Design Tools for the Development of Reliable Secure Communication Software

Johann Schumann, RIACS / NASA Ames

PC-World, 4/2004:

Microsoft Posts Critical Patches Four security bulletins cover many fixes in Windows, IE, Outlook.

> *. . . among the most critical holes Microsoft is warning about is a* **buffer overrun** *vulnerability in the Local Security Authority Subsystem Service (LSASS),. . .*
>
> *The second critical vulnerability is a* **buffer overrun** *hole in the Private Communications Transport (PCT) protocol. . .*

Introduction

Software for Secure Communications can become insecure due to *flaws* during any phase of the software life cycle

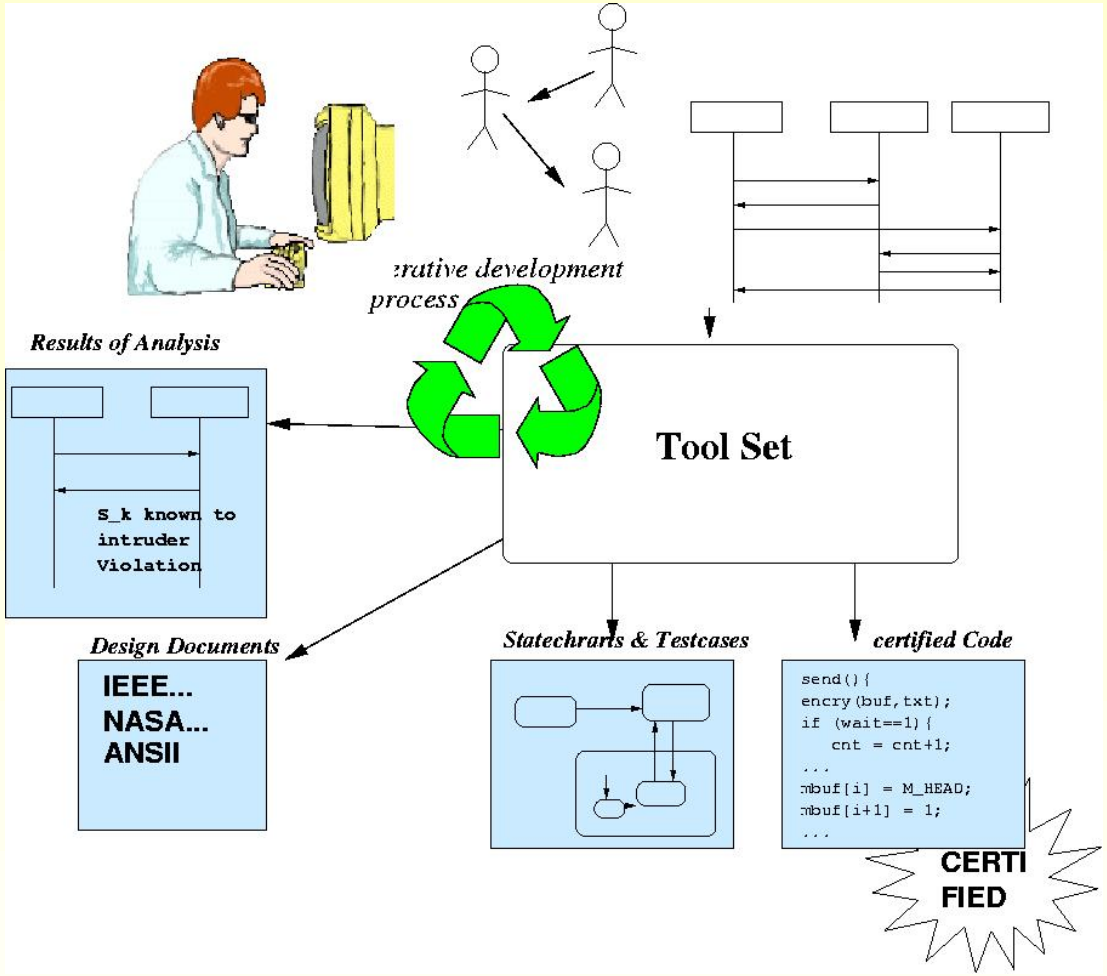
- Design
  - wrong algorithm
  - wrong requirements
- Implementation
  - buffer overrun, uninit'd variable, . . .
  - sleeper codes
- Verification and Validation
  - wrong tests
  - insufficient test coverage
- Deployment
  - wrong code (e.g., disabled crypto)
  - code tampering

Our Goal

Cost-effective development for *reliable, secure communications software*

- unified, tool-supported framework from specification to deployment
- support iterative process
- automatic certification support

Overview

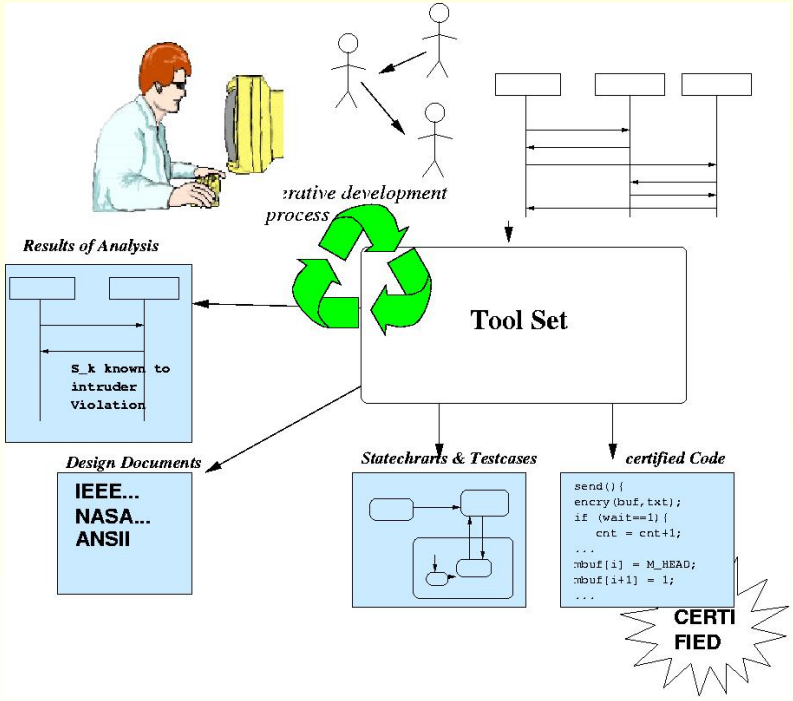

erative development process
Results of Analysis
S_k known to intruder Violation
Tool Set
Design Documents
IEEE...
NASA...
ANSII
Statechrarts & Testcases
certified Code
send(){
encry(buf,txt);
if (wait==1){
cnt = cnt+1;
...
mbuf[i] = M_HEAD;
mbuf[i+1] = 1;
...
CERTI
FIED

The Indivdiual Tools: Overview

riacs Research Institute for Advanced Computer Science

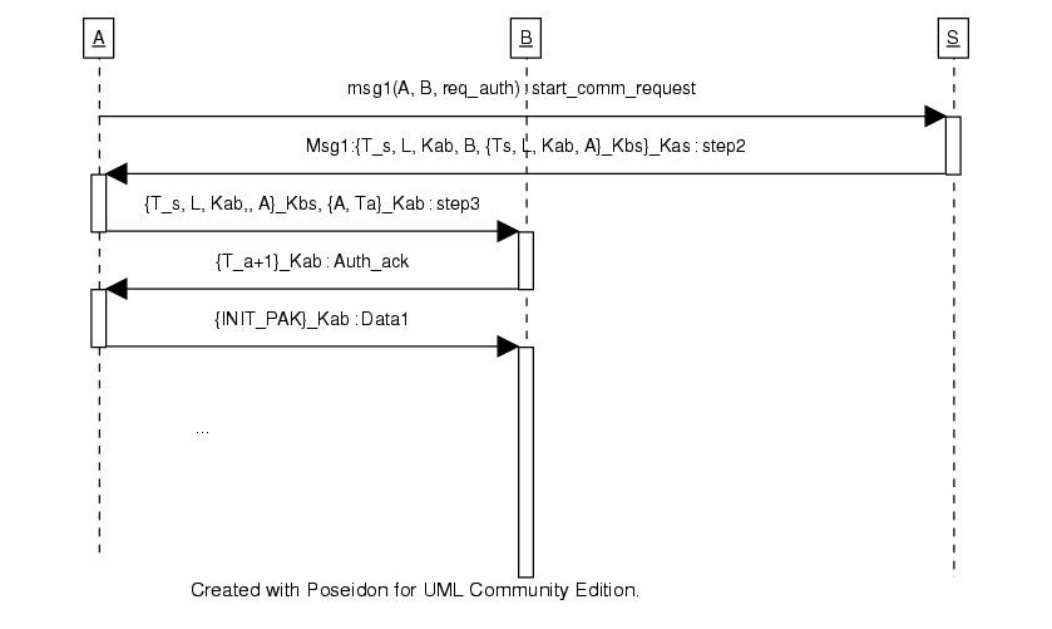
Requirements:

- iterative process
- fast turnaround
- security analysis
- reliable, secure code
- certification



- High-level Specification
  - UML for specification
- Automatic Protocol Analysis
  - Simulation, MC, . . .
- Correct Protocol Optimization
- Automatic generation of code
  - Design document gen.
  - multi target platforms
- Automatic test case generation
- Automatic certification support

Protocol Specification



- specification of protocol as UML sequence diagrams and Use Cases
- distributed representation
- OCL annotations regarding reliability and security properties
- UMLSec (?)

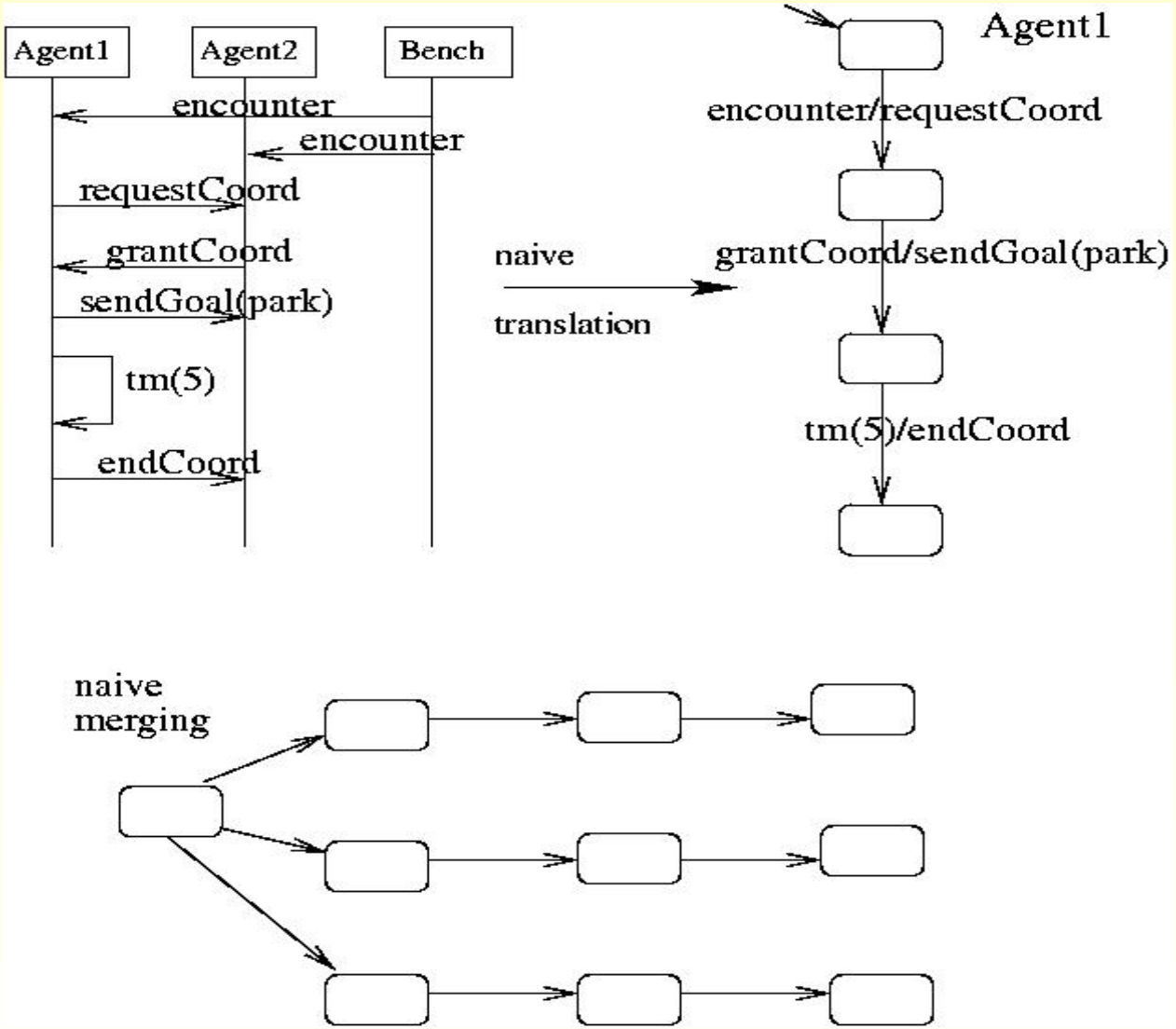
Explicit specification of protocol with safety/security properties

Generation of Design

- automatic Synthesis of UML-designs from sequence diagrams
- generation of readable, highly structured statecharts from sequence diagrams and OCL constraints
- provably correct designs
- automatic detection of inconsistencies and loops

Automatic synthesis of Statecharts facilitate rapid, consistent turn-around

Statechart Synthesis: The naive Approach



State Vector

- formed by OCL specifications on SD Messages and global invariants
- we use a *state vector* (set of variables):

```
<coordWith, hasSessionKey, msgCounter>
```

- Constraints (*domain theory* + *safety policy* + *security policy*)

```
context A::getKey()
   pre:  self.hasSessionKey = false ;          <?,F,?>
   post: self.hasSessionKey = true ;           <?,T,?>
```

- undetermined values are marked by a '?'
- not all messages need to be annotated with pre/postconditions

Automatic Synthesis of Statecharts

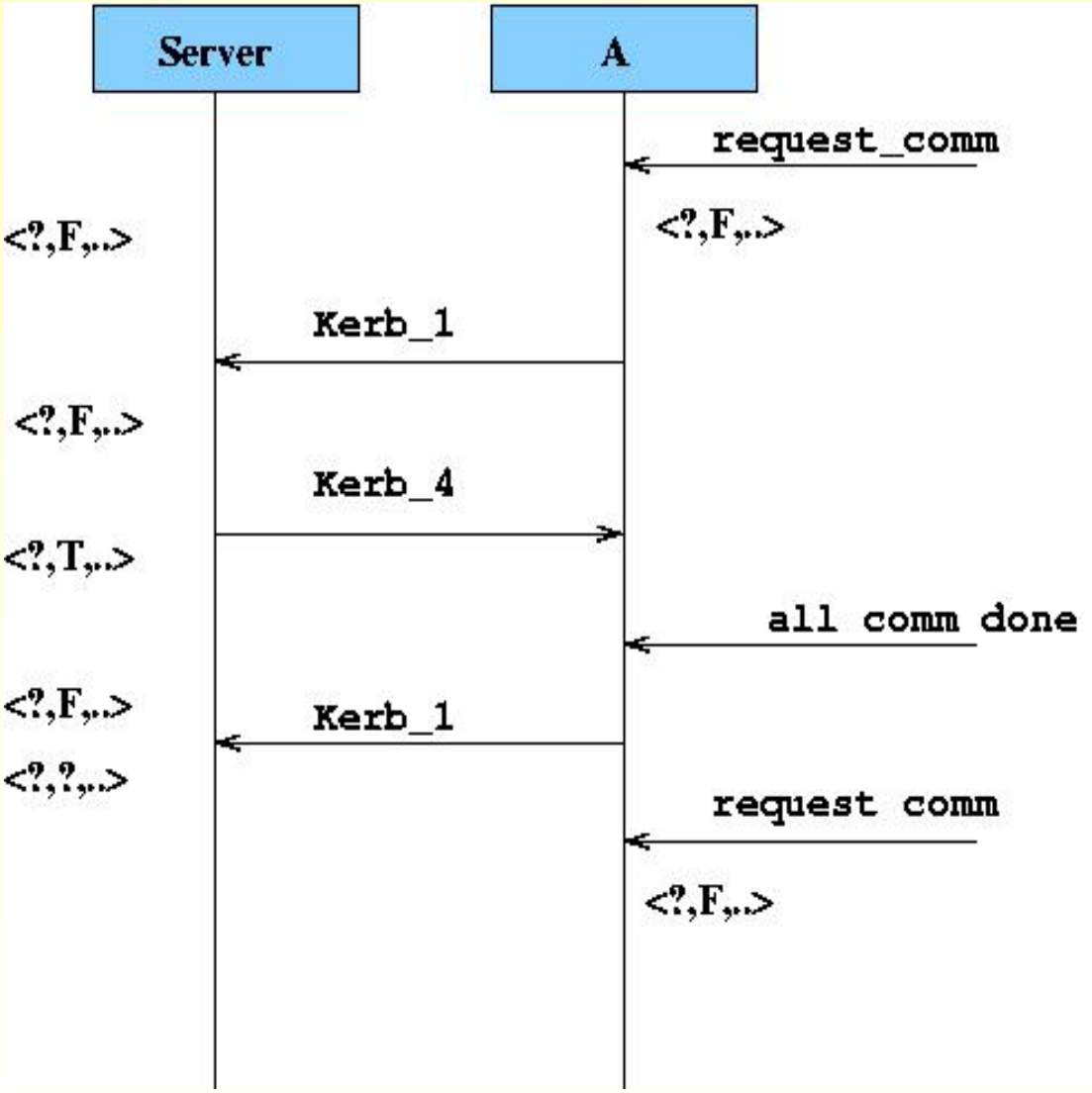
riacs Research Institute for Advanced Computer Science

1. Annotation with constraints for justified merge of SDs
2. Conflict Detection
3. Generation of a flat statchart for each SD
4. Merge of the SCs
5. Introduction of Hierarchy

Conflict Detection

Set variable assignment in SD

- *Unification*: assign values to '?'
- *Frame axiom*: if variable not set by a message, carry it over to the next message
- loop detection

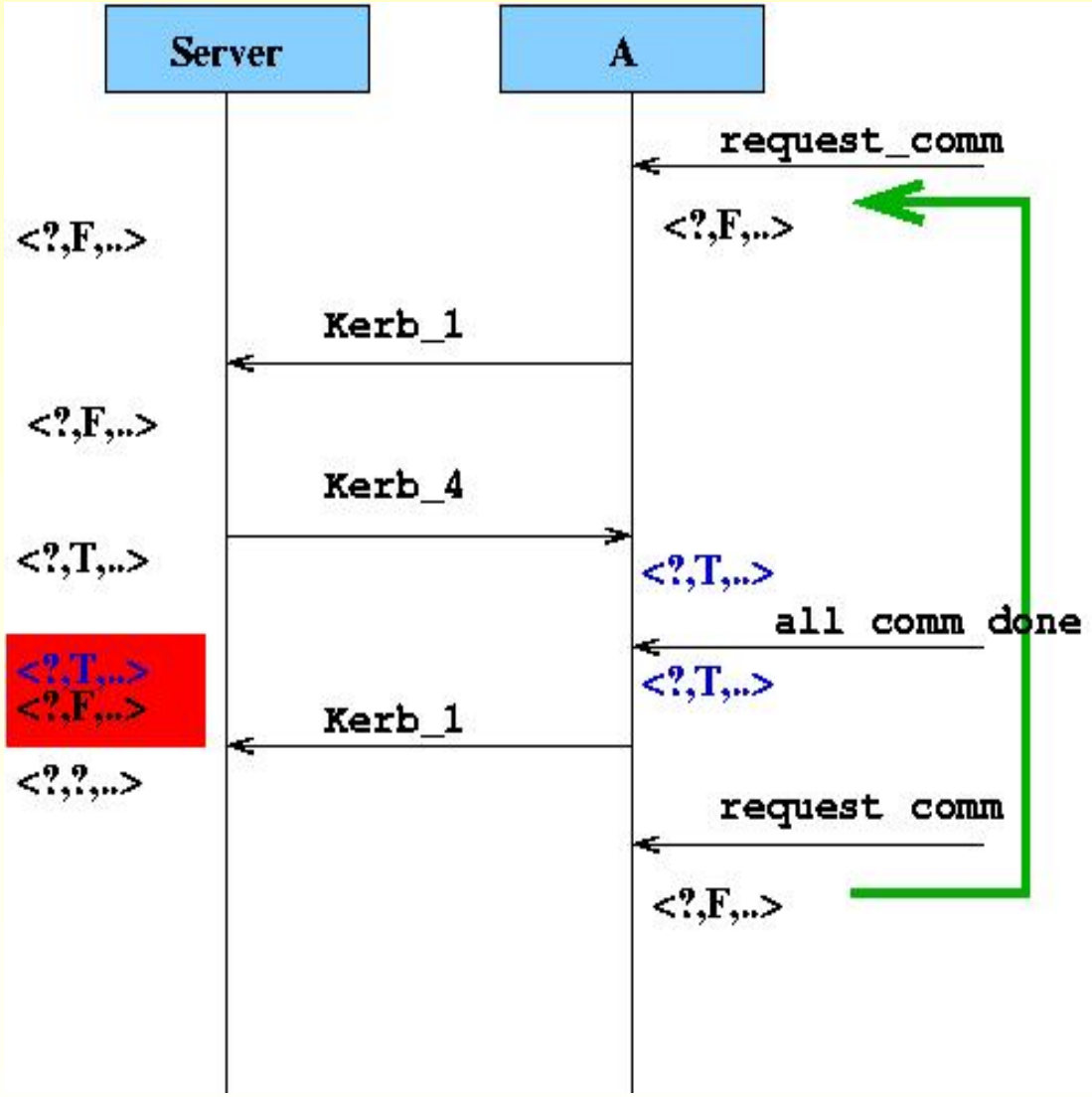


State vector: `<coordWith, hasSessionKey, msgCounter>`

Conflict Detection

Extend variable assignment in SD

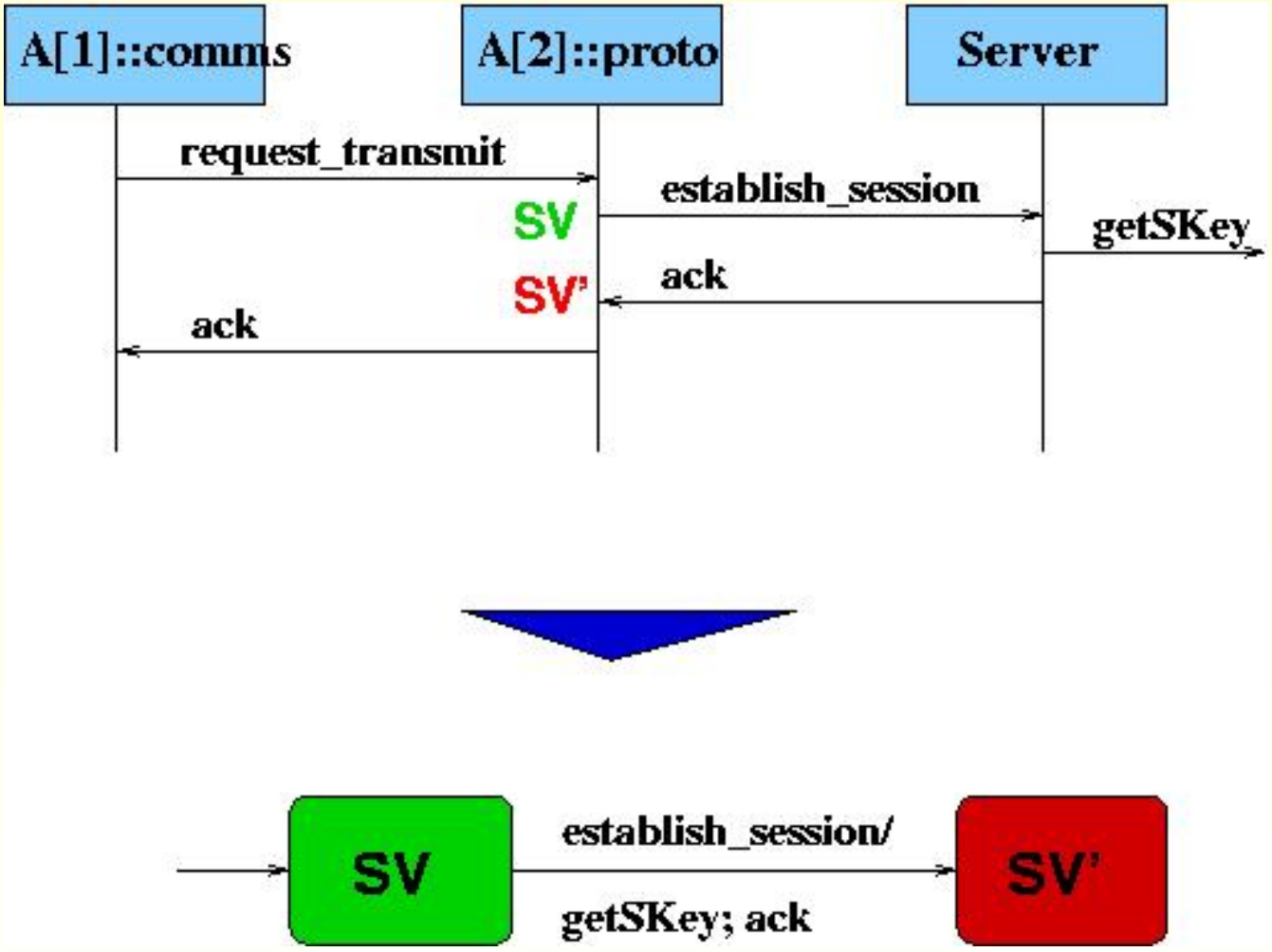
- *Unification*: assign values to '?'
- *Frame axiom*: if variable not set by a message, carry it over to the next message
- loop detection



State vector: `<coordWith, hasSessionKey, msgCounter>`

Merging SDs

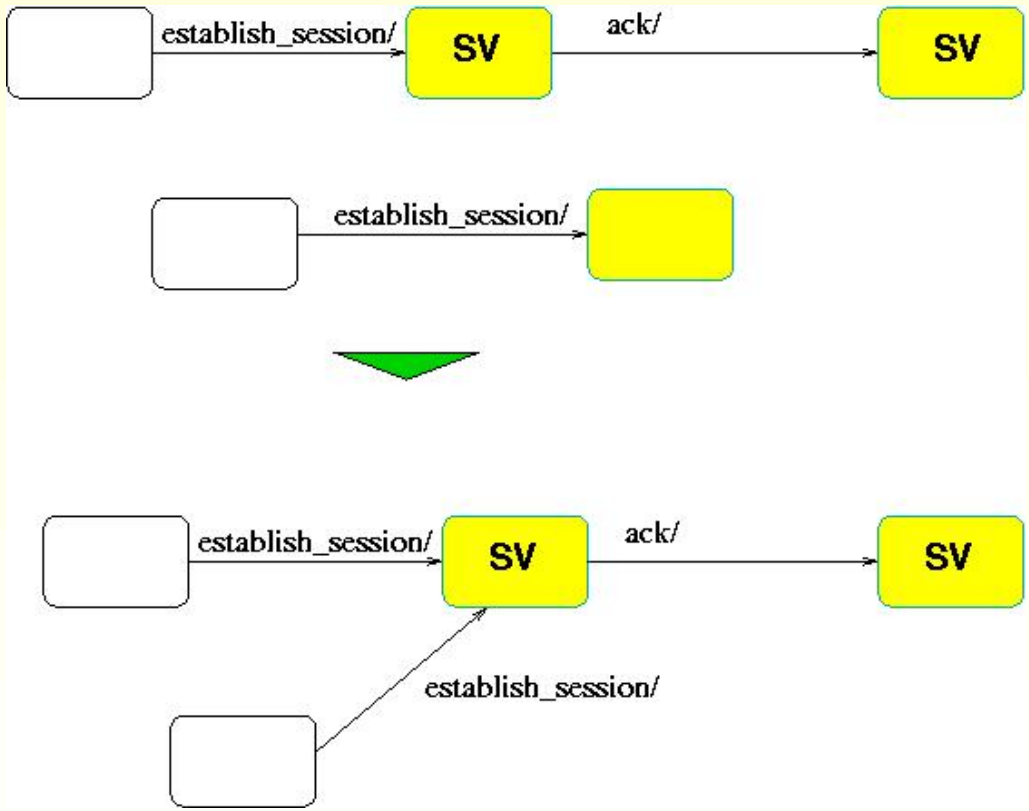
- Translate SD → flat SC



Merging SDs

- multiple flat SCs $\Rightarrow$ single flat SC

  two nodes are similar *if they have identical state vectors and have* ≥ 1 *incoming transition in common*



Introduction of Structure/Hierarchy

Synthesized statecharts must be *readable/understandable* for

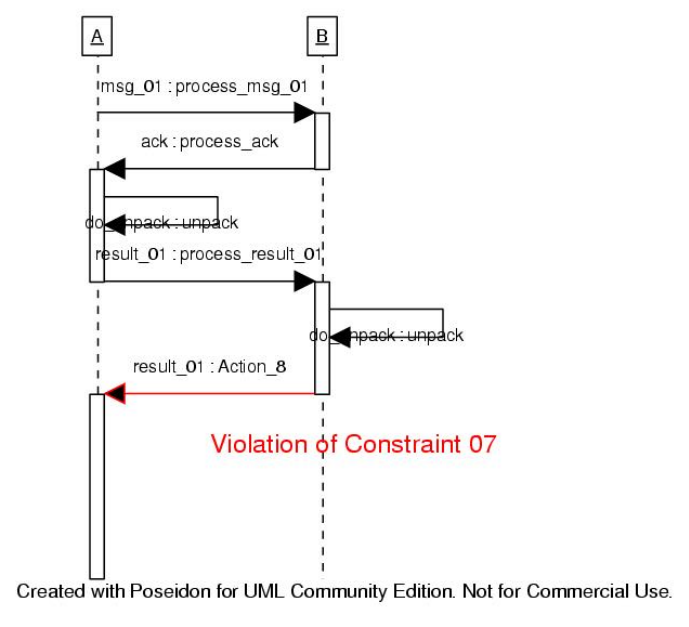
- manual refinement and modifications of the initial design
- system architecture (high-level states) often known

What is a “well-designed”, readable statechart?

- consolidation of related behavior
- separation of unrelated behavior
- introduction of meaningful abstractions
- heuristics: number of nested levels, nodes in one supernode, and inter-level transitions should have reasonable values

Protocol Analysis

- integration of OTS analysis and verification tools
  - theorem proving
  - model checking
  - simulation
- integration of results/feedback into our framework



Because of *message_meaning*

$$\vdash A \models B \mid\sim C :- \vdash A \lhd \{C\}_D \wedge \vdash A \models B \overset{D}{\rightarrowtail} A. \quad (14)$$

Because of *lemma_from_task_2*

$$\vdash pB \models pA \overset{K_{ab}}{\rightarrowtail} pB. \quad (15)$$

Because of *sees_components*

$$\vdash A \lhd B :- \vdash A \lhd C \wedge B = \iota(C). \quad (16)$$

Because of *oneof* $A = \iota(\{B,C\}) :- A = \iota(C)$. Because of *oneof* $\{\{T_a, pA \overset{K_{ab}}{\rightarrowtail} pB\}\}_{K_{ab}} = \iota(\{\{T_a, pA \overset{K_{ab}}{\rightarrowtail} pB\}\}_{K_{ab}})$. Therefore $\{\{T_a, pA \overset{K_{ab}}{\rightarrowtail} pB\}\}_{K_{ab}} = \iota(\{\{\{T_s, pA \overset{K_{ab}}{\rightarrowtail} pB\}\}_{K_{bs}}, \{\{T_a, pA \overset{K_{ab}}{\rightarrowtail} pB\}\}_{K_{ab}}\})$. Hence by (16) and by (5) $\vdash pB \lhd \{\{T_a, pA \overset{K_{ab}}{\rightarrowtail} pB\}\}_{K_{ab}}$. Hence by (14) and by (15) $\vdash pB \models pA \mid\sim (\{T_a, pA \overset{K_{ab}}{\rightarrowtail} pB\})$. Hence by (11) and by (13) $\vdash pB \models pA \models (\{T_a, pA \overset{K_{ab}}{\rightarrowtail} pB\})$. Hence by (8) and by (10) $\vdash pB \models pA \models (pA \overset{K_{ab}}{\rightarrowtail} pB)$. Hence by (7) $\vdash pB \models pA \models pA \overset{K_{ab}}{\rightarrowtail} pB$. Hence by (6) *query*. Thus we have completed the proof of (4).

q.e.d.

Protocol Optimization

- specific communication requirements require specific protocol variants
  - low bandwidth
  - low computational resources (memory, CPU)
  - high latency (e.g., 20' to Mars)
- avoid repeated "wrapping" of data
- avoid unnecessary messages and synchronisation
- without loosing correctness

logic-based protocol optimization (collaboration with Cornell) will be integrated into our framework

Formal methods based protocol optimization enables the designer to specifically tailor the communication protocol without compromising reliability and security

Certification Support

We are developing a *product-oriented* approach to certification

- don’t verify the generator, but focus on produced artifacts
- common techniques:
  - testing and simulation
  - code review
  - program analysis
  - model checking
  - program verification
- important requirements
  - automatic processing
  - tamper-proof certificates
  - no annotations to be provided by the user

Our Approach: Synthesizing Certifiable Code

riacs Research Institute for Advanced Computer Science

Our Approach: Synthesizing Certifiable Code

riacs Research Institute for Advanced Computer Science

Demonstrate that the code generator cannot introduce errors for each piece of generated code

Basic Idea I:

Combine automatic software construction (synthesis)
with *automatic* software inspection (certification)

Our Approach: Synthesizing Certifiable Code

Demonstrate that the code generator cannot introduce errors for each piece of generated code

Basic Idea I:

Combine automatic software construction (synthesis) with *automatic* software inspection (certification)

Basic Idea II:

Certify generated programs, not the generator

Our Approach: Synthesizing Certifiable Code

Demonstrate that the code generator cannot introduce errors for each piece of generated code

Basic Idea I:

Combine automatic software construction (synthesis) with *automatic* software inspection (certification)

Basic Idea II:

Certify generated programs, not the generator

Basic Idea III:

Introduce code certificates

Our Approach: Synthesizing Certifiable Code

riacs Research Institute for Advanced Computer Science

Demonstrate that the code generator cannot introduce errors for each piece of generated code

Basic Idea I:

Combine automatic software construction (synthesis) with *automatic* software inspection (certification)

Basic Idea II:

Certify generated programs, not the generator

Basic Idea III:

Introduce code certificates

Basic Idea IV:

Use Floyd-Hoare program verification techniques

Certifiable Properties

Most software errors are violations of safety properties

$\Rightarrow$ see Introduction

$\Rightarrow$ on the Aerospace top-ten code review list

- language-specific properties
  - **array bounds (memory safety)** (pack/unpack)
  - **variable initialization-before-use**
  - underflow/overflow (e.g., message counter)
- domain-specific properties
  - **module input-use**
  - volatile memory access limitations
  - security properties
- effectiveness properties
  - Worst Execution Time analysis
  - Data rates

Generation and Processing of Safety Obligations

riacs Research Institute for Advanced Computer Science

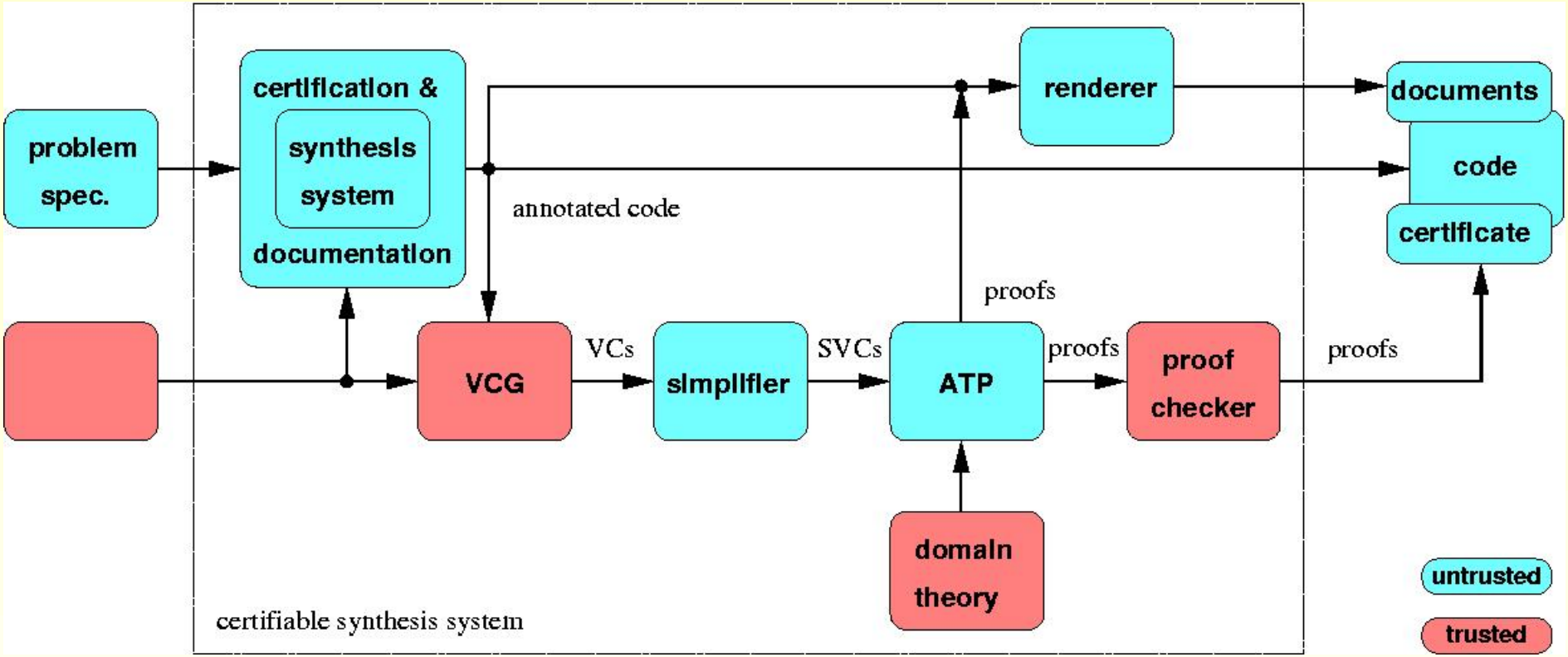
- annotation of program with pre-/post-conditions
- generation of logic formulas
- formal proof of these formulas

Example: (variable initialization)

```
for( i=0; i< 10; i++){
    x[i] = 5;
}
```

- post-condition: $x(0:9)$ is initialized
- loop invariant: if $x(0:i-1)$ is initialized, then $x(i)$ is initialized
- proof obligation has to demonstrate that from code and invariants, the post-condition can be infered.

Certifiable Program Generation: Architecture



- only few and small trusted components
- easy combination with Proof Carrying Code (PCC) to produce tamper-proof code

Conclusions

tool-supported framework for the reliable development and deployment of secure communication software

- supports entire life-cycle from specification and analysis to tamperproof code
- basic technologies already developed
  - Protocol analysis/verification
  - Scenario to Statechart
  - Automatic Code Certification
- areas for collaboration
  - generation of testcases
  - software process
  - protocol optimization
- two project phases:
  - Requirements analysis, inital architecture and case study
  - tool maturation