Progressive Model Generation for Adaptive Resilient System Software

Topic number: N13A-T014

Award No: N00014-13-P-1175

(UNCLASSIFIED)

Issued By: Office of Naval Research

|  |  |
| --- | --- |
| **Lead Organization:**  Grammatech-Logo1.jpg  GrammaTech, Inc.  531 Esty Street  Ithaca, NY 14850 | **Principal Investigator/Preparer:**  Michael McDougall, Ph.D.  GrammaTech, Inc.  531 Esty Street  Ithaca, NY 14850  (607) 273-7340 x. 127 mcdougall@grammatech.com |
| **Subcontractor:**  University of Pennsylvania  3451 Walnut Street  Philadelphia PA 19104-6205 | **Technical POC:**  Oleg Sokolsky  Department of Computer and Information Science  University of Pennsylvania  3330 Walnut Street  Philadelphia, PA 19104-6390 |
| **TPOC** | **TPOC** |

**Contract Period:** (07-01-2013 to 04-30-2014)

**Reporting Period**: (07-01-2013 to 01-31-2014)

DISCLAIMER

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Navy or the U.S. Government.

**STTR DATA RIGHTS**

Topic Number: N13A-T014

Contract Number: N00014-13-P-1175

Contractor Name: GrammaTech, Inc.

Contractor Address: 531 Esty Street, Ithaca, NY 14850

Expiration of SBIR Data Rights Period: 04/30/2019

The Government’s rights to use, modify, reproduce, release, perform, display, or disclose technical data or computer software marked with this legend are restricted by paragraph (b)(4) of DFARS 252-227-7018, Rights in Noncommercial Technical Data and Computer Software—Small Business Innovative Research (SBIR) Program.

Distribution Statement B. Distribution authorized to U.S. Government Agencies Only; Proprietary Information (DFARS - SBIR Data Rights); 2013/08/16. Other requests for this document shall be referred to Office of Naval Research, ATTN: ONR Code 311, One Liberty Center, 875 N. Randolph Street, Arlington, VA 22203-1995.

Contents

[1 Technical Objectives 3](#_Toc377038228)

[2 Technical Problems 3](#_Toc377038229)

[3 General Methodology 3](#_Toc377038230)

[4 Technical Results 3](#_Toc377038231)

[5 Important Findings and Conclusions 3](#_Toc377038232)

[6 Implications for Further Research 3](#_Toc377038233)

[7 Significant Hardware/Software Development 3](#_Toc377038234)

[8 Special Comments 3](#_Toc377038235)

[9 Contract Deliveries Status 3](#_Toc377038236)

# Task Objectives

In the Phase I base effort, we pursued the following objectives:

1. **Develop and validate a modeling language** appropriate for capturing security properties.
   1. Identify a suitable starting candidate, or candidates. We will use PEDL/SMEDL as our starting point, but may draw features from other modeling approaches.
   2. Test the candidate language by encoding security policies. We will use the security policy of the Chromium browser (the open source core of Google Chrome) as our primary example, and supplement it with other security properties to test important areas not covered by the Chromium policy.
   3. Based on the results of encoding policies, adapt the candidate language as required.
2. **Investigate methods for generating the models** using a combination of automatic and manual techniques. Our vision is a semi-automated process whereby a programmer supplies hints that enable automatic analysis to generate a skeleton model, which is then refined manually. Specific topics to be investigated include:
   1. Using static analysis to recover the system architecture—for example, identifying the processes in an application and how they communicate.
   2. Using static analysis to identify major program ‘modes’, or high-level states that indicate a process or application is currently playing a particular role.
   3. Using dynamic analysis to supplement the static approaches.
   4. How to design a user interface to make model generation user-friendly.
   5. How to effectively validate models. In the envisioned system, the model becomes a critical part of the application, since an incorrect model may prevent normal operation of an application. Programmers will therefore need to test and validate their models before deploying their applications, just like they need to test and validate their code.
3. **Demonstrate the suitability of the modeling language** for use in runtime monitoring.
   1. Develop techniques for configuring a software dynamic translator using PEDL and the event mapper.
   2. Investigate scalable techniques for mapping low-level events to high-level events at runtime.
4. **Write a final report** that includes a plan for Phase II development.
5. **[Option] Implement a proof-of-concept prototype of the tool.**
   1. Extend the modeling language to add any syntactic or semantic constructs needed to support implementation.
   2. Implement mode-inference analysis.
   3. Extend GrammaTech’s existing Eclipse plugin to
      1. initiate analysis from source code.
      2. aid the users as they construct PEDL from source code.
   4. Implement a prototype tool that configures a SySense runtime monitor using the model.

The University of Pennsylvania had primary responsibility for Objective 1, and advised GrammaTech on Objective 2. GrammaTech was responsible for Objectives 2 and 3. This final report (Objective 4) was written in a joint effort.

# Technical Problems

<=3 page total.

## Problem Area

GT to fill in – describe the problem being addressed.

## Problems Encountered in Task 1

The primary problem to be addressed by Task 1 is to identify the necessary language features needed to capture behavioral security models of complex applications and develop a modeling language that allows to generate efficient monitors from models.

Validation of the developed language on the chosen application, the Chrome browser, turned out to be a challenge as the code base is large and complex. We concentrated on relatively simple properties is Phase I. Still, making sure that the properties are faithful abstractions of expected behavior of the system required us to understand the system at a relatively detailed level. We were able to overcome the challenge by labor-intensive code review. However, the encountered challenge reinforced the need to develop the model extraction technology proposed in this project.

## Problems Encountered in Tasks 2 and 3

GT to fill in

# General Methodology

<=4 pages total.

## Task 1 Methodology

Penn: describe the methodology employed for Task 1. Should be <=2 pages

The methodology adopted for Task 1 was derived directly from Task 1 objectives. The main objective was the development of a modeling language suitable for capturing expected behaviors of the system by system developers and for monitoring of the specified behavior at run time. This immediately imposed two requirements on the language design. One the one hand, the level of abstraction in the language needs to be high enough so that the user can naturally express his or her intuition about the system. On the other hand, semantics of the language needs to be lined up with existing monitoring languages, such as state machines or temporal logic formulas, so that existing monitor generation techniques can be utilized.

Our approach was to first understand strengths and deficiencies of existing monitoring languages from the behavior modeling perspective. We primarily concentrated on languages supported by the leading correctness monitoring and checking tools in the runtime verification community such as MaC [FMSD2004] and MOP [STTT2012].

The next step in the research approach was to select an application of the size and complexity that is similar to the final targets of the technology targeted in this project, and gain sufficient understanding of it so that we can construct a model capturing some of the behavioral aspects of the system. Unlike the ultimate goal of the project, this model will not be constructed during system development, and without the envisioned tool support. However, the importance of this exercise was to enable us to understand, which language features would be important to make this model concise and easy to construct and understand.

The construction of the model mentioned above served as the design-by-example phase of defining the language. Once we felt that the set of considered behaviors of the chosen application were representative of the general case, we made them concrete in a more formal language description, making sure that the resulting language definition directly supports all the considered features.

Implementation of the language parser and supporting monitoring infrastructure is left for subsequent phases of the project. However, the chosen methodology has put us in a position where the design of the language developed in Phase I of the project has validated by the manual construction of the model for the Chromium case study and its monitor. This will, in turn, facilitate tool development in Phase II.

## Task 2 Methodology

GT to fill in.

## Task 3 Methodology

GT to fill in.

# Technical Results

<=15 pages

## Task 1 Results

Penn: describe results here. 5-7.5 pages.

The main outcome of the Task 1 effort is the definition of the language to be used for expressing behavioral models of systems. Before engaging in the language design, we performed gap analysis of existing languages for correctness monitoring, which is summarized below.

**Analysis of existing languages for correctness monitoring.** We compared MEDL, developed by the Penn investigators in their prior work, and MOP, developed at UIUC as part of the state-of-the-art academic toolset for correctness monitoring.

MEDL introduced the concepts of events and conditions, where instantaneous events were used to trigger changes in conditions, which persisted over time intervals between event arrivals. Thus, one can think of a collection of conditions defined in a MEDL specification as the state of the checker. In addition, auxiliary variables could be set in response to events and formed additional checker state. Alarms, which signified violations of monitored properties, were raised when the checker reached a certain state. While MEDL contained all the necessary features to specify properties to be monitored, it had several drawbacks that made it hard to be used to monitor large systems:

* The first drawback was the monolithic nature of a MEDL specification. All events and conditions were defined in a flat space and related only by dependencies between them using logic of MEDL specification. As the specification became large with complex dependencies, it became hard to understand and validate. At run time, a MEDL specification was evaluated by a single checker, making it difficult to introduce distributed monitoring.
* Second, a MEDL specification was static; that is, the checker manipulated the same set of conditions throughout the system execution. This made it difficult to apply MaC to many real-life systems that dynamically create objects, spawn new threads of execution, etc. An extension of MEDL for handling parameterized properties [RV2005] that can be instantiated for each monitored object has been defined in an *ad hoc* way and never fully implemented.
* Finally, MEDL design keeps the state space of the checker implicit, making it hard to characterize the change of state that would be effected by a given event. Checker state in MEDL was partially captured by auxiliary variables and partially by conditions. To complicate matters, not all conditions contributed by checker state as some conditions were formed from other conditions by propositional operators of MEDL.

By contrast, MOP has been designed to support parametric monitoring from the outset, making it easy to specify monitors with large numbers of parameters and efficiently manage large numbers of monitor instances by efficiently routing observations to the right instances. In addition to temporal logic-based specifications, MOP allows properties to be expressed as state machines and relies on an efficient state-machine monitoring algorithm [TACAS2002]. On the other hand, MOP, like MEDL, lacks facilities to structure large models captures models as collections of individual properties

**SMEDL overview.** The central concept of SMEDL is a monitoring object, or object for short. An object can be an abstraction of a system object, whose state is tracked by the monitoring system. Alternatively, an object can be an abstract entity that represents interactions between multiple system objects. An object can have a state, which reflects history of its evolution. Objects can be created dynamically during an execution of a system. As the system evolves, multiple instances of the same object may be created. To distinguish between instances, a part of the object state is designated as the object identity and remains immutable during the object's lifetime. Objects in SMEDL can have their state changed by event arrivals. At runtime, each object instance produces a separate checker that encapsulates runtime state of the object and performs event updates.

The second concept of SMEDL is an event. SMEDL events are instantaneous occurrences that ultimately originate from observations of the system execution. Events are decorated with a set of attributes, which are formal parameters of the event. An event occurrence carries a set of actual parameters, which are values assigned to formal parameters of the event. Events are delivered to checkers that correspond to objects matching the values of event parameters. Thus, a single event can be processed by multiple checkers.

Each SMEDL object includes an interface, which declares events that are associated with this object, object state, which can include references to other objects, and the specification of object behaviors. We adopt a *scenario-based* behavior specification style, where each scenario describes a partial behavior, and the complete behavior is obtained by composing scenarios.

Consider an example of a SMEDL object. In a web browser, a *tab* is a panel that displays the contents of a URL and handles all aspects of processing a page. For example, it deals with storing and retrieving cookies. It is associated with a URL that is currently being processed and displayed. The same tab may display a different URL if the user makes a new request. We may want to monitor the handling of a URL by a tab, in particular processing of the cookies. This makes the URL part of the monitored tab state. A browser may have several tabs open at the same time, displaying different pages. To distinguish the tab, we need an identity. Tabs, generally, do not have a name or a set of attributes that uniquely identify it. In such a case, we represent the identity using an *opaque* keyword, meaning that the contents of this attribute is not interpreted and is assumed only to be unique. It can be, for example, a reference to the system object that does not have a meaning in the monitor context. The specification of a tab object, showing two simple scenarios, is shown below. The first scenario represents the cookie integrity property. In this scenario, an event that stores a cookie is accepted when the domain of the cookie is a subdomain of the current URL of the tab. The event does not change the state of the tab object. Otherwise, an error is raised. The second scenario represents a change of the URL that the tab displays. In contrast to the first scenario, the URL change updates the state of the checker by an action that is performed in response to the event. However, no alarms can be raised within this scenario.

**object** Tab  
 **identity  
 opaque** id  
 **state** URL currentUrl  
 Renderer rengine  
 **events  
 imported** store(Tab,Cookie), render(Renderer,URL)  
 **exported** **error** cookieIntegrityAlarm  
 **scenarios** CookieIntegrity:  
 Tab → store(**this**,cookie) *when* isSubdomain(cookie.domain, currentUrl.getHost()) → Tab  
 **else** **raise** cookieIntegrityAlarm → Tab  
 PageRendering:  
 Tab → Kernel.displayURL(**this**, url) → render(rengine, url) {currentUrl=url} → Tab

This simple example highlights several syntactic and semantic aspects of SMEDL, which will be made precise in the following sections. Scenarios of objects are specified in terms of events. Events are associated with objects. However, scenarios of an object can include events of other objects. This is a means of coordination between object behaviors. Events can be observations from the monitored system or can be raised by the checker. In particular, alarms are in the latter category. However, not all events raised by the checker are alarms. They can also be used in scenarios of other objects and allow for different checkers to coordinate.

**SMEDL syntax.** Before presenting the syntax for SMEDL objects, we introduce the building blocks such as types and expressions used in scenario actions. Several places in the description below make references to Java syntax. This is a notational convenience to avoid spelling out large grammar specifications that describe parts that are generally well understood. There is no implied dependence on Java parsers or Java runtime in SMEDL implementations. Reserved keywords of SMEDL are shown in bold in the grammar expressions below and cannot be used as identifiers.

*Types.* Primitive types are integers, floating point numbers, strings, and container objects such as sets. Container types are expected to contain standard operations to manipulate container contents. Names and parameters of these operations match methods of the corresponding objects from the Java standard library. The **opaque** type is a primitive type, on which only the equality operation is defined. Composite types are either objects defined in SMEDL or Java objects treated as data types. Methods of Java objects can be used as operations in expressions.

*Expressions.* Expressions describe operations on values of state variables and event parameters. For simplicity, expressions borrow a subset of Java expression syntax. They include infix notation for arithmetic and logical operations on numerical and Boolean values, respectively. Invocation of methods for object types, such as container manipulation operations, is represented using the common dot operator.

*Object definitions.* Object definitions are given by the following grammar. Scenario definitions, referenced in the grammar, are discussed below. Each object contains a set of state variables, partitioned into object identity and data state, a non-empty set of events that are delivered to the object, and a non-empty set of scenarios. The sets of state and identity variables can be empty. If identity variables are absent, the object is *static* and cannot be explicitly instantiated. Static objects have only one instance.

object ::=  
 [ **identity** variable\_declaration+ ]  
 [ **state**  variable\_declaration+ ]  
 **events  
 imported** event\_definition ( “,” event\_definition )\*  
 [ **exported** event\_definition ( “,” event\_definition )\* ]  
 **scenarios** [ scenario\_definition ]+

variable\\_declaration ::=  
 type identifier ( “,” indentifier )\*

event\_definition ::=  
 [ **error**] identifier [ “(“ parameter\_list “)” ]

parameter\_list ::=

type ( “,” type )\* | *empty*

Variable declarations for identity and state variable are lists of typed identifiers. Event definitions are partitioned into imported and exported events, and can be decorated with lists of typed parameters. The set of imported events in an object cannot be empty. Otherwise, the object cannot be used to monitor system behaviors, because it would not have any inputs to observe. Exported events are generated by the object itself. The set of declared exported events can be empty. As explained in the section on SMEDL semantics, each object has a default exported event that represents a monitoring violation. Declared exported events can be used as means of supplying more detailed information about the observed behavior, or as means of coordination between monitors of different SMEDL objects.

*Scenario definitions.* An object can engage in a number of scenarios. Each scenario is given a name and is defined by a set of sequences of events that advance scenarios from location to location. An event may be supplemented by an action that changes the values of state variables of the object and can raise other events. Intuitively, a scenario waits in a location for the specified event to arrive. When it arrives, scenario execution is triggered. If present, the optional action is executed, and the scenario moves into the location where it waits for the next event. Location in a scenario may be named, or it may be anonymous. Named locations are used in the composition of scenarios, as discussed below. The first location of each scenario is named. When the name of the first location is the same as the name of the object, the scenario is ready to be triggered immediately upon object instantiation. Otherwise, the scenario can be triggered only after the named location is reached by one of the other scenarios.

scenario\_definition ::=  
 [ **atomic** ] identifier “:” trace\_definition+

trace\_definition ::= identifier “→” step\_definition+

step\_definition ::=  
 event\_instance “→” [ action ] [ identifier [ “(“ state\_update\_list “)” ] ]

[ **else** “→” [ action ] [ identifier [ “(“ state\_update\_list “)” ] ]

event\_instance ::=  
 identifier “(“ identifier\_list “)” [ **when** boolean\_expression ]

action ::= “{“ nonempty\_action\_item\_list “}”

action\_item ::= state\_update | raise\_stmt | instantiation\_stmt

state\_update ::=   
 target = expression |  
 target( expression\_list )

raise\_stmt ::= **raise** identifier “(“ expression\_list “)”

instantiation\_stmt ::= **new** identifier “(“ state\_update\_list “)”

A scenario may be designated **atomic**, meaning that the object cannot engage in any other scenario until the end of the sequence of events in the scenario is reached. Otherwise, scenarios are interleaved, in that an incoming event may trigger execution of a scenario while another scenario may be partially executed. An event may be used in multiple scenarios. When such an event arrives, all scenarios that are ready to accept this event are triggered.

A scenario is, by definition, a linear sequence of events. Alternatives in expected behaviors can be expressed as multiple scenarios from the same named location. Additional named locations can be introduced within a scenario for that purpose. As an example, consider the PageRendering scenario shown in the overview subsection in the Tab object:

Tab → Kernel.displayURL(**this**, url) → render(rengine, url) → Tab(currentUrl=url)

Suppose we also need to express the possibility that the tab may reject rendering after a request from the kernel, as an allowed alternative to the render event, we can modify the scenario to contain two event sequences, as follows:

Tab → Kernel.displayURL(**this**, url) → RenderRequested → render(rengine, url) →  
 Tab(currentUrl=url)  
RenderRequested → reject(url) → Tab

In the modified scenario, RenderRequested is a new location that allows two alternative actions, render and reject. Note that RenderRequested is not an initial location and is reached only after displayURL event is observed.

Scenario definitions allow us to specify two events following each other without a named location in between. This is a notational convenience that allows us to avoid naming locations that not be referenced elsewhere in the model. In every such case, we syntactically introduce a unique default location into the scenario definition. In the following section, when we give semantics for SMEDL objects, we assume that events and locations alternate every scenario definition.

Actions used in scenarios contain items of three kinds: state updates, event emissions, and object instantiations. We do not spell out detailed definitions of expressions or expression lists that are used in event items. Expressions are assumed to evaluate to a value that is used in state-update assignments or to set identity variables of a new object instance. Targets of state-update assignments are state variables. If a state variable is another object, a target can be a state variable in that object or, recursively, its sub-objects. If a state variable is a collection, a target can be a method of the collection class.

**SMEDL semantics.** Behavior of a SMEDL object is given by a collection of finite-state automata extended with shared data variables. There is a separate automaton for each scenario defined for the object. All of the automata share the state variables of the object. Transitions of the automata are defined by the event sequences in the scenario. With each scenario, we associate an *alphabet*, the set of events that are used in the event sequences of the scenario.

*State of the object automata.* The current state of the object automaton is determined by the values of state variables (data state) and by the scenario state, that is, the current location within each scenario defined in the object. The scenario state is a tuple of locations, containing one location per scenario defined in the object. This location is called the current state of the scenario.

In addition, the state of an object contains a queue of outstanding event instances that have been delivered to the object but have not been reacted to by the state machines. This set is called the current input of the object.

*Transitions of the object automata.* Transitions are given by the scenario definitions and are triggered by arriving events. The event instance at the head of the input queue is applied to every scenario that has the event in its alphabet. If there is a scenario that does not have a step from the current location labeled with this event, a scenario violation is raised. Otherwise, the current location is changed to the one following the event in the selected step, and the action defined in the step is performed, updating the state. The event instance is then removed from the current input. If an executed action contains a raise event, an event instance is formed with attribute values given by the values of expressions in the raise statement. The new event instance is added to the input queues of object instances with the values of identity variables equal to the values of event attributes.

**Chromium case study.** To evaluate the use of the language for modeling security properties of real systems, we considered a variant of the cookie integrity policy shown above and applied it to the Chromium browser code. Implementation of the runtime monitor and instrumentation of the browser code has been performed manually as a proof of concept.

Informally, the cookie integrity policy states that, if the user has chosen the setting to avoid third-party cookies, a cookie can be stored by the browser only if domain of the cookie is a subdomain of the web page displayed in the current tab. To simplify the monitor implementation, we decided not to perform the check of subdomain inclusion within the browser. Instead, Chromium has a routine to perform the check of user settings. We restated the cookie integrity property to state that the cookie can be stored by the browser only if the check has been performed for the cookie, expressed as a scenario in SMEDL as follows:

CookieIntegrity: Tab → check(tab,cookie) → store(cookie) → Tab

If the browser attempts to store the cookie without performing the check first, the monitor would still be in the initial scenario location, which does not allow a store step. A scenario violation would be raised.

To implement the monitor with minimal changes to the existing Chromium codebase to avoid modifying its existing behavior and to keep our task manageable. We extended the existing CookieOptions class, which holds cookie-related properties, by adding two boolean flags to store whether a policy has been checked against this cookie and whether the checked policy passed. The CookieOptions class is included as input parameter, alongside the cookie itself, to retrieval and storage methods of the CookieMonster, which manages cookie storage in Chromium. We instrumented the calls to the storage methods to perform the checks of the flag values.

In our experiments we were able to observe successful as well as unsuccessful calls to the CookieMonster storage methods. The failed calls attempted to store a cookie, for which the flags did not record that the settings check has been performed. The failure can be caused by a flaw in Chromium or, more likely, by improper instrumentation, which failed to identify all places in the code where the checking routine is called. Even in the latter case, the outcome underscores the importance of developing tools for proper monitor construction that would avoid false alarms.

## Task 2 Results

## Task 3 Results

# Important Findings and Conclusions

<= 2 pages

Penn: List important findings and conclusions

Important conclusions from Task 1 are that capturing security properties of large software systems is possible using SMEDL and that monitors to determine property violations can be constructed.

The case study underscored the importance of developing tool support for property specification, and especially for automatic and provably correct monitor generation and deployment. We plan to develop such tool support during Phase II of the project.

MM will merge with GT findings and conclusions.

# Implications for Further Research

This should primarily be a summary of Phase 2 direction. We will derive it from the Phase 2 proposal (and Section 5 above).

# Significant Hardware/Software Development

Penn has developed monitoring code that, after manually instrumenting the Chrome browser, allowed us to check that cookies are not stored without checking user-set cookie policies. While not significant in itself, the monitoring code can serve as the prototype for automatic monitor generation and instrumentation to be developed in Phase II.

# Special Comments

I think this is a list of references.

# Contract Deliveries Status

GT will fill this in.