STATL Definition

Steven T. Eckmann   Giovanni Vigna   Richard A. Kemmerer

Reliable Software Group
Department of Computer Science
University of California
Santa Barbara, CA 93106
[eckmann,vigna,kemm]@cs.ucsb.edu

June 5, 2001

Abstract

STATL is an extensible state/transition-based attack description language designed to support intrusion detection. The language allows one to describe computer penetrations as sequences of actions that an attacker performs to compromise a computer system. A STATL description of an attack scenario can be used by an intrusion detection system to analyze a stream of events and detect possible ongoing intrusions. Since intrusion detection is performed in different domains (i.e., the network or the hosts) and in different operating environments (e.g., Linux, Solaris, or Windows NT) it is important to have an extensible language that can be easily tailored to different target environments. STATL defines domain-independent features of attack scenarios and provides constructs for extending the language to describe attacks in particular domains and environments. The STATL language has been successfully used in describing both network-based and host-based attacks, and it has been tailored to very different environments, e.g., Sun Microsystems' Solaris and Microsoft's Windows NT. An implementation of the runtime support for the STATL language has been developed and a toolset of intrusion detection systems based on STATL has been implemented. The toolset was used in a recent intrusion detection evaluation effort, delivering very favorable results. This report presents the details of the STATL syntax and semantics.

Introduction

Intrusion detection systems analyze information about the activities performed in a computer system or network, looking for evidence of malicious behavior. The information may come in the form of audit records produced by the operating system auditing facilities, log messages produced by different types of sensors and network devices, or in the form of raw network traffic obtained by eavesdropping a network segment. These data sources are used by intrusion detection systems (IDSs) in two different ways, according to two different paradigms: *anomaly detection* and *misuse detection*. In anomaly detection systems [JV94, KRL97, GWC98], historical data about a system's activity and specifications of the intended behavior of users and applications are used to build a profile of the "normal" operation of the system. Then, the intrusion detection system tries to identify patterns of activity that deviate from the defined profile. Misuse detection systems take a complementary approach [Rea99, CIS99, Pax98]. Misuse detection tools are equipped with a number of attack descriptions (or "signatures") that are matched against the stream of audit data looking for evidence that the modeled attack is occurring. Misuse and anomaly detection both have advantages and disadvantages. Misuse detection systems can perform focused analysis of the audit data and they usually produce only a few false positives, but they can detect only those attacks that have been modeled. Anomaly detection systems have the advantage of being able to detect previously unknown attacks. This advantage is paid for in terms of the large number of false positives and the difficulty of training a system with respect to a very dynamic environment. A recent DARPA-sponsored evaluation of intrusion detection systems [Lab99] provided interesting feedback on the characteristics of several anomaly and misuse detection systems confirming the advantages and disadvantages inherent in the two approaches.

A critical feature of misuse detection systems is the description of attack signatures. Several *attack languages* have been proposed by the research community. However, most of the existing languages have been developed to support intrusion detection in particular domains and environments. These languages suffer from the idiosyncrasies of their operational settings and are difficult to extend to new environments. In addition, the semantic model and the abstractions provided to the signature developer are either too low-level or not clearly defined.

Our experience with intrusion detection in different domains suggested the design of a domain-independent attack description language that could be extended in a well-defined way to match different operating environments. The result of this research is STATL, which is a language to support misuse detection. STATL defines the domain-independent features of attack scenarios and also provides constructs for extending the language to describe attacks in particular domains and environments. The STATL language has been successfully used in describing both network-based and host-based attacks, and it has been tailored to very different environments, e.g., Sun Microsystems' Solaris and Microsoft's Windows NT. An implementation of the runtime support for the STATL language has also been developed and a toolset of intrusion detection systems based on STATL has been implemented [VEK00]. Even though the language was originally developed to support the development of intrusion detection systems based on the *State Transition Analysis Technique* (STAT) [IKP95], STATL is general and flexible enough to be used as a common language for different misuse detection systems.

This report presents STATL's syntax and semantics. Chapter 1 describes the details of STATL's syntax. Chapter 2 beginning on page 12 describes the details of STATL's semantics. Appendix A contains the complete STATL grammar.

Chapter 1

Syntax

A STATL specification is the description of an attack scenario. The attack is modeled as a sequence of steps that take a system from an initial safe state to a final compromised state. This modeling approach is naturally supported by a state/transition-based language. One of the advantages of this approach is that state/transition specifications can be represented graphically by means of state transition diagrams (STDs). Therefore, even though STATL is primarily a text-based language, the STATL development environment includes a graphical editor that allows one to directly visualize the STD representing an attack scenario.

The STATL language provides constructs to represent an attack as a composition of *states* and *transitions*. States are used to characterize different snapshots of a system during the evolution of an attack. Obviously, it is not feasible to represent the complete state of a system (e.g., volatile memory, file system, etc.); therefore, a STATL scenario uses variables to record just those parts of the system state needed to define an attack signature (e.g., the value of a counter or the ownership of a file).

Transitions are used to represent evolution from one state to another. A transition has an associated *action* that is a specification of the event that may cause the scenario to move to a new state. For example, an action may be the opening of a TCP connection or the execution of an application. The space of possible relevant actions is constrained by a *transition assertion*, which is a filter condition on events that may possibly match the action. For, example an assertion may require that a TCP connection is opened with a specific destination port or that an application being executed should be part of a predefined set of security-critical applications.

It is possible that several occurrences of the same attack are active at the same time. A STATL attack scenario therefore has its semantics defined in terms of a set of *instances*, each of which represents an attack currently in progress.

The evolution of the set of instances of a scenario is determined by the type of transitions in the scenario definition. A transition can be *consuming*, *nonconsuming*, or *unwinding*. A nonconsuming transition is used to represent a step of an occurring attack that does not prevent further occurrences of attacks from spawning from the transition's source state. Therefore, when a nonconsuming transition fires the source state remains valid, and the destination state becomes valid too. For example, if an attack has two steps that are the creation of a link named "`-i`" to a SUID shell script and the execution of the script through the created link, then the second step does not invalidate the previous state. That is, another execution of the script through the same link may occur. Semantically, the firing of a nonconsuming transition causes the creation of a *new* instance. The original instance is still in the original state, while the new instance is in the state that is the destination state of the fired transition. In contrast, the firing of a consuming transition makes the source state of a particular attack occurrence invalid. For example, an action that deletes a file may invalidate the source state. The file cannot be deleted twice and therefore the original state is invalid. Semantically, the firing of a consuming transition does not generate a new scenario instance; it simply changes the state of the original one. Unwinding transitions represent a form of "rollback" and they are used to describe events and conditions that may invalidate the progress of one or more scenario instances and require the return to an earlier state. For example, the deletion of a file may invalidate a condition needed for an attack to complete, and, therefore, the corresponding scenario instances may be brought back to a previous state, for example before the file was created. The details of scenario execution are in Section 2.3.

Type name	Description
`bool`	boolean – `true` or `false`
`char`	C++ `char`
`int`	integer in default architecture size
`int_8`	8-bit int
`int_16`	16-bit int
`int_32`	32-bit int
`int_64`	64-bit int
`long`	C++ `long`
`u_int`	unsigned int in default architecture size
`u_int_8`	8-bit unsigned int
`u_int_16`	16-bit unsigned int
`u_int_32`	32-bit unsigned int
`u_int_64`	64-bit unsigned int
`short`	C++ `short`
`string`	C++ `string`
`timer`	for declaring timers and specifying timer events
`timeval`	`struct { long sec; long msec; }`

Table 1.1: STATL data types.

The STATL language also provides the built-in concept of timers for specifying signatures that involve time intervals and timeouts.

An attack description language should have a rigorously defined syntax and semantics. This chapter presents the details of STATL's syntax and includes fragmentary examples for each of the syntax rules. The next chapter presents STATL's semantics.

In the syntax rules, literal keywords are in **boldface** and other literal text is enclosed in single quotes. Optional items are enclosed in square brackets '[', ']', items that may appear zero or more times are enclosed in curly braces '{', '}'. Alternatives are separated by '|', and grouped with parentheses where necessary to indicate associativity. Examples may include ellipses (...) to indicate that details have been left out; the ellipses are not part of STATL.

1.1 Lexical elements

STATL identifiers consist of letters, digits, and the underscore character '_', and start with a letter. For example `host_name` and `IPaddr2` are identifiers. STATL identifiers are case-sensitive, so `IPaddress` is different from `IPAddress`. STATL compound identifiers use standard object-oriented dot notation, as in "object.attribute". STATL keywords are reserved words and may not be used as identifiers. For example, since `scenario` is a keyword, it may not be used as a variable name.

STATL includes two kinds of comments: any text between "/*" and "*/" (except "*/"), including the delimiters, is a comment. Any text following "//" to the end of the line, including the "//" marker, is a comment. Whitespace may appear anywhere in a STATL specification except within tokens (keywords, identifiers, and multiple-character operators).

1.2 Data types

STATL includes several builtin types: `int` and `u_int` in various sizes, `bool`, `string`, `timeval` (for timestamps), and `timer`. It also includes arrays, plus containers `vector`, `set`, `list`, and `map`. Table 1.1 summarizes STATL's builtin types.

It is not possible to define new data types within a STATL scenario. Application-specific types must be defined within the application-specific extension library. For example, NetSTAT scenarios may use different types than USTAT scenarios, but both use `int` and `timeval`. The extension process is described elsewhere.

1.3 Scenario

A scenario uses zero or more libraries of application-specific types, events, functions, and predicates. A scenario has a name, may have parameters, may contain annotations and constant and variable declarations, and most importantly, contains the states and transitions that define the "attack signature" – what to match and what to do with matches. A scenario may also define supporting functions to be used in state and transition assertions and code blocks:

Scenario ::=
{ **use** *LibraryID* {',' *LibraryID*} ';' }
scenario *ScenarioID*
[*ScenarioParameters*]
'{'
[*FrontMatter*]
{*State* | *Transition* | *NamedAction*}
'}'
{ *FunctionDefinition* }

A scenario must have at least one transition and two states – the initial state and a final state. The initial state must have no incoming transitions, and final states have no outgoing transitions. Scenario parameters are specified as a list of comma-separated typed identifiers:

ScenarioParameters ::=
'(' *Parameter* {',' *Parameter*} ')'
Parameter ::= *Type ParameterId*

Example:

```
scenario example (string host, int count)
{ ... }
```

The example scenario has two parameters, `host` and `count`. Parameters are accessible by the scenario instances as constants.

1.4 Front Matter

Scenarios may include annotations, and may declare constants and variables:

Front Matter ::=
{(*Annotation* | *ConstDecl* | *VarDecl*)}

ConstDecl ::=
const *Type ConstId* { '[' [*size*] ']' } '=' *InitialValue* ';'
VarDecl ::=
[**global**] *Type VarId* { '[' [*size*] ']' } ['=' *InitialValue*] ';'

Annotations are described in Section 1.12. A variable declared "global" is shared by all instances of the scenario. A variable not declared "global" is instantiated privately in each instance of the scenario. Variables may be assigned initial values.

Example:

```
use tcpip;
scenario example
{
  const int bufsize = 1024;
  global int count = 0;
  Host server;
  ...
}
```

This example declares a constant integer `bufsize` with value 1024 and declares a global variable `count` with initial value 0. This variable will be shared by all instances of the scenario. That is, if a scenario instance increments the `count` variable, the update is seen by all of the other instances of the scenario. The variable declaration in the example also includes a variable named `server` of type `Host` (a `netlib` type). Because `server` is a local variable (i.e., its declaration does not contain the keyword **global**), each instance of the scenario will have its own copy of `server`.

1.5 State

"State" is one of the two fundamental concepts in STATL. States have names so they can be referred to in transitions and in the graphical representation of the scenario (i.e., in the STD). Each state may have annotations, an assertion, and a code block, but these elements are optional:

State ::=
    [**initial**]
    state *StateId* {*Annotation*}
    '{'
        [*Assertion*]
        [*CodeBlock*]
    '}'

Exactly one state must be designated as the initial state. When a scenario plugin is loaded into an IDS a first instance (the "root" instance) is created in the initial state.

The state assertion, if present, is tested before entry to the state, after testing the assertion of the transition that leads to the state. A state's assertion is implicitly True if none is specified. A state's code block is executed after the incoming transition's assertion and the state's assertion have been evaluated and found to be True, and after the incoming transition's code block (if it exists) is executed.

Example:

```
scenario example
{
  const int threshold = 64;
  int counter;
  ...
  initial
  state s1 { }
  ...
  state s3
  {
    counter > threshold
    { log("counter over threshold limit"); }
  }
  ...
}
```

In this example state `s1` is designated as the initial state. It has neither an assertion nor a code block. State `s3` has an assertion and a code block. The assertion specifies that the value of local variable `counter` is greater than the value of constant `threshold`. The code block calls the built-in procedure `log` to write a message to the IDS's log file.

1.6 Transition

"Transition" is the second of the two fundamental concepts in STATL. Each transition has a name and must indicate the pair of states that it connects. Transitions may have the same source and destination state; that is, loops are allowed. In addition, a transition must specify a type (`consuming`, `nonconsuming`, or `unwinding`), may have annotations, must specify an event type to match, and may have a code block:

Transition ::=
  transition *TransitionID* '(' *StateId* '->' *StateId* ')'
  (**consuming** | **nonconsuming** | **unwinding**)
  {*Annotation*}
  '{'
    ('[' *EventSpec* ']' | *ActionId*)
    {*Annotation*}
    [':' *Assertion*]
    [*CodeBlock*]
  '}'

A transition's event is specified either directly (see Section 1.7) or by reference to a named action (see Section 1.8). In the former case the transition's assertion is just the assertion in the transition. In the latter case, if the named action includes an assertion and the transition also includes an assertion, then the resulting assertion is the conjunction of the two assertions. An example is given in Section 1.8, after named actions are defined.

A transition's code block is executed after evaluating the transition's assertion and the destination state's assertion, and before executing the destination state's code block. A transition's code block may perform any computation supported by STATL and the IDS extension in use, but is typically used to copy event field values into the global or local environment for later reference.

Example:

```
use bsmlib, unixlib;
scenario example
{
  int userid;
  ...
  transition t2 (s1 -> s2) nonconsuming
  {
    [READ r] : r.euid != r.ruid
    { userid = r.euid; }
  }
  ...
}
```

In this example, `t2` is a nonconsuming transition that leads from state `s1` to state `s2`. The event spec indicates that the transition should match events of type `READ`, with a filter condition specifying that the `euid` and `ruid` fields of the event must differ for the transition to fire. The transition's code block copies the `euid` field of event `r` into the local variable `userid` for later reference. Note that this scenario uses both *bsmlib* and *unixlib* extensions, which define BSM events and UNIX-related abstractions, respectively.

1.7 Event Spec

"Event specs" are the essential elements of transitions. They specify what events (signature actions) to match and under what conditions.

EventSpec ::=
(*BasicEventSpec* [*SubEventSpec*]) | *TimerEvent*

BasicEventSpec ::= *EventType EventId*

SubEventSpec ::= '[' *EventSpec* { ',' *EventSpec* } ']'
EventType ::= **ANY** | *ApplEventType*
'(' *ApplEventType* {'|' *ApplEventType* } ')'

An event spec is either a basic event spec optionally followed by a subevent spec, or a timer event. A *basic event spec* identifies the built-in meta-event "type" `ANY`, which matches any event, or an application-specific event type (e.g., `READ`) or a disjunction of application-specific event types (e.g., `(UDP | TCP)`), and a name that will be used to reference the matching event. A basic event spec identifying a single type matches an event of the same type only. A basic event spec that is the disjunction of two or more event types matches an event of any of the types in the disjunction. A subevent spec identifies a set of event specs. Subevent specs enable complex, tree-structured event patterns. A subevent spec matches a set of subevents if each event spec in the subevent spec matches an event in the set of subevents. Event matching is formally defined in section 2.3.3.

Timer events provide a means of specifying intervals and timeouts in signatures (e.g., to specify that n events of a certain type within any t-second interval constitute an attack).

Example:

```
[(READ | WRITE) access] :
         access.euid != access.ruid
```

Example:

```
[IP d1 [TCP t1]] :
      (d1.src == 192.168.0.1) && (t1.dst == 23)
```

The first example is a USTAT event spec that matches read or write events in which the effective and real user-ids differ. The second example is a NetSTAT event spec (with a subevent spec) that matches any IP datagram containing a TCP segment, with source IP address `192.168.0.1` and destination port `23`.

The built-in meta-event type `ANY` is effectively the same as disjunction over all application-specific event types, but is easier to specify (and more efficient to implement as a special case).

1.8 NamedSigAction

A named signature action has a name and specifies an event spec, and may have annotations:

NamedSigAction ::=
action *ActionId* {*Annotation*}
'{'
('[' *EventSpec* ']' | *ActionId*)
{*Annotation*}
[':' *Assertion*]
'}'

Named signature actions may be used to improve clarity and maintainability when multiple transitions have identical or similar event specs, for example, having the same event type but slightly different assertions. In such cases the common part can be factored out, put into a named action, and then used in the similar transitions.

Example:

```
use bsmlib, unixlib;
scenario example
{
  ...
  action a1
  {
    [WRITE r] : r.euid != 0
  }

  transition t1 (s1 -> s2)
  {
    a1: r.euid != r.ruid
  }

  transition t2 (s1 -> s3)
  {
    a1: r.euid == r.ruid
  }
  ...
}
```

In this example transitions `t1` and `t2` both use named action `a1` as their action, but with different assertions. This is equivalent to:

```
use bsmlib, unixlib;
scenario example
{
  ...
  transition t1 (s1 -> s2)
  {
    [WRITE r] : (r.euid != 0) && (r.euid != r.ruid)
  }

  transition t2 (s1 -> s3)
  {
    [WRITE r] : (r.euid != 0) && (r.euid == r.ruid)
  }
  ...
}
```

1.9 CodeBlock

Transitions and states may have codeblocks that are executed after the corresponding transition and state assertions have been evaluated and found to be True. A codeblock is a sequence of statements enclosed in braces:

CodeBlock ::=
  '{'
    {*statement*}
  '}'

The statements in a codeblock can be declarations, assignments, *for* and *while* loops, *if-then-else*, procedure calls, etc. (see the grammar in Appendix A for a complete list). Semantically, the statements in a STATL code block are executed in order, in the context of the global and local environments of the scenario instance in which the code block is executed.

1.10 Assertions

Assertions appear as filter conditions in states and in event specs (which are the matching element of transitions). STATL assertions are boolean expressions built up from literal constants, variable and constant names, function calls, and common arithmetic and relational operators (see Appendix A for a complete expression grammar). A STATL assertion is evaluated in the context of the global and local environments of the scenario instance where it is evaluated. Assertions may use, but may not change, the value of any name in the global or local environment. In addition, transition assertions may refer to events named in the event spec and to the fields of those events.

1.11 Timers

Timers are useful to express attacks in which some event or set of events must (or must not) happen within an interval following some other event or set of events. Timers can also be used to prevent "zombie" scenarios – scenarios that have no possible evolution – from wasting memory resources.

Timers are declared like variables using the built-in type `timer`. There are both local and global timers. All timers must be explicitly declared. Timers are started in code blocks using the built-in procedure `timer_start`. Timer expiration is treated as an event, and these events may be matched by using "timer events" as transition event specs.

Example:

```
scenario example
{
  timer t1;
  . . .
  state s1
  {
    { timer_start(t1, 30); }
  }

  transition expire (s1->s2)
  { [timer t1] }
  ...
}
```

The code block of state `s1` starts timer `t1`, which will expire in 30 seconds (i.e., at a time 30 seconds later than the timestamp on the event that led to state `s1`). The timer event `timer t1` matches the expiration of the timer named `t1`. When timer `t1` expires, transition `expire` will fire, leading to state `s2`.

A single timer may appear in multiple transitions; every enabled transition that has `timer t` as its event spec fires when timer `t` expires.

1.12 Annotations

STATL annotations provide a standard way to extend the language with application-specific features. The annotations to a scenario are processed by a specialized component of the offline STAT architecture, called the *Analyzer* [VK99]. The Analyzer's task is to perform any IDS-specific preprocessing on a STATL scenario before the scenario is actually translated into an executable representation. For example, the NetSTAT Analyzer uses the annotations of a STATL scenario to determine where a probe should be placed in a target network and how the probe should be configured in order to be sure to detect the described attack. The Analyzer relies on another component, the *Factbase*, to obtain information about the target environment. For example, in NetSTAT the Factbase contains information about the network topology and the services deployed.

Syntactically, an annotation is an annotation tag optionally followed by a list of expressions, all enclosed in the annotation delimiters "<*" and "*>":

Annotation ::=
  <* *Id* {*Expression* { ',' *Expression*} } ***>**

Annotations may be attached to states, transitions, actions, named actions, and scenarios. Each IDS defines any scenario customization it needs as annotations, and the associated Analyzer is expected to process them (any annotations left by the Analyzer will be ignored by the STATL translator).

An example of an application-specific language extension is the `ENDPOINT_PORTS` annotation often used in NetSTAT actions to specify which two network interfaces are communicating. For example, the event spec:

```
[IP d [TCP t]]
  <* ENDPOINT_PORTS a, b *> :
  ...
```

represents a TCP/IP segment exchanged between network interfaces `a` and `b`.

Annotations may be attached to states, transitions, and named actions by putting them just before the entity's opening brace. For example,

```
  state s1 <* annotation *>  ...
  transition t1 (s1->s2) consuming
    <* annotation *>  ...
```

An annotation in the scenario front matter is at scenario scope. An example of this kind is the set of conditions in NetSTAT scenarios that customize the attack signature to an installation-specific Factbase. These are specified using the NetSTAT `CONSTRAINT` annotation:

Example:

```
  scenario example ( ... )
  {
    Host server;
    Service x;
    ...

    <* CONSTRAINT
         server in Network.hosts &&
         x in server.services
    *>
    ...
  }
```

In this example the annotation tag is `CONSTRAINT` and the argument list consists of one expression.

1.13 Other language features

This section discusses language features not described elsewhere.

1.13.1 Builtin predicates and functions

Currently STATL provides the following builtin functions:

- `log`(*format-string*, *args*) – write a string to the log file. Arguments are like those for the standard C `printf` function.
- `msg`(*format-string*, *args*) – write a string to the message file. Similar to `log`.
- `timer_start`(*timer-name*, *seconds*) – inserts a timeout event into the local or global timer queue. For more information, see “starting a timer” on page 24.
- `enqueue_event`(*event-name*, *event-type*, *timestamp*) – this function wraps an application event in a STAT event and inserts it into the event stream according to the specified timestamp. The extension library must provide an implementation of this function. See section 2.3.4 for more information about synthetic events.

1.13.2 Specification functions

Specification functions are auxiliary functions included in a scenario file. They perform computations that would be difficult or impossible to write directly in an assertion, or that are reused multiple times in assertions or codeblocks. For example, in the scenario fragment:

```
transition check (s0->compromised) nonconsuming
{
  [E e] : complicated_test(e.a,e.b)
  {
    // do something...
  }
}

bool
function complicated_test ( int a, int b )
{
  // complicated computation here ...
}
```

the specification function `complicated_test` is used in the assertion of transition `check`, with two fields of event `e` as arguments.

1.13.3 File inclusion

STATL scenario files may include other files using `#include`, as in C or C++. For example,

```
#include "somestuff.stat"
```

would interpolate the contents of `somestuff.h` where the `#include` appears. This might be useful for including specifications functions in multiple scenario files.[1]

[1] But if a specification function is needed by multiple scenarios, it might be better to write the shared functions as STAT extension functions, making them available to all scenarios that use the extension.

Chapter 2

Semantics

The syntax of STATL and examples of its use were presented in the previous chapter. This chapter defines the semantics of STATL in terms of the semantics of a single STATL scenario. There is no "sharing" between scenarios, so they are effectively independent of each other. Therefore, when the semantics of a STATL scenario have been defined, the semantics of STATL have been defined.

The STAT runtime core implements the STATL semantics defined here, and also implements several additional features. The two most important of these additional features, semantically, are *responses* and *synthetic events*. Responses may be dynamically attached to or detached from scenario states, and specify actions to take when the states are reached. Typical responses are to log a detection or send an alert. Generating a synthetic event and inserting it into the sensor's event stream is another kind of response. Synthetic events are discussed in Section 2.3.4, but most features of the STAT runtime core are not discussed in this report. In particular, in this chapter responses are treated as abstract entities with unspecified semantics.

Section 2.1 presents the most abstract view of a STATL scenario, modeled as an event-driven, reactive system. Then section 2.2 introduces the entities that are needed to specify the semantics of a STATL scenario, and section 2.3 defines the semantics of a STATL scenario. Section 2.4 presents an example of applying STATL semantics to a particular scenario and event stream. Section 2.5 describes mechanisms for extending STATL. Section 2.6 is a glossary of symbols used in this chapter.

2.1 Abstract view of STATL scenarios

A STATL scenario can be modeled abstractly as an event-driven, reactive system. At this highest level of abstraction, three kinds of reactions of a system S to an event ϵ can be identified:

- **none** – S does nothing when event ϵ occurs,
- **change state** – S changes its internal state when event ϵ occurs,
- **respond** – S generates some externally visible response (e.g., an IDMEF Alert [Cur01]) when event ϵ occurs.

In general, the signature of an event-driven, reactive system S is:

$$\mathsf{S} : \Sigma \times \mathrm{E} \to \Sigma \times \wp(\mathcal{R})$$

where Σ denotes the domain of system states, E denotes the domain of events, $\mathcal{R}$ denotes the domain of responses, and $\wp(\mathcal{R})$ is the powerset of $\mathcal{R}$. In other words, a system S takes a system state σ and an event ϵ and produces a new system state σ' and a set R of responses:

$$\mathsf{S}(\sigma, \epsilon) \to (\sigma', R)$$

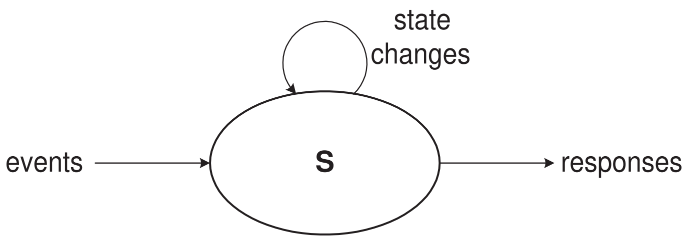

Figure 2.1: An abstract event-driven, reactive system.

Figure 2.1 depicts an abstract system. In STATL, responses are associated with state changes, so there is never a response without a state change. Informally, $\sigma' = \sigma$ implies that $R = \emptyset$ (i.e., S has produced no reaction), $\sigma' \neq \sigma$ means that S has changed its state, and $R \neq \emptyset$ means that S has produced a response.

In the rest of the paper this abstract view is refined by defining the static and dynamic elements of a STATL scenario and mapping the three kinds of abstract reaction – none, change state, respond – to a scenario's detailed semantics. Every scenario element defined in the next section is involved either in deciding whether a transition fires, or determining what happens to the system state when a transition does fire. These two things – when does a transition fire, and what happens when it does – together define the dynamic semantics of STATL, which is called the *scenario evolution model*.

2.2 Basic semantic definitions

A STATL *scenario* is defined formally as a 6-tuple:

$$\mathsf{S} : (S, T, s_0, I, g, q)$$

where S is a set of states, T is a set of transitions, s_0 identifies the initial state, I is an instance set, g is a global environment, and q is a global timer queue. The static elements of a scenario, defined in Section 2.2.1, are *states*[1] and *transitions* and their static subelements, and the identity of the scenario's initial state (as specified with the `initial` keyword). A state consists of a name, a *state assertion*, and a *state codeblock*. State assertions are Boolean functions and state codeblocks are sequences of executable statements. Both are interpreted over the domain of the scenario's dynamic environments, which are introduced in the next paragraph and defined in Section 2.2.2. A transition consists of a name, an *event spec*, a *transition assertion*, a *transition codeblock*, and a pair of state names indicating the source and destination states with which the transition is associated. Event specs indicate the kinds of events that a transition should match. Transition assertions are Boolean functions and transition codeblocks are sequences of executable statements. Both are interpreted over the domain of the scenario's dynamic environments and the "current" *event*.

The dynamic entities associated with a scenario are a *global environment*, a *global timer queue*, and an *instance set*. A global environment is a *name-value mapping* associating values with the scenario's global variables, parameters, and constants. A global timer queue is a queue of timeout events that apply to all instances of a scenario. An instance set is a set of *scenario instances*. A scenario instance (or just *instance*) is a sequence of *snapshots*. Each snapshop consists of a "current" state, a *local environment*, and a "causative" event. A local environment is a name-value mapping associating values with the scenario's local variables. An event is a STAT event, which abstracts and encapsulates an application event.

The following two subsections formalize the static and dynamic entities just introduced. These entities are then used in Section 2.3 to define the semantics of a STATL scenario.

2.2.1 Static entities

The static entities associated with a STATL scenario are just the states and transitions defined in the scenario.

[1]Each italicized term is defined in one of the following two subsections, depending on whether it is identified as static or dynamic.

2.2.1.1 State

The set of states defined in scenario S is denoted by $\mathsf{S}.S$, or just S when the particular scenario is understood or unimportant. A scenario state $s \in S$ has a name, an assertion, and a codeblock:[2]

$$s : \mathcal{N}_S \times \mathcal{A}_S \times \mathcal{C}_S$$

where $\mathcal{N}_S$ denotes the domain of state names[3], $\mathcal{A}_S$ denotes the domain of state assertions, and $\mathcal{C}_S$ denotes the domain of state codeblocks. The symbols $s.\mathrm{name}$, $s.\mathrm{assert}$, and $s.\mathrm{code}$ denote the name, assertion, and codeblock, respectively, of state s.

2.2.1.2 Transition

The set of transitions defined in scenario S is denoted by $\mathsf{S}.T$, or just T when the particular scenario is understood or unimportant. A scenario transition $t \in T$ has a name, a type, an event spec, an assertion, a codeblock, and a pair of state names indicating the source and destination states with which the transition is associated:

$$t : \mathcal{N}_T \times \mathcal{T}_T \times \mathcal{E} \times \mathcal{A}_T \times \mathcal{C}_T \times \mathcal{N}_S \times \mathcal{N}_S$$

where $\mathcal{N}_T$ denotes the domain of transition names, $\mathcal{T}_T$ denotes the domain of transition types[4], $\mathcal{E}$ denotes the domain of event specs, $\mathcal{A}_T$ denotes the domain of transition assertions, and $\mathcal{C}_T$ denotes the domain of transition codeblocks. The symbols $t.\mathrm{name}$, $t.\mathrm{type}$, $t.\mathrm{espec}$, $t.\mathrm{assert}$, $t.\mathrm{code}$, $t.\mathrm{from}$, and $t.\mathrm{to}$ denote the name, type, event spec, assertion, codeblock, source-state, and destination-state, respectively, of transition t.

2.2.1.3 Event spec

The domain of event specs is denoted by $\mathcal{E}$. An event spec $e \in \mathcal{E}$ consists of an event type, an event name, and a possibly empty set of (sub)event specs:

$$e : \mathcal{T}_E \times \mathcal{N}_E \times \wp(\mathcal{E})$$

where $\mathcal{T}_E$ denotes the domain of event types, $\mathcal{N}_E$ denotes the domain of event names, and $\wp(\mathcal{E})$ is the powerset of $\mathcal{E}$. The symbols $e.\mathrm{type}$, $e.\mathrm{name}$, and $e.\mathrm{subs}$ denote the type, name, and (sub)event specs, respectively, of event spec e.

Event specs are explained in section 2.3.3, in the context of the semantics of STATL event matching.

2.2.1.4 Assertion

Transition assertions and state assertions are predicates that determine whether transitions fire when an event matches an event spec. A missing assertion is interpreted as `true`.

A state assertion is evaluated in the context of the scenario's global environment and an instance's local environment (defined in Sections 2.2.2.2 and 2.2.2.3, respectively). A transition assertion is evaluated in the context of the scenario's global environment, an instance's local environment, and the "current" event:

- a state assertion *s.assert* is a Boolean function of a global environment and a local environment:

  $$s.assert : \mathcal{G} \times \mathcal{L} \rightarrow \text{Boolean}$$

  where $\mathcal{G}$ denotes the domain of global environments and $\mathcal{L}$ denotes the domain of local environments.

[2] Initial states may not have an assertion or a codeblock, because both assertions and codeblocks are defined (later) with respect to firing transitions that lead to a state, and no transition leads to the initial state.

[3] In fact, $\mathcal{N}_S = \mathcal{N}$, the domain of STATL names. Transition names, denoted by $\mathcal{N}_T$, object names, denoted by $\mathcal{N}_O$, and event names, denoted by $\mathcal{N}_E$, also are drawn from the same domain. They are distinguished only to make clear what sort of entity is being named in certain contexts.

[4] $\mathcal{T}_T = \{\mathtt{consuming}, \mathtt{nonconsuming}, \mathtt{unwinding}\}$

- a transition assertion *t.assert* is a Boolean function of a global environment, a local environment, and the "current" event:

  $$t.assert : \mathcal{G} \times \mathcal{L} \times \mathrm{E} \rightarrow \text{Boolean}$$

  where E denotes the domain of STAT events.

Section 2.3.2.3 defines how assertions are evaluated.

2.2.1.5 Codeblock

Transition codeblocks and state codeblocks are sequences of executable statements that change a scenario's global environment and an instance's local environment when a transition fires. A missing codeblock is interpreted as a "no-op" – no change to the global or local environment. A state codeblock is executed in the context of the scenario's global environment and an instance's local environment. A transition codeblock is executed in the context of the scenario's global environment, an instance's local environment, and the "current" event.

- A state codeblock is a function of a global environment, a global timer queue, and a local environment. A state codeblock produces an updated global environment, global timer queue, and local environment:

  $$s.\text{code} : \mathcal{G} \times \mathcal{Q} \times \mathcal{L} \rightarrow \mathcal{G} \times \mathcal{Q} \times \mathcal{L}$$

  That is, given a global environment g, global timer queue q, and local environment l, state codeblock $s.\text{code}$ produces a new global environment g', a new global timer queue q', and a new local environment l'.[5]

  $$s.\text{code}(g, q, l) \rightarrow (g', q', l')$$

- A transition codeblock is a function of a global environment, a global timer queue, a local environment, and the "current" event. A transition codeblock produces an updated global environment, global timer queue, and local environment:

  $$t.\text{code} : \mathcal{G} \times \mathcal{Q} \times \mathcal{L} \times \mathrm{E} \rightarrow \mathcal{G} \times \mathcal{Q} \times \mathcal{L}$$

  That is, given a global environment g, a global timer queue q, a local environment l, and an event ϵ, transition codeblock $t.\text{code}$ produces a new global environment g', a new global timer queue q', and a new local environment l'.

  $$t.\text{code}(g, q, l, \epsilon) \rightarrow (g', q', l')$$

Section 2.3.2.4 defines how codeblocks are executed.

2.2.2 Dynamic entities

For modeling a STATL scenario S as a reactive, event-driven system, the static entities in the previous section provide the system specification, and the dynamic entities in this section define the system "state".

[5] Responses don't show up here because responses are not part of STATL. Section 2.3.1 describes how responses occur after execution of $t.\text{code}$ and $s.\text{code}$.

2.2.2.1 Name-value mapping

The domain of name-value mappings is denoted by $\mathcal{M}$. Name-value mappings are used to define global and local environments (in Sections 2.2.2.2 and 2.2.2.3). A name-value mapping $m \in \mathcal{M}$ maps names to values:

$$m : \mathcal{N}_O \rightarrow \mathcal{V}$$

where $\mathcal{N}_O$ denotes the domain of object names and $\mathcal{V}$ denotes the domain of object values. For example,

$$\{\,\texttt{length} : 1024, \texttt{hostname} : \texttt{"mailserver"}\,\}$$

is a name-value mapping that maps the name `length` to the integer value 1024 and the name `hostname` to the string value `"mailserver"`. Names must follow the syntactic rules for STATL identifiers, and values may be chosen from any builtin or application-specific type domain.

Given a name-value mapping $m = \{n_0 : v_0, \ldots, n_p : v_p\}$, it must be the case that $n_i = n_j \rightarrow i = j$; i.e, names are unique within a given name-value mapping. Also, $m(n_j) = v_j$ for $0 \leq j \leq p$. That is, a name-value mapping is a function that returns the value associated with a specified name, or `undefined` if the specified name is not in dom(m).

2.2.2.2 Global environment

The domain of global environments is denoted by $\mathcal{G} = \mathcal{M}$. The global environment of scenario S is denoted by $\mathsf{S}.g$. A global environment $g \in \mathcal{G}$ is a name-value mapping for scenario global variables, which are also called "scenario variables". The names in a global environment are exactly the scenario's declared global variables, parameters, and constants.[6] All *scenario instances* (see Sections 2.2.2.6 and 2.2.2.7) of a single customized version of the scenario[7] share a single global environment. For example, given the scenario

```
scenario example ( int timeout )
{
  int userid;
  string username;
  global int counter;

  ...
}
```

each customized version of the scenario would have a global environment containing names `counter` and `timeout`, with associated values. A particular global environment for a version of the scenario customized with $\texttt{timeout} = 30$ might look like this:

$$g = \{\texttt{counter} : 7, \texttt{timeout} : 30\}$$

so g treated as a function would have $g(\texttt{counter}) = 7$ and $g(\texttt{timeout}) = 30$. All instances of this customized version of the scenario would share this global environment. A different customized version of the scenario, say with $\texttt{timeout} = 120$, would provide a different global environment to be shared by all of its instances.

2.2.2.3 Local environment

The domain of local environments is denoted by $\mathcal{L} = \mathcal{M}$. A local environment $l \in \mathcal{L}$ is a name-value mapping for scenario local variables, which are also called "instance variables". Each *scenario instance* (see Sections 2.2.2.6 and 2.2.2.7) has its own local environment. The names in a local environment are exactly the scenario's declared local variables (i.e., variables that are not declared `global`). For example, given the scenario

[6] Parameters and constants are in the global environment by convention. Since they are semantically immutable, they could just as well be in local environments. An implementation would probably substitute their values at compile-time or load-time.

[7] A scenario is *customized* by providing a value for each of its parameters.

```
scenario example
{
  int userid;
  string username;
  global int counter;

  ...
}
```

each instance of the scenario would have a local environment containing names `userid` and `username`, with associated values. A particular local environment l for one of the scenario's instances might look like this:

$$l = \{\texttt{userid} : 105, \texttt{username} : \texttt{"alice"}\}$$

so l treated as a function would have $l(\texttt{userid}) = 105$ and $l(\texttt{username}) = \texttt{"alice"}$.

2.2.2.4 Event

Events are not part of the syntax of STATL, but are needed to define the semantics of STATL. It is important to distinguish between "application events", which are elements of a raw event stream, and "STAT events", which are essentially wrappers around application events, and are processed by the STAT event-processing engine. The domain of STAT events is denoted by E, and the domain of application events is denoted by Æ. Each STAT event $\epsilon \in \mathrm{E}$ contains an application event type, a timestamp, an application event, and a possibly empty set of subevents, each of which is a STAT event of the same form:

$$\epsilon : \mathcal{T}_E \times \mathrm{T} \times \text{Æ} \times \wp(\mathrm{E})$$

where $\mathcal{T}_E$ denotes the domain of event types and T denotes the time domain. By abstracting and encapsulating application events, which have an arbitrary, application-defined format, STAT events present a consistent, abstract event format to the STATL event matching algorithm. It is assumed that application preprocessors (event-stream readers) extract application event types and timestamps, and construct STAT-event wrappers around application events. The symbols $\epsilon.\mathrm{type}$, $\epsilon.\mathrm{time}$, $\epsilon.\mathrm{appevent}$, and $\epsilon.\mathrm{subs}$ denote the type, timestamp, application event, and subevents, respectively, of event ϵ.

Further explanation of *event* is deferred to Section 2.3.3.

2.2.2.5 Timer queue

The domain of timer queues is denoted by $\mathcal{Q}$. Each timer queue $q \in \mathcal{Q}$ is a timestamp-ordered list of timeout events. Discussion of timer queues is deferred to the definition of `timer_start` on page 24, which also describes timeout events, timer event specs, global and local timer queues, and how timeout events in timer queues are processed as events.

2.2.2.6 Snapshot

The domain of snapshots is denoted by Σ. Each snapshot $\sigma \in \Sigma$ contains a state name, a local environment, a local timer queue, and an event:

$$\sigma : \mathcal{N}_S \times \mathcal{L} \times \mathcal{Q} \times \mathrm{E}$$

Snapshots are the building blocks for scenario instances. Intuitively, a snapshot identifies how far an attack has progressed by recording the current state, the values of local variables, and the event that led to the current state. The symbols $\sigma.\mathrm{state}$, $\sigma.\mathrm{env}$, $\sigma.\mathrm{q}$, and $\sigma.\mathrm{event}$ denote the state, local environment, local timer queue, and event, respectively, of snapshot σ. The current state $\sigma.\mathrm{state}$ implicitly identifies which transitions are *enabled* (i.e., available to match events) in snapshot σ. The enabled transitions in a set of transitions T with respect to snapshot σ are those whose source state is the same as $\sigma.\mathrm{state}$:

$$\mathrm{enabled}(T, \sigma) \stackrel{def}{=} \{t_j \in T \ni t_j.\mathrm{from} = \sigma.\mathrm{state}\}$$

where '$\ni$' means "such that".

2.2.2.7 Scenario instance

The domain of scenario instances (or just "instances") is denoted by $\mathcal{I}$. An instance $i \in \mathcal{I}$ is a non-empty sequence of snapshots:

$$i : \Sigma^{+}$$

By representing a scenario instance i as a sequence of snapshots $\langle \sigma_0 \ldots \sigma_n \rangle$, an instance records the complete history of an attack that (partially) matches a particular scenario S. Each snapshot in an instance, other than the first, corresponds to a transition that fired in reaction to a particular event.[8] The first snapshot in every instance of a scenario S is an *initial* snapshot σ_0:

$$\sigma_0 \stackrel{def}{=} (\mathsf{S}.s_0, l_0, \langle\rangle, \mathtt{null})$$

where l_0 is a local environment in which each local variable has its initial value, the timer queue is empty, and there is no event.

The enabled transitions in a set of transitions T with respect to instance i are determined by the last snapshot of i, denoted by $i.\mathrm{last}$:

$$\mathrm{enabled}(T, i) \stackrel{def}{=} \mathrm{enabled}(T, i.\mathrm{last}) = \{t_j \in T \ni t_j.\mathrm{from} = i.\mathrm{last.state}\}$$

2.2.2.8 Instance set

The instance set of scenario S is denoted by $\mathsf{S}.I$, or just I when the particular scenario is understood or unimportant. An instance set I is a set of instances:

$$I : \wp(\mathcal{I})$$

Intuitively, $\mathsf{S}.I = \{i_0, i_1, \ldots, i_m\}$ records all partial matches of S up to the "current" event. That is, I contains all the active instances of a particular attack. A scenario starts from an initial safe state and reaches a final state through transitions. For every scenario S, the initial instance set is:

$$\mathsf{S}.I \stackrel{def}{=} \{i_0\}, \text{where } i_0 \stackrel{def}{=} \langle \sigma_0 \rangle$$

That is, every scenario begins its dynamic existence with an instance set that contains a single instance, which, in turn, consists of a single initial snapshot. The enabled transitions of i_0 are just the transitions that exit the scenario's initial state s_0:

$$\mathrm{enabled}(T, i_0) \stackrel{def}{=} \mathrm{enabled}(T, i_0.\mathrm{last}) = \mathrm{enabled}(T, \sigma_0) = \{t_j \in T \ni t_j.\mathit{from} = s_0\}$$

There is no notion of an enabled transition set with respect to an instance set I. Instead, each instance $i \in I$ has an enabled transition set as defined in Section 2.2.2.7.

2.3 The dynamic semantics of a scenario

Intuitively, a scenario begins its dynamic existence with an initial instance set (defined in Section 2.2.2.8), an initial global environment g_0 in which each global variable has its initial value, and initial timer queue q_0 that is empty. The instance set, global environment, and timer queue *evolve* as events are processed and transitions fire. An enabled transition fires when its event spec *matches* the event being processed, its assertion is satisfied, and the assertion of its destination state is satisfied. As mentioned in Section 2.2.1.2 there are three types of transitions: nonconsuming, consuming, and unwinding. Nonconsuming transitions represent attack steps that do not prevent further occurrences of

[8] The snapshots in an instance do not record the identity of the transition that fired. Instead, they record the *consequences* of firing the transition: the destination state reached and the new local environment.

attacks from spawning from the transition's source state. Therefore, when a nonconsuming transition fires the source state remains valid, and the destination state becomes valid too. In contrast, the firing of a consuming transition makes the source state of a particular attack occurrence invalid. Unwinding transitions represent a form of "rollback" and they are used to describe events and conditions that may invalidate the progress of one or more scenario instances and require the return to an earlier state.

The instance set evolves differently depending on what type of transition fires. Instances are created and added to an instance set when nonconsuming transitions fire. Existing instances are modified when consuming transitions fire. Instances are destroyed and deleted from an instance set when unwinding transitions fire. When a transition fires, the global environment and the local environment of the affected instance are updated by "executing" the codeblocks of the transition and the destination state.

The algorithm in Figure 2.2 defines how scenario S applied to current event ϵ_c evolves from (S, T, s_0, I, g, q) to (S, T, s_0, I', g', q'). The auxiliary set F of to-be-fired transition-instance pairs is initially empty, as is the new instance set I'. The new global environment g' and new global timer queue q' are initially the same as the old g and q, respectively.

foreach $i \in I$
  foreach $t \in \text{enabled}(T, i)$
    let $\epsilon' = \text{select\_satisfying\_event}(\text{matching\_events}(\{\epsilon_c\}, \{t.\text{espec}\}), i, t, g)$
    if $\epsilon' \neq$ `null`
    then
      insert (t, i, ϵ') into F
    end
  end
end
let $I' = \{i \in I \ni (\nexists\, t, \epsilon' \ni (t, i, \epsilon') \in F)\}$
foreach $(t, i, \epsilon') \in F \ni t.\text{type} =$ `nonconsuming`
  $\text{fire\_nonconsuming\_transition}(t, i, \epsilon', I', g', q')$
end
foreach $(t, i, \epsilon') \in F \ni t.\text{type} =$ `consuming`
  $\text{fire\_consuming\_transition}(t, i, \epsilon', I', g', q')$
end
foreach $(t, i, \epsilon') \in F \ni t.\text{type} =$ `unwinding`
  $\text{fire\_unwinding\_transition}(t, i, \epsilon', I', g', q')$
end

$\text{select\_satisfying\_event}(\varepsilon, i, t, g) \stackrel{def}{=}$
  if $\exists\, \epsilon_j \in \varepsilon \ni$
    $\text{eval\_trans\_assertion}(t.\text{assert}, g, i.\text{last.env}, \epsilon_j)$ **and**
    $\text{eval\_state\_assertion}(t.\text{to.assert}, g, i.\text{last.env})$
  then
    return some $\epsilon_j \in \varepsilon \ni$
      $\text{eval\_trans\_assertion}(t.\text{assert}, g, i.\text{last.env}, \epsilon_j)$ **and**
      $\text{eval\_state\_assertion}(t.\text{to.assert}, g, i.\text{last.env})$
  else
    return `null`
  end

Figure 2.2: STATL event processing algorithm

Informally, the algorithm says to first identify all the transitions that should fire, along with the instances in which they should fire and the (sub)event that causes each transition to fire.[9] Next initialize I' to the set of instances in I that have no transitions to be fired. Then fire any nonconsuming transitions that were identified in the first step. After that fire the identified consuming transitions. Finally, fire the identified unwinding transitions. The following subsections define the functions matching_events, eval_trans_assertion, eval_state_assertion, and fire_*_transition in reverse order, presenting a view of how a STATL scenario reacts to events. First instance set evolution is defined in terms of transition firing, then codeblock execution and assertion evaluation are described, and finally matching between an event spec and an event is specified. Section 2.4 presents an example to illustrate the definitions in this section.

2.3.1 Transition firing and instance set evolution

This section describes what happens to a scenario's instance set when a transition fires. Informally, when a nonconsuming transition is fired, a copy of the instance is created and a new snapshot is appended to the copy. The new snapshot has as its state the destination state of the transition that fired. It has as its event the event that caused the transition to fire. And it has as its local environment the local environment of the original instance, possibly updated by executing the transition codeblock and the destination state codeblock (defined in Section 2.3.2.4). Executing the codeblocks also may update $\mathsf{S}.g$, $\mathsf{S}.q$, and the new instance's timer queue, which is inherited from the original instance. The original instance is added to the new instance set. If the destination state is not a final state the new instance is also added to the new instance set. A *final state* is a state s with no exiting transitions; that is, one for which $\{t_j \in T \ni t_j.\text{from} = s\}$ is empty.

When a consuming transition is fired, a copy of the instance is created and a new snapshot is appended to the copy. The new snapshot and other dynamic elements are constructed as in the case of a nonconsuming transition. If the destination state is not a final state, the new instance is added to the new instance set. The original instance is *not* added to the new instance set.[10]

When an unwinding transition fires, one or more instances are deleted from the instance set (i.e., they are not in the new instance set). The instances that get deleted are just those that are derived from the snapshot that is being unwound to. In order to give an intuitive and precise description of unwinding it is necessary to define transitions more formally. The following subsections consider the three transition types separately, in each case assuming that the instance i in which the transition is firing is of the form $i = \langle \sigma_0 \dots \sigma_n \rangle$. The functions $\text{fire\_*\_transition}(t, i, \epsilon, I', g', q')$ are defined in terms of how I', g', and q' change.

Although responses are not strictly part of STATL, it should be noted that the STAT runtime core calls response functions attached to a state s immediately after executing the codeblock $s.\text{code}$.

2.3.1.1 Nonconsuming transitions

The function $\text{fire\_nonconsuming\_transition}(t, i, \epsilon, I', g', q')$ is defined as follows:

$$
\begin{aligned}
I' &\stackrel{def}{=} \begin{cases} I' \cup \{i, \langle \sigma_0 \dots \sigma_n \sigma_{n+1} \rangle\} & \text{if } t.\text{to is not final} \\ I' \cup \{i\} & \text{otherwise} \end{cases} \\
g' &\stackrel{def}{=} \text{exec}(t.\text{to.code}, \texttt{null}, \text{exec}(t.\text{code}, \epsilon, ((g', \sigma_n.\text{env}), (q', \sigma_n.\text{q})))).\text{maps.first} \\
q' &\stackrel{def}{=} \text{exec}(t.\text{to.code}, \texttt{null}, \text{exec}(t.\text{code}, \epsilon, ((g', \sigma_n.\text{env}), (q', \sigma_n.\text{q})))).\text{queues.first}
\end{aligned}
$$

where σ_{n+1} is constructed as follows:

$$
\begin{aligned}
\sigma_{n+1}.\text{state} &\stackrel{def}{=} t.\text{to} \\
\sigma_{n+1}.\text{env} &\stackrel{def}{=} \text{exec}(t.\text{to.code}, \texttt{null}, \text{exec}(t.\text{code}, \epsilon, ((g', \sigma_n.\text{env}), (q', \sigma_n.\text{q})))).\text{maps.second}
\end{aligned}
$$

[9] Each event may contain any number of subevents, any of which might "match" as defined in Section 2.3.3. The function $\texttt{select\_satisfying\_event}$ nondeterministically selects one (sub)event as *the* match when multiple matches are possible. Section 2.3.3 discusses this nondeterminism in more detail. In the usual case events do not contain isomorphic subevents and at most one match is possible.

[10] Copying the original instance instead of modifying it directly handles the case of multiple consuming transitions to be fired in the same instance.

$$\sigma_{n+1}.\mathrm{q} \stackrel{def}{=} \mathrm{exec}(t.\mathrm{to.code}, \mathtt{null}, \mathrm{exec}(t.\mathrm{code}, \epsilon, ((g', \sigma_n.\mathrm{env}), (q', \sigma_n.\mathrm{q})))).\mathrm{queues.second}$$
$$\sigma_{n+1}.\mathrm{event} \stackrel{def}{=} \epsilon$$

The global timer queue q' is unchanged if no global timers were started, otherwise it is changed according to the definition of `timer_start` on page 24. The new instance's timer queue $\sigma_{n+1}.\mathrm{q}$ is $\sigma_n.\mathrm{q}$ if no local timers were started, otherwise it is changed according to the definition of `timer_start` on page 24. The original and new instances are added to the new instance set I' if the destination state is not a final state (i.e., a state with no exiting transitions). The codeblock execution function

$$\mathrm{exec} : \mathcal{C} \times \mathrm{E} \times (\mathcal{M}^+ \times \mathcal{Q}^2) \rightarrow \mathcal{M}^+ \times \mathcal{Q}^2$$

is defined in Section 2.3.2.4. The call to exec takes as its third argument a pair whose first element is a mapping list and whose second element is a pair of timer queues. The call returns possibly modified copies of the mappings and queues. $\mathrm{exec}(\ldots).\mathrm{maps}$ is the updated mapping list, and $\mathrm{exec}(\ldots).\mathrm{queues}$ is the updated pair of timer queues. The first element in each case is the global entry and the second element is the local entry.

The global and local environments and timer queues are updated by (1) executing the transition codeblock in the context of the current event, the given global environment, and the local environment from the original instance, then (2) executing the state codeblock in the context of no event and the name-value mappings produced by executing the transition codeblock. The global environment is changed by assignments to global variables. The local environment is changed by assignments to local variables. The global timer queue is changed by `timer_start` calls applied to global timers. The local timer queue is changed by `timer_start` calls applied to local timers.

2.3.1.2 Consuming transitions

The function $\mathrm{fire\_consuming\_transition}(t, i, \epsilon, I', g', q')$ is defined as follows. The new global environment g' and global timer queue q' are defined as for consuming transitions. The new instance set I' is defined as follows:

$$I' \stackrel{def}{=} \begin{cases} I' \cup \{\langle \sigma_0 \ldots \sigma_n \sigma_{n+1} \rangle\} & \text{if } t.\text{to is not final} \\ I' & \text{otherwise} \end{cases}$$

where the new snapshot σ_{n+1} is constructed as for consuming transitions. The only difference between a consuming transition and a nonconsuming transition is that when a consuming transition is fired, the original instance is not added to the new instance set.[11]

2.3.1.3 Unwinding transitions

The function $\mathrm{fire\_unwinding\_transition}(t, i, \epsilon, I', g', q')$ is defined as follows. The new values of the global environment g' and global timer queue q' are defined as for consuming and nonconsuming transitions. To derive the new value of the instance set I', define

$$p \stackrel{def}{=} \sigma_0 \sigma_1 \ldots \sigma_j \sigma_{j+1}$$

where σ_j is the *first* snapshot in i with $\sigma_j.\mathrm{state} = t.\mathrm{to}$.[12] I' is then defined as follows:

$$I' \stackrel{def}{=} (I' \setminus \{i' \in I' \ni p \in i'\}) \cup \{\langle \sigma_0 \ldots \sigma_j \rangle\}$$

This says to delete from I' any instance i' that has p as a prefix, because any such instance must be derived from the instance that is being unwound and must have progressed past the state (snapshot) that is being unwound to. Then add to I' the instance $\langle \sigma_0 \ldots \sigma_j \rangle$ (if it is not already in I'), which is the instance that is being unwound to. Notice that instances are being deleted from I', not from I, and this is done after all nonconsuming and consuming transitions have fired.

[11] A subtle consequence of the definition of consuming transitions is that it is possible for the instance set I' to become empty.

[12] The definition could have specified "last" instead of "first", or it could have specified that the scenario writer has a choice when multiple snapshots in the instance have the target state.

2.3.2 Dynamic semantics of assertions and codeblocks

This section defines how assertions are evaluated and codeblocks are executed. It starts by defining fundamental concepts common to these two aspects of STATL semantics: substitutions of values for names, and expression evaluation.

2.3.2.1 Substitutions

Three kinds of value-for-name substitutions take place in evaluating expressions:

- event field names are replaced by their values in the current application event
- names of variables declared in codeblocks are replaced by their values in the name-value mapping associated with the codeblock
- names of local variables, global variables, scenario parameters, and scenario constants are replaced by their values in the appropriate name-value mapping – the local environment for local variables, and the global environment for the others

Formally, the value associated with a name id in the context of an event ϵ and a list of name-value mappings $(m_0, \ldots, m_n)$ is defined by the subst function as follows:

$$\text{subst}(id, \epsilon, (m_0, \ldots, m_n)) \stackrel{def}{=} \begin{cases} \text{value}(\epsilon) & \text{if } id = \epsilon \\ m_j(id) & \text{otherwise, where } id \in \text{dom}(m_j) \\ & \quad \text{and } \forall\, k > j, id \notin \text{dom}(m_k) \end{cases}$$

The value of a substitution depends on which mappings each id appears in: the event is tried first, then the mappings are "searched" from right to left. Because of the way mapping lists are constructed – m_0 is always the global environment, m_1 is always the local environment, m_2, if it exists, corresponds to a codeblock, and m_3, etc., correspond to nested blocks – inner block declarations hide outer block declarations, block declarations hide the local and global environments, and the local environment hides the global environment.

In some cases id will be of the form *name.field* and the obvious extension of the above definition applies. For example, if the event name is `e` and the event type has a field `host`, then the substitition subst(`e.host`, `e`, ...) denotes the value of the `host` field of the event value denoted by `e`.

2.3.2.2 Expression evaluation

STATL expressions are evaluated by simultaneously substituting the appropriate values for all names in the expression, computing function values, and applying STATL operators, which have their usual meanings. Application functions and predicates have application-defined meanings.

More formally, the value of a STATL expression *expr* in the context of an event ϵ and a list of name-value mappings $(m_0, \ldots, m_n)$ is denoted by $\text{eval}(expr, \epsilon, (m_0, \ldots, m_n))$. The eval function is defined operationally as follows:

1. simultaneously apply substitutions $\text{subst}(n_1, \epsilon, (m_0, \ldots, m_n))$, ..., $\text{subst}(n_j, \epsilon, (m_0, \ldots, m_n))$ to *expr*, where $n_1, \ldots, n_j$ are the names appearing in *expr*.
2. compute values of any function references and apply builtin operators according to grouping (parentheses), associativity, and precedence.

2.3.2.3 Evaluation of assertions

The evaluation of a state assertion or transition assertion is just the evaluation of a STATL expression, as described in the previous subsection. Formally,

$$\begin{aligned} \text{eval\_trans\_assertion}(a, g, l, \epsilon) &\stackrel{def}{=} \text{eval}(a, \epsilon, (g, l)) \\ \text{eval\_state\_assertion}(a, g, l) &\stackrel{def}{=} \text{eval}(a, \text{null}, (g, l)) \end{aligned}$$

Transition assertions are evaluated in the context of the event, the global environment, and the local environment. State assertions are evaluated in the context of the global environment and local environment.

2.3.2.4 Execution of codeblocks

This section defines the codeblock execution function exec, which "executes" a codeblock in the context of an event, a list of name-value mappings, and a pair of timer queues. It returns an updated list of name-value mappings and an updated pair of timer queues. The meaning of assignment and sequencing are defined formally below. The remaining STATL statement types are given only informal descriptions, because they are like ordinary programming language statements and they do not directly affect a scenario's global environment or a scenario instance's local environment.

The signature of the exec function is as follows:

$$\mathrm{exec} : \mathcal{C} \times \mathrm{E} \times (\mathcal{M}^+ \times \mathcal{Q}^2) \rightarrow \mathcal{M}^+ \times \mathcal{Q}^2$$

In general, an exec call always returns modified copies of the input name-value mapping list and timer queue pair. The mappings may be modified by assignment statements in the codeblock, but no mappings are added or deleted, and no name-value pairs are added to or deleted from any mapping. The pair of timer queues may be modified, if the codeblock contains any `timer_start` calls:

$$\mathrm{exec}(c, \epsilon, ((m_0, \ldots, m_n), (q_g, q_l))) = ((m'_0, \ldots, m'_n), (q'_g, q'_l))$$

where $m'_0, \ldots, m'_n, q'_g, q'_l$ are defined in the following paragraphs for the most important statement types.

Declarations Executing a sequence of block declarations produces a new name-value mapping, and following statements are then executed in the context of this new name-value mapping appended to the given list of mappings. Formally,

$$\mathrm{exec}(\{\mathit{decls}; s\}, \epsilon, ((m_0, \ldots, m_n), (q_g, q_l))) \stackrel{\mathit{def}}{=} ((m'_0, \ldots, m'_n), (q'_g, q'_l))$$

where

$$\mathrm{exec}(\{s\}, \epsilon, ((m_0, \ldots, m_n, m_{n+1}), (q_g, q_l))) = ((m'_0, \ldots, m'_n, m'_{n+1}), (q'_g, q'_l))$$

where m_{n+1} is a name-value mapping constructed from *decls* in the obvious way: a name-value pair for each declaration, each pair having either the declared initial value or the default initial value for the type.

Assignment statement Executing an assignment statement modifies one name-value pair in one of the name-value mappings that provides its context. Formally,

$$\mathrm{exec}(\{\mathit{name} = \mathit{expr};\}, \epsilon, ((m_0, \ldots, m_n), (q_g, q_l))) \stackrel{\mathit{def}}{=} ((m'_0, \ldots, m'_n), (q_g, q_l))$$

where

$$m'_j = \begin{cases} m_j & \text{if } \mathit{name} \notin \mathrm{dom}(m_j) \\ & \quad \text{or } \exists\, k > j \ni \mathit{name} \in \mathrm{dom}(m_k) \\ \{\ldots, \mathit{name} : \mathrm{eval}(\mathit{expr}, \epsilon, (m_0, \ldots, m_n)), \ldots\} & \text{if } \mathit{name} \in \mathrm{dom}(m_j) \\ & \quad \text{and } \forall\, k > j, \mathit{name} \notin \mathrm{dom}(m_k) \end{cases}$$

The first and last ellipses indicate that the rest of the changed mapping m'_j is not modified. Timer queues are not changed by assignment.

Sequence Executing a sequence of two or more statements consists of executing the first statement in the context of the initial list of name-value mappings and pair of timer queues, then executing the second statement in the context of the new list of name-value mappings and pair of timer queues that results from executing the first statement, and so on. Formally,

$$\mathrm{exec}(\{s_1; s_2\}, \epsilon, ((m_0, \ldots, m_n), (q_g, q_l))) \stackrel{\mathit{def}}{=} \mathrm{exec}(\{s_2\}, \epsilon, \mathrm{exec}(\{s_1\}, \epsilon, ((m_0, \ldots, m_n), (q_g, q_l))))$$

Starting a timer The procedure call `timer_start(t,`n`)` generates a special kind of synthetic event called a *timeout*, with name `t` and timestamp $t_c + n$ (i.e., n seconds past the current time). The generated timeout is inserted, in timestamp order, into q_g if `t` is global, or into q_l if `t` is local. Generating a timeout is informally referred to as "starting" the timer, and a pending timeout (i.e., a queued timeout) is informally called a "running" timer.

Formally,

$$\text{exec}(\texttt{timer\_start(t,}n), \epsilon, ((m_0, \ldots, m_n), (q_g, q_l))) \stackrel{def}{=} ((m_0, \ldots, m_n), (q'_g, q'_l))$$

where $q'_g = q_g$ if `t` is local and $q'_l = q_l$ if `t` is global. One of the two timer queues, call it q, changes as follows:

1. If q already contains a timeout event (`t`,m) for some m, then delete that timeout event.
2. Insert $\tau_{new} = (\texttt{t}, t_c + n)$ into q in timestamp order. Given $q = \langle \tau_1, \ldots, \tau_s \rangle$:
   - if $q = \langle\rangle$ then $q' = \langle \tau_{new} \rangle$
   - if $t_c + n \leq \tau_1.\text{time}$ then $q' = \langle \tau_{new}, \tau_1, \ldots, \tau_s \rangle$
   - if $t_c + n > \tau_s.\text{time}$ then $q' = \langle \tau_1, \ldots, \tau_s, \tau_{new} \rangle$
   - otherwise $q' = \langle \ldots, \tau_j, \tau_{new}, \tau_{j+1} \ldots \rangle$ where $\tau_j.\text{time} < t_c + n \leq \tau_{j+1}.\text{time}$

Timeout events are semantically different from other events in two respects:

- Timeouts may have global or local scope, like variables (and timers must in fact be declared like variables). Generation of a global timeout causes the timeout to be inserted into the scenario's global timer queue. Generation of a local timeout causes the timeout to be inserted into the generating instance's timer queue. When a new instance is created, it inherits its parent's timer queue.
- Generating a timeout `t` in a given context (i.e., globally or in a particular instance) when `t` is already pending in that context causes the pending `t` to be deleted. Informally, starting a timer that is already running resets that timer to a new value.

Timers are conceptually interpolated into the event stream: a timeout becomes the current event when its timestamp is less than or equal to the timestamp of the event after the current event. When multiple timeouts have the same timestamp, they are inserted into the event stream in arbitrary order. When a timeout `t` becomes the current event, it matches the event spec `timer t`. A single timer may appear in multiple transitions; every enabled transition that has `timer t` as its action fires when timeout `t` occurs (or equivalently, when timer `t` expires). A timeout event is deleted from the timer queue when it becomes the current event.

Other statements The control-flow statements in STATL are if-else and switch conditionals; for, while, and do-while loops; goto, break, and continue; and return, associated with scenario-defined functions. Each of these statement types has essentially the same semantics as in C++.

2.3.3 Event matching

This section defines STATL event matching – how do event specs match events – in terms of the matching_events function used in Figure 2.2 on page 19. Recall that a STAT event $\epsilon \in$ E is a 4-tuple consisting of an event type, a timestamp, an application event, and a set of subevents, and that an event spec $e \in \mathcal{E}$ is a 3-tuple consisting of an event type, an event name, and a possibly empty set of subevent specs.[13] Given event ϵ and event spec e, the STAT event matching algorithm is specified in Figure 2.3. Event matching is defined recursively because both events and event specs are defined recursively.

[13]STATL event specs include disjunction of event types. Semantically, whenever (A|B) appears in an event spec, the containing transition is equivalent to two new transitions, one containing A in place of (A|B), the other containing B in place of (A|B). This equivalence generalizes to more than two disjuncts.

$\text{matching\_events}(\mathit{events}, \mathit{eventspecs}) \stackrel{def}{=}$
    foreach $e_i \in \mathit{eventspecs}$
        let $\mathit{events}_i = \cup_{\epsilon_j \in \mathit{events}}\ \text{matching\_event}(\epsilon_j, e_i)$
    end
    return $\cup_{\mathit{events}_i}\ \text{select\_event}(\mathit{events}_i)$

$\text{matching\_event}(\epsilon, e) \stackrel{def}{=}$
    if $(\epsilon.\text{type} = e.\text{type})$
    then
        let $\mathit{subevents} = \text{matching\_events}(\epsilon.\text{subs}, e.\text{subs})$
        if $(\texttt{null} \notin \mathit{subevents})$
        then
            return $(\epsilon.\text{type}, \epsilon.\text{time}, \mathit{subevents})$
        end
    end
    let $\mathit{events} = \cup_{\epsilon_j \in \epsilon.\text{subs}}\ \text{matching\_event}(\epsilon_j, e)$
    return $\text{select\_event}(\mathit{events})$

$\text{select\_event}(\mathit{events}) \stackrel{def}{=}$
    if $\exists\, \epsilon_j \in \mathit{events} \ni \epsilon_j \neq \texttt{null}$
    then
        return some non-null $\epsilon_j \in \mathit{events}$
    else
        return `null`
    end

Figure 2.3: STATL event matching algorithm

The functions in Figure 2.3 specify, respectively, (a) the set of events that is returned when a set of events is matched against a set of event specs, (b) what it means for a single event to match a single event spec (and the returned event), and (c) how an event is nondeterministically selected from a set of events.

The definition of matching_events says to first identify the set of matching events for each event spec, then return a set consisting of one matching event for each event spec. Intuitively, a set of events matches a set of event specs if each event spec in the set of event specs matches some event in the set of events. Also, when any event spec in the set matches multiple events, one of the matches is nondeterministically selected to be returned as the match for that event spec.

The definition of matching_event says that if the event and the event spec have the same type, and each subevent spec has a matching subevent, then return an event constructed from the original event's type and timestamp, and the set of matching subevents. Otherwise recursively check each subevent, and if any of them matches the event spec, select one nondeterministically as the return value. Intuitively, an event and an event spec match if either (a) they have the same type and the event's subevents match the event spec's subevent specs, or (b) they have different types but the event has a subevent that matches the event spec.

The definition of select_event nondeterministically selects one event from a set of events.

In the simplest case, when the event has no subevents and the event spec has no subevent specs, only their types are checked. For example, the event $(\texttt{IP\_datagram}, t_1, æ, \emptyset)$, where æ is an application event treated as an opaque object, matches the event spec $(\texttt{IP\_datagram}, \texttt{ip}, \emptyset)$ but does not match the event spec $(\texttt{TCP\_segment}, \texttt{tcp}, \emptyset)$.

Matching is more complex when the event has subevents or the event spec has subevent specs. For example, the event $(\texttt{IP\_datagram}, t_1, æ, \emptyset)$ does not match the event spec

$$(\texttt{IP\_datagram}, \texttt{ip}, \{(\texttt{TCP\_segment}, \texttt{tcp}, \emptyset)\})$$

because the event has no subevent that could match the subevent spec $(\texttt{TCP\_segment}, \texttt{tcp}, \emptyset)$. An event that does match the given event spec is

$$(\texttt{ETH\_packet}, t_1, æ_1, \{(\texttt{IP\_datagram}, t_1, æ_2, \{(\texttt{TCP\_segment}, t_1, æ_3, \emptyset)\})\})$$

The matching algorithm returns the part of the event that matches, in this case

$$(\texttt{IP\_datagram}, t_1, æ_2, \{(\texttt{TCP\_segment}, t_1, æ_3, \emptyset)\})$$

When a complex event has multiple parts that match an event spec, any one of the matching parts may be returned for use in the event processing algorithm. This affects scenario evolution if the different matching parts have different event field values that lead to different environment updates.

To illustrate more complex event matching, events are depicted using an informal tree representation. For example, consider the two events shown in Figure 2.4. The event spec $(\texttt{TCP}, \texttt{t}, \{(\texttt{RPC}, \texttt{r}, \emptyset)\})$ is matched by the event in

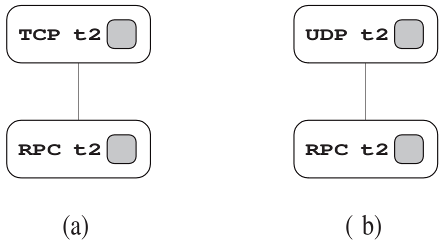

Figure 2.4: Matching complex events.

Figure 2.4 (a), because it has the right type (TCP), and its subevent spec $(\texttt{RPC}, \texttt{r}, \emptyset)$ is matched by the event's only subevent. The event in Figure 2.4 (b) does not match the given event spec, because the event's type is not TCP. The event spec $(\texttt{RPC}, \texttt{r}, \emptyset)$ is matched by either of the events in Figure 2.4 according to the second part of the definition of event_match: each event has a subevent with type RPC, which matches the subevent's type.

Because subevents are unordered (they form a set), matching may be ambiguous if there is more than one instance of any event type anywhere in the event "tree". That is, the matching algorithm does not define which subevents are tried first, so when several subtrees of the event tree match, one of those subtrees is selected nondeterministically.

This is important because a transition's codeblock will typically modify the global and/or local environments with values from the matched event. When multiple matches are possible, the event names in transition assertions and codeblocks refer just to the parts of the selected matching event. If the application events within the STAT events differ, then different results are possible.

To illustrate, consider the event depicted in Figure 2.5 and the event spec $(\texttt{A}, \texttt{a}, \{(\texttt{B}, \texttt{b}, \emptyset), (\texttt{D}, \texttt{d}, \emptyset)\})$. The event contains two subevents and one subsubevent that match this event spec. The potential matches are labeled 1, 2, and 3. The definition of matching_event uses select_event, which selects one of the matches nondeterministically. To see how ambiguity in matching can lead to different results, suppose event type A has an integer field port, and that the value of port in the three potentially matching A event instances is 80, 25, and 80, respectively. Further suppose the A event type has an integer field length, and suppose one of the two (sub)events with port equal 80 has length equal 256 and the other 512. With these conditions, if the codeblock has a statement like "total_length += length", where total_length is an instance (local) variable, then the local environment will differ after the transition fires, depending on which A subevent matched first.

2.3.4 Event streams and synthetic events

An *event stream* ε is a finite or infinite sequence $\langle \epsilon_1 \ldots \epsilon_j \ldots \rangle$ of STAT events satisfying the property $\epsilon_j.\text{time} \leq \epsilon_{j+1}.\text{time}$ for all $0 < j < \text{length}(\varepsilon)$. That is, the events in an event stream are in timestamp order. The *current time* t_c with respect to event stream ε is defined as $\epsilon_c.\text{time}$, where ϵ_c is the *current event*. The current event is the event being processed by a sensor using the algorithm defined in Figure 2.2 on page 19. When a sensor is "between" events

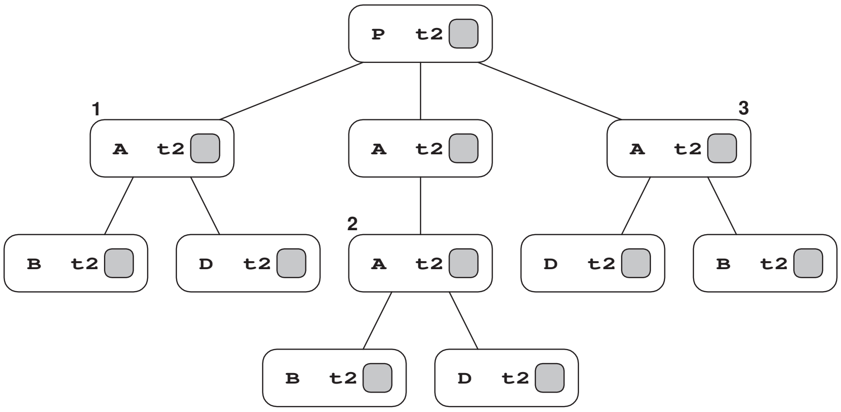

Figure 2.5: Ambiguous matching.

the current event is by definition the event most recently processed. Before any events have been processed the current event is undefined, and the current time is therefore also undefined.

A *synthetic event* is an event generated by a codeblock using the `new` operator. Synthetic events are inserted into an event stream using the `enqueue_event` procedure, which maintains the time-ordered property of the event stream with respect to the part of the event stream that has not yet been processed. That is, given event stream ε with current event ϵ_c, the new event stream ε' that results from inserting synthetic event ϵ_s is defined as follows:

$$\varepsilon' = \begin{cases} \langle \ldots \epsilon_c \epsilon_s \ldots \rangle & \text{if } \epsilon_s.\text{time} \leq \epsilon_c.\text{time} \\ \langle \ldots \epsilon_s \epsilon_k \ldots \rangle & \text{otherwise, where } \epsilon_{k-1}.\text{time} < \epsilon_s.\text{time} \leq \epsilon_k.\text{time} \end{cases}$$

That is, the inserted event will be the "next" event if its timestamp is the same as or before the current time, otherwise it will be inserted after all events with earlier timestamps and before all events with the same or later timestamp.

For example, the codeblock

```
{
  HALFOPENTCP e;
  e = new HALFOPENTCP(attacker_addr, attacker_port, victim_addr,
                      victim_port, 0);
  enqueue_event(e, HALFOPENTCP, 0);
}
```

constructs a `HALFOPENTCP` event and inserts it into the event stream with timestamp t_c, where t_c is the current time (i.e., the timestamp on the event being processed), so the new `HALFOPENTCP` event will be the next event processed, after processing of the current event completes.

The definition of an application- or domain-specific event type must include a constructor to enable a scenario writer to create synthetic events of that type. The `enqueue_event` procedure takes a reference to an event, an event type, and a timestamp, and inserts a STAT event with the given type, timestamp, and application event into the application's event stream.

2.4 Applying the semantics to an example

This section illustrates the preceding definitions of STATL semantics by simulating the execution of a small scenario applied to a short event stream. The scenario uses three simple event types, `E1`, `E2`, and `E3`, each of which has a single attribute of interest, `color`. The scenario matches an `E1` event followed within `timeout` seconds (default 30) by an `E2` event and an `E3` event, all having the same color. A summary state transition diagram for the scenario is in Figure 2.6. The complete scenario follows in Figure 2.7.

The transition named `t1` matches `E1` events and it is nonconsuming, which means that any number of `E1` events can be awaiting matching `E2` events. Transition `t2` matches `E2` events and it is consuming, which means that at most

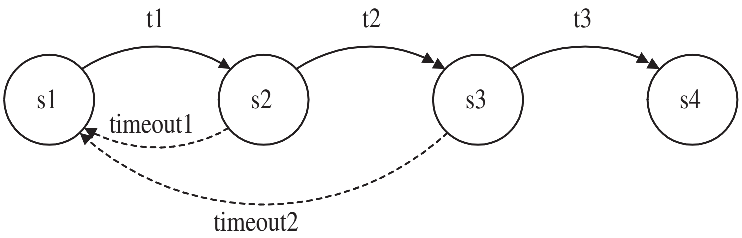

Figure 2.6: STD for the scenario to be executed.

```
scenario example ( int timeout = 30 )
{
  timer timer1;
  int color;
  global int count;

  initial
  state s1 { }

  transition t1 (s1->s2) nonconsuming
  {
    [e E1]
    { color = e.color; timer_start(t1, timeout); }
  }

  state s2 { }

  transition t2 (s2->s3) consuming
  { [e E2] : e.color == color }

  state s3 { }

  transition t3 (s3->s4) consuming
  { [e E3] : e.color == color }

  state s4
  { ++count; log("got %d matches", count); }

  transition timeout1 (s2->s1) unwinding
  { [timer timer1] }

  transition timeout2 (s3->s1) unwinding
  { [timer timer1] }
}
```

Figure 2.7: STATL for the scenario to be executed.

one `E2` event will lead to state `s3` for each `E1` event. Transition `t3` matches `E3` events and it also is consuming, which means that at most one `E3` event will lead to state `s4` for each `E1` event. The codeblock attached to state `s4` increments a global counter and logs the count. If an `E1` event is seen but no matching $\langle$`E2`,`E3`$\rangle$ event sequence appears before the 30-second timer expires, then transitions `timeout1` and `timeout2` unwind the instance back to state `s1`.

The event stream to be processed is $\varepsilon = \langle \epsilon_1 \epsilon_2 \epsilon_3 \epsilon_4 \epsilon_5 \epsilon_6 \rangle$ where

$$
\begin{aligned}
\epsilon_1 &= \mathtt{E1}(0, \mathrm{red}) \\
\epsilon_2 &= \mathtt{E1}(4, \mathrm{red}) \\
\epsilon_3 &= \mathtt{E1}(7, \mathrm{blue}) \\
\epsilon_4 &= \mathtt{E2}(10, \mathrm{red})
\end{aligned}
$$

$$\begin{aligned}\epsilon_5 &= \texttt{E3}(33, \text{red}) \\ \epsilon_6 &= \texttt{E2}(35, \text{blue})\end{aligned}$$

Each event is shown with its timestamp and its color. The initial situation is that

$$\texttt{example} = (S, T, \texttt{s1}, I_0, g_0)$$

where S and T are the static sets of states and transitions, respectively, and the dynamic entities, I_0 and g_0, have the following values:

$$\begin{aligned}I_0 &= \{i_0\}, \text{where } i_0 = \langle(\texttt{s1}, \{\texttt{color} : \texttt{undef}\}, \langle\rangle, \texttt{null})\rangle \\ g_0 &= \{\texttt{count} : 0\}\end{aligned}$$

The initial instance set contains a single instance, which consists of a single snapshot in the scenario's initial state with an empty timer queue. The global timer queue is not shown because there are no global timers in the example scenario. The rest of this section shows how I and g evolve from I_0 and g_0 as the event stream ε is processed by the algorithm in Figure 2.2 on page 19.

Processing event ϵ_1 at time 0 The first step in processing an event is to initialize the auxiliary set F to be empty and the new global environment with the value of the old global environment, in this case

$$g_1 = g_0 = \{\texttt{count} : 0\}$$

I_0 contains only one instance, i_0, so the first loop (**foreach** $i \in I$) is executed exactly once. The inner loop (**foreach** $t \in \text{enabled}(T, i)$) also is executed once, because $\text{enabled}(T, i_0)$ is the set $\{\texttt{t1}\}$. The event spec of `t1` is `[E1 e]`. The call to $\text{matching\_events}(\{\epsilon_1\}, [\texttt{E1 e}])$ returns $\{\epsilon_1\}$, because the event and the event spec have type `E1`. The call to $\text{select\_satisfying\_event}(\{\epsilon_1\}, i_0, \texttt{t1}, g_0)$ returns ϵ_1 because ϵ_1 is a satisfying event (neither the transition `t1` nor the destination state `s2` has an assertion, so they are implicitly true) and it is the only satisfying event. The tuple $(\texttt{t1}, i_0, \epsilon_1)$ is therefore inserted into F for use in the rest of the algorithm. There was only one transition to try in the inner loop and only one instance to try in the outer loop, so both loops are exited with

$$F = \{(\texttt{t1}, i_0, \epsilon_1)\}$$

That is, transition `t1` should be fired in instance i_0 with event ϵ_1.

The next step of the algorithm is to initialize I', which will be called I_1 while processing ϵ_1. There are no instances in I_0 that are not in F, so I_1 is initially empty:

$$I_1 = \emptyset$$

Since `t1` is a nonconsuming transition, the next step is to call $\text{fire\_nonconsuming\_transition}(\texttt{t1}, i_0, \epsilon_1, I_1, g_1)$. This executes the codeblock of transition `t1` (and would execute the codeblock of state `s2` if `s2` had a codeblock). Following the rules in sections 2.3.1.1 and 2.3.2.4, I_1 and g_1 are computed as follows:

$$\begin{aligned}I_1 &= I_1 \cup \{i_0, i_1\} \text{ where } i_1 \text{ is a new instance defined below} \\ &= \{i_0, i_1\} \\ g_1 &= \text{exec}(\text{null}, \text{null}, \text{exec}(\texttt{t1}.\text{code}, \epsilon_1, (g_1, i_0.\text{last.env}), (q', i_0.\text{last.q}))).\text{maps.first} \\ &= \text{exec}(\texttt{t1}.\text{code}, \epsilon_1, (g_1, i_0.\text{last.env}), (\langle\rangle, \langle\rangle))).\text{maps.first} \\ &= \text{exec}(\{\texttt{color = e.color; timer\_start(timer1,30);}\}, \epsilon_1, (g_1, i_0.\text{last.env}), (\langle\rangle, \langle\rangle))).\text{maps.first} \\ &= \text{exec}(\{\texttt{timer\_start(timer1,30)}\}, \epsilon_1,\end{aligned}$$

$$
\begin{aligned}
& \operatorname{exec}(\{\texttt{color = e.color}\}, \epsilon_1, (g_1, \{\texttt{color}: \text{undef}\}), (\langle\rangle, \langle\rangle))).\text{maps.first} \\
= \ & \operatorname{exec}(\{\texttt{timer\_start(timer1, 30)}\}, \epsilon_1, (g_1, \{\texttt{color}: \text{red}\}), (\langle\rangle, \langle\rangle)).\text{maps.first} \\
= \ & ((g_1, \{\texttt{color}: \text{red}\}), (\langle\rangle, \langle\tau_1\rangle)).\text{maps.first} \\
= \ & g_1 \\
= \ & \{\texttt{count}: 0\}
\end{aligned}
$$

The computation of the new value of g_1 proceeds by plugging values for the current situation into the formula for g' in section 2.3.1.1. Next the rule for sequential statements in section 2.3.2.4 is applied, producing a nested exec call. Then the inner exec function is applied to the `color` assignment, which changes the local environment. Next the exec function is applied to the `timer_start` call, which doesn't change either environment but does generate a timeout event $\tau_1 = (\texttt{timer1}, 30)$, which is inserted into the timer queue of instance i_1 (see definition of i_1 just below). Finally, the first element of the resulting name-value mapping list is returned as the new value of g_1. It happens to be the same as the old value of g_1 because the codeblock for transition `t1` does not modify the global environment.

The value of i_1 is, by definition, a copy of i_0 with a new snapshot σ_1 appended, where σ_1 is constructed according to section 2.3.1.1 as follows:

$$
\begin{aligned}
\sigma_1.\text{state} &= \texttt{s2} \\
\sigma_1.\text{env} &= ((g_1, \{\texttt{color}: \text{red}\}), (\langle\rangle, \langle\tau_1\rangle)).\text{maps.second} \\
&= \{\texttt{color}: \text{red}\} \\
\sigma_1.\text{q} &= ((g_1, \{\texttt{color}: \text{red}\}), (\langle\rangle, \langle\tau_1\rangle)).\text{queues.second} \\
&= \langle\tau_1\rangle \\
\sigma_1.\text{event} &= \epsilon_1
\end{aligned}
$$

The value for σ_1.env is computed as for g_1 above, but the second element of the mapping list is returned instead of the first. The value for σ_1.q is computed as for σ_1.env, but the second element of the list of timer queues is returned instead of the second element of the mapping list.

Therefore, the constructed value of i_1 is as follows:

$$
i_1 = \langle(\texttt{s1}, \{\texttt{color}: \text{undef}\}, \langle\rangle, \texttt{null}), (\texttt{s2}, \{\texttt{color}: \text{red}\}, \langle\tau_1\rangle, \epsilon_1)\rangle
$$

Summarizing, firing transition `t1` in instance i_0 leaves the global environment unchanged, and the instance set has a new instance i_1 that is a copy of i_0 with a new snapshot appended, which has a new timeout event queued. F contained only one transition to fire, so the values of I_1 and g_1 just computed are the starting values for processing the next event.

Processing of ϵ_1 is completed, and ϵ_2 becomes the current event (because it is the next event in the event stream and its timestamp is less than that of any queued timeout).

Processing event ϵ_2 at time 4 The algorithm is initialized with F empty and $g_2 = g_1 = \{\texttt{count} : 0\}$. There are two instances, i_0 and i_1, in I_1. Each instance will be used in the algorithm's first loop to identify transitions that should fire. Consider i_0 first. Since $\epsilon_2 = \texttt{E1}(2, \text{red})$ has type `E1`, its initial processing with respect to i_0 is the same as for ϵ_1. Therefore, $(\texttt{t1}, i_0, \epsilon_2)$ is inserted into F. Next consider i_1. The state of i_1 is `s2`, so enabled(T, i_1) is $\{\texttt{t2}, \texttt{timeout1}\}$. Since the event type in ϵ_2, `E1`, does not match the event type in `t2`'s event spec, `E2`, no transitions are to fire in instance i_1, so no additional changes are made to F. After processing both instances in I_1, the set of transitions to be fired is as follows:

$$
F = \{(\texttt{t1}, i_0, \epsilon_2)\}
$$

The new instance set, I_2, is initialized to the set of instances in the old instance, I_1, that have no transitions to be fired:

$$
I_2 = \{i_1\}
$$

The rest of the processing is the same as for ϵ_1: the global environment is unchanged, the original instance i_0 is added to the new instance set, and a new instance i_2 also is added to the instance set:

$$\begin{aligned} I_2 &= \{i_0, i_1, i_2\} \\ g_2 &= g_2 \\ &= \{\texttt{count} : 0\} \end{aligned}$$

where

$$i_2 = \langle(\texttt{s1}, \{\texttt{color} : \texttt{undef}\}, \langle\rangle, \texttt{null}), (\texttt{s2}, \{\texttt{color} : \text{red}\}, \langle\tau_2\rangle, \epsilon_2)\rangle$$

where a new timeout event $\tau_2 = (\texttt{timer1}, 34)$ has been generated and inserted into instance i_2's previously empty timer queue.

The values of I_2 and g_2 just computed are the starting values for processing the next event.

Processing event ϵ_3 at time 7 The next event in the event stream is ϵ_3, which is to be processed with respect to I_2 and g_2. Again the algorithm is initialized by setting F to empty and g_3 to the global environment's previous value, g_2. The first step is again to collect transitions that should be fired, by checking each enabled transition of each instance in I_2. I_2 contains three instances, i_0, i_1, and i_2.

Since $\epsilon_3 = \texttt{E1}(7, \text{blue})$ has type $\texttt{E1}$, its processing with respect to i_0 is the same as for ϵ_1. Therefore, $(\texttt{t1}, i_0, \epsilon_2)$ is inserted into F. Next consider i_1. The state of i_1 is $\texttt{s2}$, so $\text{enabled}(T, i_1)$ is $\{\texttt{t2}, \texttt{timeout1}\}$. Since the event type in ϵ_2, $\texttt{E1}$, does not match the event type in $\texttt{t2}$'s event spec, $\texttt{E2}$, no transitions are to fire in instance i_1, so no additional changes are made to F. Finally, i_2 is similar to i_1 (its event spec doesn't match the event type of ϵ_3), so no additional changes are made to F. After processing all instances, the set of transitions to be fired is as follows:

$$F = \{(\texttt{t1}, i_0, \epsilon_3)\}$$

The new instance set, I_3, is initialized to the set of instances in the old instance, I_2, that have no transitions to be fired:

$$I_3 = \{i_1, i_2\}$$

The rest of the processing is the same as for ϵ_2, so again the global environment is unchanged, the instance i_0 is copied into the new instance set, and a new instance i_3 is added to the new instance set:

$$\begin{aligned} I_3 &= \{i_0, i_1, i_2, i_3\} \\ g_3 &= g_3 \\ &= \{\texttt{count} : 0\} \end{aligned}$$

where

$$i_3 = \langle(\texttt{s1}, \{\texttt{color} : \texttt{undef}\}, \langle\rangle, \texttt{null}), (\texttt{s2}, \{\texttt{color} : \text{blue}\}, \langle\tau_3\rangle, \epsilon_3)\rangle$$

where a new timeout event $\tau_3 = (\texttt{timer1}, 37)$ has been generated and inserted into instance i_3's previously empty timer queue.

The values of I_3 and g_3 just computed are the starting values for processing the next event:

$$\begin{aligned} I_3 &= \{i_0, i_1, i_2, i_3\} \\ &= \{\langle(\texttt{s1}, \{\texttt{color} : \texttt{undef}\}, \langle\rangle, \texttt{null})\rangle, \\ &\quad \langle(\texttt{s1}, \{\texttt{color} : \texttt{undef}\}, \langle\rangle, \texttt{null}), (\texttt{s2}, \{\texttt{color} : \text{red}\}, \langle\tau_1\rangle, \epsilon_1)\rangle, \\ &\quad \langle(\texttt{s1}, \{\texttt{color} : \texttt{undef}\}, \langle\rangle, \texttt{null}), (\texttt{s2}, \{\texttt{color} : \text{red}\}, \langle\tau_2\rangle, \epsilon_2)\rangle, \\ &\quad \langle(\texttt{s1}, \{\texttt{color} : \texttt{undef}\}, \langle\rangle, \texttt{null}), (\texttt{s2}, \{\texttt{color} : \text{blue}\}, \langle\tau_3\rangle, \epsilon_3)\rangle\} \\ g_3 &= \{\texttt{count} : 0\} \end{aligned}$$

where the three queued timer events are $\tau_1 = (\texttt{timer1}, 30)$ in instance i_1, $\tau_2 = (\texttt{timer1}, 34)$ in instance i_2, $\tau_3 = (\texttt{timer1}, 37)$ in instance i_3.

Processing event ϵ_4 at time 10 The next event in the event stream is $\epsilon_4 = \texttt{E2}(10, \mathrm{red})$, which is to be processed with respect to I_3 and g_3. Again the algorithm is initialized by setting F to empty and g_4 to the global environment's previous value, g_3. The first step is again to collect transitions that should be fired, by checking each enabled transition of each instance in I_3. I_3 contains four instances, i_0, i_1, i_2, and i_3. The event type in ϵ_4 is E2. Instance i_0 has been considered in the preceding steps, where it was noted that the only enabled transition, t1, has an event spec with event type E1. This does not match E2, so no transitions should fire in i_0.

Instances i_1, i_2, and i_3 are all in state s2, which has two enabled transitions: t2 and timeout1. The latter has a timer event spec, so it cannot match an E2 event. The event type of t2 is the same as the event's event type, and the assertion attached to t2 is satisfied for instances i_1 and i_2 (instance i_3 has color blue, not red, in the local environment of its last snapshot). Therefore after the top loop F contains two tuples:

$$F = \{(\texttt{t2}, i_1, \epsilon_4), (\texttt{t2}, i_2, \epsilon_4)\}$$

The new instance set, I_4, is initialized to the set of instances in the old instance, I_3, that have no transitions to be fired:

$$I_4 = \{i_0, i_3\}$$

The next step is to fire the transitions in F: consuming transition t2 in instances i_1 and i_2. Following the rules in sections 2.3.1.2 and 2.3.2.4, I_4 and g_4 are computed as follows for i_1:

$$\begin{aligned} I_4 &= I_4 \cup \{i_4\} \text{ where } i_4 \text{ is defined below} \\ &= \{i_0, i_3, i_4\} \\ g_4 &= g_3 \\ &= \{\texttt{count} : 0\} \end{aligned}$$

where

$$i_4 = \langle(\texttt{s1}, \{\texttt{color} : \texttt{undef}\}, \langle\rangle, \texttt{null}), (\texttt{s2}, \{\texttt{color} : \mathrm{red}\}, \langle\tau_1\rangle, \epsilon_1), (\texttt{s3}, \{\texttt{color} : \mathrm{red}\}, \langle\tau_1\rangle, \epsilon_4)\rangle$$

That is, i_4 is i_1 (not a copy of i_1) with a new snapshot containing the new destination state s3 and the activating event ϵ_4.

Next, I_4 and g_4 are updated as follows for i_2:

$$\begin{aligned} I_4 &= I_4 \cup \{i_5\} \text{ where } i_5 \text{ is defined below} \\ &= \{i_0, i_3, i_4, i_5\} \\ g_4 &= g_4 \\ &= \{\texttt{count} : 0\} \end{aligned}$$

where

$$i_5 = \langle(\texttt{s1}, \{\texttt{color} : \texttt{undef}\}, \langle\rangle, \texttt{null}), (\texttt{s2}, \{\texttt{color} : \mathrm{red}\}, \langle\tau_2\rangle, \epsilon_2), (\texttt{s3}, \{\texttt{color} : \mathrm{red}\}, \langle\tau_2\rangle, \epsilon_4)\rangle$$

That is, i_5 is i_2 (not a copy of i_2) with a new snapshot containing the new destination state s3 and the activating event ϵ_4.

Summarizing, two instances progressed from state s2 to state s3 via consuming transition t2, so the new instances i_4 and i_5 replace old instances i_1 and i_2.

The next event in the event queue, ϵ_5, has a timestamp of 33. There are three queued timeout events, one of which has an earlier timestamp: τ_1 with timestamp 30. Therefore the next event to be processed is τ_1, with respect to I_4 and g_4:

$$
\begin{aligned}
I_4 &= \{i_0, i_3, i_4, i_5\} \\
&= \{\langle(\texttt{s1}, \{\texttt{color}:\text{undef}\}, \langle\rangle, \texttt{null})\rangle, \\
&\quad \langle(\texttt{s1}, \{\texttt{color}:\text{undef}\}, \langle\rangle, \texttt{null}), (\texttt{s2}, \{\texttt{color}:\text{blue}\}, \langle\tau_3\rangle, \epsilon_3)\rangle, \\
&\quad \langle(\texttt{s1}, \{\texttt{color}:\text{undef}\}, \langle\rangle, \texttt{null}), (\texttt{s2}, \{\texttt{color}:\text{red}\}, \langle\tau_1\rangle, \epsilon_1), (\texttt{s3}, \{\texttt{color}:\text{red}\}, \langle\tau_1\rangle, \epsilon_4)\rangle, \\
&\quad \langle(\texttt{s1}, \{\texttt{color}:\text{undef}\}, \langle\rangle, \texttt{null}), (\texttt{s2}, \{\texttt{color}:\text{red}\}, \langle\tau_2\rangle, \epsilon_2), (\texttt{s3}, \{\texttt{color}:\text{red}\}, \langle\tau_2\rangle, \epsilon_4)\rangle\} \\
g_4 &= \{\texttt{count}:0\}
\end{aligned}
$$

where the three queued timer events are $\tau_1 = (\texttt{timer1}, 30)$ in instance i_4, $\tau_2 = (\texttt{timer1}, 34)$ in instance i_5, and $\tau_3 = (\texttt{timer1}, 37)$ in instance i_3.

Processing timeout event τ_1 at time 30 The current event is $\tau_1 = (\texttt{timer1}, 30)$, which is to be processed with respect to I_4 and g_4.[14] Again the algorithm is initialized by setting F to empty and g_5 to the global environment's previous value, g_4. The first step is again to collect transitions that should be fired, by checking each enabled transition of each instance in I_4. I_4 contains four instances, i_0, i_3, i_4, and i_5. Since τ_1 is a local timer event queued in instance i_4, only transitions in i_4 can match. Therefore only i_4 need be considered. Instance i_4 has an enabled transition named `timeout` that has the right label, `timer1`. F is therefore $\{(\texttt{timeout}, i_4, \tau_1)\}$. As usual, I_5 is initialized with the subset of instances in I_4 that have no transitions to be fired:

$$I_5 = \{i_0, i_3, i_5\}$$

The next step is to fire the transitions in F: unwinding transition `timeout` in instance i_4. Following the rules in sections 2.3.1.3 and 2.3.2.4, I_5 and g_5 are computed as follows for i_4:

$$
\begin{aligned}
p &= \langle(\texttt{s1}, \{\texttt{color}:\text{undef}\}, \langle\rangle, \texttt{null}), (\texttt{s2}, \{\texttt{color}:\text{red}\}, \langle\tau_1\rangle, \epsilon_1)\rangle \\
I_5 &= (I_5 \setminus \{i' \in I_5 \ni p \in i'\}) \cup \{\langle(\texttt{s1}, \{\texttt{color}:\text{undef}\}, \langle\rangle, \texttt{null})\rangle\} \\
&= \{i_0, i_3, i_5\} \\
g_5 &= g_4 \\
&= \{\texttt{count}:0\}
\end{aligned}
$$

The next event in the event queue, ϵ_5, has a timestamp of 33. The earliest timestamp on any queued timeout event is 34 (on τ_2 in instance i_5). Therefore the next event to be processed is ϵ_5, with respect to I_5 and g_5:

$$
\begin{aligned}
I_5 &= \{i_0, i_3, i_5\} \\
&= \{\langle(\texttt{s1}, \{\texttt{color}:\text{undef}\}, \langle\rangle, \texttt{null})\rangle, \\
&\quad \langle(\texttt{s1}, \{\texttt{color}:\text{undef}\}, \langle\rangle, \texttt{null}), (\texttt{s2}, \{\texttt{color}:\text{blue}\}, \langle\tau_3\rangle, \epsilon_3)\rangle, \\
&\quad \langle(\texttt{s1}, \{\texttt{color}:\text{undef}\}, \langle\rangle, \texttt{null}), (\texttt{s2}, \{\texttt{color}:\text{red}\}, \langle\tau_2\rangle, \epsilon_2), (\texttt{s3}, \{\texttt{color}:\text{red}\}, \langle\tau_2\rangle, \epsilon_4)\rangle\} \\
g_5 &= \{\texttt{count}:0\}
\end{aligned}
$$

where the two queued timer events are $\tau_2 = (\texttt{timer1}, 34)$ in instance i_5 and $\tau_3 = (\texttt{timer1}, 37)$ in instance i_3.

Processing event ϵ_5 at time 33 The next event in the event stream is $\epsilon_5 = \texttt{E3}(33, red)$, which is to be processed with respect to I_5 and g_5. Again the algorithm is initialized by setting F to empty and g_6 to the global environment's previous value, g_5. The first step is again to collect transitions that should be fired, by checking each enabled transition of each instance in I_5. I_5 contains three instances, i_0, i_3, and i_5. The event type in ϵ_5 is `E3`. Instance i_0 has been considered in the preceding steps, where it was noted that the only enabled transition, `t1`, has an event spec with

[14] When τ_1 becomes the current event, it is deleted from i_4's timer queue, which becomes empty.

event type `E1`. This does not match `E3`, so no transitions should fire in i_0. Instance i_3 is in state `s2`, which has two enabled transitions: `t2` and `timeout1`. The latter has a timer event spec, so it cannot match an `E3` event. The event type of `t2` is `E2`, which doesn't match the event's event type (`E3`). Instance i_5 is in state `s3`, which has two enabled transitions: `t3` and `timeout2`. The event type of `t3` is the same as the event's event type, and the assertion attached to `t3` is satisfied for instance i_5. Therefore after the top loop F contains one tuple:

$$F = \{(\texttt{t3}, i_5, \epsilon_5)\}$$

The new instance set, I_6, is initialized to the set of instances in the old instance, I_5, that have no transitions to be fired:

$$I_6 = \{i_0, i_3\}$$

The next step is to fire the transitions in F: consuming transition `t3` in instance i_5. Following the rules in sections 2.3.1.2 and 2.3.2.4, I_6 and g_6 are computed as follows for i_5:

$$\begin{aligned}
I_6 &= \{i_0, i_3\} \\
g_6 &= \text{exec}(\texttt{s4}.\text{code}, \text{null}, \text{exec}(\text{null}, \epsilon_5, (g_6, i_5.\text{last.env}), (q', i_5.\text{last.q}))).\text{maps.first} \\
&= \text{exec}(\texttt{s4}.\text{code}, \text{null}, (g_6, i_5.\text{last.env}), (\langle\rangle, \langle\tau_2\rangle))).\text{maps.first} \\
&= \text{exec}(\{\texttt{++count; log(...);}\}, \text{null}, \text{null}, (\{\texttt{count} : 0\}, i_5.\text{last.env}), (\langle\rangle, \langle\tau_2\rangle))).\text{maps.first} \\
&= \text{exec}(\{\texttt{log(...)}\}, \text{null}, \\
&\qquad \text{exec}(\{\texttt{++count}\}, \text{null}, (\{\texttt{count} : 0\}, i_5.\text{last.env}), (\langle\rangle, \langle\tau_2\rangle)))).\text{maps.first} \\
&= \text{exec}(\{\texttt{log(...)}\}, \text{null}, (\{\texttt{count} : 1\}, i_5.\text{last.env}), (\langle\rangle, \langle\tau_2\rangle))).\text{maps.first} \\
&= ((\{\texttt{count} : 1\}, i_5.\text{last.env}), (\langle\rangle, \langle\tau_2\rangle))).\text{maps.first} \\
&= \{\texttt{count} : 1\}
\end{aligned}$$

The instance set I_6 is not changed by firing `t3` in instance i_5 because `s4` is a final state.

Summarizing, one instance, i_5, progressed to its final state, so it is no longer in the instance set (and its queued timer event, τ_2, disappeared with it). The global variable `count` was incremented.

The next event in the event stream is ϵ_6 with timestamp 35. This is earlier than the timestamp of the only remaining queued timeout event (τ_3 with timestamp 37), so ϵ_6 is processed next, in the context of I_6 and g_6:

$$\begin{aligned}
I_6 &= \{i_0, i_3\} \\
&= \{\langle(\texttt{s1}, \{\texttt{color} : \text{undef}\}, \langle\rangle, \texttt{null})\rangle, \\
&\qquad \langle(\texttt{s1}, \{\texttt{color} : \text{undef}\}, \langle\rangle, \texttt{null}), (\texttt{s2}, \{\texttt{color} : \text{blue}\}, \langle\tau_3\rangle, \epsilon_3)\rangle\} \\
g_6 &= \{\texttt{count} : 1\}
\end{aligned}$$

where the queued timer event is $\tau_3 = (\texttt{timer1}, 37)$ in instance i_3.

Processing event ϵ_6 at time 35 The next event, $\epsilon_6 = \texttt{E2}(35, \text{blue})$, is another `E2` event, so it is processed like the previous `E2` event, ϵ_4. Again the algorithm is initialized by setting F to empty and g_7 to the global environment's previous value, g_6. The first step is again to collect transitions that should be fired, by checking each enabled transition of each instance in I_6. I_6 contains two instances, i_0 and i_3. Instance i_0 has been considered in the preceding steps, where it was noted that the only enabled transition, `t1`, has an event spec with event type `E1`. This does not match `E2`, so no transitions should fire in i_0.

Instance i_3 is in state `s2`, which has two enabled transitions: `t2` and `timeout1`. The latter has a timer event spec, so it cannot match an `E2` event. The event type of `t2` is the same as the event's event type, and the assertion attached to `t2` is satisfied for instance i_3 . Therefore after the top loop F contains one tuple:

$$F = \{(\texttt{t2}, i_3, \epsilon_6)\}$$

The new instance set, I_7, is initialized to the set of instances in the old instance, I_6, that have no transitions to be fired:

$$I_7 = \{i_0\}$$

The next step is to fire the transitions in F: consuming transition `t2` in instance i_3. Following the rules in sections 2.3.1.2 and 2.3.2.4, I_7 and g_7 are computed as follows for i_3:

$$\begin{aligned} I_7 &= I_7 \cup \{i_6\} \text{ where } i_6 \text{ is defined below} \\ &= \{i_0, i_6\} \\ g_7 &= g_6 \\ &= \{\texttt{count} : 1\} \end{aligned}$$

where

$$i_6 = \langle(\texttt{s1}, \{\texttt{color} : \texttt{undef}\}, \langle\rangle, \texttt{null}), (\texttt{s2}, \{\texttt{color} : \text{blue}\}, \langle\tau_3\rangle, \epsilon_3), (\texttt{s3}, \{\texttt{color} : \text{blue}\}, \langle\tau_3\rangle, \epsilon_6)\rangle$$

That is, i_6 is i_3 (not a copy of i_3) with a new snapshot containing the new destination state `s3` and the activating event ϵ_6.

Summarizing, one instance progressed from state `s2` to state `s3` via consuming transition `t2`, so the new instance i_6 replaces old instance i_3. The global environment is unchanged.

There are no events remaining in the event queue, but there is still one queued timeout event: $\tau_3 = (\texttt{timer1}, 37)$ in instance i_6:

$$\begin{aligned} I_7 &= \{i_0, i_6\} \\ &= \{\langle(\texttt{s1}, \{\texttt{color} : \texttt{undef}\}, \langle\rangle, \texttt{null})\rangle, \\ &\quad \langle(\texttt{s1}, \{\texttt{color} : \texttt{undef}\}, \langle\rangle, \texttt{null}), (\texttt{s2}, \{\texttt{color} : \text{blue}\}, \langle\tau_3\rangle, \epsilon_3), (\texttt{s3}, \{\texttt{color} : \text{blue}\}, \langle\tau_3\rangle, \epsilon_6)\rangle \\ g_7 &= \{\texttt{count} : 1\} \end{aligned}$$

where the queued timer event is $\tau_3 = (\texttt{timer1}, 37)$ in instance i_6.

Processing timeout event τ_3 at time 37 Instance i_6 has a queued timeout event: $\tau_3 = (\texttt{timer1}, 37)$. Since it is queued in instance i_6, only transitions in i_6 can match. Transition `timeout2` is enabled in instance i_6 and it is an unwinding transition, so processing τ_3 is similar to processing τ_1 earlier. The only change in the instance set is that i_6 is unwound. The final configuration is:

$$\begin{aligned} I_8 &= \{i_0\} \\ &= \{\langle(\texttt{s1}, \{\texttt{color} : \texttt{undef}\}, \langle\rangle, \texttt{null})\rangle\} \\ g_8 &= \{\texttt{count} : 1\} \end{aligned}$$

and there are no remaining events or queued timeout events to process.

2.5 Semantic extensions

There are two ways to extend the STATL language: *annotations* and *domain libraries*. In both cases the semantic extension is entirely up to the application, and thus beyond the scope of this paper. This section describes only how annotation and library semantics are related to the semantic description given in previous sections.

STATL annotations provide a standard way to extend compile-time semantics by specifying scenario features that are handled by an application-specific preprocessor called the *Analyzer*. The Analyzer's task is to perform any required IDS-specific preprocessing on a STATL scenario before the scenario is actually translated into an executable representation. For example, the NetSTAT Analyzer uses the annotations of a STATL scenario to determine where a probe should be placed in a target network and how the probe should be configured in order to be sure to detect the described attack. The Analyzer relies on another component, the *Factbase*, to obtain information about the target environment. For example, in NetSTAT the Factbase contains information about the network topology and the services deployed. Each IDS defines any scenario customization it needs as annotations, and the associated Analyzer is expected to process them (any annotations left by the Analyzer will be ignored by the STATL translator). Since annotations are processed at compile-time, they are not related to the previously defined run-time semantics of STATL at all.

Domain libraries extend run-time semantics by specifying new event types and object types as C++ classes. The new event types may be used in event specs and will presumably appear in the application's event stream, and therefore are used in the matching_events function defined in Section 2.3.3. New object types and new functions that apply to the new event types and object types may be used in assertions and codeblocks. Therefore assertion evaluation and codeblock execution depend on the semantics of domain libraries, as described in Sections 2.3.2.3 and 2.3.2.4.

2.6 Glossary of symbols used in semantic definition

This table lists symbols used in the semantic definition of STATL, identifies where they are defined and used, and describes them.

Symbol	References	Meaning
$\ni$	2.2.2.6	"such that"
$\mathcal{A}_T$	2.2.1.2	domain of transition assertions
a	2.3.2.3	an assertion
Æ	2.2.2.4	domain of application events
æ	2.3.3	an application event
$\mathcal{C}$	2.3.1.1, 2.3.2.4	domain of codeblocks
$\mathcal{C}_S$	2.2.1.1	domain of state codeblocks
$\mathcal{C}_T$	2.2.1.2	domain of transition codeblocks
$\mathcal{E}$	2.2.1.2, 2.2.1.3, 2.3.3	domain of event specs
e	2.2.1.3, 2.3.3	an event spec
E	2.1, 2.2.1.4, 2.2.1.5, 2.2.2.4, 2.2.2.6, 2.3.1.1, 2.3.2.4, 2.3.3	domain of (STAT) events
ϵ	2.1, 2.2.1.5, 2.2.2.4, 2.3, 2.3.1, 2.3.1.1, 2.3.1.2, 2.3.1.3, 2.3.2.1, 2.3.2.2, 2.3.2.3, 2.3.2.4, 2.3.3	an event
ϵ_c	2.3.4	the current event
ε	2.3, 2.3.4	a set of events
$\mathcal{G}$	2.2.1.4, 2.2.1.5, 2.2.2.2	domain of global environments
g	2.2, 2.2.1.5, 2.2.2.2, 2.3, 2.3.1, 2.3.1.1, 2.3.1.2, 2.3.1.3, 2.3.2.3	a global environment
I	2.2, 2.2.2.8, 2.3, 2.3.1, 2.3.1.1, 2.3.1.2, 2.3.1.3	an instance set
$\mathcal{I}$	2.2.2.7, 2.2.2.8	domain of scenario instances
i	2.2.2.7, 2.3, 2.3.1, 2.3.1.1, 2.3.1.2, 2.3.1.3	a scenario instance
$\mathcal{L}$	2.2.1.4, 2.2.1.5, 2.2.2.3, 2.2.2.6	domain of local environments
l	2.2.1.5, 2.2.2.3, 2.3.2.3	a local environment
$\mathcal{M}$	2.2.2.1, 2.2.2.2, 2.2.2.3, 2.3.1.1, 2.3.2.4	domain of name-value mappings
m	2.2.2.1, 2.3.2.1, 2.3.2.2, 2.3.2.4	a name-value mapping
$\mathcal{N}$	2.2.1.1	domain of STATL names
$\mathcal{N}_E$	2.2.1.1, 2.2.1.3	domain of event names
$\mathcal{N}_O$	2.2.1.1, 2.2.2.1	domain of object names
$\mathcal{N}_S$	2.2.1.1, 2.2.1.2, 2.2.2.6	domain of state names
$\mathcal{N}_T$	2.2.1.1, 2.2.1.2	domain of transition names

Symbol	References	Meaning
$\mathcal{Q}$	2.2.2.7	domain of timer queues
q	2.2, 2.2.2	a timer queue
$\mathcal{R}$	2.1	domain of responses
S	2.2, 2.2.1.1, 2.3	a set of states
S	2.1, 2.2, 2.2.1.1, 2.2.1.2, 2.2.2, 2.2.2.2, 2.2.2.7, 2.2.2.8, 2.3, 2.3.1	a scenario
s	2.2.1.1, 2.2.1.4, 2.2.1.5, 2.3.1	a scenario state
Σ	2.1, 2.2.2.6, 2.2.2.7	domain of system states or snapshots
σ	2.1, 2.2.2.6, 2.2.2.7, 2.2.2.8, 2.3.1, 2.3.1.1, 2.3.1.2, 2.3.1.3	a system state or a snapshot
T	2.2, 2.2.1.2, 2.2.2.6, 2.2.2.7, 2.2.2.8, 2.3, 2.3.1	a set of transitions
T	2.2.2.4	domain of times
$\mathcal{T}_E$	2.2.1.3, 2.2.2.4	domain of event types
$\mathcal{T}_T$	2.2.1.2	domain of transition types
t	2.2.1.2, 2.2.1.4, 2.2.1.5, 2.2.2.6, 2.2.2.7, 2.2.2.8, 2.3, 2.3.1, 2.3.1.1, 2.3.1.2, 2.3.1.3, 2.3.3	a scenario transition or a timestamp
t_c	2.3.2.4, 2.3.4	the current time
τ	2.4	a timeout event
$\mathcal{V}$	2.2.2.1	domain of values

Bibliography

[CIS99] CISCO. NetRanger Intrusion Detection System. Technical Information, April 1999.

[Cur01] D. Curry. Intrusion Detection Message Exchange Format: Extensible Markup Language (XML) Document Type Definition. `draft-ietf-idwg-idmef-xml-03.txt`, February 2001.

[GWC98] A.K. Ghosh, J. Wanken, and F. Charron. Detecting Anomalous and Unknown Intrusions Against Programs. In *Proceedings of the Annual Computer Security Application Conference (ACSAC'98)*, Scottsdale, AZ, December 1998.

[IKP95] K. Ilgun, R. A. Kemmerer, and P. A. Porras. State Transition Analysis: A Rule-Based Intrusion Detection System. *IEEE Transactions on Software Engineering*, 21(3), March 1995.

[JV94] H. S. Javitz and A. Valdes. The NIDES Statistical Component Description and Justification. Technical report, SRI International, Menlo Park, CA, March 1994.

[KRL97] C. Ko, M. Ruschitzka, and K. Levitt. Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-based Approach. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 175–187, 1997.

[Lab99] MIT Lincoln Laboratory. DARPA Intrusion Detection Evaluation. http://www.ll.mit.edu/IST/ideval/, 1999.

[Pax98] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, TX, January 1998.

[Rea99] Internet Security Systems. *Introduction to RealSecure Version 3.0*, January 1999.

[VEK00] G. Vigna, S. Eckmann, and R. Kemmerer. The STAT Tool Suite. In *Proceedings of DISCEX 2000*, Hilton Head, South Carolina, January 2000. IEEE Computer Society Press.

[VK99] G. Vigna and R.A. Kemmerer. NetSTAT: A Network-based Intrusion Detection System. *Journal of Computer Security*, 7(1):37–71, 1999.

Appendix A

STATL grammar

This is the grammar used in the STATL parser. It is written for the ANTLR translator generator, which uses the following notation:

- names beginning with uppercase letters are tokens, all other names are nontokens.
- parentheses not followed by `?`, `*`, or `+` indicate grouping.
- optional items are enclosed in '`(`' and '`)?`'.
- items that may appear zero or more times are enclosed in '`(`' and '`)*`'.
- items that may appear one or more times are enclosed in '`(`' and '`)+`'.
- syntactic predicates are indicated by '`=>`', as in:

```
( rule1 ) => rule2
```

  where `rule1` is some syntax fragment that must be matched in order for there to be a match for `rule2`.

```
scenarioList : ( scenario )+ ;

scenario :
        ( "use" IDENT ( COMMA IDENT )* SEMI )*
        "scenario" scenarioId
        ( scenarioParameters )?
        LCURLY
            ( frontMatter )?
            (   state
              | transition
              | namedAction
            )*
        RCURLY
        ( function )*
        ;

scenarioParameters : LPAREN ( parameter ( COMMA parameter )* )?  RPAREN ;

parameter : typeSpec identifier declaratorBrackets ;

frontMatter : ( declaration | annotation )+ ;

state :
        ( "initial" )?
        "state" stateId
```

```
        ( annotation )*
        LCURLY
            ( stateAssertion )?
            ( codeBlock )?
        RCURLY
        ;

transition :
        "transition" transitionId
        transStates
        (   "consuming" | "nonconsuming" | "unwinding" )
        ( annotation )*
        LCURLY
            (   LBRACK evtSpec RBRACK
                ( annotation )*
                ( COLON actionAssertion )?
              | (actionId ( COLON actionAssertion )? )
            )
            ( codeBlock )?
        RCURLY
        ;

transStates : LPAREN stateId ARROW stateId RPAREN ;

codeBlock : compoundStatement ;

evtSpec :
        basicEvtSpec ( subEvtSpec )?
      | timerEvent
        ;

basicEvtSpec : evtType evtId ;

subEvtSpec : LBRACK evtSpec ( COMMA evtSpec )* RBRACK ;

timerEvent : "timer" timerId ;

namedAction :
        "action" actionId
        ( annotation )*
        LCURLY
          LBRACK evtSpec RBRACK
          ( annotation )*
          ( COLON actionAssertion )?
        RCURLY
        ;

stateAssertion : assertion ;

actionAssertion : assertion ;

assertion : logicalOrExpression ;

evtType :
        LPAREN eventId ( BOR eventId )+ RPAREN
      | eventId
        ;

annotation :
        ANNOTATION_START
          annotationId
          ( logicalOrExpression ( COMMA logicalOrExpression )* )?
        ANNOTATION_END
        ;

function : type "function" functionId ( funcParams )? codeBlock ;
```

```
funcParams : LPAREN ( funcParam ( COMMA funcParam )* )?  RPAREN ;

funcParam : typeSpec identifier declaratorBrackets ;

scenarioId : identifier ;

stateId : identifier ;

transitionId : identifier ;

eventId : i:identifier ;

evtId : identifier ;

actionId : identifier ;

timerId : identifier ;

functionId : identifier ;

portId : identifier ;

annotationId : identifier ;

//
// ---------------   DECLARATIONS   ---------------
//

declaration : modifiers typeSpec variableDefinitions SEMI ;

modifiers : ( modifier )* ;

modifier : "global" | "const" ;

variableDefinitions : variableDeclarator ( COMMA variableDeclarator )* ;

variableDeclarator : identifier declaratorBrackets varInitializer ;

declaratorBrackets : ( LBRACK ( NUM_INT )?  RBRACK )?  ;

varInitializer : ( ASSIGN initializer )? ;

arrayInitializer : LCURLY ( initializer ( COMMA initializer )* )?  RCURLY ;

initializer : expression | arrayInitializer ;

//
// ---------------   STATEMENTS   ---------------
//

compoundStatement : LCURLY ( statement )* RCURLY ;

statement :
        compoundStatement
      | "timer_start"
        LPAREN
          identifier COMMA expression COMMA expression
        RPAREN

      | (declaration) => declaration

      | IDENT COLON statement

      | expression SEMI

      | "if" LPAREN expression RPAREN statement
        ( "else" statement )?
```

```
        | "for"
          LPAREN
              (forInit)?        SEMI
              (expression)?     SEMI
              (expressionList)?
          RPAREN
          statement

        | "while" LPAREN expression RPAREN statement

        | "do" statement "while" LPAREN expression RPAREN SEMI

        | "goto" IDENT SEMI

        | "break" SEMI

        | "continue" SEMI

        | "return" (expression)? SEMI

        | "switch" LPAREN expression RPAREN
          LCURLY ( cases )* RCURLY

        | SEMI
          ;

cases : ( ("case" e:expression | "default") COLON )+ ( stmtList )? ;

stmtList: ( statement )+ ;

forInit : (declaration) => declaration | expressionList ;

//
// ---------------   EXPRESSIONS   ---------------
//

expression : assignmentExpression ;

expressionList : expression ( COMMA expression )* ;

assignmentExpression :
        conditionalExpression
        ( (   ASSIGN
            | PLUS_ASSIGN
            | MINUS_ASSIGN
            | STAR_ASSIGN
            | DIV_ASSIGN
            | MOD_ASSIGN
            | SR_ASSIGN
            | SL_ASSIGN
            | BAND_ASSIGN
            | BXOR_ASSIGN
            | BOR_ASSIGN
          )
            assignmentExpression
        )?
        ;

conditionalExpression :
        logicalOrExpression
        ( QUESTION conditionalExpression COLON conditionalExpression )?
        ;

logicalOrExpression :
        logicalAndExpression ( LOR logicalAndExpression )*
        ;
```

```
logicalAndExpression :
        inclusiveOrExpression ( LAND inclusiveOrExpression )*
        ;

inclusiveOrExpression :
        exclusiveOrExpression ( BOR exclusiveOrExpression )*
        ;

exclusiveOrExpression :
        andExpression ( BXOR andExpression )*
        ;

andExpression :
        equalityExpression ( BAND equalityExpression )*
        ;

equalityExpression :
        relationalExpression (( NOT_EQUAL | EQUAL ) relationalExpression )*
        ;

relationalExpression :
        shiftExpression
        ( (   LT
            | GT
            | LE
            | GE
            | "in" )
          shiftExpression
        )*
        ;

shiftExpression :
        additiveExpression (( SL | SR ) additiveExpression )*
        ;

additiveExpression :
        multiplicativeExpression (( PLUS | MINUS ) multiplicativeExpression )*
        ;

multiplicativeExpression :
        castExpression (( STAR | DIV | MOD ) castExpression )*
        ;

castExpression :
        ( LPAREN typeSpec RPAREN castExpression ) =>
            LPAREN typeSpec RPAREN castExpression

      | INC castExpression
      | DEC castExpression
      | MINUS castExpression
      | PLUS castExpression
      | STAR castExpression
      | BAND castExpression
      | BNOT castExpression
      | LNOT castExpression
      | postfixExpression
        ;


postfixExpression :
        primaryExpression
        (  ( DOT IDENT )+
         | ( LBRACK expression RBRACK )+
        )?

        ( ( LPAREN ( expressionList | /*nothing*/ ) RPAREN )+
```

```
          | INC
          | DEC
          | // nothing
        )
        ;

primaryExpression :
        IDENT
      | constant
      | "true"
      | "false"
      | LPAREN expression RPAREN
        ;

constant :
        NUM_INT
      | CHAR_LITERAL
      | STRING_LITERAL
        ;

typeSpec : type ;

type :  ( "list" | "set" | "vector" | "map" ) => containerType
      | simpleType
        ;

simpleType : identifier | builtInType ;

containerType :
        ( "list" | "set" | "vector" ) LT simpleType GT
      | "map" LT simpleType COMMA simpleType GT
        ;

builtInType :
        "char"
      | "short"
      | "int"
      | "long"
      | "int8"
      | "int16"
      | "int32"
      | "u_int8"
      | "u_int16"
      | "u_int32"
      | "string"
      | "timer"
      | "timeval"
        ;

identifier : IDENT ;


//
// ---------------   OPERATORS   ---------------
//

ARROW             :    "->"   ;
QUESTION          :    '?'    ;
LPAREN            :    '('    ;
RPAREN            :    ')'    ;
LBRACK            :    '['    ;
RBRACK            :    ']'    ;
LCURLY            :    ''    ;
RCURLY            :    ''    ;
COLON             :    ':'    ;
COMMA             :    ','    ;
DOT               :    '.'    ;
```

```
ASSIGN             :     '='     ;
EQUAL              :     "=="    ;
LNOT               :     '!'     ;
BNOT               :     '~'     ;
NOT_EQUAL          :     "!="    ;
DIV                :     '/'     ;
DIV_ASSIGN         :     "/="    ;
PLUS               :     '+'     ;
PLUS_ASSIGN        :     "+="    ;
INC                :     "++"    ;
MINUS              :     '-'     ;
MINUS_ASSIGN       :     "-="    ;
DEC                :     "--"    ;
STAR               :     '*'     ;
STAR_ASSIGN        :     "*="    ;
MOD                :     '%'     ;
MOD_ASSIGN         :     "%="    ;
SR                 :     ">>"    ;
SR_ASSIGN          :     ">>="   ;
GE                 :     ">="    ;
GT                 :     ">"     ;
SL                 :     "<<"    ;
SL_ASSIGN          :     "<<="   ;
LE                 :     "<="    ;
LT                 :     '<'     ;
BXOR               :     '^'     ;
BXOR_ASSIGN        :     "^="    ;
BOR                :     '|'     ;
BOR_ASSIGN         :     "|="    ;
LOR                :     "||"    ;
BAND               :     '&'     ;
BAND_ASSIGN        :     "&="    ;
LAND               :     "&&"    ;
SEMI               :     ';'     ;
ANNOTATION_START   :     "<*"    ;
ANNOTATION_END     :     "*>"    ;
```