E^*: A Language for Event Monitoring

Secure Systems Lab, Stony Brook University.

E^* is a language for system monitoring. It is loosely based on the BMSL [4, 7, 5, 1] language. E^* specifications view system behaviors in terms of the sequences of events they generate. Events often (but not necessarily) correspond to function call invocations in the monitored system, with event arguments corresponding to the call's parameters.

E^* is a rule-based language, where the rules are triggered when a runtime system delivers events. Each rule consists of an event pattern, which includes event names and conditions on event arguments for triggering a rule. When a rule is triggered, its right-hand side ("action") is executed. An E^* specification isn't limited to being passive: it can use functions within rule rhs to call back into the monitored system, and these functions may modify that system in any way.

The boundary between specifications and programs is fuzzy. So, E^* ends up being a full programming language on its own. It supports loops, functions and a rich set of types. In particular, all standard primitive types (integers of various widths, booleans, doubles and strings) are supported. In addition, frequently used container types, such as tuples, vectors, sets and dictionaries are supported. Similar to Python (and unlike C++), these types are fully integrated into the language, easing their use. Unlike Python (and like C++), E^* is strongly typed, and supports `const` types.

The E^* compiler (`esc`) compiles E^* source code into C++. E^* specifications are first processed by the C-preprocessor so that they can make use of `#include`'s and macros. Some of the key features of the E^* compiler are:

- *Sublinear rule matching algorithm.* Rule based languages tend to promote a programming style where a large number of rules are used. Often, such languages specify a sequential matching semantics, which means that the matching time will increase linearly with the number of rules. In E^*, we argue that sequential matching is bad in terms of performance as well as understandability. We have developed *condition factorization* [3, 6] algorithms that factorize rule matching so that typically, matching takes time proportional to the number of *successfully matched* rules, rather than *all* of the rules in the program.

- *Performance of C++ without its complexity.* Most constructs in E^* have a straight-forward translation to C++. At the same time, by leaving out many of the complex features of C++, the learning curve for E^* should be less daunting. Of immediate help should be the E^* type checker that provides understandable error messages on complex nested data structures, something that C++ compilers are terrible at.

- *Regular expression support.* Regular expressions are first class entities within the event-matching component of E^*, instead of being treated as strings. This simplifies their specification, e.g., the need for quotes and escaping is greatly reduced. Regular expression matching is integrated into the condition factorization algorithm by factorizing RE matching using regular expressions residues [2].

- *Ease of building and interfacing with runtime systems.* Interfacing to a new runtime system requires the definition of an interface in E^*, specifying event names and their arguments, as well as any support functions provided by the runtime system. The runtime should also implement this interface, and provide a header file that declares these functions [7, 1].

- *Support for anomaly detection.* E^* includes features to support anomaly detection, based on our earlier work on specification-based intrusion detection [5, 4].

1 Types

E^* supports all primitive data types, plus the following additional ones:

- *Enumerated data type* that is similar to the enum type in the C-language.
- *Array*, *tuple*, *set*, *list* and *dictionary* types are the type constructors available in E^* for constructing aggregate data types from primitive types.

  List is implemented using C++ STL vector type. Similarly, tuple, set and dictionary types are mapped to STL's tuple, unordered_set and unordered_map types.

  A restriction in the language is that arrays cannot be used as a parameter to a function or any other entity that takes parameters. This should be considered more of an implementation limitation, and lacks a design rationale. E^* programs use of vectors almost exclusively, since there is no particular advantage to using arrays. So, this restriction is not relevant in practice.
- A *struct* type is used to describe low-level data such as network packets, or structs in C-language. Structs in E^* are more expressive than the C-language, but we omit the details here.
- A *class* type denotes an abstract data type, with the details of the representation made opaque to E^* specifications. Note that there are no methods or other details associated with a class, hence a class declaration specifies just its name. The only way to operate on class types is through *external functions* that are described subsequently in this section.

Other than class types, the remaining types are native to E^*, in the sense that variables of these types can be defined and manipulated in E^*.

1.1 Assignment and parameter passing semantics

Assignment always uses value semantics, i.e., the value of the rhs is copied into the storage that holds the value of the lhs of the assignment. Parameter passing is by value for primitive data types. Aggregate types are passed by reference. A const prefix on an aggregate-type parameter means that a const reference will be passed.

2 Global Declarations

At the top-level, an E^* specification consists of global declarations that have three sections: (a) the external interface used by the E^* specification, (b) user-defined functions in E^*, and (c) module declarations. We describe each of these sections below, with further details appearing in subsequent sections.

2.1 External Interface

An E^* specification starts with a list of global declarations that define the interface used by the specification. An external *runtime system* delivers events to the E^* specification via this interface. Typically, the an interface declaration would be stored in a header file that is then `#include`'d by any E^* specification that uses this interface. Interface declarations can include:

- *events* that can be delivered over the interface, together with their argument types

- *functions* provided by the monitored system. We call these *external functions* to highlight the fact that their definition is outside E^*.
- *class* declarations specify data types that can be accessed via the interface, but as discussed before, the representation itself is to opaque to E^*. Class types can be manipulated only via external functions.
- *struct* declarations specify struct-type data that can be exchanged over the interface.

2.2 E^* Functions

User-defined functions can take zero or more arguments of any valid E^* data type, and can be recursive. However, they cannot take other functions as argument. User-defined functions use the following syntax in E^*:

$$\langle type \rangle \;\; \langle id \rangle (param_1, \ldots, param_n) \;\; [\texttt{const}] \;\; \{\langle stmt \rangle\}$$

Every non-void function must contain a `return` statement. A *const* prefix can be used in a function parameter type declaration to indicate that this function does not modify this parameter. If a function (a) does not change any global state, and (b) is deterministic, this is indicated by a *const* suffix following the function name. (Note that the E^* compiler checks (a), but relies on the programmer to ensure (b).)

2.3 Modules

The core of an E^* specification consists of rules that occur inside one or more *modules.* A module can use a previously defined module by *instantiating* with parameters. There must always be a module named *main*, which is implicitly instantiated by the E^* compiler[1]. The set of active rules in an E^* specification includes those in the *main* module and all other modules that are directly or indirectly instantiated by it.

3 Expressions and Statements

E^* supports all operations on primitive data types that can be found in the C-language. Expressions may contain function invocations, where the function may be an external function, a built-in function, or a user-defined function.

A statement may be simple or compound. A simple statement consists of a single function call or an assignment. A compound statement is either a sequence of statements enclosed within braces, an if-then-else or a for-statement. (See E^* grammar for details.)

3.1 Event Patterns

The full E^* language supports both *primitive* patterns, which match a single event, and *sequential* patterns, which match a sequence of events. However, the current implementation omits sequential patterns for simplicity, and hence we limit this language description to primitive patterns. Primitive event patterns match a single event and are of the form:

[1] Actually, the compiler insists that there be either a main function or a main module. If there is a main function, then that E^* program works like a program in any other standard programming language, without events and rules.

⟨*ppat*⟩ ::= ⟨*event*⟩[`|` ⟨*eventCond*⟩] // ⟨*eventCond*⟩ is a condition on event arguments.
| ⟨*ppat1*⟩ `\/` ⟨*ppat2*⟩ // Matches either ⟨*ppat*1⟩ or ⟨*ppat*2⟩.
| `!`⟨*ppat1*⟩ // Does not match ⟨*ppat*1⟩.

In the first form, ⟨*eventCond*⟩ is a condition on the event patterns. The second form of primitive patterns generalizes the first form so as to match one of many events, while the third form allows negation of patterns.

Conditions can reference event arguments on the left-hand side of comparisons. They can also use const functions (built-in or user-defined) on the lhs[2].

The right-hand sides of conditions must be constant-values expressions, i.e., they must evaluate to constant values at compile time. Note that module parameters that are constants at module instantiation time can be used in the rhs.

These restrictions are placed in order to support efficient matching of patterns. Note that patterns are *not ordered,* and hence they are *not* matched one by one. Instead, a decision tree structure is built that shares the matching computations across all patterns. Using a single traversal of the decision tree, all matching patterns will be identified. Tests (or their components) are not repeated on any path of this decision tree.

⟨*eventCond*⟩ ::= ⟨*boolExpr*⟩ | `!`⟨*eventCond*⟩ | ⟨*eventCond*⟩ ⟨*andOr*⟩ ⟨*eventCond*⟩

⟨*andOr*⟩ ::= `||` | `&&` | `|||` | `&&&`

⟨*boolExpr*⟩ ::= ⟨*lhs*⟩ | ⟨*lhs*⟩ ⟨*relOp*⟩ ⟨*const_expr*⟩ | ⟨*lhs*⟩ `&` ⟨*const_expr*⟩ (`==` | `!=`) ⟨*const_expr*⟩

⟨*lhs*⟩ ::= ⟨*id*⟩ | ⟨*id*⟩`(`⟨*lhs*⟩,...,⟨*lhs*⟩`)`

⟨*relOp*⟩ ::= `==` | `!=` | `<` | `<=` | `>` | `>=` | `=~` | `in`

Note that E^* event conditions, in one regard, are actually more powerful than conditions expressible in most languages: they permit regular expression comparisons. The form of other types of expressions is limited: the right-hand sides of expressions needs to be a constant. (It can be a constant-valued expression.)

Some conditions that do not follow the syntax of event conditions can be transformed to do so, e.g., $x - 5 > 0$ can be transformed to $x > 5$. But other forms, e.g., conditions that involve multiple variables, cannot be transformed away. Such conditions can be made into an if-statement at the outermost level of the reaction component of the rule.

Regular Expression Syntax and Semantics E^* compiler includes a regular expression matching engine. It supports POSIX-style extended regular expressions with POSIX-style semantics, with a few extensions, restrictions and exceptions.

- Note that regular expressions are intended to be compiled into matching code by the E^* compiler. Variables of regular expression type are not intended to be supported, although it is possible that the compiler won't complain. (However, variables of regular expression type are useless since there are no functions available at runtime to operate on them.)
- Regular expressions may be passed as module parameters. This works since module parameters are instantiated at compile-time.

[2]To avoid side-effects during event matching, we don't permit non-const functions.

- Regular expression literals appearing on the rhs of a regular expression operator (namely `=~`) may be simply be delimited by white-space, or they may be enclosed in double or single quotes, e.g.,

  ```
  x =~ [_A-za-z][_A-Za-z0-9]*
  ```

  In other contexts, regular expression literals should be enclosed within single or double quotes, preceded immediately by an `r` or `R`, e.g.,

  ```
  r'a[yz]*w'.
  ```

As with POSIX regular expressions, the following character classes are supported, and they have their usual meaning.

```
[:alnum:] [:alpha:] [:cntrl:] [:digit:] [:graph:] [:lower:]
[:print:] [:punct:] [:space:] [:upper:] [:xdigit:]
```

In addition, the class `[:any:]` matches any character. We assume that strings are null-terminated, so regular expressions cannot contain null characters.

Since regular expressions are matched against string-valued event arguments, some POSIX conventions are not applicable. For instance, `^` and `$` are treated literally, instead of matching the beginning (or end) of a line. Although `.` continues to have the same meaning as in POSIX, its use is discouraged because newlines have no special role in our setting. Users are encouraged to use `[:any:]` instead.

3.2 Scope and visibility of event arguments

For negated patterns, event arguments are not available past the negation operator. In particular, arguments of a negated event cannot be used in the rule body.

In a disjunctive pattern, there can be two or more last events, so the arguments of any of these events can be accessed in the reaction part of a rule. Obviously, this won't make sense if the last events don't agree on argument bindings. In particular, only arguments that have the same name and type across these events can be accessed in the reaction.

E^* compiler will complain about unused event parameters. If the parameter name begins with an underscore, then this error message is not flagged. In addition, each occurrence of an underscore is treated as a distinct variable. (This provision does not apply to all variables starting with `_`, but only when the entire variable name is just an underscore.)

4 Rules

E^* specifications consist of rules of the form $pat \rightarrow action$ where *pat* is a pattern on events, and *action* is a statement. These statements are executed when an event matching *pat* is delivered by the runtime system. Rules may optionally be given a label.

Note that the ordering of rules within a module is irrelevant for matching purposes: each rule is matched independent of others. This is not a fundamental restriction: if you want the effect of matching patterns in the order they are listed, you need to explicitly include conditions in subsequent rules that explicitly negate the conditions for matching previous rules. This process is illustrated below:

$$\begin{array}{rcl} e \mid C_1 & \rightarrow & R_1 \\ e \mid C_2 & \rightarrow & R_2 \\ e \mid C_3 & \rightarrow & R_3 \end{array} \qquad\qquad \begin{array}{rcl} e \mid C_1 & \rightarrow & R_1 \\ e \mid !C_1 \;\&\&\; C_2 & \rightarrow & R_2 \\ e \mid !C_1 \;\&\&\; !C_2 \;\&\&\; C_3 & \rightarrow & R_3 \end{array}$$

This may seem cubmersome, but there is an important benefit to our semantics: the semantics of a pattern is self-contained, as opposed to having implicit dependencies on all of the preceding patterns. This semantic clarity is far more important than seeming compactness, especially in security applications. Indeed, experience with systems such as firewalls and access control systems shows that it is enormously difficult to understand the semantics of large systems of rules when ordered matching is employed. The benefits of compactness are far outweighed by these semantic difficulties even for rulesets of modest size, say, 10 rules.

We could have added support for ordered matching semantics within modules that have a small set of rules. To do so, we need to give the programmer the option of using ordered or unordered semantics. Not only does this complicate the language, but it also allows programmers a "quick and dirty" option early on, thereby making it difficult to learn (or switch) to use the more disciplined approach.

In addition to semantic clarity, unordered matching avoids unnecessary serialization of matching: it is no longer necessary to rule out the first 999 patterns before checking the 1000th pattern. This leads to faster matching. In fact, E^* implementation uses a fast rule matching algorithm called *condition factorization* [6, 3] that has been extended to support regular expressions using the *regular expression derivative* [2] approach.

Unlike pattern matching, the actions performed by rules in different modules *are* ordered: they will be performed in the order in which the modules were instantiated[3]. Actions associated with rules within the same module are executed in the order of rules.

We could have said that, similar to matching, reactions are executed in arbitrary order. But this does not work well: pattern-matching is side-effect free, so the order of trying the matches will have no impact in determining which of the paterns matched. Unfortunately, actions do have side-effects, and so different orders of execution may have different semantics. So, if we allow E^* to reorder actions arbitrarily, then the specifications will have indeterminate semantics, or we must require that the actions be side-effect free. Neither of these options seems acceptable, so we choose ordered semantics for the execution of reactions. In future, this semantics may be augmented with static analysis that warns users about potentially conflicting actions.

5 Modules

Modules enable the development of parameterized E^* specifications. Module parameters can be data values (inluding regular expression values), events, functions[4] other modules, or state variables.

One module can make use of another module by *instantiating* it as follows:

$$\texttt{instantiate m(}\langle expr_1\rangle, \ldots, \langle expr_n\rangle\texttt{)}$$

where `m` is the name of a previously declared module. Let it be of the form

$$\texttt{module m(}\langle param_1\rangle, \ldots, \langle param_n\rangle\texttt{)}\ \ \{\langle body\rangle\}$$

[3]If events correspond to functions, then there would typically be two events for each function, one corresponding to a call and the other to return. In this case, a LIFO ordering may be more natural, i.e., calls would be invoked in the order of module instantiation, but returns will be executed in the reverse order.

[4]The compiler does not yet support function parameters, and perhaps not modules either.

The effect of the instantiation statement is that of inserting ⟨*body*⟩ in its place, after substituting all occurrences of $param_i$ with $expr_i$.

At the top level, there should be a module named `main`. The runtime system will deliver all the events to (an instance of) this module. The compiler-generated code for this module will then distribute the event to all the modules instantiated by the main module, and their descendants.

Module instantiations should be followed by one or more rules of the form described earlier. Following the rules, there can be zero or more *on* statements, which provide a way to maintain statistics associated with one or more rules. The rules of interest are identified using their labels. The keyword *all* is used to associate an on-statement with all the rules.

6 On Statements and Anomaly Detection

`On`-statements can be used compute and maintain statistical information to support anomaly detection. They are associated with one or more transition rules, i.e., statistical information is updated when those transitions are taken. This information is typically about parameters such as event arguments or state variable values at the time of transition. There can be two broad categories of anomalies we hope to detect:

- Individual events that are anomalous with respect to the distribution observed during training
- Entire distributions that are anomalous with respect to what was observed during training

At the language level, we currently support only the first kind. As compared to the first, the second type of anomaly detection cannot be timely: no single event will skew the distribution so much as to result in an alarm. Thus, one needs to compare distributions periodically. Lack of timeliness also implies lack of specificity: we can say that something is wrong, but we cannot point to a possible event that is the culprit. (In particular, because the detection results from a periodic comparison, there is no reason to believe that the last few events had anything to do with the alarm.)

An anomalous event is the execution of one of the rules associated with an on statement, together with a parameter value that is anomalous. This parameter may be categorical or ordinal. Although ordinal data may seem common, order is often meaningless in the context of use, e.g., file names, IP addresses/ports, etc. Moreover, combinations of parameters can be considered a single tuple-type parameter, and their simplest treatment is as categorical data (even if the components of the tuple are all ordinal).

6.1 Anomaly Detection for Categorical Data

Anomaly can be detected by maintaining a distribution of categorical data in terms of their frequency of occurrence. With each event, we can compare the parameter value against the associated frequency to derive an event probability. An anomaly is defined as an event with a probability below a specified threshold. This can be expressed using an on-statement of the following form[5]:

⟨*on_stmt*⟩ ::= `on` ⟨*trans*⟩ [⟨*wrt*⟩] `learn` ⟨*expr*⟩ [`when` ⟨*cond*⟩] `detect when` ⟨*cond*⟩ `maxprob` ⟨*const_expr*⟩

⟨*trans_spec*⟩ ::= [`eachof`] (`all` | `{`⟨*id*⟩ (`,` ⟨*id*⟩)*`}`)

⟨*wrt*⟩ ::= `specialize` ⟨*expr*⟩ `maxsize` ⟨*const_expr*⟩

⟨*param_spec*⟩ ::= ⟨*expr*⟩ `maxsize` ⟨*const_expr*⟩ [`decay rate` ⟨*const_expr*⟩ `decay time` ⟨*const_expr*⟩]

[5]The grammar in the appendix is out of date for on-statements; use the grammar given here.

An on-statement is associated with a set of transitions listed in its ⟨*trans_spec*⟩. The keyword `all` is a shorthand for explicitly listing all of the transitions in the current module.

By default, the same anomaly detector is used across all of these transitions. An optional keyword `eachof` indicates that a separate anomaly detectors should be maintained for each transition.

A ⟨*param_spec*⟩ specifies the parameter value to be maintained. It can be any expression that consists of state variables and constants. Parameter values are stored in a set that uses an LRU-type algorithm to bound memory use. The maximum size of this table is specified using the keyword `maxsize`. For each distinct parameter value, this set stores the relative number of times this value has been observed. In addition, an exponential decay algorithm is supported so that old information can be weighted less than recent information. This algorithm is specified using two parameters `decay rate` and `decay time`. Specifically, occurrence counts are scaled by the decay rate every decay time seconds.

The parameter value is accumulated in the LRU set only if the condition in ⟨*learn_spec*⟩ holds. Similarly, the statement is used for detection only if the condition in ⟨*detect_spec*⟩ holds. In addition, the probability of event occurrence, based on the relative counts maintained in the LRU set, should be less than the value specified using the keyword `maxprob`.

References

[1] Thomas Bowen, Dana Chee, Mark Segal, R. Sekar, Tushar Shanbhag and Prem Uppuluri, "Building Survivable Systems: An Integrated Approach based on Intrusion Detection and Damage Containment," DARPA Information Survivability Conference and Exposition (DISCEX), 2000.

[2] Janusz Brzozowski, "Derivatives of regular expressions," Journal of the ACM (JACM) 1964.

[3] R. Sekar, R. Ramesh, and I.V. Ramakrishnan, "Adaptive pattern matching," SIAM Journal on Computing, 1995.

[4] R. Sekar, Guang Yang, Shobhit Verma and Tushar Shanbhag, "A High-Performance Network Intrusion Detection System," ACM Conference on Computer and Communications Security (CCS), 1999.

[5] R. Sekar, Ajay Gupta, James Frullo, Tushar Shanbhag, Abhishek Tiwari, Henglin Yang and Sheng Zhou, "Specification-based anomaly detection: a new approach for detecting network intrusions," ACM Conference on Computer and Communications Security (CCS), 2002.

[6] Alok Tongaonkar and R. Sekar. "Condition Factorization: A Technique for Building Fast and Compact Packet Matching Automata," IEEE Transactions on Information Forensics and Security, 2016.

[7] Prem Uppuluri and R. Sekar, "Experiences with Specification Based Intrusion Detection System," Recent Advances in Intrusion Detection (RAID), 2001.

Appendix A: Runtime Interface with Host V3 Example

Interfacing to a runtime requires the following steps:

- Create interface definition for E^*
- Create the runtime code that will report events to E^* specifications
- Create a simple script or Makefile to handle compilation and linking of E^* files.

These steps are described in more detail below, with examples taken from the Host runtime interface.

A.1 Defining the interface

As described earlier, an interface defines:

- all (runtime-system-defined) external classes and external functions that operate on them. Attention should be paid to ensure that any functions used in rule lhs are `const` functions, which means that they must be *determininistic* as well as *side-effect free.* Non-const functions can only be used in the rhs of rules.
- all of the events associated with the interface.

For instance, Host interface is defined in a file called `HostExtes.h`[6]. This file happens to define an interface named `ESHost`. To understand the role played by an interface name in the code generated by E^* compiler, note that `HostExtes.h` will be included into an E^* file, e.g., `test.es`. E^* compiler will generate a C++ file `test.es.C` from `test.es`. It will insert a pair of includes in this code: `#include ESHost_pre.h` at the beginning of `test.es.C` and `#include ESHost_post.h` at its end. These header files must exist and be part of the runtime system.

To make changes to an interface, e.g., to add a function `f` to `ESHost`, `HostExtes.h` needs to be modified to add the declaration of `f`. In addition, the implementation of `f` needs to be included in `ESHost_pre.h` or `ESHost_post.h`, or in another file that will be linked with `test.es.C`. (Note that, at a minimum, a declaration for `f` must be present in `ESHost_pre.h` or you will get a compiler error when compiling `test.es.C`.)

Note that E^* code is strictly a layer above the runtime. Runtime system can deliver events to E^*, and E^* can use functions to execute operations on the runtime, but there is no direct provision for the runtime system to call an E^* function. Nevertheless, such an upcall can be simulated using events: define an event corresponding to each desired upcall, and write a rule in E^* for this event so that it performs the desired function. Values can be returned by defining an external variable, e.g., `rv` that will set using an external function, say, `setrv`.

A.2 Linkage between runtime and E^*-generated code

Given an interface specification, E^* compiler generates one function corresponding to each event, and this functions takes the same arguments as the event. Each of these functions is a member of a `__main` class in order to avoid polluting the global namespace. To report an event, the runtime system simply calls the corresponding function generated by the compiler.

[6] Like C and C++ header files, E^* header files are also named with a trailing `.h`.

In order for the generated code to compile without errors, all external functions and classes mentioned the interface must be declared in the "pre" header file associated with the interface, e.g., the `ESHost_pre.h` header file for the interface `ESHost`.

In order for the E^*-generated code to compile into an executable, the generated `.o` file (e.g., the file `test.es.o` when compiling the specification named `test.es`) needs to be linked with the runtime system, which must contain the definition of all external functions and classes.

Requirements mentioned so far must be satisfied by any runtime system. Although there are many more specifics that can be implemented differently by different runtime systems, it is helpful to know the specifics of the Host V3 interface, as it can provide a more concrete roadmap for implementation. Specifically, note that Host V3 contains an interface class called `HostExt` that contains two events for each Host event. Specifically, for an event such as `read`, there are two methods `read_pre` and `read` in `HostExt`. For a few events, there may be no pre-event.

When a Host event `x` occurs, note that this typically corresponds to an audit log consumer calling the method `x` on `Host`. `Host::x` contains a call to `HostExt::x_pre` at its beginning, and a call to `HostExt::x` at its end.

`ESHost` interface has been implemented by defining a subclass `ESHost` of `Host`. The constructor of `ESHost` installs an extension ext in `Host`. This extension belongs to class `ESExt` which is derived from `HostExt`. For each method `x_pre` and `x` in `HostExt`, `ESExt` simply calls the corresponding functions `__main.x_pre` and `__main.x` . Note that `ESExt` class is defined in `ESHost_post.h`, while `ESHost.h` is included in `ESHost_pre.h`.

A.3 Scripting compilation and linking of E^* code

To simplify the use of E^*-extensions, it is useful to develop scripts to automate the steps in compilining and linking them to produce an executable. Once again, there are many ways to do this, but the specifics of Host V3 can provide a helpful roadmap.

For Host V3, instead of writing a script to handle the compilation and linking of E^* code, the necessary logic has been put into a makefile. The makefile contains rules for generating `x.es.C` from an E^* source file `x.es`. Note that a valid top-level E^* file must define a module named `main`, which is then compiled into a class called `__main` that is defined in the generated C++ file `x.es.C`. The makefile then compiles `x.es.C` into `x.es.o` using a C++ compiler, and then links to the rest of the Host code to yield a binary named `x.ese`. All of this steps will be run automatically when the user types the command `make x.ese`.

A.4 Output

Print statements in E^* write to the stream stored in the variable `esout`, which is `cout` by default. The runtime can initialize `esout` to a different value if it wants the output to go elsewhere.

A.5 Support for dynamic addition of extensions

Currently, all of the E^*-compiled code is linked statically with the Host code to produce an executable. This means that there is no way to change the extensions, or add new ones while Host is already running.

Supporting dynamic addition of extensions is relatively easy. First, Host should be modified so that it can accept a set of extensions, e.g., it can contain a vector of extensions. Each extension should then be compiled into a shared library, so that it can be loaded dynamically. Hopefully, it should be possible to unload them dynamically as well, so that we can easily experiment with new extensions, or make changes to existing ones.

To summarize, the main changes needed are (a) make a small change to Host so that it maintains a list of extensions, and invokes pre and post operations on each extension for each event, (b) compile the E^* generated C++ code into a shared library rather than a simple object file.

Appendix B: Lexical Structure

- *Comments:* C++-style comments are supported. Note that comments are processed (and stripped out) by the C-preprocessor in our implementation.
- *Identifiers:* As in most other languages, these begin with an upper case or lower case letter, or an underscore, and may contain more letters, underscores or digits.
- *Keywords, literals* and *operators* are discussed in more detail below.

B.1 Keywords

- Type-related keywords are:

```
void string class event
bool bit enum char short int long double
struct list tuple dict
unsigned const with
```

- Built-in functions are:

```
length append union remove
insert erase clear print printr
```

  Note that `print` has variable arity, and automatically inserts a space between the values printed. It also introduces a new line at the end. In contrast, `printr` is a "print raw" function that does not introduce spaces or new lines.

- *Compound-statement* related keywords are: `if else foreach do return`
- *Module-related* keywords are:

```
module statemachine instantiate
states start final map timeout in
```

- *On-statement* related keywords are:

```
on all eachof when
specialize maxsize falseAlarmRate falsealarmrate
frequency histogram
learn detect
```

B.2 Operators

Symbol	Explanation
`-->`, `->`	Used in *pat*`-->`*action*
`|`	to separate conditions from events, for bit-wise or, and in regexp
`+`	For numeric addition and string concatenation
`-`, `*`, `/`	Arithmetic operators
`==`, `!=`	Equality/disequality on all types
`>`, `<`, `>=`, `<=`	Inequality operators on primitive types
`,`	Used as separator in various lists
`.`	Used to select fields of a struct
`=`	Assignment operator
`=~`	Regular expression comparison, with left-hand side being a variable and the right-hand side a regular expression constant
`;`	Terminates statements
`:`	Used in struct derivation, labels and in dictionaries
`{`, `}`	Used to construct lists, sets and dictionaries.
`[`, `]`	Array subscripting, tuple field access and regexp char class
`(`,`)`	Parenthesis, and for constructing tuples
`&&&`	Sequential conjunction, second operand evaluated only if the first operand is true
`&&`	Unordered conjunction, either operand may be evaluated first
`|||`	Sequential disjunction, second operand evaluated only if the first operand is false
`||`	Unordered disjunction, either operand may be evaluated first
`!`	Logical negation
`\/`	Disjunction operator for event patterns
`&`, `~`, `^`	Bit-wise operators
`^`, `$`, `*`, `+`, `?`, `.`, `|` `\n`	Regular expression operators

B.3 Literals

- *Boolean* constants include `true` and `false`.
- *Integers* in E^* can be in binary, octal, decimal, or hexadecimal. Binary numbers consist of zeroes and ones, with a `b` or `B` suffix. (No intervening space between the digits and the suffix.) Octal numbers begin with a zero, while a hexadecimal begins with a `0x`.

  Unlike languages such as C/C++, there is no need to distinguish between integer constants of different types such as unsigned, long, short etc. The correct integral type is inferred from the context.
- *IP addresses* can be specified using dotted decimal notation, as in 127.0.0.1.
- *Floating point* numbers use the familiar syntax, as in C/C++.
- *Strings* are enclosed by single or double quotes. Within string literals, the usual escape sequences are supported: `\a`, `\b`, `\f`, `\n`, `\r`, `\t`, and `\v`. In addition, `\0oo` and `\xhh` escape

sequences can be used to enter characters as ASCII code, where `o` and `h` denote octal and hexadecimal digits respectively.

As in most other languages, string constants cannot contain newlines.

- *Regular expressions.* See below.

B.4 Regular Expressions

Regular expression are unique in E^* in that they can only be literal constants. There is no "regular expression" data present at runtime: you cannot have variables of regular expression type, nor can you pass them into functions.[7] Module parameters can in fact be regular expressions, as their passing is handled entirely at compile time.

[7] Of course, it is possible to represent regular expressions as strings, and pass them into libraries that match them. However, these are just plain strings in E^* language.

Appendix C: Grammar

⟨*type*⟩ ::= [`const`] ⟨*type1*⟩

⟨*type1*⟩ ::= `void` | `bool` | `string` | `bit`
| `unsigned` | `double` | [`unsigned`] (`char` | `short` | `int` | `long`)
| ⟨*id*⟩
| ⟨*type*⟩ (`list` | `set`) // We support set, list, tuple and dictionary types.
| `tuple` (⟨*type*⟩ (`,` ⟨*type*⟩)*)
| `dict` (⟨*type*⟩ `,` ⟨*type*⟩)

⟨*spec*⟩ ::= ⟨*decl*⟩*

⟨*decl*⟩ ::= ⟨*event_decl*⟩`;` // Event signature.
| ⟨*class_decl*⟩`;` // Externally defined, opaque, encapsulated data type.
| ⟨*fun_decl*⟩`;` // Function signature.
| ⟨*enum_decl*⟩[`;`] // Definition of an enumerated type.
| ⟨*fun_defn*⟩[`;`] // Function signature *and* body.
| ⟨*struct_decl*⟩[`;`] // Used to describe low-level data, e.g., network packets.
| ⟨*module_defn*⟩[`;`] // Module definitions.

⟨*event_decl*⟩ ::= `event` ⟨*id*⟩ [(⟨*param_list*⟩)]

⟨*class_decl*⟩ ::= `class` ⟨*id*⟩`;`

⟨*fun_decl*⟩ ::= ⟨*type*⟩ ⟨*id*⟩ `(`[⟨*param_list*⟩]`)` [`const`]

⟨*param_list*⟩ ::= ⟨*param_decl*⟩(`,` ⟨*param_decl*⟩)*

⟨*param_decl*⟩ ::= ⟨*type*⟩ [⟨*id*⟩]

⟨*enum_decl*⟩ ::= `enum` ⟨*id*⟩ `{`⟨*id*⟩(`,` ⟨*id*⟩)*`};`

⟨*fun_defn*⟩ ::= ⟨*fun_decl*⟩ `{`⟨*var_decl*⟩* ⟨*stmt*⟩+`}`[`;`]

⟨*struct_decl*⟩ ::= `struct` ⟨*id*⟩ [`:` ⟨*parent_spec*⟩ (`or` ⟨*parent_spec*⟩)*] `{`⟨*var_decl*⟩*`}`

⟨*parent_spec*⟩ ::= ⟨*id*⟩ [`with` ⟨*expr*⟩]

⟨*module_defn*⟩ ::= ⟨*module_hdr*⟩ `{`⟨*module_body*⟩`}` // Specifications parameterized using this construct.

⟨*module_hdr*⟩ ::= [`module` | `statemachine`] ⟨*id*⟩ `(`[⟨*module_param*⟩(`,` ⟨*module_param*⟩)*])

⟨*module_param*⟩ ::= ⟨*param_decl*⟩ | ⟨*fun_decl*⟩ | ⟨*event_decl*⟩ | ⟨*module_hdr*⟩
// Modules can take other modules (as well as functions and events) as parameters.

⟨*module_body*⟩ ::= ⟨*module_decl*⟩* ⟨*rule*⟩* ⟨*on*⟩*

⟨*module_decl*⟩ ::= `states` `{`⟨*id*⟩(`,` ⟨*id*⟩)*`};` // The states of the state machine.
| ⟨*var_decl*⟩ // Variable declaration (with or without initialization).
| (`start` | `final`) ⟨*id*⟩`;` // Only a single start and a single final state.
| `map` ⟨*event*⟩ `when` ⟨*expr*⟩`;` // Specifies events of interest to a state machine instance.
| `timeout` ⟨*time*⟩ [`in`] (`all` | `{`⟨*id*⟩(`,` ⟨*id*⟩)*`});` // Timeouts for one or more states.
| `instantiate` ⟨*fun_call*⟩ // Module instantiation.

⟨*var_decl*⟩ ::= ⟨*type*⟩ ⟨*var*⟩ (`,` ⟨*var*⟩)*`;`

⟨*var*⟩ ::= ⟨*sim_var*⟩ | ⟨*array_var*⟩

⟨*sim_var*⟩ ::= ⟨*id*⟩ [`=` ⟨*expr*⟩]

⟨*array_var*⟩ ::= ⟨*id*⟩[[⟨*expr*⟩]]+ // Array subscripting, ***or*** tuple or dictionary access.

⟨*on*⟩ ::= `on` ⟨*trans_spec*⟩ [⟨*specialize*⟩] (⟨*freq_spec*⟩|⟨*val_spec*⟩) `falseAlarmRate` ⟨*expr*⟩
[`learn if` ⟨*expr*⟩] [`detect if` ⟨*expr*⟩]`;`

⟨*trans_spec*⟩ ::= [`eachof`] (`all` | `{`⟨*id*⟩ (`,` ⟨*id*⟩)*`}`) [`when` ⟨*expr*⟩] // refers to transitions by their labels.
// eachof indicates that statistics should be maintained separately for each transition.

⟨*specialize*⟩ ::= `specialize (`⟨*expr_list*⟩`) maxsize` ⟨*expr*⟩ // Maintain separate statistics for each
// value of ⟨*expr_list*⟩ but limit to maxsize most frequently occurring values.

⟨*expr_list*⟩ ::= ⟨*expr*⟩(`,` ⟨*expr*⟩)*

⟨*freq_spec*⟩ ::= `histogram frequency` ⟨*time*⟩ `[` ⟨*expr*⟩ `]` // Specifies minimum and # of timescales.

⟨*val_spec*⟩ ::= `histogram` ⟨*expr*⟩ // Value distribution.

⟨*time*⟩ ::= ⟨*expr*⟩ (`us` | `ms` | `s` | `sec`) // Also permits `seconds`, `milliseconds`, etc.

⟨*rule*⟩ ::= [⟨*id*⟩`:`] ⟨*event_pat*⟩ `->` ⟨*stmt*⟩ // `-->` can also be used.

⟨*event_pat*⟩ ::= ⟨*event*⟩[`|` ⟨*eventCond*⟩] // Matched when ⟨*event*⟩ occurs and ⟨*eventCond*⟩ holds.
| `!`⟨*event_pat*⟩ // Event non-occurrence, ⟨*event_pat*⟩ cannot contain sequence operators.
| ⟨*event_pat*⟩ `\/` ⟨*event_pat*⟩ // Disjunction operator.

⟨*eventCond*⟩ ::= ⟨*boolExpr*⟩ | `!`⟨*eventCond*⟩ | ⟨*eventCond*⟩ ⟨*andOr*⟩ ⟨*eventCond*⟩

⟨*andOr*⟩ ::= `||` | `&&` | `|||` | `&&&`

⟨*boolExpr*⟩ ::= ⟨*id*⟩ | ⟨*lhs*⟩ ⟨*relOp*⟩ ⟨*const_expr*⟩

⟨*lhs*⟩ ::= ⟨*id*⟩[`&` ⟨*const_expr*⟩]

⟨*relOp*⟩ ::= `==` | `!=` | `<` | `<=` | `>` | `>=` | `=~` | `in`

⟨*lv_expr*⟩ ::= ⟨*id*⟩ | ⟨*lv_expr*⟩`.`⟨*id*⟩ | ⟨*lv_expr*⟩`[`⟨*expr*⟩`]` // Note: `[]` and `.` needed in struct definition.

⟨*expr*⟩ ::= ⟨*literal*⟩
| ⟨*lv_expr*⟩
| ⟨*unop*⟩ ⟨*expr*⟩
| ⟨*expr*⟩ ⟨*binop*⟩ ⟨*expr*⟩
| ⟨*fun_call*⟩
| (⟨*expr_list*⟩) // Tuple value.
| [⟨*expr_list*⟩] // List value.
| {⟨*expr_list*⟩} // Set value.
| { ⟨*expr*⟩:⟨*expr*⟩(`,` ⟨*expr*⟩:⟨*expr*⟩)* } // Dictionary value.

⟨*unop*⟩ ::= `-` | `!` | `~`

⟨*fun_call*⟩ ::= ⟨*fun_name*⟩`(`⟨*expr_list*⟩`)`

⟨*fun_name*⟩ ::= ⟨*id*⟩ // User-defined or externally defined functions.
| `empty` | `clear` | `length` | `append` | `union` | `insert` | `erase`
| | `remove` | `print` | `printr`
// Predefined functions that work on many types. Print also has variable arity.

⟨*asg_stmt*⟩ ::= ⟨*lv_expr*⟩ `=` ⟨*expr*⟩

⟨*stmt*⟩ ::= `;` // Empty statement.
| ⟨*asg_stmt*⟩`;`
| ⟨*fun_call*⟩`;`
| `return` ⟨*expr*⟩[`;`] // Valid only within a function definition.
| `if` ⟨*expr*⟩ ⟨*stmt*⟩ [`else` ⟨*stmt*⟩]
| `{`⟨*var_decl*⟩* ⟨*stmt*⟩+`}`[`;`]
| `foreach` ⟨*id*⟩ `in` ⟨*expr*⟩ `do` ⟨*stmt*⟩
| `while` ⟨*expr*⟩ `do` ⟨*stmt*⟩

⟨*binop*⟩ ::= `+` | `-` | `*` | `/` | `%` // Almost all operators from C-language.
| `&` | `|` | `^` | `<<` | `>>`
| `==` | `!=` | `>` | `>=` | `<` | `<=` | `=~` | `in`
| `&&` | `||`

Appendix D: Unimplemented Features

D.1 Sequential Event Patterns

Sequential patterns match a sequence of events, and are of the following form. Here, let ϵ represent the pattern that matches the empty sequence of events.

$\langle spat \rangle$ ::= $\langle ppat \rangle$ // Base case is a primitive pattern.
| $\langle spat1 \rangle$ `:` $\langle spat2 \rangle$ // If $\langle spat1 \rangle$ matches $e_1, \ldots, e_k$ and $\langle spat2 \rangle$ matches $e_{k+1}, \ldots, e_l$ // then $\langle spat \rangle$ will match $e_1, \ldots, e_l$.
| $\langle spat1 \rangle$ `\/` $\langle spat2 \rangle$ // Matches either $\langle spat1 \rangle$ or $\langle spat2 \rangle$.
| $\langle spat1 \rangle$`?` // Matches either the empty sequence or a sequence that matches $\langle spat1 \rangle$.
| $\langle spat1 \rangle$`+` // Matches one of: $\langle spat1 \rangle$, $\langle spat1 \rangle$`:`$\langle spat1 \rangle$, $\langle spat1 \rangle$`:`$\langle spat1 \rangle$`:`$\langle spat1 \rangle, \ldots$.
| $\langle spat1 \rangle$`*` // Kleene closure: equivalent to `(`$\langle spat1 \rangle$`+)?`.

Use of negation on sequential patterns is highly error-prone, so E^* limits the negation operation to primitive patterns.

Correct treatment of assignments in the middle of patterns is tricky. For instance, consider the pattern p:

$$\mathtt{e}_1(x)|(a = x) \ : \ e_1* \ : \ e_2$$

and the event history H:

$$\mathtt{e}_1(1)e_1(2)e_1(3)e_1(4)e_2$$

Recall that we declare a match when p matches a suffix of H. Thus, there are 4 possible ways to match p with H: the first one binds a to 1, the second binds a to 2, and so on. To provide a simple declarative semantics to pattern matching, it is necessary to consider and evaluate all of these cases. For this purpose, we restrict the use of mutable variables within event patterns as follows:

- The only assignments permitted in event patterns are those that assign to *rule variables.* Rule variables are not explicitly declared: they are recognized by virtue of being assigned within an event pattern, and their types inferred from those of the expressions assigned to them.

  The scope of a rule variable is limited to a single rule. Moreover, they cannot be assigned in the reaction component of a rule. Consequently, a rule variable value won't change due to the execution of rules, or the matching of patterns other than the one in which the variable appears.[8] These factors are critical in ensuring a simple semantics for pattern-matching.

- Since the values of mutable variables may change due to the execution of other rules, their use within an event pattern will complicate semantics. So, we do not permit mutable variables to be used within event patterns.

Note: All of the above restrictions can be lifted for the special case of patterns *without* sequencing operators. There are no viable matching prefixes maintained by the matcher for such patterns, so we don't need to worry about their invalidation due to updates.

[8] **Need to check what happens if a rule variable is assigned twice. uch assignments can complicate semantics in some cases, so we might as well disallow them.** Updates to rule variables within an event pattern do not seem to pose a problem, since each viable matching prefix has its own bindings for rule variables. In such a setting, it should be easy to provide a simple semantics even if there are updates to rule variables. (

Note that a pattern $pat*$ is equivalent to $\epsilon \vee pat*$, and since ϵ does not bind any event arguments, there are no common set of argument bindings available after $pat*$. So, all event bindings made within sub-pattern are invalidated after a `*`, `+` and `?` operator. For similar reasons, no new event variable bindings are available after an event sub-pattern of the form `!`pat.

Similar consistency rules can be enforced for rule variables, but we are currently not doing this. In particular, it may be desirable to have patterns such as

$$(e|ne = ne + 1)*$$

where ne is a state variable that is used to count the number of occurrences of the event e. Similarly, we may want to use a pattern

$$(e_1|ne1 = ne1 + 1) \vee (e_1|ne2 = ne2 + 1)$$

These would become invalid if the consistency rules applied to event arguments are also extended to state variables. (An earlier version of the language did enforce consistency rules on state variables, so one could consider the current change as provisional, until we get more experience.)

D.1.1 Patterns with Back References

Similar to regular expressions, the actual event sequence matching an event pattern can be remembered for subsequent use. This feature requires the addition of the following form to sequential event patterns:

$\langle spat \rangle$ ::= `(`$\langle id \rangle = \langle spat \rangle$`)`

Storage overheads posed by such patterns can be significant, especially since there can be many matches that may be active at the same time, so this feature should be used with extreme care.

D.2 Abstract Events

For simplicity, the current E^* compiler does not support abstract events. Users can achieve a similar effect using cpp macros.

Abstract events provide a way to decompose complex event patterns into simpler ones. They can be given intuitive names so that one does not have to parse a complex pattern every time one examines a specification. In the full E^* language, abstract events may defined as follows:

$$\langle id \rangle(x_1, \ldots, x_n) ::= \langle spat \rangle$$

where $\langle id \rangle$ is the newly defined abstract event that has $x_1, \ldots, x_n$ are its arguments. All of these variables *must* appear within $\langle spat \rangle$.

D.3 State Machines

State machines are very similar to modules, while providing a few additional features that are not available in modules. In particular, state machines support session-style specifications. Instances of the state machine can be spawned dynamically, with each instance having its own copy of the entire state of the machine. For instance, one can define a state machine that maintains the status of evey TCP connection between remote and local hosts in an enterprise. To support this functionality, state machines permit a few additional constructs in their body. We describe them below. Most

of these features are already supported by the lexer and parser, but the code generation features have not been implemented yet.

The "states" of a state machine are specified using a `states` statement. It combines the definition of an enumerated data type defining valid states of the state machine, together with the declaration of a state-variable called `state`. The start and final states of the state machine are specified using the `start` and `final` declarations.

At runtime, there will, in general, be multiple instances of each state machine. A `map` statement is designed to identify which instance should receive any given event. For instance, we may create an instance of a state machine on each `fork` system call, and store its pid in a variable, say, `id`. All subsequent systems call by the same process may then be delivered to this instance by comparing the pid with the stored value of `id`. The `map` statement is restricted to ensure that a hash-table look up is all that is needed to determine the (single) state machine instance to which any given event should be delivered

When the final state of a state machine is reached, that instance is automatically destroyed.

The *action* component of rules in a state machine must update `state`. If no event is delivered to a state machine for a long time, it is possible to specify a transition to another state on a (predefined) *timeout* event. Timeout values for different states are specified using `timeout` declarations.

It should be possible to limit maximum memory used by instances of a state machine by specifying a maximum number of instances. When the limit is reached, an approximately LRU algorithm should be used to purge inactive instances to create space for new ones.