Smarmy – A language to verify the security of software architectures

Thomas Heyman
Riccardo Scandariato
Wouter Joosen

Report CW 629, December 2012
Abstract

This document presents Smarmy, a formal modelling language intended to verify the security of software architectures. Smarmy is created with the practising software architect in mind. Its semantics are very close to how a software architecture would be modelled in UML, while its syntax is similar to Java. Smarmy is supported by Smarmelade, which is an integrated development environment built on top of the Eclipse framework. Smarmelade uses model finding to verify Smarmy models, and generate and visualise counterexamples to help the architect in debugging.

The purpose of the Smarmy approach is to systematically uncover the context of an architecture, which consists of the assumptions that that architecture makes on its environment which are necessary to uphold its security requirements. The context represents the minimal but necessary preconditions on the deployment environment of the architecture. It can serve as an explicit target for a risk assessment, or as a checklist to ensure that the system-to-be is deployed securely.
Smarmy

A language to verify the security of software architectures

Thomas Heyman
Riccardo Scandariato
Wouter Joosen
This document presents Smarmy, a formal modelling language intended to verify the security of software architectures. Smarmy is created with the practising software architect in mind. Its semantics are very close to how a software architecture would be modelled in UML, while its syntax is similar to Java. Smarmy is supported by Smarmelade, which is an integrated development environment built on top of the Eclipse framework. Smarmelade uses model finding to verify Smarmy models, and generate and visualise counterexamples to help the architect in debugging.

The purpose of the Smarmy approach is to systematically uncover the context of an architecture, which consists of the assumptions that that architecture makes on its environment which are necessary to uphold its security requirements. The context represents the minimal but necessary preconditions on the deployment environment of the architecture. It can serve as an explicit target for a risk assessment, or as a checklist to ensure that the system-to-be is deployed securely.
Contents

1. Introduction .................................................. 4
   1.1. Background and motivation ................................. 4
   1.2. Purpose of this document ................................ 5

2. Smarmy .......................................................... 6
   2.1. An illustrative example—a tweet client .................. 6
      2.1.1. Creating an initial model ............................ 6
      2.1.2. Verifying the model .................................. 8
   2.2. A more thorough example—the Secure Logger .......... 11
      2.2.1. Creating an initial model ............................ 11
      2.2.2. Verifying the model .................................. 15
   2.3. Language reference ........................................ 16
      2.3.1. The core language ..................................... 16
      2.3.2. The meta model ........................................ 23

3. Smarmelade ..................................................... 25
   3.1. Overview .................................................. 25
   3.2. Configuring and using Smarmelade ......................... 26

4. Conclusion ..................................................... 28

A. The tweeter model ............................................ 29

B. The Secure Logger model ..................................... 31
1. Introduction

Smarmy, supported by Smarmelade, is an approach to systematically uncover assumptions on which the security of a software architecture is founded, and whether that architecture upholds its security requirements given those assumptions. Why this is a good idea is summarized in Section 1.1. How this is handled in this document is presented in Section 1.2.

1.1. Background and motivation

Smarmy, supported by Smarmelade, is created to verify the security of software architectures by uncovering trust assumptions. According to Haley et al. [Haley et al., 2004], these trust assumptions are key to deciding which parts of the environment of the system need further analysis and which do not. Trust assumptions are the decision about how much to trust the supplied indicative (i.e., objectively true) properties of the environment of a software system. These assumptions are common in all security analysis methods—after all, the result of the security analysis of a software system is necessarily relative to the environment of that system. For instance, authentication based on passwords assumes that the passwords of user accounts are kept private. This is a non-trivial assumption—what about when the web browser on a public computer stores your password, or when you are actually giving your password to a malicious site (as in so-called phishing attacks)? None of these situations actually involves the application to which you want to authenticate. In other words, correctly managing a password is an inherent part of the environment of the system, and can never be achieved completely by that system itself. This moves part of the responsibility of authentication to the environment (e.g., the web browser, operating system, and network) of that system.

Documenting trust assumptions is crucial during risk assessment. For instance, the threat that information is disclosed over a connection could be discarded because that communication channel is behind a firewall. If the underlying assumption (e.g., "the communication channel is behind a firewall") is not explicitly documented, then any change to the environment of that system risks voiding the results of the security analysis, thereby rendering the system insecure. Therefore, an explicit enumeration of the set of assumptions in which the analysis is grounded is necessary to back up the analysis result and to build a supporting assurance argument. A security analysis should make these trust assumptions explicit by either specifying what guarantees the environment offers, or what properties the environment needs to uphold.

Smarmelade helps to uncover trust assumptions in Smarmy specifications by starting from as few assumptions on the modelled architecture as possible, and generating counterexamples that hint at important security considerations that the architect overlooked. It is then up to the architect to add explicit assumptions to the Smarmy specification that mitigate those counterexamples. These

---

1Note that trust, as used here, is not to be confused with trustworthiness, which is often defined as the holistic property that encompasses security, safety and reliability [Bernstein, 2005].
explicit assumptions can then be used as an explicit target for risk assessment, and as a driver for a risk based architectural decomposition. Both Smarmy and Smarmelade are follow-up work of the research presented in [Heyman et al., 2010, Heyman et al., 2012].

1.2. Purpose of this document

The purpose of this document is to serve as a tutorial for both the Smarmy language and Smarmelade IDE. Smarmy has been created to easily model software architectures. It does this by having 1) semantics that are close to “standard” software architecture vocabulary (i.e., it talks about components, connectors, interfaces and invocations), and 2) provide a Java-like syntax that is both unambiguous and intuitive to use. The Smarmy language is documented in Section 2.

Smarmelade is an integrated development environment that supports both creating and verifying Smarmy models. It offers all standard features of contemporary IDEs (e.g., editing, syntax highlighting, model navigation and error marking) and some advanced Smarmy-specific features (e.g., requirement verification, counterexample visualisation, and automatic scope minimization). Smarmelade is documented in Section 3.

The original Smarmy source files presented in this document, as well as links to the supporting publications, can be found online at [Heyman, 2012].
2. Smarmy

We introduce Smarmy by means of example, and use it to model a simple fictitious twitter client in Section 2.1. We then move on to a more detailed example by modelling and verifying the Secure Logger security pattern in Section 2.2. A succinct reference to the Smarmy language itself is presented in Section 2.3.

2.1. An illustrative example—a tweet client

As a first introduction, we will model a fictitious “tweeter” service that allows clients to send text messages which are then added to their user account. In order to protect these accounts from non authentic tweets, the tweeter service expects clients to submit their user id and password as well, which is then verified with the data stored in the client account on the server. Given this simple setup, we would like to verify the following: every message in a user’s account was tweeted earlier from a client that provided the correct user id and password.

2.1.1. Creating an initial model

In Smarmy, everything is contained in modules. Modules, in turn, are contained in packages, which provide name space functionality.

Listing 2.1: Introducing the tweeter module.

```plaintext
1 package tutorial;
2
3 module Tweeter {
4 ...  
5 }
```

Modules can contain definitions of interfaces and components. In the case of our tweeter client, its high-level architecture consists of a Client and Server component. The Server exposes an interface TweeterIF which contains one operation, ‘tweet’. This operation expects three arguments, i.e., a message, user id, and matching password. Its type is boolean, as Smarmy is a declarative language, all parts of its specification are either true or false—in this case, ‘tweet’ is true if the tweet succeeds, false otherwise. The signature of the ‘tweet’ operation is therefore boolean tweet(Message msg, UserID id, Password pwd).

This means that we will also need to define some domain specific types that are used in the definition of our Server, i.e., Message, UserID, and Password. All this is combined in the following code snippet.
Listing 2.2: Defining the tweeter client and server.

```java
class Message {}
class UserID {}
class Password {}

interface TweeterIF {
    boolean tweet(Message msg, UserID id, Password pwd);
}

class Client {}

component Server implements TweeterIF {
    boolean tweet(Message msg, UserID id, Password pwd) {
        ...
    }
}
```

We are now ready to fill in the specification of the tweet operation. Intuitively, a tweet succeeds if the Server recognizes that the specified Password matches the UserID. In that case, the Message is added to that UserID’s list of tweets.

But how do we know whether the id matches the pwd? And what does it mean for the msg to be added? There is a missing ingredient in our model so far—we need the concept of a user account.

Listing 2.3: Specifying the tweet operation with the help of a user account type.

```java
class UserAccount {
    one UserID id;
    one Password password;
    set Message messages;
}

component Server implements TweeterIF {
    set UserAccount accounts;
    boolean tweet(Message msg, UserID id, Password pwd) {
        let UserAccount acct {
            if (acct.id = id and acct.password = pwd and acct in this.accounts) then msg in acct.messages
        }
    }
}
```

A UserAccount is another domain specific type. However, unlike the previous types, it is not atomic and has attributes. An attribute consists of a cardinality, i.e., one (0..1), one (1..1), set (0..*) or some (1..*), a type, and a name. Attributes can be accessed with ’.’ notation, just as in Java. As in Java, accessor and mutator operations can be written to obtain and change the values of an attribute.

The Server stores a list of UserAccounts by means of a component attribute. The tweet operation can then be specified as follows: “If the Server has a UserAccount with matching UserID and Password in its internal list of accounts, then the Message is added to that UserAccount. How this is translated to Smarmy is shown on line 32. Note the surrounding let block, which is the Smarmy way of introducing
local variables. A `let` contains one or more type-identifier combinations, separated by a ‘,’; that can be used in the code block following the `let`.

Does this model behave as intended? Don’t take my word for it, let’s generate some example instances. By specifying `illustration` commands, we can instruct Smarmelade to visualize instances of the tweeter model that uphold some property. For instance, let’s specify that we want to see a tweeter instance that successfully places a Message. This is done as follows.

Listing 2.4: Specifying an example model illustration.

```plaintext
illustration PlaceATweet {
  show {
    sometimes {
      some { Server srv; Message msg; UserID id; Password pwd | srv.tweet(msg,id,pwd) }
    }
  }
}
```

An `illustration` has a name and a body. The body contains a `show` block, which in turn contains the property we want to observe. In this case, `show` contains two new expressions that we have not encountered yet. The inner expression, `some`, expresses a cardinality on a set—the set being all instances of Servers, Messages, UserID and Passwords so that `srv.tweet(msg,id,pwd)` holds, and the cardinality `some` expressing that there needs to be at least one element in that set. Other cardinalities, as with attribute definitions, are `lone`, `one`, `set`, and `no`, signifying an empty set. The outer expression, `sometimes`, is a so-called temporal expression. It expresses temporal constraints on the code inside the block that follows it. All operations take place at a specific time. We need to specify when we want to see this tweet invocation take place. In this case, we just want to see `tweeat` happen at an arbitrary moment, i.e., `sometimes`. Other temporal specifications would be `never` and `always`; we will see more examples later.

Smarmelade can now generate and visualize instances of `PlaceATweet`. The IDE does this by leveraging the Alloy Analyzer under the hood. To generate an instance, right-click on the `PlaceATweet` illustration in the outline view, and select ‘Run this command’. The progression of the command is shown in the progress view. After the command completes, clicking the ‘OK’ link opens a visualisation of the generated instance.

### 2.1.2. Verifying the model

Now that we have our initial architectural model, we can begin the verification process. The first question is, of course, what are we to verify? The answer depends on our model. As with the `illustration`, we have to specify one or more security requirements that our architecture should uphold, in the form of goals.

Listing 2.5: Specifying the security requirement of the model.

```plaintext
goal OnlyCorrectClientsCanTweet {
  check {
    always {
      forall { Message m; UserAccount acct; Server srv |
        if (acct in srv.accounts and m in acct.messages) then earlier {
```
some { Client c | invoked c, srv.tweet(m,acct.userid,acct.password) }
}

Note that goals normally are elicited in a requirements engineering process. This goal, OnlyCorrectClientsCanTweet, specifies the following. It should always be true that for all Messages, UserAccounts and Servers, if the Server contains a certain UserAccount which in turn contains that Message, then there was a Client that earlier invoked the tweet operation on that Server with that Message and the credentials of the UserAccount as its arguments.

By right-clicking on the OnlyCorrectClientsCanTweet goal in the outline view and selecting 'Run this command', a counterexample is produced that shows how our seemingly simple security requirement is violated. The counterexample shows a situation where a Server already contains Messages from the start. In other words, we will have to ensure that all Servers are initialized properly—that is, with no Messages in their accounts. This property is encoded in the ServersCorrectlyInitialized() predicate. By adding it to the OnlyCorrectClientsCanTweet goal as an assumption, we instruct Smarmelade to only take situations into account where ServersCorrectlyInitialized() holds.

Listing 2.6: Adding an assumption to the model.

boolean ServersCorrectlyInitialized() {
    initially {
        forall { Server srv; UserAccount acct | if (acct in srv.accounts) then no {acct.messages} }
    }
}
goal OnlyCorrectClientsCanTweet {
    assuming
    ServersCorrectlyInitialized();
    check { ... }
}

While adding this assumption takes care of the previous counterexample, we are now presented with a different situation. Here, an arbitrary component that is not a Client invokes the tweet operation on a Server. Having ‘unofficial’ Clients making use of the tweeting functionality violates the OnlyCorrectClientsCanTweet requirement, and we need to introduce another assumption to mitigate this case. The result is this.

Listing 2.7: Adding a second assumption to the model.

boolean OnlyClientsTweet() {
    always {
        forall { Component c; Server s; Message m; UserID id; Password pwd |
            if (invoked c, s.tweet(m.id,pwd)) then c in {Client} }
    }
}
goal OnlyCorrectClientsCanTweet {
    assuming
More counterexamples appear. The next one shows a situation where new Messages are added to a UserAccount illegitimately, i.e., without there being a corresponding invocation of the tweet operation. This is solved by assuming that the integrity of the message contents of all UserAccounts is protected—every Message in its contents is the result of another component executing the tweet operation with the correct UserID and Password at an earlier time. The next counterexample hints at an invocation of the tweet operation being spoofed—it seemingly originated from a Client, but was sent from another component. This is resolved by assuming that all connectors authenticate their callers. This process continues until no more counterexamples are found.

At this point it is important to revisit the uncovered assumptions and verify whether they are all still required. Often, by adding new assumptions, an old assumption will become superfluous. This can be easily checked by right-clicking on the goal in the outline view, and selecting ‘Minimize context’. In this case, we see that our first assumption, ServersCorrectlyInitialized, is no longer required, and can be removed. The final context is the following.

Listing 2.8: Final list of assumptions for the tweeter architecture.

```java
    boolean OnlyClientsTweet() {
        forall { Component c; Server s; Message m; UserID id; Password pwd |
            if (invoked c, s.tweet(m, id, pwd)) then c in [Client] }
    }

    boolean IntegrityOfUserAccounts() {
        forall { UserAccount acct; Message m |
            if (m in acct.messages) then (some { Component c; Server srv |
                acct in srv.accounts and earlier { executed c, srv.tweet(m, acct.userid, acct.password) } })
        }
    }

    boolean ConnectorsAuthenticated() {
        forall { Connector c | c.callersAuthenticated() }
    }

    boolean IntegrityOfUserAccountDetails() {
        forall { UserAccount acct; Password pwd; UserID id |
            if (initially { acct.password = pwd and acct.userid = id}) then
                (always { acct.password = pwd and acct.userid = id})
        }
    }

    boolean NoSharedAccounts() {
        forall { Server s1; Server s2; UserAccount acct |
            if (acct in s1.accounts and acct in s2.accounts) then s1 = s2 }
    }
```
2.2. A more thorough example—the Secure Logger

In this second example, we delve a little deeper in some of the previously introduced concepts. As an example, we create a detailed model of the Secure Logger security pattern as documented in [Steel et al., 2005]. The Secure Logger serves to log data securely in a tamper proof fashion—that is, making both modification and deletion of logged messages detectable. So let’s model the pattern, and see what assumptions we can uncover.

2.2.1. Creating an initial model

First, let’s revisit the module concept introduced earlier. Modules in Smarmy not only serve to group related modelling concepts, they can also accept type parameters to facilitate reusing modules in other projects (similar to Java generics). As an example, consider the following definition of the SecureLogger module. As we do not know (and do not really care) what it is that our logger will store, we can abstract it with a type parameter, Message. These type parameters are treated as atomic types.

```
Listing 2.9: Introducing the SecureLogger module in Smarmy.

package patterns;


class Principal[] // Definition of a new atomic type.
...

// Definition of a new complex type, SignedMessage, which is an abstraction
// of a ProcessedMessage which has been cryptographically signed.
class SignedMessage {
    // A field declaration contains a cardinality (lone, one,some or set),
    // the type of the field, and a name.
    one ProcessedMessage content;
    one ProcessedMessage signedContent;
    one Principal signedBy;
...
}

} // End of the module declaration.
```

If this module is subsequently imported and reused in another project, a value for Message will have to be specified as follows.

```
Listing 2.10: Importing the SecureLogger module in another Smarmy file.

import patterns.SecureLogger<MyMessage> as LOG;
...
class MyMessage {}

component LoggerClient {
    lone LOG.Logger myLog;
    boolean doSomethingUsefulWith(MyMessage msg) {
```
The specification of the doSomethingUsefulWith operation shows how one component can call an operation of another component. Smarmy distinguishes between calling an operation locally and remotely. Components can only communicate by means of invoking operations defined in one of the interfaces implemented by another component. Simply calling myLog.log(msg) without the invoke keyword would try to call an operation locally on a different component, and would result in an error. Similarly, calling an operation that is not declared in one of the interfaces of another component results in an error. The invoke keyword serves to remind the modeller that the analysis will introduce Invocations to realize the communication between the calling and receiving components. Similarly, the invoked and executed keywords can be used to assert whether a specific invocation was indeed invoked, resp. executed. We will see an example of this later on line 231.

Back to our model of the Secure Logger. In order to specify what we mean by ’signing’ a SignedMessage, we need to introduce a named instance of the Principal type, i.e., the LoggerEntity. We can then say that a SignedMessage is signed properly if it has the LoggerEntity as its signedBy principal. Named instances can be introduced by the object keyword, as follows.

Listing 2.11: Using objects to specify the sign operation.

```java
// Definition of two specific named instances of the Principal type.
object AdminEntity in Principal;
object LoggerEntity in Principal;
...

// Definition of a new complex type, SignedMessage, which is an abstraction
// of a ProcessedMessage which has been cryptographically signed.
class SignedMessage {
    ...
    // This defines an operation on the SignedMessage type.
    boolean sign() {
        // The body of operations always consists of one expression.
        this.signedContent = this.content and this.signedBy = LoggerEntity
    }
}
```

Based on the original documentation, we can construct the following model. More details on how we go from informal documentation to formal model are presented in [Heyman et al., 2012].

Listing 2.12: Definition of the Logger component in Smarmy.

```java
// Definition of the Log interface.
interface LogIF {
    void log(Message m);
}

// Definition of the Logger component. Note that the Logger implements the Log
// interface.
component Logger implements LogIF {
```
The Logger contains two attributes: a set of SignedMessages and the next sequence number to assign, as one integer value.

```java
set SignedMessage contains;
one int nextUID;

// This is the implementation of the log operation.
boolean log(Message m) {
    some { ProcessedMessage pm; SignedMessage s |
        pm.content = m and
        0 <= pm.id and pm.id < this.nextUID and
        s.content = pm and (s in this.contains) and
        s.sign(LoggerEntity)
    }
    }
}
```

Up till now, we have only encountered boolean operations. As Smarmy is a declarative modelling language, the majority of operations will be boolean in nature—i.e., they describe properties on the behavior of a component or the model in general that are either true or false. However, operations can also return explicit return values so that the calling operations can process them. Consider for instance the Smarmy version of the verify operation, which returns ‘1’ if the Logger is intact, and ‘0’ if it is not. This operation is subsequently used by the read operation.

Listing 2.13: An example of returning non boolean values and subsequently using the return value in other operations in Smarmy.

```java

component Logger implements LogIF {
    ...
    int read(Message m) {
        // The implementation of the read operation.
        int v = verify() {
            if (v = 1 and some { SignedMessage sm | (sm in this.contains) and
            sm.content.content = m and sm.content.id >= 0 and sm.content.id < nextUID })
            then returns 1 else returns 0
        }
        }
        ...
    }
    ...
    int verify() {
        if allEntriesAccountedFor() and entriesAreValid() then returns 1 else returns 0
    }
    ...
}
```

As shown on line 183, values can be returned by means of the returns keyword. However, there is an important difference between returns in Smarmy, and return statements from imperative languages such as Java: as Smarmy is a declarative modelling language, the operation eventually returns the specified value. However, the evaluation of an operation does not stop when the first return statement is encountered, as is the case in imperative programming languages—the entire expression that
comprises the operation body is evaluated at the same time.

Back to the Secure Logger. As Smarmy is declarative in nature, it allows some powerful expressions. Consider for instance the following example, which highlights the possibility to iterate over all members of a type, and set comprehensions, in which all members of a type that have specific properties are grouped.

Listing 2.14: An example property definition that leverages declarative features of Smarmy.

```java
component Logger implements LogIF {
...

// Definition of a property on the Logger.

boolean allEntriesAccountedFor() {
// The forall statement allows iteration over a set of values. Note that a
// type implicitly defines a set of all its members.
forall { int i |
if (i >= 0 and i < this.nextUID) then
// Note that set comprehensions are supported. The following expression
// considers the set of SignedMessages in the contains set of this
// Logger and with the correct id. This set is then evaluated to contain
// at least one element with the some keyword.
some { SignedMessage s | (s in this.contains) and (s.content.id = i) }
}
}
...

}
```

The above definition of allEntriesAccountedFor is an example of a local operation of the Logger. It is not defined in the LogIF, and can never be called from another component—it only serves to ease the specification of the Logger internals.

Apart from defining types, interfaces and components, models can also introduce stand-alone operations (i.e., predicates). The following example predicate shows a global property that, given a Logger as argument, determines whether the nextUID value of that Logger is correct.

Listing 2.15: Temporal blocks in Smarmy.

```java
boolean CalculateNextUID(Logger l) {
initially {
 l.nextUID = #{Message m | some { Component c | executed c, l.log(m) }} and
 from now on {
 l.nextUID = previous { l.nextUID } +
 #[Message m | some { Component c | executed c, l.log(m) }]
 }
}
```

Up till now, we have occasionally encountered the concept of time. Time, in Smarmy, is implicit—every operation implicitly defines a time value for ‘now’. The modeller can express temporal constraints by means of temporal blocks. The first four, initially (at the first time instance), always (at all time instances), never (at no time instance) and sometimes (at some time instances), are global time blocks. They can be used as a regular boolean expression in any place where a normal boolean expression
would be valid. The last six, earlier (at some time instance up to and including now), from now on (at all time instances after and including now), later (at some time instance after and including now), so far (at all time instances up to and including now), previous (at the time instance directly before now) and next (at the time instance directly after now), are nested time blocks. They can only be used within another temporal scope, i.e., in operation bodies or in any of the global time blocks.

Consider, for instance, the initially block on line 211. The expression nested in that block implicitly constrains the state of the model at the first time instance. The nested from now on block on line 213 constrains all time instances after the implicit time instance, in other words, all times after the initial time instance. Within that block, the current value of nextUID is compared with the previous value by means of the previous temporal block.

This concludes our initial model of the Secure Logger. A complete version is included in Appendix B. We now turn to its verification.

### 2.2.2. Verifying the model

We have already encountered both model instance generation via the illustration command, and model verification via the goal command. For instance, the Test illustration shows model instances where there is at least one Logger instance, as modelled on line 220.

As Smarmelade leverages model finding for verifying Smarmy models, it can not verify infinitely large model instances (e.g., infinitely many Loggers containing an infinite amount of Messages). Instead, the verification is only guaranteed to be complete up to a specific scope. The scope of the analysis is the maximum amount of instances of every type that will be used during the verification. For instance, a scope of 3 but 2 Messages would consider all situations with 3 Components, 3 Connectors, 3 Invocations, ..., and 2 Messages. The default scope is 3. However, this can be explicitly set by the modeller. For instance, the scope of the Test illustration is three, with maximum two Messages and one time instance.

Listing 2.16: Expressing commands in Smarmy.

219 illustration Test {
220   show { some Logger }
221   scope 3 but 2 Message, 1 Time
222 }
223
224 goal NothingDeleted {
225   assuming
226     forall { Logger l | CalculateNextUID(l) },
227     ...
228     SecureConnectors();
229   check {
230     always {
231       forall { Component c; Logger l; Message m | if invoked c, l.log(m) then (  
232         later {
233           from now on |
234             let int v = l.verify(), int r = l.read(m) { if (v = 1) then (r = 1) }  
235           )}
236       )
237     }
The higher the scope, the more exhaustive the analysis (but the higher the verification time, and the larger the counterexamples). In order to find the smallest scope in which counterexamples can be found, which significantly helps the debugging process, the modeller can instruct Smarmelade to minimize the scope by right-clicking on a goal and selecting 'Minimize scope'. After completion, clicking the 'OK' link in the progress view will show the minimum scope for all explicitly defined types.

As with the tweeter example, the analysis uncovers many counterexamples that are resolved by adding explicit assumptions. An example assumption is the following.

Listing 2.17: Integration of the meta model in Smarmy.

```java
boolean SecureConnectors() {
    for (Connector c : c.isSecure())
        return true;
}
```

Line 243 refers to the Connector type. This is part of the built-in architectural meta model supported by Smarmy. We will take a closer look at those types in Section 2.3. Here, the example assumption that all Connectors are secure is modelled on line 243 as an iteration over all instances of the Connector class. A complete list of the assumptions uncovered in the Secure Logger model is included in Appendix B.

### 2.3. Language reference

The fundamentals of the Smarmy language are summarized in Section 2.3.1. Smarmy contains a built-in architectural meta model. This is introduced in Section 2.3.2.

#### 2.3.1. The core language

**Types and named instances**

A type consists of a name, preceded by the keyword `class`, and a body, surrounded by curly brackets. A type body consists of an arbitrary amount of attribute definitions, and an arbitrary amount of operations. A type can also inherit from other types with the extends keyword. By extending another type, the type inherits all of the attributes and operations of the type it extends. Finally, named instances of types can be created by means of the `object` keyword. That named instance can be referenced as a global variable.

The following example introduces a Child type that extends Person. It has two attributes, an age which is exactly one integer, and a family which is a set (i.e., \(\ast\)) of Persons. A Child has one operation, `hasAsFamily(Person p)`, which is true when the specified Person is in the family attribute of that Child.
class Child extends Person {
    one int age;
    set Person family;

    boolean hasAsFamily(Person p) {
        p in this.family
    }

    object susana in Child;

    Interfaces

    Interfaces are specific types that have a name, preceded by the interface keyword, and a body surrounded by curly brackets. Their bodies can not contain attributes, however, and are limited to containing operation headers (as in Java). Interfaces are used to group remotely accessible component operations.

    The following example introduces a ShoppingCart interface that introduces two operations, addToCart(Item i) and checkout().

    interface ShoppingCart {
        boolean addToCart(Item i);
        boolean checkout();
    }

    Components

    Components are specific types that have a name, preceded by the component keyword, and a body surrounded by curly brackets. Their bodies are identical to class bodies, with the difference that component attributes can only have cardinalities of lone (0..1) or set (0..*), never one (1..1) or some (1..*).

    Components can implement an arbitrary amount of interfaces by means of the implements keyword, followed by a comma separated list of interface names. A component has to explicitly override all operations in the interfaces it implements. Components do not support extending other types.

    The following example introduces a ShoppingCart component that implements the ShoppingCart.

    component ShoppingCart implements ShoppingCart {
        lone ShoppingCart cart;

        boolean addToCart(Item i) {
            i in this.cart.contents
        }

        boolean checkout() {
            this.cart.finalize()
        }
Operations

Behavior of Smarmy models is encoded in operations, which consist of a header and a body. An operation is defined either within a type or component, or outside of any type, i.e., stand alone. Every operation header has a return type (which is either an explicitly defined type, or is boolean or void), followed by the operation name, and a comma separated list of arguments between round brackets. Its body is surrounded by curly brackets, and always contains exactly one expression. The type of the expression must match the return type of the operation.

An operation returns values with the returns expression. A returns expression has as its type the type it returns. The expression of an operation can contain an arbitrary amount of returns expressions. However, evaluating an operation expression does not stop when a returns is encountered.

The following example introduces a stand alone operation that accepts two integers as arguments and returns the larger one.

```cpp
268 } 269 }

int largestOf(int a, int b) {
  if (a > b) then (returns a) else (returns b)
}
```

Boolean, void and nothing

As Smarmy is a declarative language, all operations are effectively boolean in nature—either their expression is realized and the appropriate value returned, or not. Therefore, the default return value of operations that do not return values explicitly by means of the returns expression, is boolean. The only exception is nothing, which is a void expression. Neither boolean nor void operations can return values by means of a returns expression.

Boolean expressions

Smarmy contains the basic boolean expressions and, or, not. Both and and or are binary infix operators that expect one boolean expression on their left and another on their right. The negation not is a unary operator, written before a boolean expression.

Implications are expressed by means of an if then else expression. It expects a boolean expression between the if and then keywords. The expression after then will be evaluated in the case that the condition is true. If it is not, and the optional else clause is present, the expression following it will be evaluated. Note that the types of both then and else expressions must match. Equivalences can similarly be expressed by means of the iff then expression. Round brackets can be used to surround expressions for operator precedence reasons.

The following example expresses that susana is family of robert if and only if the converse is true.

```cpp
270 int largestOf(int a, int b) {
  if (a > b) then (returns a) else (returns b)
}
```

```
if susana.hasAsFamily(robert) then robert.hasAsFamily(susana)
```
Set comprehensions

Smarmy is built around sets. Every type, including Components and Interfaces, implicitly defines a set of instances of that type. In order to refer to the set of all instances of a type T, that type is surrounded with curly braces, as in [T]. Attributes of types define relationships (which are also sets) between types. The ‘element of’ relation is denoted by the keyword in. As an example, if susana is a Child, the following is true.

```
susana in [Child]
```

Finally, sets can be explicitly constructed by means of set comprehensions. A set comprehension consists of one or more variable definitions, followed by constraints on those variables. A set comprehension is the set consisting of all instances of those types that uphold the constraints. A set comprehension is surrounded by curly braces. Then, the local variables are defined, separated by semicolons. A pipe character separates the variable definitions from the constraints.

The following example defines the set of all Persons that are related to susana.

```
{ Person p | susana.hasAsFamily(p) }
```

Cardinality expressions

Cardinality expressions are boolean expressions that constrain the amount of elements in a set. All sets can be constrained with the keywords lone, one, some, set followed by a set, which constrain the size of that set to (0..1), (1..1), (1..*), (0..*), respectively. Alternatively, the cardinality of a set can be obtained with the integer expression #.

For example, the following expression states that the amount of Children is exactly three, while at least one of them is family of susana.

```
{ #[Child] = 3 } and some { Child c | susana.hasAsFamily(c) }
```

Iterating with forall

Similar to set comprehensions, elements of a set can be iterated by means of a forall expression. This is a boolean expression that consists of one or more variable definitions, and an expression that constrains those variables.

For example, the following expression combines a forall with a cardinality expression to state that all Items should be added to at least one ShoppingHandler.

```
forall { Item i | some { ShoppingHandler sh | sh.addToCart(i) } }
```

Temporal expressions

Temporal expressions are boolean expressions that modify the implicit time for the code block that follows. The temporal expressions initially (at the first time instance), always (at all time instances),
never (at no time instance) and sometimes (at some time instances), are global time blocks that can be used as regular boolean expressions.

The temporal expressions earlier (at some time instance up to and including now), from now on (at all time instances after and including now), later (at some time instance after and including now), so far (at all time instances up to and including now), previous (at the time instance directly before now) and next (at the time instance directly after now), are nested time blocks. They are also regular boolean expressions, but can only be used within another temporal scope.

Both operations and assuming clauses of commands implicitly define a temporal scope—operations define time as the moment at which they are executed, and assumptions define time as always.

The following example states that when it is executed, an item is added to the cart of a ShoppingHandler. At the next time instance, that ShoppingHandler is checked out.

```java
boolean buy(ShoppingHandler sh, Item i) {
    sh.addToCart(i) and next { sh.checkout() }
}
```

### Declaring local variables with let

Local variables can be introduced with let expressions. These expressions consists of the keyword let, followed by one or more variable declarations, and a code block. The declared variables are defined in the scope of that block.

Variable declarations in a let can also include an initializer by adding an '=' followed by an expression of the correct type. The local variable will then be initialized to that value.

The following example declares two local integer variables and initializes them. It contains one boolean expression which is true.

```java
let int a = 1; int b = 2 { b > a }
```

### Calling local operations

Local operations of which no explicit return value is expected, can be called with '.' notation, as in Java. However, when you want to make use of the return value of an operation, that operation has to be called from within the initializer of a let statement.

The following example assumes that a ShoppingHandler has a local operation getTotalPrice(), which returns the price of all items in the shopping cart as an integer. It defines a payAmount operation that, given a ShoppingHandler and integer amount, determines whether that amount is a correct amount for payment.

```java
component ShoppingHandler implements ShopIF {
    ...
    boolean payAmount(int amount)
    let int total = this.getTotalPrice() {
        total = amount
    }
```
Calling remote operations

Remote operations, i.e., operations that are defined in one of the interfaces of a component and are called from another component, have to be invoked by means of the `invoke` keyword. This keyword signifies that explicit communication will take place between these components by means of an Invocation.

Obtaining the result of invoking a remote operation is similar to the local case—the `invoke` expression needs to be used as initializer of a local variable in a `let`. Remote operations can also be called asynchronously without a `let`. In that case, the return value will not be available.

The following illustrates how both the getTotalPrice and payAmount operations are called remotely from another component. The first call obtains a return value, while the second call happens asynchronously.

```java
component Client {
  lume ShoppingHandler session;
  boolean finish() {
    let int amount = invoke session.getTotalPrice() {
      invoke session.payAmount(amount)
    }
  }
}
```

However, remote operations can be called directly (i.e., without making use of `invoke`) from without components. That is, commands and global operations can call remote operations as if they were local. The reason for this is that these model elements can be used to specify intended behaviour—that is, properties that the model should uphold, as opposed to specification that states how the architecture achieves this.

Two other expressions are available from without components to facilitate expressing model properties: `invoked` and `executed`. The first, `invoked`, is a boolean expression with a syntax similar to `invoke`. However, it accepts two arguments, separated by a comma. The first argument denotes a calling component, and the second argument an operation call. It is true if the specified component indeed invoked that operation. The second, `executed`, is similar to `invoked` with the difference that it is true if the expression that follows it is actually executed, i.e., if the postconditions of the invoked operation become true.

The following example expresses that whenever the finish operation of a Client is true, payAmount should eventually be executed by that Client.

```java
forall | Client c; ShoppingHandler shop |
  some | int amount |
  if c.finish() then eventually | executed c, shop.payAmount(amount) |
```
Commands

A command is either an illustration or a goal. Both have a name and a body. Both bodies optionally accept a comma separated list of assumptions, terminated by a semicolon. These assumptions are boolean expressions that are assumed to hold as preconditions for the expression to be executed. Following the optional assumptions, the illustration expects a show block which contains an expression that should be evaluated, and the goal expects a check block which contains an expression that should be verified. Both optionally contain a scope statement, which lists an integer denoting the global scope of the command, followed by a comma separated list of scopes for specific types.

The following example expresses a goal to be verified, i.e., that all ShoppingHandlers are eventually paid, for a global scope of 3 but only two Clients. Additionally, the goal assumes that no two Clients share the same ShoppingHandler.

```plaintext
goal AllShoppingHandlersArePaid {
  assuming
    no { Client c1; Client c2 | c1.session = c2.session };
  check {
    forall { ShoppingHandler shop |
      some { Client c; int amount | eventually { executed c, shop.payAmount(amount) } }
  }
}
```

Packages, modules and import statements

Every Smarmy file has to start with a package declaration, which states the name space of the concepts defined in that file, followed by a semicolon. Every Smarmy file is also expected to contain a module, which serves as a unit for importation in other models. An import statement contains the fully qualified name of the module to be imported, as well as a local alias to refer to that imported module.

The following example illustrates how a type can be defined in the module MyModule, with namespace my.namespace.

```plaintext
package my.namespace;

module MyModule {

  class Dog {
    boolean bark() {
      ...
    }
  }
}
```
MyModule can now be imported and used from another file as follows.

```java
import my.namespace.MyModule as DOG;

class Kennel {
    set DOG/Dog dogs;
    boolean makeNoise() {
        forall { DOG/Dog dog | if (dog in this.dogs) then dog.bark() }
    }
}
```

### 2.3.2. The meta model

Embedded in Smarmy is the architectural meta model shown in Figure 2.1. All types from the meta model and their interrelations are directly accessible in Smarmy as types, resp. attributes of those types.

![UML Diagram](image)

Figure 2.1: Our architectural meta-model (using UML syntax).

In addition, some types contain some extra built-in operations to make modelling easier. These are summarized below.

**Component.** The Component type has the following operations built-in. The operation boolean connectedTo(Component other) denotes whether the component is connected to other via one or more connectors. The operation boolean learns(Object data) denotes whether the component ‘learns’ a specific piece of data. Learning is defined as observing that data being exchanged as argument to an Invocation. The operation boolean canIntercept(Invocation inv) denotes whether the component can intercept the specified inv, that is, whether the component is connected to a connector over which inv is exchanged.

**Connector.** The Connector type has the following operations built-in. The operation boolean isReliable() denotes whether this connector keeps the Components it connects always connected; in other words, connected components are never disconnected. The operation boolean callersAuthenticated() denotes whether there are any Invocations buffered by this Connector that have a spoofed caller, i.e., where the orig_caller is different from the actual caller. The operation boolean receiversAuthenticated()
denotes whether this Connector only connects the intended receivers and the caller of the Invocations it buffers. The operation boolean tamperProof() denotes whether any of the Invocations buffered by this Connector have been tampered with. An Invocation is considered modified when there exists an equivalent Invocation, i.e., with the same caller, receivers, of the same Operation, and invoked at the same time. The operation boolean isLocal() denotes whether this Connector connects Components that are hosted on the same Node. The operation boolean hasFiniteCapacity(int cap) denotes whether the amount of Invocations buffered by that Connector are at most cap. The operation boolean atMostOnceInvocationSemantics() denotes whether the Invocations buffered by this Connector are executed at most once. The operation boolean atLeastOnceInvocationSemantics() denotes whether the Invocations buffered by this Connector are executed at least once. The operation boolean deleteProof() denotes whether any Invocations buffered by this Connector are removed from it without being executed.

**Invocation.** The Invocation type has the following operations built-in. The operation boolean live() denotes whether that Invocation is actually executed at some time. The operation boolean instantaneous() denotes whether the Invocation is executed at the same time that it was invoked; in other words, that there is no delay between invocation and execution. The operation boolean hasAuthenticCaller() denotes whether the caller and authentic_caller of that Invocation match. The operation boolean executedBy(Component cmp) denotes whether the Invocation is actually executed by cmp.
3. Smarmelade

3.1. Overview

Smarmelade was created to support creating and verifying Smarmy models, and can be obtained at [Heyman, 2012]. It is built as a plugin to the Eclipse framework\(^1\), and supports the normal Eclipse features—editor support, syntax highlighting, error detection, an outline view, the incoming and outgoing call graph, and so on. What this looks like in practise is shown in Figure 3.1. Smarmy models are automatically translated to Alloy models [Jackson, 2006], and verified by the Alloy Analyzer\(^2\) in the backend. Verification results are then fed back to Smarmelade, either as a notification that the verified property holds, or as a visualized counterexample.

Internally, Smarmelade is built on top of the Chameleon framework [van Dooren et al., 2012], which is a generic object oriented framework for language development. Every Smarmy concept is encoded as a class which contains references to other Smarmy concepts. These classes also contain methods to verify their own integrity. Based on these Chameleon features, apart from the semantic advantages of Smarmy (i.e., that the modelling language is focussed on specifying architectural models), Smarmelade offers additional advantages by enforcing the following rules.

**Relaxation rules.** One crucial thing to consider when creating models for security verification, is that they do not contain hidden assumptions that are not realistic. These hidden assumptions can be largely mitigated by ensuring that the models uphold certain relaxation rules [Heyman et al., 2010]. Smarmelade helps in enforcing this. By construction, all relationships in Smarmy are time dependent. Additionally, constraints are placed on component attributes so that hidden assumptions on the existence of other components are avoided.

**Separating implementation from specification.** In general, two types of predicates exist. The first type consists of implementation related predicates, such as component operations that can be called from other components. The second type consists of specification related predicates, or ‘oracles’, which are useful to express assumptions and goals, such as the executed predicate which verifies whether an invocation is actually executed within a specified component. Obviously, the implementation of another component can not depend on these oracle predicates, as there is no realistic way to implement them in code. Smarmelade enforces this separation.

**Properly nesting temporal blocks.** Some temporal expressions only make sense within a well-defined temporal scope. Smarmelade automatically enforces that temporal expressions are well formed.

Additionally, Smarmelade provides the following features.

\(^{1}\)http://eclipse.org/
\(^{2}\)http://alloy.mit.edu/alloy/
3.2 Configuring and Using Smarmelade

Context minimization. Smarmelade can detect when assumptions are made superfluous, helping to ensure that the context contains the minimal set of assumptions on which the model depends.

Scope minimization. Smarmelade can automatically compute the minimal scope in which a counterexample can be found. This helps in allowing the modeler to look at the smallest counterexample first, which, in turn, provides hints as to what is the root cause of that uncovered issue. For instance, if the minimal scope for time is two, odds are high that some mutable relation is to blame for the counterexample.

3.2 Configuring and using Smarmelade

All Smarmy models are contained in projects. A project is created by selecting ‘File’ → ‘New’ → ‘Project...’ from the menu. Choose ‘New Chameleon Project’ from the pop-up menu, choose ‘Smarmy’ from the language selection menu, and finally choose both a name and location for your new project, and optionally add additional source folders. By default, a new project contains a ‘src’ folder in which the Smarmy source files are to be placed, and an ‘alloy’ folder which contains output files for the Alloy Analyzer. This folder also contains the architectural meta model in the ‘alloy/util/metamodel.ais’ file.

Smarmy source files can be added by right-clicking on the ‘src’ folder, and adding subfolders, resp. new files. Note that 1) all Smarmy source files have to end with the ‘.sma’ file extension, and 2)
the path relative to the ‘src’ folder should match the **package** declaration in the source file. As new files are empty, an error will be generated until the modeller adds a suitable **package** and **module** declaration to the file.

A typical Smarmelade modelling session looks like Figure 3.1. There are a few things to note here. For optimal modelling experience, the following views should be enabled (with the ‘Window’ → ‘Show view’ → ‘Other...’).

**Outline.** This view shows an outline of the entire document, which significantly facilitates navigation. By clicking on a node, the editor jumps to the corresponding definition. This view also enables executing commands by right clicking on either illustrations or goals, and selecting ‘Run this command’, ‘Minimize scope’, or ‘Minimize context’. The output of executed commands is shown in the Progress view.

**Progress.** This view shows the progression of commands that are executed, or have been executed. The result of executed commands can be observed by clicking on the resulting ‘OK’ link.

**Call Hierarchy.** Given a model element, this view enables visualising both incoming calls (i.e., parts of the model that depend on that model element) and outgoing calls (i.e., parts of the model on which the selected model element depends). This view is activated by selecting a model element (either in the editor or in the outline view), and clicking either the green incoming tree or outgoing tree icon on the top right of the call hierarchy view.

**Problems.** This view shows modelling errors and warnings.

**Chameleon Model.** This view is able to show the entire document in a tree like structure (corresponding to the internal parse tree), by clicking on ‘Show Document Model’ and expanding the nodes as required. By clicking on a node, the editor jumps to the corresponding definition. This tree corresponds to the internal parse tree of the document, and can be used for debugging purposes.

The Smarmelade editor supports control-click navigation. By holding down the control button (command on a Mac) and hovering the mouse over the specification of the model, underlying modelling text changes either to green (i.e., the model element was recognized as defined in the model), gray (i.e., the model element was recognized as built into the tool), or red (i.e., the model element is unknown). Clicking on a green link jumps to the definition of the underlying concept. This is a convenient way to verify that model imports, etc., are resolved correctly.

A final note on the build process, is that a full build is only triggered upon saving the source file. In order to avoid verification problems, it is recommended to save the source file before attempting to run a command—failing to do so will most likely not result in verifying the intended code. The same goes for newly opened documents: they should be saved once before triggering any commands.
4. Conclusion

Sarmy is a declarative modelling language, tailored for specifying and verifying models of software architectures. It combines the semantics of a security-centric architectural meta model with a syntax that tries to be Java-like, while still retaining expressive declarative features. Smarmelade is a full-featured integrated development environment for Sarmy. It is built on top of the Eclipse framework and translates Sarmy code to Alloy, which is subsequently verified by the Alloy Analyzer.
A. The tweeter model

```java
package tutorial;

module tweeter {

class Message {}
class UserID {}
class Password {}

interface TweeterIF {
    boolean tweet(Message msg, UserID id, Password pwd);
}

component Client {}

class UserAccount {
    one UserID userid;
    one Password password;
    set Message messages;
}

component Server implements TweeterIF {
    set UserAccount accounts;

    boolean tweet(Message msg, UserID id, Password pwd) {
        let UserAccount acct {
            if (acct.userid = id and acct.password = pwd and acct in this.accounts) then msg in acct.messages
        }
    }

    illustration PlaceATweet {
        show {
            sometimes {
                some { Server srv; Message msg; UserID id; Password pwd | srv.tweet(msg.id,pwd) }
            }
        }
    }

    boolean ServersCorrectlyInitialized() {
        initially { forall { Server srv; UserAccount acct | if (acct in srv.accounts) then no acct.messages } }
    }

    boolean OnlyClientsTweet() {
        forall { Component c; Server s; Message m; UserID id; Password pwd |
```
THE TWEETER MODEL

45} if (invoked c, s.tweet(m, id, pwd)) then c in {Client} }
46
47
48 boolean IntegrityOfUserAccounts() {
49 for all UserAccount acct; Message m |
50 if (m in acct.messages) then (some { Component c; Server srv |
51 acct in srv.accounts and earlier { executed c, srv.tweet(m, acct.userid, acct.password) } })
52 }
53
54
55 boolean ConnectorsAuthenticated() {
56 for all { Connector c | c.callersAuthenticated() }
57
58
59 boolean IntegrityOfUserAccountDetails() {
60 for all { UserAccount acct; Password pwd; UserID id |
61 if (initially { acct.password = pwd and acct.userid = id}) then
62 (always { acct.password = pwd and acct.userid = id}) }
63 }
64
65 boolean NoSharedAccounts() {
66 for all { Server s1; Server s2; UserAccount acct |
67 if (acct in s1.accounts and acct in s2.accounts) then s1 = s2 }
68 }
69
70 goal OnlyCorrectClientsCanTweet {
71 assuming
72 OnlyClientsTweet(),
73 IntegrityOfUserAccounts(),
74 ConnectorsAuthenticated(),
75 IntegrityOfUserAccountDetails(),
76 NoSharedAccounts();
77 check {
78 always {
79 for all { Message m; UserAccount acct; Server srv |
80 if (acct in srv.accounts and m in acct.messages) then earlier {
81 some { Client c | invoked c, srv.tweet(m, acct.userid, acct.password) }
82 }
83 }
84 }
85 }
86
87
88 }
B. The Secure Logger model

```java
package patterns;

module SecureLogger<+Message> {

class Principal {}

object AdminEntity in Principal;

object LoggerEntity in Principal;

class ProcessedMessage {
    one Message content;
    one int id;
}

class SignedMessage {
    one ProcessedMessage content;
    one ProcessedMessage signedContent;
    one Principal signedBy;

    boolean sign(Principal signer) {
        this.signedContent = this.content and this.signedBy = LoggerEntity
    }

    boolean isValid() {
        this.content = this.signedContent and this.signedBy = LoggerEntity
    }
}

interface LogIF {
    void log(Message m);
    int read(Message m);
}

boolean CalculateNextUID(Logger l) {
    initially {
        l.nextUID = #{Message m | some { Component c | executed c, l.log(m) }} and
        from now on {
            l.nextUID = previous { l.nextUID } +
            #{Message m | some { Component c | executed c, l.log(m) }}
        }
    }
}
```
component Logger implements LogIF {
    set SignedMessage contains;
    lone int nextUID;

    boolean log(Message m) {
        some { ProcessedMessage pm; SignedMessage s |
            pm.content = m and
            0 <= pm.id and
            pm.id < this.nextUID and
            s.content = pm and
            (s in this.contains) and
            s.sign(LoggerEntity)
        }
    }

    int read(Message m) {
        let int v = verify() {
            if (v = 1 and some { SignedMessage sm | (sm in this.contains) and
                sm.content.content = m and sm.content.id >= 0 and sm.content.id < nextUID }
            ) then (returns 1) else (returns 0)
        }
    }

    int verify() {
        if allEntriesAccountedFor() and entriesAreValid() then (returns 1) else (returns 0)
    }

    boolean allEntriesAccountedFor() {
        forall { int i |
            if (i >= 0 and i < this.nextUID) then
                some { SignedMessage s | (s in this.contains) and (s.content.id = i) }
        }
    }

    boolean entriesAreValid() {
        forall { int i |
            if (0 <= i and i < this.nextUID) then
                one { SignedMessage s | (s in this.contains) and (s.content.id = i and s isValid()) }
        }
    }

    boolean LoggerSignaturesCannotBeForged() {
        forall { SignedMessage sm |
            let ProcessedMessage pm = sm.signedContent,
                Message pmc = pm.content,
                int pmi = pm.id,
                Principal pr = sm.signedBy {
                from now on {
                    (pm = sm.signedContent and
                    pmc = sm.signedContent.content and
                    pmi = sm.signedContent.id
                } or ( pr != sm.signedBy )
        }
    }
}
### THE SECURE LOGGER MODEL

```java
98 } } } } }  
99 } } } } } }  
100 } } } } } }  
101 } } } } } }  
102 boolean LoggerDoesNotOverwriteEntries() {
103  forall { SignedMessage sm |
104      let ProcessedMessage pm = sm.content, int i = pm.id |
105      if sm.isValid() then so far |
106      no { SignedMessage sm1 | sm != sm1 and sm1.isValid() and sm1.content.id = i } |
107 } } } } } }  
108 } } } } } }  
109 } } } } } }  
110 } } } } } }  
111 } } } } } }  
112 boolean OnlyStoreLoggedItems() {  
113  forall { Logger l | Message m |
114      let int res = l.read(m) |
115      if res = 1 then later { some { Component c | executed c, l.log(m) } } |
116 } } } } } }  
117 } } } } } }  
118 } } } } } }  
119 } } } } } }  
120 boolean SecureConnectors() {  
121  forall { Connector c | c.isSecure() } |
122 } } } } } }  
123 } } } } } }  
124 illustration TestIlllustration {  
125      show { nothing } |
126 } } } } } }  
127 } } } } } }  
128 goal NothingDeleted {  
129  assuming |
130  forall { Logger l | CalculateNextUID(l) },  
131 LoggerSignaturesCanNotBeForged(),  
132 LoggerDoesNotOverwriteEntries(),  
133 SecureConnectors(),  
134  forall { Invocation i | i.live() };  
135  check {  
136      always { forall { Component c | Logger l | Message m |
137         if invoked c, l.log(m) then (  
138         later { from now on {  
139         let int v = l.verify(), int r = l.read(m) { if (v = 1) then (r = 1)  
140         }  
141      } } } } |
142 } } } } } }  
143 scope 3 but 3 Component, 2 Time, 2 Connector  
144 } } } } } }  
145 } } } } } }  
146 } } } } } }  
147 } } } } } }  
148 }
```
Bibliography


