Undertaking the Tokeneer Challenge in Event-B

Victor Rivera, Sukiriti Bhattacharya, and Nestor Catano

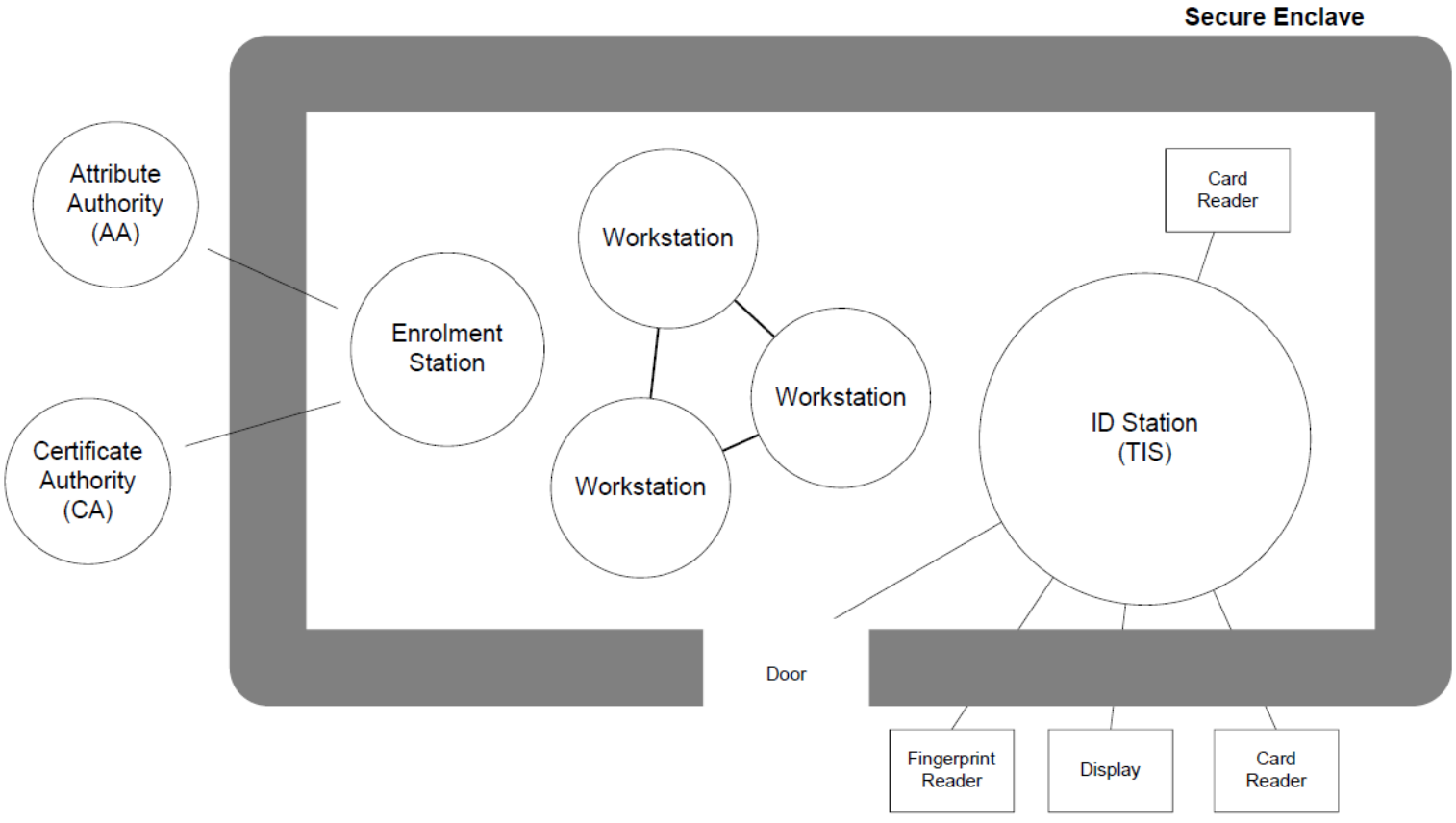
Innopolis University
Technologies and Software Development Institute
Software Engineering Lab.

May 15^{th}, Austin, TX, USA
FormaliSE 2016

The Tokeneer Project

The Tokeneer project

The Tokeneer ID Station (TIS) is responsible for reading a smart card (Token) and, based on a number of protocols and checks, ensuring that any person trying to access the enclave is indeed permitted to enter the enclave, and giving the corresponding grants as a user or administrator.


Secure Enclave
Attribute Authority (AA)
Certificate Authority (CA)
Enrolment Station
Workstation
Workstation
Workstation
Card Reader
ID Station (TIS)
Door
Fingerprint Reader
Display
Card Reader

The Tokeneer project

- It was initiated by the U.S. National Security Agency (NSA) (2003)
- The main idea: as a demonstrator of highly secure, low defect, high-assurance software system.
- The NSA commissioned Praxis to re-develop the software for the Tokeneer ID Station (TIS)

The Tokeneer project

- The specification and implementation were completed in 2003 and made publicly available by the NSA and Praxis in October 2008.
- Forming an ideal base for further research in program verification at both industrial and academic communities.

Tokeneer specification/implementation

Praxis specified TIS in Z and implemented and tested it in Ada, following the System Requirement Specification (SRS) and System Test Specification (STS) documents (documents also publicly available).

Tokeneer Challenges

Tokeneer Challenges

Challenge 1: Re-implement Tokeneer

To use different specification languages, programming languages, verification tools to re-implement Tokeneer.

Tokeneer Challenges

Challenge 1: Re-implement Tokeneer

To use different specification languages, programming languages, verification tools to re-implement Tokeneer.

Challenge 2: Proof of Security Properties

The Tokeneer specification contains 3 security properties. Praxis presented a full demonstration of one of them and a partial demonstration to another one: to fully proof all security properties.

Challenge 1

Challenge 1:
An approach for software development

Steps

Steps

1. To model TIS in Event-B (a formal methods based on Step-wise Refinement) based on the existing Z specifications of Tokeneer. This allows the TIS to be described in different levels of abstraction.

Steps

1. To model TIS in Event-B (a formal methods based on Step-wise Refinement) based on the existing Z specifications of Tokeneer. This allows the TIS to be described in different levels of abstraction.
2. To use Rodin (an IDE for Event-B) to verify the Event-B model of TIS.

Steps

1. To model TIS in Event-B (a formal methods based on Step-wise Refinement) based on the existing Z specifications of Tokeneer. This allows the TIS to be described in different levels of abstraction.
2. To use Rodin (an IDE for Event-B) to verify the Event-B model of TIS.
3. To use EventB2Java (a tool of Event-B) to generate Java code for the Event-B model of the TIS.

Steps

1. To model TIS in Event-B (a formal methods based on Step-wise Refinement) based on the existing Z specifications of Tokeneer. This allows the TIS to be described in different levels of abstraction.
2. To use Rodin (an IDE for Event-B) to verify the Event-B model of TIS.
3. To use EventB2Java (a tool of Event-B) to generate Java code for the Event-B model of the TIS.
4. To propagate the requirements in the STS document all the way down to the JUnit test cases drawn from the generated Java code, so that upon a test case failure, it is possible to back-trace the failed requirements to the corresponding parts in the model.

A brief description of Event-B

The Event-B method

Event-B models are complete developments of discrete transition systems:

- systems go through a series of stages, named refinements.
- each refinement is a description of the system with a higher level of detail.
- each refinement is provably consistent with the previous one (the proof obligations).
- Event-B models are composed of *context*s (static part) and *machine*s (dynamic part).

Event-B: Rodin

- Rodin platform is an open-source Eclipse IDE for the development and verification of the Event-B models.
- Rodin automatically generates the set of proof obligations (PO) necessary to prove consistency of machines.
- Rodin comes with a series of plug-ins that extend its functionality. For instance:
  - AtelierB provers to help users to automatically discharge proof obligations.
  - EventB2java generates Java implementations of Event-B models.

Experimental results

Step 1: Event-B model of TIS

TIS Event-B model consists of an Abstract machine and 6 refinements:

Machine	**LOC**
Abstract	43
certificate_L1	72
certificate_L2	125
entry_L1	219
entry_L2	264
enrol	213
admin	517
Total	1453

We used the existing Z specification of TIS as a requirement document.

Step 2: Event-B model of TIS – Verification

- Event-B is based on the idea of structuring a development into many small steps to achieve a high degree of automation.

Machine	**LOC**	**#POs**	**Aut**
Abstract	43	9	100
certificate_L1	72	36	88.9
certificate_L2	125	58	93.1
entry_L1	219	59	88.13
entry_L2	264	36	83.3
enrol	213	4	100
admin	517	132	90.9
Total	1453	334	90.1

Step 2: Findings

- Our initial Event-B model was inconsistent. We inspected the model and found and corrected the inconsistencies.
- We were able to achieve a high degree of proof automation in Rodin (90.1% were discharged automatically using Rodin's proof engines).

Step 2: Findings

- We gained confidence about our Event-B model: conditions expressed in the SRS document were formally introduced as invariants and events in Event-B.
- We were also able to encode and prove all three security properties of TIS (Challenge 2).
- We were able to prove the soundness of the system w.r.t those conditions by discharging all POs with Rodin.

Step 2: Findings

- However, there is no clear way to be sure that the formal specifications in Event-B are sound w.r.t. the English description of the requirements.

Step 3: Java implementation of TIS

- Once we finished to model the TIS in Event-B, we used EventB2Java to generate a Java implementation of the Event-B model.

Step 3: Findings

- EventB2Java automatically generated 2704 lines of Java code.
- The only difficulty we found with the translation is that EventB2Java does not generate initial values for Event-B constants in Java.
- Those values were manually set to adhere to axioms written in Event-B.
- Nevertheless, these initial values are clearly described in the STS document.

Step 4: JUnit testing of the implementation

- We manually wrote executable JUnit tests from the STS document given by Praxis.
- The document also provides the needed input to the unit under test and the generated JUnit tests evaluate its output before assigning any verdict about its success or failure.
- We ran the set of tests against the code with a supplied input data, then we compared the results obtained against the expected results. Any mismatch in the result implies that the model needs to be improved.

Step 4: Findings

- The initial Java implementation of the TIS did not pass all the tests.
- We inspected our Event-B model, found errors, and generated Java code again.
- We repeated this process until the generated Java code successfully passed all the tests.
- We are confident about the behaviour of the implementation since it meets the expectations defined in the STS document.

Tokeneer experiment

In applying this approach, users have the following advantages:

i finding inconsistencies in the Event-B model by
   - discharging POs and
   - performing tests in Java;

ii Event-B refinement chains tend to be short;

iii allows experts from different domains to work together;

iv once the model is correct and behaves correctly, it is ready to be implemented. This approach ends with an initial Java program.