Designing and Testing a High Confidence ASN.1 Compiler

HCSS ‘04

National Security Agency
Information Assurance Directorate
United States of America

galois

Overview

- The Challenge of ASN.1
  - Complexity
  - Exposure / impact / risk
- A High Assurance Response
  - Meeting community's needs
- Our Approach
- Next Steps

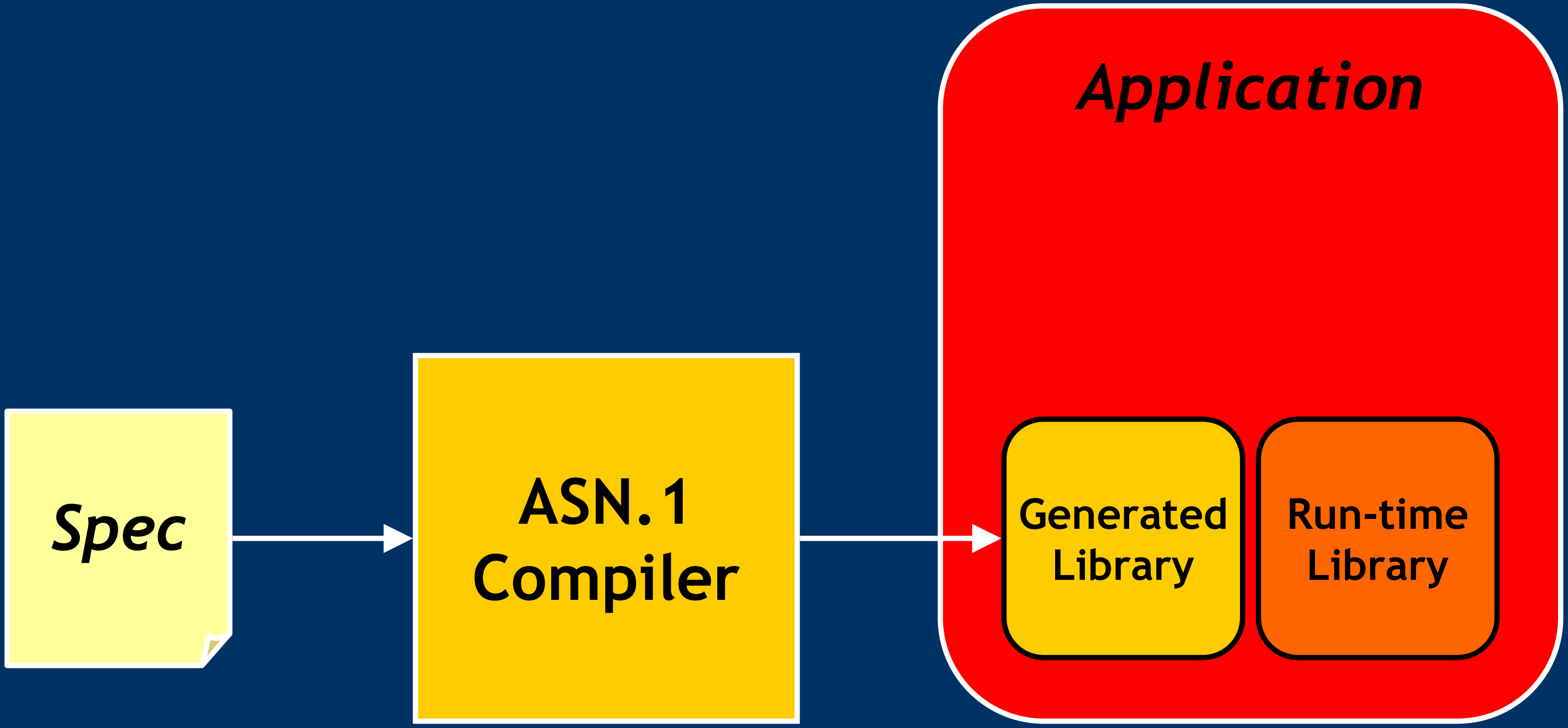
Outline of the Challenge

- ASN.1 and its uses
- Complexity of ASN.1
- Likelihood of errors in generated code
- Consequences of those errors
- Existing tools
- A High Assurance tool

ASN.1 is Everywhere

- **ASN.1 enables description of data in platform-independent manner, ensuring that messages are:**
  - Mutually intelligible
  - Given the same semantics by both parties
- **ASN.1 is used in most network protocols, *e.g.*:**
  - SET, SNMP, TCAP, CMIS/CMIP, PKCS, MHS, ACSE, CSTA, NSDP, DPA, TDP, ETSI, DMH, ICAO, IMTC, DAVIC, DSS1, PKIX, IIF, LSM, MHEG, NSP, ROS(E), FTAM, JTMP, VT, RPI, RR, SCAI, TME, WMtp, GDMO, SMTP, X.400, X.500, X.509, SSL, …
- **ASN.1 encoding and decoding code is ubiquitous**
  - In practically every network device or application

ASN.1 Compiler



Spec

ASN.1 Compiler

Application

Generated Library

Run-time Library

ASN.1 is Large

- As a language, ASN.1 is very large:
  - Sums (e.g., CHOICE, SET)
  - Records, with subtyping (e.g., SEQUENCE)
  - Recursive data types (e.g., SEQUENCE OF, SET OF, user-defined)
  - Many (~26) primitive types
  - Constraints (X.680, X.682)
  - Information objects (X.681)
  - Parameterization (X.683)
- Writing a compiler a difficult, error-prone task

ASN.1: Complexity through Density

- **Even core elements (X.680 ASN.1 definition and X.690 BER/DER/CER definition) very dense:**
  - Precise semantics of ASN.1 is very difficult to extract
  - It is difficult to *know* when you've got it right, or for two parties to agree on what *is* right
- **Deciding when decode should reject messages**
  - Crucial but very difficult
  - Constraints semantics given in terms of concrete syntax
- **Semantics of type equality obscure**
  - Given in terms of lists of lexical tokens
  - Assignment uses this to resolve implicit subtyping/overloading
- **Language constructs subtly non-compositional:**
  - Semantics can depend upon its context, but structure gives no hint
  - Cross-feature interference (*esp.* CHOICE and tagging)
- ***Every* type is a special case**
- **Multiplicity: 13 string types**
  - Each defined in terms of International Register Tables

Generating Code for ASN.1

- **Additional pitfalls for the compiler implementor:**
  - Numerous opportunities for overflowing machine representation:
    - Arbitrary precision integers and reals
    - Arbitrarily long octet streams (led to recent bug Microsoft ASN.1 library)
  - Similar concepts get treated very differently:
    - *e.g.*, long tags *vs.* long lengths *vs.* long values
    - Barrier to problem understanding, good code design

Consequences of Failure

- **High impact:**
  - Leads to attacker ingress, vulnerability to DoS
  - ASN.1 code often run in "privileged" mode
- **Costs of fixing ASN.1 problems estimated to be much greater than Y2K reparation[1]:**
  - More equipment affected
  - Repairs must be done more quickly, more often
  - More regression testing required (configuration complexity)
  - Attacks lead to outages, plus take time & money to discover, repair

[1] "Critical Infrastructure Protection Issues", Bill Hancock, V.P. Security and Chief Security Officer, Exodus, *ITU Workshop on Creating Trust in Critical Network Infrastructures*, May 2002

Existing Tools Unsatisfactory

- **Open source:**
  - Evaluation/certification possible
  - eSNACC:
    - Not fully featured
    - Produces incorrect code for encoding INTEGER
- **Closed source:**
  - Barrier to evaluation/certification
  - OSS Nokalva:
    - Fully featured
    - Produces incorrect code for decimal encoding of REAL
    - API not developer-friendly
  - Others from Objective Systems Inc., Sun, Atos Origin:
    - Sun, Atos Origin: old versions of ASN.1
    - Objective Systems Inc.: fully featured, code not examined
- **Hand-written encode/decode:**
  - Expensive to produce; no reuse of certification possible
- **Hand-written validation code:**
  - Expensive to produce; no reuse of certification possible

Requirements for HA ASN.1 Compiler

- Supports all of ASN.1 X.680 (07/2002)
  - Some legacy support also required (macros, ANY DEFINED BY)
  - Desirable: X.681, X.682, X.683 (07/2002)
  - Supports BER/DER/PER
- Is easy for developers to use
  - Good error messages
  - Produces code that is easy to use (well-designed APIs)
- Produces correct, robust encode/decode routines every time
  - Passes NSA C group evaluation
  - Becomes part of “approved” tool chain
  - Produces code that obeys properties that may be used in certifying the parent application

Galois and ASN.1

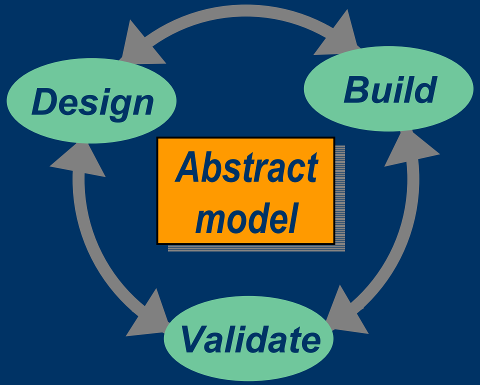
- **In 2002, Galois was working on H-CDSA, sponsored by NSA R2:**
  - High confidence reworking of Intel's Common Data Security Architecture
- **Needed to parse X.509 certificates**
- **Team used own innovative approach:**
  - Parse ASN.1 definition into Haskell type
  - Use Haskell polymorphism and class system to derive encode/decode routines
  - Very similar to Slind *et al*. polytypic approach
- **Client saw value in investigating how far these ideas could be taken**

Showing Galois' Approach Valid

- **NSA R2 funded Galois to build proof-of-concept for ideal tool:**
  - Supports meaningful subset of ASN.1
  - Supported features behavior on legal inputs thoroughly tested
  - Design and implementation demonstrably amenable to evaluation
  - Tool friendly, API friendly

Galois' Methodology

- Specify and develop in problem domain:
  - Makes use of formal methods tractable
  - Allows focus on crucial properties
- Apply mathematical rigor early
- Use functional programming to express model in executable form
- Progress rapidly to implementation



galois

Designed for Manifest Correctness

- **Parser grammar almost identical to X.680 grammar:**
  - Direct comparison feasible
- **Code generation correct by construction, transformation and translation:**
  - V1: type-driven specification of encode/decode
    - Derivation system gives a formal semantics to ASN.1
  - V2: lambda calculus implementation of encode/decode
    - Inlined, specialized version of V1
    - Gives a formal semantics to individual ASN.1 specifications
  - EnDe C: domain-specific language for encode/decode
    - Translated from V2
    - Gives an operational semantics to individual ASN.1 specifications
  - C: Final code target
    - Translated from EnDe C

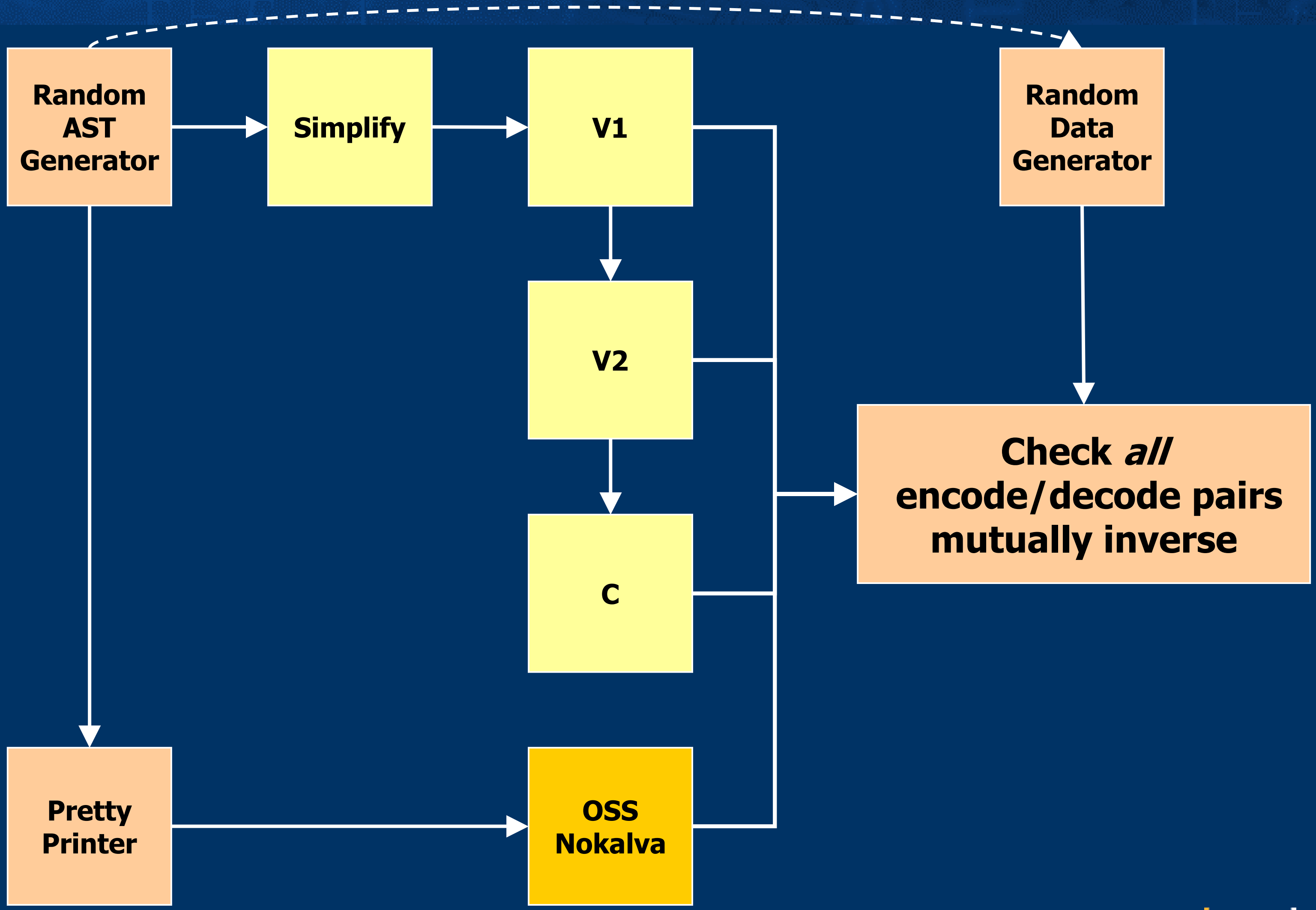
Designed For Robustness

- **Mapping to C for each EnDe C construct considered in isolation**
- **Each mapping designed with robustness properties in mind:**
  - Use ADT-style API for all types
    - Our code handles all allocation, user handles freeing
  - Encode calculates the buffer size required before encoding; allocates accordingly
  - All buffers have associated lengths
  - All `malloc`s are guarded
  - All pointer dereferences guarded
- **Run-time library designed from same principles**

Random Coverage Testing

- **Handwritten tests**
  - Unit tests
  - Regression testing
- **Randomly-generated test data**
  - Coverage metrics for the input space
    - Gives an idea of how *representative* a test set is
  - Tests expected behavior on valid inputs for:
    - Parser
    - Static Analysis
    - Code generation
- **Rejection behavior**
  - Testing framework currently at design stage
  - Corresponding approach planned

Expected Behavior of Generated Code



Random AST Generator

Simplify

V1

V2

C

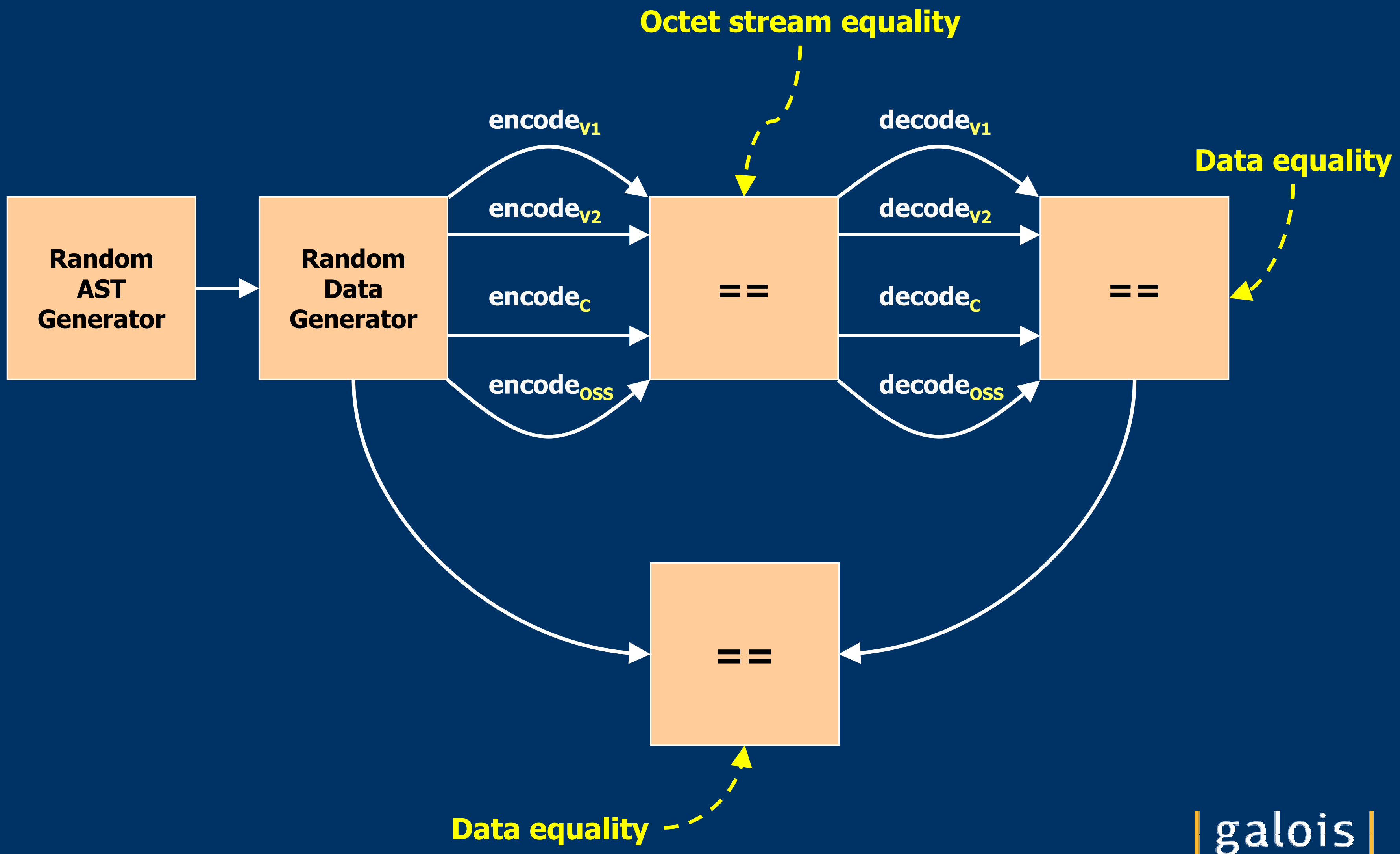
Random Data Generator

Check *all* encode/decode pairs mutually inverse

Pretty Printer

OSS Nokalva

Mutually Inverse Encode/Decode Pairs



Octet stream equality

encode$_{V1}$

encode$_{V2}$

encode$_{C}$

encode$_{OSS}$

decode$_{V1}$

decode$_{V2}$

decode$_{C}$

decode$_{OSS}$

Data equality

Random AST Generator

Random Data Generator

==

==

==

Data equality

Haskell Property Code

```
prop_decodeEncodeT1_All :: T1 -> Bool
prop_decodeEncodeT1_All x =
  let x'  = Right (x, [])

      os  = H.encTLV_T1 x  in
      os == I.encTLV_T1 x  &&
      os == C.encodeT1  x  &&
      os == E.encodeT1  x  &&

      H.decTLV_T1 os == x' &&
      I.decTLV_T1 os == x' &&
      C.decodeT1  os == x' &&
      E.decodeT1  os == x'
```

Test Results

Phase	# Tests	# Defects
Parser	290k	6
Static Analysis	4k	22
Back-end	8k	34

Manifest Correctness and Testing

- **If the design is so good, why the need for testing?**
  - Design ≠ implementation, even in high-level languages like Haskell
    - Developers make errors
  - Design was of little help with primitive types
    - Biggest problem is correctly *interpreting* X.690 spec
    - Several defects related to primitives
  - Design was of no help with transition between static analysis phase and back-end
    - Most defects in this phase transition
- **Even so, number of defects surprisingly low for project of this complexity**

Project Summary

- Ran from October 2003 to March 2004
- Team:
  - Galois (3 developers)
  - SPRE:
    - Reliability evaluation
    - Ongoing (due end April 2004)
  - NCSU:
    - Studying Galois software process
    - Led to work with Programatica (OHSU OGI)
      - Deriving reliability measure from certificate graph

Compiler Delivered

- Supports most primitives, and important compound types
- Supports value notation for supported types
- Detailed, accurate and "friendly" error messages
  - Type errors, validation errors
  - Syntax error reporting not yet friendly
- Generates Haskell and C
- Windows and Linux
  - Compiler and compiled code
- Test plan fully implemented for valid inputs to parser, static analysis, code generation

Possible Next Steps

- **Define precise, formal ASN.1 semantics**
  - Based on core elements, not concrete syntax
- **Define precise EnDe C semantics:**
  - Formally express translation to C code
  - Formally express desirable robustness properties
- **Generate proof scripts to enable machine-assisted proof of correctness, robustness properties**

Conclusions

- **ASN.1 is intricate and complex**
  - Innovative techniques were required
  - Formal, semantics-based, transformational code-generation yielded very low defect count for compound types
  - Test plan exposed bugs in other compilers, our code, for primitive types
- **Success of this project shows that it is feasible to build a fully-featured high assurance ASN.1 compiler**
  - Semantic approach enabled the production of a solid compiler in short time
  - Techniques will scale: full ASN.1 introduces no new fundamental challenges

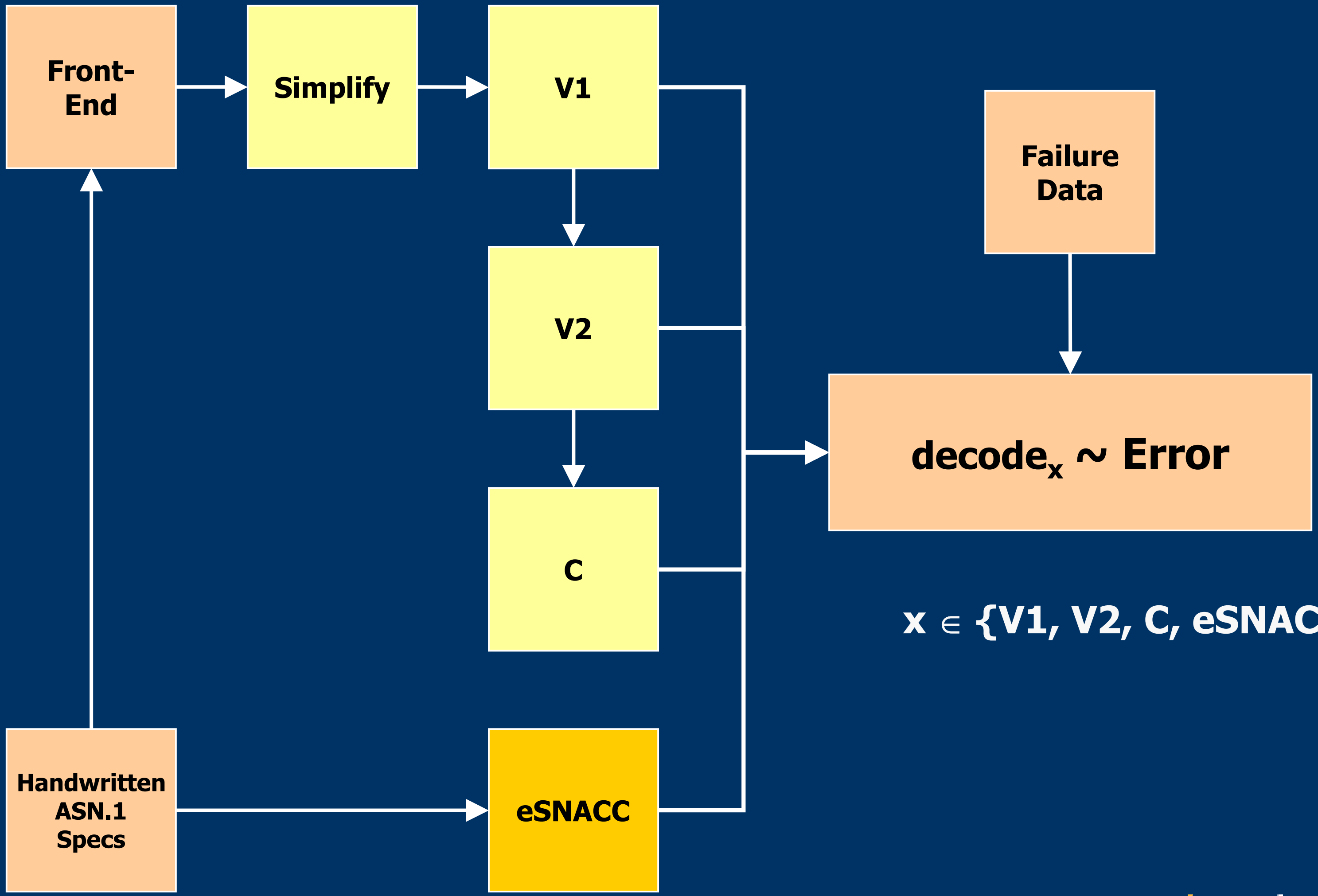
Formal ASN.1 Semantics

- **X.680 (and others) defines ASN.1 in terms of concrete syntax:**
  - Dense, hard to read
  - Details and interrelationships must be painstakingly teased out
  - ASN.1’s constraint language is particularly baroque
- **Formal ASN.1 semantics:**
  - Express the core elements of ASN.1, without extraneous detail
  - Translate every ASN.1 spec into a core ASN.1 spec
  - Well-defined, easily understood semantics
  - Suitable for use in formal proof
- **Inherently useful as a reference**

Enhancing Robustness Design

- **Augment EnDe C operational semantics with a model of memory usage**
- **Benefit:**
  - Formally express the mapping to C
  - Formally express robustness properties, *i.e.*, that generated code does not introduce:
    - Space leaks
    - Integer overflow
    - Buffer under/overrun
    - Stack overflow
    - Dangling pointers
  - Automatic generation of test data
    - Deepen confidence that code generation preserves these properties

Decode Rejection Behavior



$x \in \{V1, V2, C, eSNACC\}$

Generating Proof Scripts

- **For a given ASN.1 specification, generate a theorem prover proof script that:**
  - Automatically *proves* that encode/decode pairs are all mutually inverse
  - Automatically *proves* that generated code preserves robustness properties
- **Formal correctness of each compilation**
  - Appears to be a tractable approach (*cf*. Slind *et al*.)
  - Adds more confidence to correctness of compiler
  - Provides artifacts for use by compiler user in their certification effort:
    - Requires some requirements input from C group