2009

Tisa: A Language Design and Modular Verification Technique for Temporal Policies in Web Services

Hridesh Rajan
Iowa State University, hridesh@iastate.edu

Jia Tao
Iowa State University

Steve Shaner
University of Central Florida

Gary T. Leavens
University of Central Florida

Tisa: A Language Design and Modular Verification Technique for Temporal Policies in Web Services

Abstract

Web services are distributed software components, that are decoupled from each other using interfaces with specified functional behaviors. However, such behavioral specifications are insufficient to demonstrate compliance with certain temporal non-functional policies. An example is demonstrating that a patient's health-related query sent to a health care service is answered only by a doctor (and not by a secretary). Demonstrating compliance with such policies is important for satisfying governmental privacy regulations. It is often necessary to expose the internals of the web service implementation for demonstrating such compliance, which may compromise modularity. In this work, we provide a language design that enables such demonstrations, while hiding majority of the service's source code. The key idea is to use greybox specifications to allow service providers to selectively hide and expose parts of their implementation. The overall problem of showing compliance is then reduced to two subproblems: whether the desired properties are satisfied by the service's greybox specification, and whether this greybox specification is satisfied by the service's implementation. We specify policies using LTL and solve the first problem by model checking. We solve the second problem by refinement techniques.

Disciplines

Computer Sciences | Programming Languages and Compilers

Tisa: A Language Design and Modular Verification Technique for Temporal Policies in Web Services [⋆]

Hridesh Rajan[1], Jia Tao[1], Steve Shaner[1], and Gary T. Leavens[2]

(1) Iowa State University, Ames, Iowa, USA
{hridesh,jtao,smshaner}@iastate.edu
(2) University of Central Florida, Orlando, Florida, USA leavens@eecs.ucf.edu

Abstract. Web services are distributed software components, that are decoupled from each other using interfaces with specified functional behaviors. However, such behavioral specifications are insufficient to demonstrate compliance with certain temporal non-functional policies. An example is demonstrating that a patient's health-related query sent to a health care service is answered only by a doctor (and not by a secretary). Demonstrating compliance with such policies is important for satisfying governmental privacy regulations. It is often necessary to expose the internals of the web service implementation for demonstrating such compliance, which may compromise modularity. In this work, we provide a language design that enables such demonstrations, while hiding majority of the service's source code. The key idea is to use greybox specifications to allow service providers to selectively hide and expose parts of their implementation. The overall problem of showing compliance is then reduced to two subproblems: whether the desired properties are satisfied by the service's greybox specification, and whether this greybox specification is satisfied by the service's implementation. We specify policies using LTL and solve the first problem by model checking. We solve the second problem by refinement techniques.

1 Introduction

Web services promote abstraction, loose coupling and interoperability of clients and services [1]. The key idea of web services is to introduce a published interface (often a description written in an XML-based language such as WSDL [2]), for communication between services and clients [1]. By allowing components to be decoupled using a specified interface, web services enable platform-independent integration.

Behavioral Contracts for Web Services. A behavioral contract for a web service specifies, for each of the web service's methods the relationships between its inputs and outputs. Such a contract treats the implementation of the service as a black box, hiding all the service's internal states from its clients. The benefit of this encapsulation is that clients do not depend upon the service's changeable design decisions. To illustrate, consider a healthcare service that allows patients to make appointments and ask prescription and health-related questions from healthcare practioners [3].

An example JML-like contract [4] for such a service follows.

[⋆] Rajan and Tao were supported in part by the NSF grant CNS 06-27354. Rajan, Shaner and Leavens were supported in part by the NSF grant CNS 08-08913.

```
service Patient {
   /*@ requires pId >= 0; ensures result >=0; @*/
   int query(int pId, int msg);
   /*@ requires qId >= 0; ensures result >=0; @*/
   int retrieve(int qId);
}
```

The service description in this contract is written in a form similar to our language, Tisa, to make comparisons easier. It specifies that a service named `Patient` makes two web-methods available: `query` and `retrieve`. The `query` method takes a patient identifier and a message as arguments. The message is represented as an integer for simplicity (think of it as an index into a table of pre-defined questions, such as "does the test show I have AIDS?"). The precondition of calling this web-method is that the patient identifier is positive; the postcondition is that it returns a positive result. The `retrieve` method takes a query identifier as argument; its precondition is that this identifier must be positive. Its postcondition is that the result is also positive. These contracts could be checked by observing the interface of the web-methods [5–9].

Demonstrating Compliance to Temporal Policies. Let us now consider the following policy inspired from Barth *et al.*'s work [3]: "a health question about a patient should only be answered by the doctor", "furthermore such answers should only be disclosed to the concerned patients". We will refer to these as "HIPAA policies" as they are similar to regulations in the US health insurance portability and accountability act (HIPAA). The behavioral contract above is insufficient for demonstrating compliance with the HIPAA policies, as it does not provide sufficient details about the internal state of the service. For example, the entity that is finally receiving the query is hidden by `query`'s contract. Demonstrating compliance to such policies is important. In our example, a patient may feel much better about their queries regarding an AIDS test result, if such compliances were demonstrated by the service.

Compliance and Modularity at Conflict. Alternatively suppose the implementation of the two web-methods `query` and `retrieve` were available, including the component services that they use. Then demonstrating compliance to the two HIPAA policies would be equivalent to ensuring that the implementation avoids non-compliant states. However, by making code for these methods available, clients might write code that depends on implementation design decisions. As a result, changing these design decisions will become harder, as these changes could break client's code [10].

We thus believe that, for web services, modularity [10] and verification of temporal policies are fundamentally in conflict. To make the service implementation evolvable, modularity requires hiding the design decisions that are likely to change. But to demonstrate compliance to key temporal policies, internal states need to be exposed.

A Language Design and Verification Logic. To reconcile these requirements, we propose a technique based on greybox specifications [11] that exposes only some internal states. This technique enables web service providers to demonstrate compliance to temporal policies, such that above, by exposing only parts of their implementation. A client can verify that the service complies with the desired policies by inspecting a greybox specification. Providers can also choose to hide many implementation details, so the service's implementation can evolve as long as it refines the specification [12, 13].

To illustrate, consider the greybox specification shown in Figure 1. This example has three services. In each service the methods are web-methods that may be

```
1  service Secretary {
2   int query(int pId, int msg) {
3    preserve pId > 0 && msg > 0;
4    if (msg >= 2) {
5     query(pId,msg)@Doctor
6    }
7    else {
8     /* Appointment? */
9     establish result > 0
10   }
11  }
12  int retrieve(int qId) {
13   requires qId > 0 ensures result > 0
14  }
15 }
16 service Doctor {
17  int query(int pId, int msg) { /* Re: Test */
18   requires pId > 0 && msg >= 2 ensures result > 0
19  }
20  int retrieve(int qId) {
21   requires qId > 0 ensures result > 0
22  }
23 }
24 service Patient {
25  int query(int pId, int msg) {
26   query(pId, msg)@Secretary;
27  }
28  int retrieve(int qId) {
29   preserve qId > 0;
30   if ((qId/1000)==1) { retrieve(qId)@Secretary}
31   else if ((qId/1000)==2) { retrieve(qId)@Doctor}
32  } }
```

Fig. 1. An Example Greybox Specification

called by clients and other services. Specification expressions of the form **`preserve`** e, **`establish`** e, and **`requires`** e_1 **`ensures`** e_2 are used within these methods to hide internal details. The code that is not hidden by specification expressions is exposed. Calls to web-methods are written using an at-sign (`@`), such as `query(pId, msg)@Secretary`. For simplicity, Tisa only allows integers to be passed as arguments in such remote calls, thus we encode questions using integers: 1 for appointments, 2 for prescriptions, and higher numbers for health-related questions. Contrary to standard black box specifications, internal states of the service, including calls to other services are exposed. By analyzing lines 26 and 4–6 (in that order) one could conclude that "health questions by patients are answered by the doctor." Demonstrating compliance to temporal policies thus becomes possible. Note that this specification only exposes selected details about the implementation. For example, the specification of `retrieve` on line 13 hides all details of how this service responds to appointment questions. Therefore, it hides the design decisions made in the implementation of creating, storing, and forwarding responses.

Contributions. An important contribution is the identification of the conflict between verification of temporal policies and modularity in web services. We show how to resolve this conflict using greybox specifications. Our language, *Tisa*, supports specification of policies specified in a variant of linear temporal logic [14], greybox specification [11] and a simple notion of refinement [12, 13, 15] for modular reasoning about correctness of implementations with respect to such policies. As usual, implementations are hidden, but policies and greybox specifications are public. To demonstrate these claims, we present two preliminary verification techniques: one checks if a greybox specification satisfies a temporal policy, the second checks whether a service implementation refines its greybox specification. (The first technique could be used by the clients to select a service whose specification satisfies their desired policies.) We also show soundness: that the composition of these two verification techniques, applied modularly by clients and all service providers, implies that the web service implementation satisfies the specified temporal policies. In practice, some additional technique, such as proof-carrying code [16] would be needed to satisfy clients that web services in fact satisfy their specifications.

```
program ::= decl* client
decl ::= classdecl | servicedecl
classdecl ::= class c extends d { field* meth* }
servicedecl ::= service w { field* meth* }
client ::= client w { e }
field ::= t f;
meth ::= t m (form*) { e }
form ::= t var,  where var≠this and var≠thisSite
t ::= c | int
e ::= n | e == e | e != e | e > e | e < e | e >= e | e <= e
      | e + e | e - e | e * e | ! e | e && e | e '||' e | isNull(e)
      | if (e) { e } else { e } | new c() | var
      | null | e.m(e*) | e.f | e.f = e | cast c e | form = e; e
      | e; e | w | m(e*)@e | refining spec { e }
```

$n \in \mathcal{N}$, the set of numeric, integer literals
$c, d \in \{\texttt{Object}, \texttt{Site}\} \cup \mathcal{C}$,
$\mathcal{C}$ is the set of class names
$f \in \mathcal{F}$, the set of field names
$m \in \mathcal{M}$, the set of method names
$var \in \{\texttt{this}, \texttt{thisSite}\} \cup \mathcal{V}$,
$\mathcal{V}$ is the set of variable names
$w \in \mathcal{W} \subseteq \mathcal{C}$,
$\mathcal{W}$ is the set of web service names

Fig. 2. Abstract syntax, based on [23, Figure 3.1, 3.7].

2 Tisa Language Design

In this section, we describe Tisa, an object-oriented (OO) language that incorporates ideas from existing work on specification languages, web services authentication languages and modeling languages. In particular, Tisa's design is inspired by Argus [17] and the work of Gordon and Pucella [18]. (Furthermore, some of our descriptions of the language syntax are adapted from Ptolemy [19].) Tisa is a distributed programming language with statically created web services and a single client, each of which has its own address space. Web services are named and declare web-methods, which can be called by the client and by other services. As a small, core language, the technical presentation of Tisa shares much in common with MiniMAO$_1$ [20], a variant of Featherweight Java [21] and Classic Java [22]. Tisa has classes, objects, inheritance, and subtyping, but it does not have **super**, interfaces, exception handling, built-in value types, privacy modifiers, or abstract methods. Tisa features new mechanisms for declaring policies and greybox specifications. Our description starts with its programming features, and then describes its specification features.

2.1 Program Syntax

The syntax of Tisa executable programs is shown in Figure 2 and explained below. A Tisa program consists of zero or more declarations, and a client (see Figure 3). Declarations are either class declarations or web service declarations.

Each web service has a name (w) representing that web service; thus web service names can be thought of as web sites. (The mapping of web services to actual computers is not specified in the language itself.) A web service can be thought of as a singleton object; however, each web service has a separate address space and its methods can only be called using a remote procedure call.

An example web service declaration for the service `Patient` appears on lines 49–62 in Figure 3. This service contains two web-methods declaration, named `query` and `retrieve`. The web-method `query` takes a patient Id and message as arguments and returns a unique query Id generated according to the input arguments. The web-method `retrieve` takes query Id as an argument and returns an answer message which en-

```
class Query extends Object {
 int pId; int msg; int qId;
}
class Queue extends Object { //...
 int add(int pId, int msg, int qId){
  /* add to inner list */; qId
 } }
service Secretary {
 Queue queryQ; Hashtable responses;
 int ticket; Log log;
 int query(int pId, int msg) {
  refining preserve pId > 0 && msg > 0 {
    log.recordCurrentTime()
  };
  if (msg >= 2) {
   query(pId, msg)@Doctor
  } else { /* Re: Appointment */
   refining establish result > 0 {
    ticket = ticket + 1;
    queryQ.add(pId, msg, ticket + 1000)
 } } }
 int respond(int qId,int pId,int msg){
  /* Encode patient's information */
  responses.add(qId, pId*1000 + msg);
  queryQ.remove(qId)
 }
 int retrieve(int qId) {
  refining requires qId > 0
    ensures result > 0 {
   responses.get(qId)
} } }
service Doctor {
 Queue topQ; Queue medQ; Queue lowQ;
 int query(int pId, int msg) {
  refining requires pId > 0 && msg >= 2
    ensures result > 0 {
   ticket = ticket + 1;
   if (msg > 500) {
    topQ.add(pId, msg, ticket + 2000)
   } else if (msg > 250) {
    medQ.add(pId, msg, ticket + 2000)
   } else {
    lowQ.add(pId, msg, ticket + 2000)
   };
  q.qId
 } }
 /* retrieve similar to Secretary's */
}
service Patient {
 int query(int pId, int msg) {
  query(pId, msg)@Secretary
 }
 int retrieve(int qId) {
  if ((qId/1000) == 1) {
   retrieve(qId)@Secretary
  } else if((qId/1000) == 2) {
   retrieve(qId)@Doctor
} } }
client User{
 int qid = query(101,3)@Patient;
 retrieve(qid)@Patient
}
```

Fig. 3. An Example Tisa Implementation

codes a patient Id. A client declares a name and runs an expression that is the main expression of the program. We next explain class declarations and expressions.

Class Declarations. Class declarations may not be nested. Each class has a name (c) and names its superclass (d), and may declare finite number of fields (*field**) and methods (*meth**). Field declarations are written with a class name, giving the field's type, followed by a field name. Methods also have a C++ or Java-like syntax, although their body is an expression.

Expressions. Tisa is an expression language. Thus the syntax for expressions includes integer literals, various standard integer and logical operations, several standard OO expressions and also some expressions that are specific to web services. The logical operations operate on integers, with 0 representing false, and all other integer values representing true. An **`if`** `(`e_1`)` `{` e_2 `}` **`else`** `{` e_3 `}` expression tests if e_1 is non-zero; if so it returns the value of e_2, otherwise it returns the value of e_3.

The standard OO expressions include object construction (**`new`** c`()`), variable dereference (*var*, including **`this`**), field dereference ($e.f$), **`null`**, cast (`cast` t e), assignment to a field ($e_1.f = e_2$), sequencing (e_1`;` e_2), casts and a definition block (t *var* $= e_1$`;` e_2). The other OO expressions are standard [23, 20].

There are three new expressions: web service names, web-method calls, and refining statements. Web service names of form w are constants. A *web-method call* has the form (m`(`e*`)`@e_w), where the expression following the at-sign (e_w) denotes the name of the web service name that will execute the web-method call named m with formals

```
specification ::= servicespec*
servicespec ::= service w { wmspec* }
wmspec ::= t m (form*) { se }
form ::= t var,  where var≠thisSite
spec ::= requires sp ensures sp

se ::= sp | spec | se; se| form = se; se | m (sp*) @sp
     | if (sp) { se } else { se }
sp ::= n | sp == sp | sp != sp | sp > sp | sp < sp | sp >= sp | sp <=  sp
     | sp + sp | sp − sp | sp ⋆ sp | ! sp | sp && sp | sp ‘||’ sp
     | var | w
```

Fig. 4. Syntax for Writing Specifications in Tisa

e*. A **`refining`** statement, of the form **`refining`** *spec* `{` e `}`, is used in implementing Tisa's greybox specifications (see below). It executes the expression e, which is supposed to satisfy the specification *spec* (see Figure 4).

2.2 Specification Constructs

The syntax for writing specifications in Tisa is shown in Figure 4. In this figure, all nonterminals that are used but not defined are the same as in Figure 2. Specifications consist of several service specifications (*servicespec*). (Since we only permit integers to be sent to and returned from web-method calls, we omit class declarations from specifications.) A service specification may contain finite number of web-method specifications (*wmspec*). All fields are hidden, so field declarations are not allowed in a service specification. The body of a web-method specification contains a side-effect free expression (*se*). Many expressions from Figure 2 also appear as such side-effect free expressions, but not field-related operations, method calls, and **`isNull`**. Web-method call expressions are allowed and so are local variable definition expressions.

The main new feature of specifications, borrowed from the refinement calculus and the greybox approach, is the specification expression (*spec*). Such an expression hides (abstracts from) a piece of code in a correct implementation. The most general form of specification expression is **`requires`** sp_1 **`ensures`** sp_2, where sp_1 is a precondition expression and sp_2 is a postcondition. Such a specification expression hides program details by specifying that a correct implementation contains a **`refining`** expression whose body expression, when started in a state that satisfies sp_1, will terminate in a state that satisfies sp_2 [15].

In examples we use two sugared forms of specification expression. The expression **`preserve`** sp is sugar for **`requires`** sp **`ensures`** sp and **`establish`** sp is sugar for **`requires`** `1` **`ensures`** sp.

An example greybox specification of the web service `Patient` appears in Figure 1. The specification of the web-method `query` appears on line 26, and specifies (and thus exposes) all the code for that method. The specification of `retrieve` hides a bit more in its **`preserve`** expression (line 29). But it also exposes code that makes a web-method call `retrieve` to the `Secretary` or `Doctor`. With these greybox specifications, enough details are exposed about what the service does when invoking other services, which makes it feasible to show compliance to the HIPAA policies.

2.3 Constructs for Specifying Policies

Our simple policy specification language is similar to Linear Temporal Logic [14].

$$\Phi(\textit{specification}) ::= \mathcal{P}(\textit{specification}) \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1\, \mathbf{U}\, \phi_2 \mid \mathbf{X}\, \phi$$

The language specifies histories that are sequences of web method calls. For a given *specification*, a policy can be an atomic proposition in $\mathcal{P}(\textit{specification})$; a negation of a policy or boolean combination of policies. For simplicity here we take the set of legal propositions $\mathcal{P}(\textit{specification})$ to be all legal web-method calls in the given specification. This set can be statically computed from the specification against which the policy is to be verified by traversing the abstract syntax tree of the specification up to the depth of web-method specifications. The operator **U** is read as "until" and **X** as "next." $\phi_1 \mathbf{U} \phi_2$ states that policy ϕ_2 must be satisfied after policy ϕ_1 is satisfied along all executions of the service. $\mathbf{X}\phi$ states that policy ϕ must be satisfied in the next state (i.e., at the next web method call). We also use the following common abbreviations:

$$\begin{array}{lll} \phi_1 \vee \phi_2 \equiv \neg(\neg\phi_1 \wedge \neg\phi_2) & \phi_1 \rightarrow \phi_2 \equiv \neg\phi_1 \vee \phi_2 & \textit{true} \equiv \phi \vee \neg\phi \\ \textit{false} \equiv \neg\textit{true} & \mathbf{F}\,\phi \equiv \textit{true}\,\mathbf{U}\,\phi & \mathbf{G}\,\phi \equiv \neg\mathbf{F}\,\neg\phi \end{array}$$

The constant *true* means that the service does not have any obligation. The operator **F** is read as "eventually" and **G** as "always". Below we present two sample policies for our healthcare service example.

$\phi_1 = \mathbf{G}(\texttt{query@Patient} \wedge (\mathbf{XF}(\texttt{query@Secretary} \vee \mathbf{XF}\texttt{query@Doctor})))$
$\phi_2 = \mathbf{G}(\texttt{retrieve@Patient} \wedge \mathbf{XF}\texttt{retrieve@Doctor} \rightarrow \neg\ \mathbf{XF}\texttt{retrieve@Secretary})$

The policy ϕ_1 states that whenever there is a web-method call `query@Patient`, there is eventually a web-method call `query` at one of the sites `Secretary` or `Doctor`. This policy says that a query is eventually delivered to one of the healthcare providers. The policy ϕ_2 encodes the constraint that a health answer that comes from doctors goes directly to the patient, and is never forwarded to secretaries. In terms of the service specification, if there is a web-method call `retrieve@Patient` and it is followed by a web-method call `retrieve@Doctor`, then there is never a web-method call `retrieve` at the site `Secretary` in the same trace.

2.4 Dynamic Semantics of Tisa's Constructs

This section defines a small step operational semantics for Tisa programs (adapted from Clifton's work [23]). In the semantics, all declarations are formed into a single class table that maps class names and web service names to class and service declarations, respectively. However, despite this global view of declarations, the model of storage is distributed, with each web service having an independent store.

The operational semantics relies on four expressions, not part of Tisa's surface syntax, to record final or intermediate states of the computation. The *loc* expression represents locations in the store. The **`under`** expression is used as a way to mark when the evaluation stack needs popping. The **`evalbody`** and **`evalpost`** are used in evaluation of specification expressions. The three exceptions `NullPointerException`, `ClassCastException`, and `SpecException` record various problems orthogonal to the type system.

A configuration in the semantics contains an expression (e), an evaluation stack (J), and a store (S). The current web service name is maintained in the evaluation stack under the name **`thisSite`**. The auxiliary function *thisSite* extracts the current web service name from a stack frame. Stacks are an ordered list of frames, each frame recording the static environment, ρ, and a type environment. (The type environment, Π, Evaluation relation: $\hookrightarrow : \Gamma \rightarrow \Gamma$

(WEB METHOD CALL)

$$\frac{\begin{array}{c}\Pi = \{var_i : \mathbf{var}\ t_i \mid 1 \leq i \leq n\} \uplus \{\mathbf{this} : \mathbf{var}\ c_2\} \uplus \{\mathbf{thisSite} : \mathbf{var}\ \mathtt{Site}\} \qquad \nu = \mathbf{frame}\ \rho\ \Pi \\ \rho = \{var_i \mapsto v_i \mid 1 \leq i \leq n\} \oplus (\mathbf{this} \mapsto loc) \oplus (\mathbf{thisSite} \mapsto w) \\ (loc, c_2, t\ m(t_1 var_1, \ldots t_n var_n)\{e\}) = \mathit{find}(w, m)\end{array}}{\langle \mathbb{E}[m(v_1, \ldots, v_n)@w], J, S\rangle \hookrightarrow \langle \mathbb{E}[\mathbf{under}\ e], \nu + J, S\rangle}$$

(REFINING)

$$\frac{n \neq 0}{\left\langle \mathbb{E}[\mathbf{refining\ requires}\ n\ \mathbf{ensures}\ e'\ \{e''\}], J, S\right\rangle \hookrightarrow \left\langle \mathbb{E}[\mathbf{evalbody}\ e'' e'], J, S\right\rangle}$$

(EVALBODY)

$$\frac{\begin{array}{c}\rho = envOf(\nu) \qquad \Pi = tenvOf(\nu) \qquad w = thisSite(\nu) \qquad t = typeOf(v, S, w) \\ \rho' = \Pi \uplus \{\mathbf{result} : v\} \qquad \Pi' = \Pi \uplus \{\mathbf{result} : \mathbf{var}\ t\} \qquad \nu' = \mathbf{frame}\ \rho'\ \Pi'\end{array}}{\left\langle \mathbb{E}[\mathbf{evalbody}\ v\ e'], \nu + J, S\right\rangle \hookrightarrow \left\langle \mathbb{E}[\mathbf{under\ evalpost}\ v\ e'], \nu' + \nu + J, S\right\rangle}$$

(EVALPOST)

$$\frac{n \neq 0}{\langle \mathbb{E}[\mathbf{evalpost}\ v\ n], J, S\rangle \hookrightarrow \langle \mathbb{E}[v], J, S\rangle}$$

(UNDER)

$$\langle \mathbb{E}[\mathbf{under}\ v], \nu + J, S\rangle \hookrightarrow \langle \mathbb{E}[v], J, S\rangle$$

Fig. 5. Operational semantics of Tisa. Standard OO rules are omitted.

is only used in the type soundness proof.) The static environment ρ maps identifiers to values. A value is a number, a web service name (site), a location, or **`null`**. Stores are maps from locations to storable values, which are object records. Object records have a class and also a map from field names to values.

The semantics is presented as a set of evaluation contexts $\mathbb{E}$ and an one-step reduction relation [24] that acts on the position in the overall expression identified by the evaluation context as shown in Figure 5. Standard OO rules are presented in our technical report [25]. The key rule is (WEB METHOD CALL), which uses the auxiliary function `find` to retrieve the body of the web method from a class table CT implicitly used by the semantics. It creates the frame for execution of the web method with necessary static environment and type environment and starts execution of the web method body. The **`under`** e expression is used in the resulting configuration to mark that the stack should be popped when the evaluation of e is finished.

Evaluation of a **`refining`** expression involves 3 steps. First the precondition is evaluated (due to the context rules). If the precondition is non-zero (i.e., true), then the next configuration is **`evalbody`** $e''\ e'$, where e'' is the body and e' is the postcondition (regarded as an expression). The body is then evaluated; if it yields a value v, then the next configuration is **`under`** `evalpost` $v\ e'$, with a new stack frame that binds **`result`** to v pushed on the stack. The type of **`result`** in the type environment Π' is determined by the auxiliary function $typeOf$. Finally, the (EVALPOST) rule checks that the postcondition is true and uses the body's value as the value of the expression.

3 Verification of Policies in Tisa

A key contribution of our work is to decouple, with Tisa's language design, the verification of whether a policy is satisfied by a web service implementation into two verification tasks that can proceed modularly and independently. The first task is to verify whether a policy is satisfied by the service specification. The second task is to verify whether the service specification is satisfied by the service implementation. Three benefits follow from this modular approach. First, the service implementation need not be visible to clients, as a client uses the specification to determine whether their desired policies hold. Thus, our approach achieves modularity for service implementations. Second, regardless of the number of clients, the second verification task must only be done once; thus our approach is likely to be scalable for web service providers. Last but not the least, policy verification is performed on the (generally smaller) specification. Thus, our approach has efficiency benefits for policy verification.

Determining whether a policy is satisfied by the specification can be reduced to a standard model checking problem [14]. We claim no contribution here; rather, the novelty of our approach is in a combination of these two techniques, enabled by a careful language design. To show the feasibility of applying ideas from model checking [14] and refinement calculus [12, 13] to our problem, in the rest of this section we describe our techniques for verifying policies and refinement.

3.1 Verifying Policies

We adopt the standard automata-theoretic approach for verifying linear temporal logic formulas proposed by Vardi and Wolper [26] to verify policies in Tisa. Following Vardi and Wolper [26], a policy $\phi \in \varPhi(\mathcal{S})$ is viewed as a finite-state acceptor and a specification $\mathcal{S}$ as a finite-state generator of expression execution histories. Thus the specification $\mathcal{S}$ satisfies policy ϕ if every (potentially infinite) history generated by $\mathcal{S}$ is accepted by ϕ, in other words, if $\mathcal{S} \cap \neg\phi$ is empty.

Figure 6 shows main parts of an algorithm for constructing a finite-state machine $\mathcal{F}(\mathcal{S}) = (\mathcal{Z}, z_0, R, \varDelta)$ from a Tisa specification $\mathcal{S}$. Here, Z is a finite set of states, z_0 is the initial state, R is a total accessibility relation, $\varDelta : Z \rightarrow 2^{\mathcal{P}(\mathcal{S})}$, which determines how truth values are assigned to propositions in each state [26, pp. 5]. All rules make use of unions for joining set of states (Z) and disjoint union ($\uplus$) for joining propositions. Rules for standard OO expressions are omitted.

The (IF EXP FSM) rule demonstrates creation of non-deterministic transitions in the state machine. It computes the FSMs corresponding to the true branch and the false branch of the **`if`** expression with initial states z' and z'' and joins these two FSMs to make a new FSM with initial state z. Corresponding to the state z', which corresponds to the true branch, the proposition sp is added to $\varDelta$, which corresponds to the conditional expression evaluating to the truth value true. Similarly for the state z'', which corresponds to the false branch, the proposition $!sp$ is added to $\varDelta$, which corresponds to the conditional expression evaluating to the truth value false.

The (SPEC EXP FSM) rule models the cases for satisfaction of precondition and postcondition. The (WEB METHOD CALL FSM) rules make use of a table *NT* that maps pairs of web service names and method names (w, m) to states. This table is used to account for recursion in web-method calls. Finally, the finite-state machine for a service specification is created by first creating finite-state machines for each of its web-method specifications as if it is being called and by joining them using an extra state that becomes the new initial state.

Production relation: $NT \vdash se \rightsquigarrow (Z, z_0, R, \Delta), NT$ where $NT \in \mathcal{NT} = \mathcal{W} \times \mathcal{M} \rightarrow Z$

(IF EXP FSM)

$$\frac{NT \vdash se' \rightsquigarrow (Z', z', R', \Delta'), NT' \quad NT' \vdash se'' \rightsquigarrow (Z'', z'', R'', \Delta''), NT'' \quad Z = Z' \cup Z'' \cup \{z\} \\ \Delta = \Delta' \uplus \Delta'' \uplus \{(z', \{sp\}), (z'', \{!sp\})\} \quad R = R' \cup R'' \cup \{(z, z'), (z, z'')\}}{NT \vdash \texttt{if}\ (sp)\ \{se'\}\ \texttt{else}\ \{se''\} \rightsquigarrow (Z, z, R, \Delta), NT''}$$

(WEB METHOD CALL FSM 1)

$$\frac{\neg(\exists z :: NT(w, m) = z) \\ NT' = NT \cup ((w, m), z) \quad m(t_1, \ldots t_n)\{se\} = \mathit{find}(w, m) \quad NT' \vdash se \rightsquigarrow (Z', z', R', \Delta'), NT'' \\ Z = Z' \cup \{z\} \quad \Delta = \Delta' \uplus \{(z', \{m@w\})\} \quad R = R' \cup \{(z, z')\}}{NT \vdash m(v_1, \ldots, v_n)@w \rightsquigarrow (Z, z, R, \Delta), NT''}$$

(WEB METHOD CALL FSM 2)

$$\frac{z = NT(w, m)}{NT \vdash m(v_1, \ldots, v_n)@w \rightsquigarrow (\{z\}, z, \{\}, \{\}), NT}$$

(SPEC EXP FSM)

$$\frac{Z = \{z_1, z_2, z_3, z_4\} \quad R = \{(z, z_1), (z, z_2), (z_1, z_3), (z_1, z_4), (z_3, z')\} \\ \Delta_{pre} = \{(z_1, \{sp_1\}), (z_2, \{!sp_1\})\} \quad \Delta = \Delta_{pre} \uplus \{(z_3, \{sp_1, sp_2\}), (z_4, \{sp_1, !sp_2\})\}}{NT \vdash \texttt{requires}\ sp_1\ \texttt{ensures}\ sp_2 \rightsquigarrow (Z, z, R, \Delta), NT}$$

Fig. 6. Finite-state machine construction, built from expressions in a specification.

Given the FSM $\mathcal{F}(\mathcal{S})$ we construct a Büchi automaton [27], $\mathcal{B}(\neg\phi)$ for the policy $\phi \in \Phi(\mathcal{S})$ as shown by Vardi and Wolper [26]. Specification $\mathcal{S}$ satisfies the policy ϕ if $\mathcal{F}(\mathcal{S}) \cap \mathcal{B}(\neg\phi)$ is empty.

3.2 Verifying Refinement

Our technique for checking whether a program refines a specification in Tisa is similar to the work of Shaner, Leavens and Naumann [15]. An implementation refines a specification if it meets two criteria: first, that the code and specification are structurally similar and second, that the body of every **`refining`** expression obeys the specification it is refining. By structural similarity we mean that for every non-specification expression in the specification, the implementation has the identical expression at that position in the code. This is checked in a top-down manner as shown in Figure 7. The operational semantics rules (REFINING), (EVALBODY) and (EVALPOST) ensure that the body of every **`refining`** expression obeys the specification it is refining.

3.3 Soundness of Verification Technique

The proof of soundness of our verification technique uses the following three definitions.

Definition 1 (A Path for $\mathcal{S}$). *Let $\mathcal{S}$ be a specification and $\mathcal{F}(\mathcal{S}) = (\mathcal{Z}, z_0, R, \Delta)$ be the FSM for $\mathcal{S}$ constructed using algorithm shown in Figure 6. A* path *t* for *$\mathcal{S}$ is a (possibly infinite) sequence of pairs $(z_i, \Delta(z_i))$ starting with pair $(z_0, \Delta(z_0))$, where for each $i \geq 0$, $z_i \in Z$ and $(z_i, z_{i+1}) \in R$.*

(PROGRAM REF)
$$\frac{\forall i \in \{1..m\}\ \exists j \in \{1...n\}\ decl_j \in servicedecl \wedge servicespec_i \sqsubseteq decl_j}{servicespec_1 \ldots servicespec_m \sqsubseteq decl_1 \ldots decl_n}$$

(SP REF)
$$\frac{sp = e}{sp \sqsubseteq e}$$

(SERVICE REF)
$$\frac{\forall i \in \{1..m\}\ \exists j \in \{1...n\}\ wmspec_i \sqsubseteq meth_j}{\begin{array}{c}\texttt{service}\, w\, \{wmspec_1 \ldots wmspec_n\} \\ \sqsubseteq \texttt{service}\, w\, \{field_1 \ldots field_f\ meth_1 \ldots meth_n\}\end{array}}$$

(WEB METHOD REF)
$$\frac{se \sqsubseteq e}{\begin{array}{c} t\, m(form_1 \ldots form_n)\, \{se\} \\ \sqsubseteq t\, m(form_1 \ldots form_n)\, \{e\}\end{array}}$$

(SEQ EXP REF)
$$\frac{se_1 \sqsubseteq e_1 \qquad se_2 \sqsubseteq e_2}{\begin{array}{c} se_1\,;\, se_2 \\ \sqsubseteq e_1\,;\, e_2\end{array}}$$

(IF EXP REF)
$$\frac{sp \sqsubseteq e_b \qquad se_T \sqsubseteq e_T \qquad se_F \sqsubseteq e_F}{\begin{array}{c}\texttt{if}\,(sp)\,\{se_T\}\,\texttt{else}\,\{se_F\} \\ \sqsubseteq \texttt{if}\,(e_b)\,\{e_T\}\,\texttt{else}\,\{e_F\}\end{array}}$$

(DEF EXP REF)
$$\frac{sp \sqsubseteq e_{init} \qquad se \sqsubseteq e_{body}}{\begin{array}{c} form = sp; se \\ \sqsubseteq form = e_{init}; e_{body}\end{array}}$$

(WEBCALL EXP REF)
$$\frac{(\forall i \in \{1..n\} :: sp_i \sqsubseteq e_i) \qquad sp_w \sqsubseteq e_w}{m(sp_1, \ldots, sp_n)@sp_w \sqsubseteq m(e_1, \ldots, e_n)@e_w}$$

(SPEC EXP REF)
$$\frac{(\texttt{requires}\ sp_1\ \texttt{ensures}\ sp_2) = spec}{\texttt{requires}\ sp_1\ \texttt{ensures}\ sp_2 \sqsubseteq \texttt{refining}\, spec\, \{e\}}$$

Fig. 7. Inference rules for proving Tisa refinement.

Definition 2 (A Path for P). *Let P be a program and $\mathcal{CFG}(P) = (Z', z'_0, R', \Delta')$ be an annotated control flow graph for P, where Z' is the set of nodes representing expressions in program, R' is the control flow relation between nodes, and $\Delta' : Z' \rightarrow 2^{\mathcal{P}(P)}$ is such that for each $z'_i \in Z'$, if it represents a web-method call expression $m(..)@w$ then $(z'_i, \{m@w\}) \in \Delta'$. A* path *$t'$* for *$P$ is a (possibly infinite) sequence of pairs $(z'_i, \Delta(z'_i))$ starting with pair $(z'_0, \Delta(z'_0))$, where for each $i \geq 0$, $z'_i \in Z$ and $(z'_i, z'_{i+1}) \in R'$.*

Definition 3 (Path Refinement). *Let t be a path for $\mathcal{S}$ and t' be a path for P. Then t* is refined by *t', written $t \sqsubseteq t'$, just when one of the following holds:*

- *$t \equiv t'$ i.e., for each $i \geq 0$, $(z_i, \delta_i) \in t$ and $(z'_i, \delta'_i) \in t'$ implies $z_i = z'_i$ and $\delta_i = \delta'_i$,*
- *$t = (z, \delta) + t_1$ and $t' = (z', \delta') + t'_1$ and $\delta \Rightarrow \delta'$ and $t_1 \sqsubseteq t'_1$,*
- *$t = (z, \delta) + t_1$ and $t' = (z'_1, \delta'_1) + \ldots + (z'_n, \delta'_n) + t'_1$ and $\delta \Rightarrow (\delta'_1 \uplus \ldots \uplus \delta'_n)$ and $t_1 \sqsubseteq t'_1$, or*
- *$t = t_1 + t_2$ and $t' = t'_1 + t'_2$ and $t_1 \sqsubseteq t'_1$ and $t_2 \sqsubseteq t'_2$.*

Lemma 1. *Let $P \in$ program and $\mathcal{S} \in$ specification be given. If P refines $\mathcal{S}$, then for each path t' for P there exists a path t for $\mathcal{S}$ such that $t \sqsubseteq t'$.*

Proof Sketch: The proof for this lemma follows from structural induction on the refinement rules shown in Figure 7. Details are contained in Section A.

Lemma 2. *Given a specification $\mathcal{S}$ and a policy $\phi \in \Phi(\mathcal{S})$, the automaton $\mathcal{F}(\mathcal{S}) \cap \mathcal{B}(\neg\phi)$ accepts a language, which is empty when the specification satisfies the policy.*

The proof of this lemma follows from standard proofs in model checking, in particular, from Lemma 3.1, Theorem 2.1 and Theorem 3.3. given by Vardi and Wolper [26, pp. 4,6]. Details are contained in Section A.

Theorem 1. *Let $\mathcal{S}$ be a specification, ϕ be a policy in $\Phi(\mathcal{S})$, and P be a program. Let ϕ be satisfied by the specification $\mathcal{S}$ and P be a refinement of $\mathcal{S}$ (as defined in Figure 7). Then the policy ϕ is satisfied by the program P.*

Proof Sketch: The proof follows from lemma 1 and 2. From lemma 1, we have that each path in the program refines a path in the specification. From lemma 2 and the assumptions of this theorem, we have that ϕ is satisfied on all paths in $\mathcal{S}$. Thus, ϕ, which is written over $\mathcal{P}(\mathcal{S})$, is also satisfied for P.

4 Related Work

In this section, we discuss techniques that are closely related to our approach.

Greybox specifications. We are not the first to consider greybox specifications [11] as a solution for verification problems. Barnett and Schulte [28, 29] have considered using greybox specifications written in AsmL [30] for verifying contracts for .NET framework. Wasserman and Blum [31] also use a restricted form of greybox specifications for verification. Tyler and Soundarajan [32] and most recently Shaner, Leavens, and Naumann [15] have used greybox specifications for verification of methods that make mandatory calls to other dynamically-dispatched methods. Compared to these related ideas, to the best of our knowledge our work is the first to consider greybox specifications as a mechanism to decouple verification of web services without exposing all of their implementation details. Secondly, most of these, e.g. Shaner, Leavens, and Naumann [15] use the refinement of Hoare logic as their underlying foundation. This was insufficient to tackle the problem that we address, which required showing refinement of (a variant of) linear temporal logic. Thus adaptation of much of their work was not possible, although we were able to adapt the notion of structural refinement.

Specification and Verification Techniques for Web Services. The technique proposed by Bravetti and Zavattaro [33] for determining whether the behavioral contract of a service correctly refines its desired requirements in a composition of web-services is closely related and complementary to this work. The main difference between this work and the current work is that we verify refinement of greybox specifications by service implementations that allows us to reason about temporal policies, while hiding much of the implementation. However, we foresee a combination of our work and Bravetti and Zavattaro's work for determining fitness of a service implementation in a desired composition of web-services.

Some approaches have recently been proposed to verify contracts for web services, as seen in the works of Acciai and Boreale [34], Kuo *et al.* [8], Baresi *et al.* [6], Barbon *et al.* [5], etc. These ideas focus on verifying the behavioral contracts as defined by the externally visible interface of the web services, whereas our work provides a principled, modular technique for verifying such policies that require inspecting the web service implementation to a limited extent.

Castagna, Gesbert and Padovani present a formalism for specifying web services based on the notion of "filtering" the possible behaviors of an existing web service to conform to the behavior of some contract [7]. These filters take the form of coercions that limit when and how an available service may be consumed. These coercions permits contract subtyping and support reasoning in a language-independent way about the sequence of reads and writes performed between service clients and providers. Their contracts are intended to constrain the usage scenarios of a web service, whereas the present work describes a modular way to specify the observable behaviors that occur inside service implementations.

Bartoletti *et al.* [35] provide a formalization of web service composition in order to reason about the security properties provided by connected services. While they ignore policy language details, our work shows how the amount of overhead used to relate specifications to policies depends on the level of detail in the policy language. Furthermore, we believe greybox reasoning grants real benefits in readability and modularity over their type system. We view later work developing executable specifications for design of web services [36] as possible future work for Tisa.

Another approach [37] proposes an architecture to enforce these access policies at component web services, but again the work is tightly coupled to the WS-SensFlow and Axis implementations. Srivatsa *et al.* [38] propose an Access Control system for composite services which does not take care of the Trust in the resulting service oriented architecture. Skalka and Wang [39] introduced a trust but verify framework which is an access control system for web services, but they do not provide temporal reasoning for the verification of policies. By recording the sequence of program events in temporal order, Skalka and Smith [40] are able to verify the policies such as whether the events were happened in a reasonable order, but the mechanism does not support decoupling the model and the implementation. Other approaches [41, 42] either do not have a formal model supporting them or are tightly coupled with implementations.

Future Work and Conclusions

We have designed Tisa to be a small core language to clearly communicate how it allows users to balance compliance and modularity in web service specification. However, our desire for simplicity and clarity led us to leave for future work many practical and useful extensions. The most important future work in the area of Tisa's semantics is to investigate refinement of information flow properties. It would also be interesting to investigate the utility of Tisa's specification forms for reasoning about the composition of web services.

Verifying web services is an important problem [7, 5, 6, 8, 9], which is crucial for wider adoption of this improved modularization technique that enables new integration possibilities. There are several techniques for verifying web-services using behavioral interfaces, but none facilitates verification that requires access to internal states of the service. To that end, the key contribution of this work is to identify the conflict between verification of temporal properties and modularity requirements in web services. Our language design, Tisa, addresses these challenges. It allows service providers to demonstrate compliance to policies expressed in an LTL-like language [14]. We also showed that policies in Tisa can be verified by clients using just the specification. Furthermore, refinement of specifications by program ensures that conclusion drawn from verifying policies are valid for Tisa programs. Another key benefit of Tisa is that its greybox specifications [11] allow service providers to encapsulate changeable implementation details by hiding them using a combination of *spec* and **`refining`** expressions. Thus, Tisa provides significant modularity benefits while balancing the verification needs.

References

1. Papazoglou, M.P., Georgakopoulos, D.: Service-oriented computing: Introduction. Commun. ACM **46**(10) (2003) 24–28
2. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web services description language (WSDL) 1.1. Technical report, World Wide Web Consortium (March 2001)
3. Barth, A., Mitchell, J., Datta, A., Sundaram, S.: Privacy and utility in business processes. In: CSF'07. 279–294
4. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: a behavioral interface specification language for Java. SIGSOFT Softw. Eng. Notes **31**(3) (2006) 1–38
5. Barbon, F., Traverso, P., Pistore, M., Trainotti, M.: Run-time monitoring of instances and classes of web service compositions. In: ICWS '06. 63–71
6. Baresi, L., Ghezzi, C., Guinea, S.: Smart monitors for composed services. In: ICSOC '04. 193–202
7. Castagna, G., Gesbert, N., Padovani, L.: A theory of contracts for web services. In: POPL '08. 261–272
8. Kuo, D., Fekete, A., Greenfield, P., Nepal, S., Zic, J., Parastatidis, S., Webber, J.: Expressing and reasoning about service contracts in service-oriented computing. In: ICWS '06. 915–918
9. Wada, H., Suzuki, J., Oba, K.: Modeling non-functional aspects in service oriented architecture. In: IEEE International Conference on Services Computing (SCC'06). (2006) 222–229
10. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Communications of the ACM **15**(12) (December 1972) 1053–8
11. Büchi, M., Weck, W.: The greybox approach: When blackbox specifications hide too much. Technical Report 297, Turku Center for Computer Science (August 1999)
12. Back, R.J.R., von Wright, J.: Refinement calculus, part i: sequential nondeterministic programs. In: REX workshop. (1990) 42–66
13. Morris, J.M.: A theoretical basis for stepwise refinement and the programming calculus. Sci. Comput. Program. **9**(3) (1987) 287–306
14. Edmund M. Clarke, J., Grumberg, O., Peled, D.A.: Model checking. MIT Press, Cambridge, MA, USA (1999)
15. Shaner, S.M., Leavens, G.T., Naumann, D.A.: Modular verification of higher-order methods with mandatory calls specified by model programs. In: OOPSLA '07. 351–368
16. Necula, G.C.: Proof-carrying code. In: POPL '97. 106–119
17. Liskov, B., Scheifler, R.: Guardians and actions: Linguistic support for robust, distributed programs. TOPLAS **5**(3) (July 1983) 381–404
18. Gordon, A.D., Pucella, R.: Validating a web service security abstraction by typing. Formal Aspects of Computing **17**(3) (2005) 277–318
19. Rajan, H., Leavens, G.T.: Ptolemy: A language with quantified typed events. In: 22nd European Conference on Object-oriented Programming (ECOOP 2008). (July 2008)
20. Clifton, C., Leavens, G.T.: MiniMAO$_1$: Investigating the semantics of proceed. Science of Computer Programming **63**(3) (2006) 321–374
21. Igarashi, A., Pierce, B., Wadler, P.: Featherweight Java: A minimal core calculus for Java and GJ. In: OOPSLA '99. 132–146
22. Flatt, M., Krishnamurthi, S., Felleisen, M.: A programmer's reduction semantics for classes and mixins. In: Formal Syntax and Semantics of Java. Springer-Verlag (1999) 241–269

23. Clifton, C.: A design discipline and language features for modular reasoning in aspect-oriented programs. Technical Report 05-15, Iowa State University (Jul 2005)
24. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Information and Computation **115**(1) (Nov 1994) 38–94
25. Rajan, H., Tao, J., Shaner, S.M., Leavens, G.T.: Reconciling trust and modularity in web services. Technical Report 08-07, Dept. of Computer Sc., Iowa State U. (July 2008)
26. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: Proceedings of the First Symposium on Logic in Computer Science. (1986) 322–331
27. Buchi, J.: On a decision method in restricted second order arithmetic. Proc. Internat. Congr. Logic, Method. and Philos. Sci (1960) 1–12
28. Barnett, M., Schulte, W.: Runtime verification of .net contracts. Journal of Systems and Software **65**(3) (March 2003) 199–208
29. Barnett, M., Schulte, W.: Spying on components: A runtime verification technique. In: Workshop on Specification and Verification of Component-Based Systems. (2001)
30. Barnett, M., Schulte, W.: The ABCs of specification: AsmL, Behavior, and Components. Informatica **25**(4) (November 2001) 517–526
31. Wasserman, H., Blum, M.: Software reliability via run-time result-checking. J. ACM **44**(6) (1997) 826–849
32. Tyler, B., Soundarajan, N.: Black-box testing of grey-box behavior. In: FATES '03, 1–14.
33. Bravetti, M., Zavattaro, G.: Towards a unifying theory for choreography conformance and contract compliance. In: Software Composition. (2007) 34–50
34. Acciai, L., Boreale, M.: XPi: A typed process calculus for XML messaging. Science of Computer Programming **71**(2) (2008) 110–143
35. Bartoletti, M., Degano, P., Ferrari, G.L.: Types and effects for secure service orchestration. In: CSFW. (2006) 57–69
36. Bartoletti, M., Degano, P., Ferrari, G.L., Zunino, R.: Semantics-based design for secure web services. IEEE Trans. Software Eng. **34**(1) (2008) 33–49
37. Wei, J., Singaravelu, L., Pu, C.: Guarding sensitive information streams through the jungle of composite web services. In: ICWS '07. 455–462
38. Srivatsa, M., Iyengar, A., Mikalsen, T., Rouvellou, I., Yin, J.: An access control system for web service compositions. In: ICWS '07. 1–8
39. Skalka, C., Wang, X.S.: Trust but verify: authorization for web services. In: SWS. (2004) 47–55
40. Skalka, C., Smith, S.F.: History effects and verification. In: APLAS. (2004) 107–128
41. Biskup, J., Carminati, B., Ferrari, E., Muller, F., Wortmann, S.: Towards secure execution orders for composite web services. In: ICWS '07. 489–496
42. Vorobiev, A., Han, J.: Specifying dynamic security properties of web service based systems. In: SKG '06. 34–34
43. Choueka, Y.: Theories of automata on ω-tapes: A simplified approach. Journal of Computer and System Sciences **8**(2) (1974) 117–141
44. Emerson, E.A., Lei, C.L.: Modalities for model checking: branching time logic strikes back. Sci. Comput. Program. **8**(3) (1987) 275–306

A Appendix: Omitted Proofs

Lemma 3. *If the atomic propositions in a specification $\mathcal{S}$ are $\mathcal{P}(\mathcal{S})$ and the atomic propositions in program P are $\mathcal{P}(P)$, and* `P` *refines the specification $\mathcal{S}$ then $\mathcal{P}(\mathcal{S}) \subseteq \mathcal{P}(\mathcal{P})$.*

The proof of this lemma follows from construction of $\mathcal{P}$ and structural refinement rules shown in Figure 7. The construction of $\mathcal{P}$ picks all potential web-method calls as propositions and the refinement ensures that all web-method specifications in $\mathcal{S}$ have a corresponding web-method declaration in `P`.

Lemma 4. *Let $P \in$ program be given. If t' is a path for P, then there are paths t'_{pre} and t'_{loop} such that $t' = t'_{pre} + t'_{loop}$, t'_{pre} has finite length, and each $(z', \delta') \in t'_{loop}$ occurs infinitely often in t'_{loop}.*

Proof Sketch: If t' has finite length, then let $t'_{pre} = t'$ and t'_{loop} be the empty path.

If t' has infinite length, then since P has only a finite number of expressions, it must loop at some point. Consider all the states that occur infinitely often in t', and let t'_{loop} be the longest suffix of t' that contains only such states. Let t'_{pre} be the unique prefix of t' such that $t' = t'_{pre} + t'_{loop}$. ∎

Lemma 5. *Let $\mathcal{S}$ be a specification and let P be a program such that $\mathcal{S}$ is refined by P. Let $t + (z_{n-1}, \delta_{n-1})$ be a path for $\mathcal{S}$ and let $t' + (z'_{n-1}, \delta'_{n-1}) + (z'_n, \delta'_n)$ be a path for P. If $\delta_{n-1} \Rightarrow \delta'_{n-1}$, then there is some (z_n, δ_n) such that $t + (z_{n-1}, \delta_{n-1}) + (z_n, \delta_n)$ is a path for $\mathcal{S}$ and $\delta_n \Rightarrow \delta'_n$.*

Proof Sketch: From the definition of path for P, we have that z'_{n-1} represents an expression in P and that there is a control flow relation from z'_{n-1} to z'_n. From the derivation rules for expressions in programs, we have the following cases.

Case **`if-true`**: z'_{n-1} represents the **`if`** expression and z'_n represents the true expression. Case **`if-false`**: z'_{n-1} represents the **`if`** expression and z'_n represents the false expression. Case seq: z'_{n-1} represents the first expression and z'_n represents the second expression in the sequence. Case def: z'_{n-1} represents the definition expression and z'_n represents the second expression in the variable definition. Case refining: z'_{n-1} represents the **`refining`** expression and z'_n represents the body expression of the **`refining`** expression. Case web-method call: z'_{n-1} represents the web-method call expression and z'_n represents the body expression of the web-method.

From the assumptions we have that $\mathcal{S}$ is refined by P. From the refinement rules we have that for each expression represented by z'_{n-1} and z'_n above there is a corresponding expression se_{m-1} and se_m in $\mathcal{S}$ and the structure of these expressions and their relative order is identical. By the construction of the FSM (Figure 6) we have that for each case above corresponding to se_{m-1} and se_m there is some state z_{m-1} and z_m in Z and $(z_{m-1}, z_m) \in R$. Thus $t_{m-1} = t'_{m-1} + (z_m, \delta_m)$ is a path for P. Also for all cases except web-method call, there are no new atomic propositions corresponding to z'_n and z_m in program and specification, thus $\delta_m \Rightarrow \delta'_n$ is vacuously true.

For the case web-method call by the construction of the FSM (Figure 6), we have that the new set of propositions $\delta_m = \{m@w\}$. From the refinement rule for web-method call, we have that identical web-method call occurs in the program. From the definition of a path for P, we have that for each such occurence of a web-method call the new of propositions $\delta'_n = \{m@w\}$. Thus $\delta_m \Rightarrow \delta'_n$ holds. ∎

Proof of Lemma 1. Let $P \in$ *program* and $\mathcal{S} \in$ *specification* be given. If P refines $\mathcal{S}$, then for each path t' for P there exists a path t for $\mathcal{S}$ such that $t \sqsubseteq t'$.

Proof Sketch: Suppose P refines $\mathcal{S}$. Let t' be a path for P.

The proof is by transfinite induction, using the various cases discussed in Figure 7 that could generate t'. The well-ordering on paths that is used is that $t_1 < t_2$ if and only if t_1 is a finite, proper prefix of t_2.

Base case: Let t' be the empty path. Then by definition of refinement, the empty path for $\mathcal{S}$ is refined by t', so we can choose t as the empty path.

Inductive case: Let t' be a non-empty (and potentially infinite) path for P. We assume inductively that for all $t'_1 < t'$ there is some path t_1 for $\mathcal{S}$ such that $t_1 \sqsubseteq t'_2$. We must show that there is some path t for $\mathcal{S}$ such that $t \sqsubseteq t'$.

By Lemma 4, we can write $t' = t'_{pre} + t'_{loop}$, such that t'_{pre} has finite length, and each $(z', \delta') \in t'_{loop}$ occurs infinitely often in t'_{loop}. Let t'_{loop} be chosen so that it is the longest such path.

Now there are two cases, depending on whether t'_{loop} is empty.

If t'_{loop} is empty, then $t' = t'_{pre}$ and t' is finite. Since t' is non-empty, we can write $t' = t'_{n-1} + (z'_n, \delta'_n)$. Now there are two subcases.

The first subcase is if t'_{n-1} is empty. Then by the construction of the FSM (Figure 6), we know that the propositions that are assigned a truth value at the start of a path (i.e., δ_n) are top-level calls to web-methods. Suppose this is a call to a web method m. But by assumption the program's m refines the corresponding web method in $\mathcal{S}$, hence there must be a z_n and δ_n such that $\delta_n \Rightarrow \delta'_n$, and so in this case $[(z_n, \delta_n)] \sqsubseteq [(z'_n, \delta'_n)] = t'$.

The second subcase is if t'_{n-1} is non-empty. By definition of $<$ for paths $t'_{n-1} < t'$. So from the inductive hypothesis we get a sequence t_{n-1} such that $t_{n-1} \sqsubseteq t'_{n-1}$. Since t'_{n-1} is finite, it has a last element $(z'_{n-1}, \delta'_{n-1})$ and $t'_{n-1} = t'_{n-2} + (z'_{n-1}, \delta'_{n-1})$. Since $t_{n-1} \sqsubseteq t'_{n-1}$ it must be that t_{n-1} is finite and non-empty. Hence there is some t_{n-2} such that $t_{n-1} = t_{n-2} + (z_{n-1}, \delta_{n-1})$. Since $t_{n-1} \sqsubseteq t'_{n-1}$ it must be that $\delta_{n-1} \Rightarrow \delta'_{n-1}$. Thus by Lemma 5, there is some (z_n, δ_n) such that $t_{n-2} + (z_{n-1}, \delta_{n-1}) + (z_n, \delta_n)$ in is a path for $\mathcal{S}$ and $\delta_n \Rightarrow \delta'_n$. Letting $t = t_{n-1} + (z_n, \delta_n)$, we then have $t \sqsubseteq t'$. This ends the proof of the second subcase, when t'_{loop} is empty.

If t'_{loop} is non-empty, we can write it as $t'_{loop} = (z'_{n+1}, \delta'_{n+1}) + t'_{pp}$. Since t'_{pre} is also non-empty, we can write $t'_{pre} = t'_{n-1} + (z'_n, \delta'_n)$. By the inductive hypothesis we have that there is some t_{pre} such that $t_{pre} \sqsubseteq t'_{pre}$. As above we can write $t_{pre} = t_{n-1} + (z_n, \delta_n)$, and by the refinement relationship, we know that $\delta_n \Rightarrow \delta'_n$. Thus by applying Lemma 5 again, we have that there is some (z_{n+1}, δ_{n+1}) such that $t_{n-1} + (z_n, \delta_n) + (z_{n+1}, \delta_{n+1})$ is a path for $\mathcal{S}$ and $\delta_{n+1} \Rightarrow \delta'_{n+1}$. Thus $t_{n-1} + (z_n, \delta_n) + (z_{n+1}, \delta_{n+1}) \sqsubseteq t'_{n-1} + (z'_n, \delta'_n) + (z'_{n+1}, \delta'_{n+1})$.

Now t'_{loop} must be made up of some repetitions of a prefix t'_2 of t'_{loop} that starts with $(z'_{n+1}, \delta'_{n+1})$. This path t'_2 is also finite, and so we can find a subpath t_2 in $\mathcal{S}$ such that $t_2 \sqsubseteq t'_2$, as above. We can then paste these together to produce a path t in $\mathcal{S}$ such that $t \sqsubseteq t'$. ∎

A.1 Omitted Details on Soundness of Policy Verification

The key idea in the proof of soundness for policy verification is to give a state exploration technique to verify that the policy is satisfied by the state machine constructed by the construction algorithm in Figure 6. Furthermore, we show that the output of our construction algorithm is a valid finite-state program.

Lemma 6. *Given a policy $\phi \in \Phi(\mathcal{S})$ one can build a Büchi automaton $\mathcal{B}(\neg\phi)$ such that the language accepted by that automaton $\mathcal{L}(\mathcal{B}(\neg\phi))$ is exactly the set of computations satisfying the formula $\neg\phi$.*

The proof of this Lemma automatically follows from the proof of Theorem 2.1 and 3.3. given by Vardi and Wolper [26, pp. 4,6].

Given a finite state program $(\mathcal{Z}, s_0, \mathcal{R}, \Delta)$ one can construct an equivalent Büchi automaton $(\sigma, \mathcal{Z}, s_0, \varrho, \Delta)$, where $\sigma = 2^{\mathcal{P}(\mathcal{S})}$, $z' \in \varrho(z, \delta)$ iff $(z, z') \in \mathcal{R}$ and $\delta = \Delta(z)$ [26, pp. 5].

Lemma 7. *Given two Büchi automata $(\sigma, \mathcal{Z}, s_0, \varrho, \Delta)$ and $\mathcal{B}(\neg\phi)$ one can construct an automaton that accepts $\mathcal{L}((\sigma, \mathcal{Z}, s_0, \varrho, \Delta)) \cap \mathcal{L}(\mathcal{B}(\neg\phi))$.*

The proof of this Lemma also automatically follows from Lemma 3.1 of [26], which in turn follows from [43].

From Lemma 6 and 7 it follows that given a finite state program $(\mathcal{Z}, s_0, \mathcal{R}, \Delta)$ and a policy $\phi \in \Phi(\mathcal{S})$, one can construct an automaton that accepts a language, which is empty when the finite state program satisfies the policy. This emptiness property is known to be solvable in linear-time [44].

Lemma 8. *For a specification $\mathcal{S}$, the production relation $\rightsquigarrow$ of Figure 6 constructs a valid finite-state program.*

Proof Sketch: The key intuition behind the proof of this lemma is that from the hypothesis of the rules (IF), (SPEC), in Figure 6 (and other rules in our technical report) one can see that each of these rules generates a finite number of states. Furthermore, each of these rules maintains the structure of the finite-state program. The rule (WEB METHOD CALL) is different as it can potentially allow recursion, and thus generate potentially infinite number of states. However, this is accounted for by the (WEB METHOD CALL FSM 1) and (WEB METHOD CALL FSM 2) rules, which check membership in the table *NT* passed into the rule. The (WEB METHOD CALL FSM 1) rule requires that there is not already a state associated with the web method being called in *NT*, and ensures that subsequent relations use a table (*NT′* in the rule) that has the particular method defined. If there is a definition in the table, the (WEB METHOD CALL FSM 2) rule, is used, which does not add a new state but simply reuses the one in the table. This makes sure that the state for a particular web method call is only added to the FSM once. ∎

Proof of Lemma 2 : Given a *specification* $\mathcal{S}$ and a policy $\phi \in \Phi(\mathcal{S})$, the automaton $\mathcal{F}(\mathcal{S}) \cap \mathcal{B}(\neg\phi)$ accepts a language, which is empty when the specification satisfies the policy.

Proof Sketch: The proof follows from lemma 6, 7, and 8.