notation $[.., x], [x, ..], [.., x, ..]$

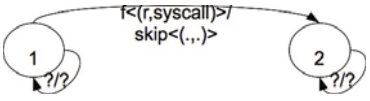

Fig. 2. TM (letter transducer version) for No System Calls Specification

to refer to an open set either from the right side, left side or both sides. Also, in the following rules mi refers to the monitor that generated an event:

$$[...e < (r, S_{\rho_{target}}) >, proceed < (., S_{\rho_{mi}}) >, ...] \xrightarrow{\text{rewrites to}} [e' < (r, S_{\rho_{mi}}; S_{\rho_{target}}) >, ...] \tag{10}$$

$$[...e < (r, S_{\rho_{target}}) >, skip < (., S_{\rho_{mi}}) >, ...] \xrightarrow{\text{rewrites to}} [e < (r, S_{\rho_{mi}}) >, ...] \tag{11}$$

Rule 10 has two versions depending on the scope of the event $e < a >$, the one shown assumes that $r = before$. The rule matches when a *proceed* event is first encountered (since an event can be repeatedly propagated from one TM monitor to another). The rule basically resumes with the action of the event $< r, S_{target} >$ while executing the monitor's statements in the monitor's own environment ρ_{mi}.
Rule 11 also matches when the first *skip* event is encountered. It evaluates to another event that skips the action statement $< S_{target} >$ and executes the statements S in the environment of the monitor ρ_{mi}

Few notes to observe here:

- An *around* scope can be expressed in this model by passing the same $e < a >$ twice, each with a different scope.
- Also, even though skipping of action $e < a >$ that has an after scope, will basically have no effect on the monitored program, since the transducers are skipping an already executed event.
- When a *skip* rule is encountered it actually skips *all* actions of monitors, including skipping the original action of the program. And that actually makes sense since the events and their actions will be wrapped with other *proceed* of *skip*.
- A TM should not have a transition of the form (q, x, y, q') where $y \in R$ unless it has certain privileges on the pervious composed transducer in the chain. This privileges should be defined using the monitoring specification language.

3.9 Example

Shown in figure 2 and figure 3, the TM monitors for composition of the specification of *No System Calls* and *File Network Wall*. We use the notation ? to

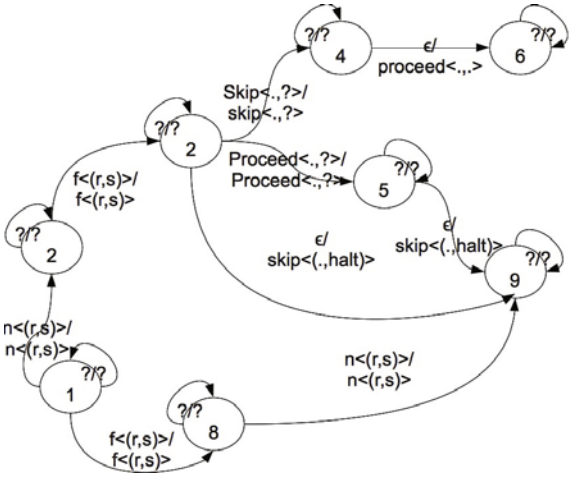

Fig. 3. File Network Wall TM (letter transducer version) after Composition with No System Calls TM

refer to other events (or statements) that are not of concern to the specification, it should generally be taken as *anything*. Also, $f < a >$ is a file access event, $n < a >$ is a network access event and ϵ can be read only when the input events from the pervious TM in the chain are consumed.

The *No System Calls* specification prevents system calls from happening by skipping the event along with its action. while the *File Network Wall* specification halts the program whenever there is a file access after a network access or vise versa.

The two specifications are dependent on each other, since a file access can in fact be a system call invocation, and if the two specifications were composed together in such a way that the *No System Calls* would be checked first, then there would be no violation to the *File Network Wall* specification if the event was already skipped by the previous specification.

4 Conclusion and Future Work

This paper describes a framework for modeling monitors which can be used to resolve conflicts between conflicting monitors. In the model we express monitors as a finite state transducer, where conflicting monitors are composed and other non-conflicting monitors can be grouped with union.

The model gives a little power of control of one monitor on another, by allowing one monitor to change in the output set of events previously received from other monitors.

A future work for the model is to build a specification language on top of the TM framework to define TM monitors and to build the appropriate composition among them.

References

1. Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L.J., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding trace matching with free variables to AspectJ. In: Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2005), pp. 345–364. ACM (2005)
2. Bauer, L., Ligatti, J., Walker, D.: A language and system for enforcing run-time security policies. Tech. Rep. TR-699-04, Princeton University (2004)
3. Bauer, L., Ligatti, J., Walker, D.: Composing security policies with polymer. SIGPLAN Not. 40, 305–314 (2005)
4. Chen, F., Roşu, G.: MOP: An efficient and generic runtime verification framework. In: Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2007), pp. 569–588. ACM (2007)
5. Erlingsson, U., Schneider, F.B.: IRM enforcement of java stack inspection. In: IEEE Symposium on Security and Privacy (SOSP 2000), pp. 246–255. IEEE (2000)
6. Evans, D.: Policy-Directed Code Safety. Ph.D. thesis, MIT (2000)
7. Evans, D., Twyman, A.: Flexible policy-directed code safety. In: IEEE Symposium on Security and Privacy (SOSP 1999), pp. 32–45. IEEE (1999)
8. Hamlen, K.W., Jones, M.: Aspect-oriented in-lined reference monitors. In: Workshop on Programming Languages and Analysis for Security (PLAS 2008), pp. 11–20. ACM (2008)
9. Jones, M., Hamlen, K.W.: Enforcing IRM security policies: two case studies. In: Intelligence and Security Informatics (ISI 2009), pp. 214–216. IEEE (2009)
10. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. In: Lee, S.H. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–353. Springer, Heidelberg (2001)
11. Ligatti, J.A.: Policy Enforcement via Program Monitoring. Ph.D. thesis, Princeton University (2006)
12. Ligatti, J., Ligatti, J., Bauer, L., Walker, D.: Edit automata: Enforcement mechanisms for run-time security policies. Journal of Information Security 4, 2–16 (2003)
13. Lomsak, D., Ligatti, J.: PoliSeer: A tool for managing complex security policies. In: International Federation for Information Processing Conference on Trust Management, IFIP-TM (2010)
14. Meredith, P.O., Jin, D., Griffth, D., Chen, F., Roşu, G.: An overview of monitoring oriented programming. Journal on Software Tools for Technology Transfer (to appear, 2011)
15. Meredith, P.O., Jin, D., Griffth, D., Chen, F., Roşu, G.: An overview of the MOP runtime verification framework. Journal on Software Techniques for Technology Transfer (to appear, 2011)
16. Roche, E., Schabes, Y. (eds.): Finite-State Language Processing. Bradford Book, MIT Press, Cambridge, Massachusetts (1997)