CS6480:
Systems and Formal Methods

Robbert van Renesse

Cornell University

Course Overview

Course Outline

- Some lectures by me
  - specification, Hoare logic, Dafny tutorial, refinement
- Paper reading by us
- Additional lectures by you
- Brand new course: shared learning experience for all of us
- Research projects
- Assignments
- Course remains under construction

Topics

- How to specify systems
- How to verify systems (refinement, “simulation”, Hoare logic)
- Survey verified systems
- Survey systems for proving and model checking

What is formal verification?

- Does software correctly implement a specification?
- Does software have desired properties (safety, liveness, other)?
- Is a particular optimization correct (equivalence, bi-simulation)?

Formal tools are used to check the above

Three parts to formal verification

- Soundness
  - If the formal verifier reports no bug, then the system does not fail
- Completeness
  - If the formal verifier reports a bug, then the system can fail
- Termination
  - The formal verifier terminates

Two types of formal verifiers

- Provers
  - Reason based on axioms and rules of inference
  - Automatic proof checking
    - but proof creation can be at least partly manual
- Model checkers
  - Manually create a model
  - Automatically explore the state space of the model

Why formally verify software systems?

- Modern software is very large (and thus hard to understand fully)
  - A car model may have over 100M lines of code
- NIST: software bugs cost $60B annually
- Vulnerable software in
  - Safety-critical systems (transportation etc.)
  - Privacy-critical systems (healthcare, etc.)
  - Money-critical systems (banking, etc.)
- Finding errors early may decrease development cost
- May make certain requirements *possible*

Testing or pen-and-paper verification may not suffice

Why not formally verify systems?

- Increases time-to-market
- May provide a false sense of safety
  - Verification validates an abstraction (or model) of a system, not the actual system
    - Finding the right abstraction level is a challenge
  - Specification may have bugs in it
  - May have missing requirements
  - May make inappropriate assumptions
  - Not all properties may have been checked
- May decrease safety
  - Verified systems may be prone to over-simplification
- May slow down adding new features
  - Or perhaps it'll help?
- Is too difficult in many cases

First few weeks

- Specification
- Hoare Logic
- Dafny
- Refinement

Textbook?

- Leslie Lamport – Specifying Systems
  - Available on-line at https://lamport.azurewebsites.net/tla/book-02-08-08.pdf
- More TBD

After that: your turn

- Give a presentation on
  - Some systems topic related to verification
  - Some verification tool or survey of tools

Possible Systems to Present

Verification and

- Operating Systems
- File Systems
- Networks
- Distributed Systems
- Concurrent Systems
- Secure Systems

Projects on Verified Operating Systems

- “Safe Kernel Extensions Without Run-Time Checking”, George Necula et al. (CMU), OSDI 1996
- “Comprehensive Formal Verification of an OS Microkernel” (seL4), Gerwin Klein et al. (NICTA), TOCS 2014
- “Safe to the Last Instruction: Automated Verification of a Type-Safe Operating System” (Verve), Jean Yang et al. (MSR), PLDI 2010
- “CertiKOS: An extensible architecture for building certified concurrent OS kernels”, Ronghui Gu et al. (Yale), OSDI 2016
- “Hyperkernel: Push-Button Verification of an OS Kernel”, Luke Nelson et al. (UW), SOSP 2017

Projects on Verified File Systems

- “Using Crash Hoare Logic for Certifying the FSCQ File System”, Haogang Chen et al. (MIT), SOSP 2015
- “Push-Button Verification of File Systems via Crash Refinement”, Helgi Sigurbjarnarson et al. (UW), OSDI 2016
- Cogent: “Verifying High-Assurance File System Implementations”, Sidney Amani et al. (NICTA), ASPLOS 2016
- “Verifying a high-performance crash-safe file system using a tree specification”, Haogang Chen et al. (MIT), SOSP 2017
- “Using Concurrent Relational Logic with Helpers for Verifying the AtomFS File System”, Mo Zou et al. (SJTU), SOSP 2019

Projects on Verified Networks

- "NetKAT: semantic foundations for networks", Carolyn Anderson et al. (Cornell), POPL 2014
- "Efficient Synthesis of Network Updates", Jedidiah McClurg et al. (CU Boulder, Cornell), PLDI 2015
- "A General Approach to Network Configuration Verification", Ryan Beckett et al. (Princeton), SIGCOMM 2017
- "Correct by Construction Networks Using Stepwise Refinement", Leonid Ryzhyk et al. (*), NSDI 2017
- "p4v: Practical Verification for Programmable Data Planes", Jed Liu et al. (*), SIGCOMM 2018
- "Verifying Software Network Functions with No Verification Expertise", Arseniy Zaostrovnykh et al. (EPFL), SOSP 2019

Projects on Verified Distributed Systems

- “Developing Correctly Replicated Databases Using Formal Tools”, Vincent Rahli et al. (Cornell), DSN 2014
- “IronFleet: Proving Practical Distributed Systems Correct”, Chris Hawblitzel et al. (MSR), SOSP 2015
- “Verdi: A Framework for Implementing and Formally Verifying Distributed Systems”, James R. Wilcox et al. (UW), PLDI 2015
- “How Amazon Web Services Uses Formal Methods”, Chris Newcombe et al. (Amazon), Comm. ACM 58(4), 2015
- “Model Checking at Scale: Automated Air Traffic Control Design Space Exploration”, Marco Gario et al. (JPL), CAV 2016
- “Grapple: A Graph System for Static Finite-State Property Checking of Large-Scale Systems Code”, Zhiqiang Zuo et al. (Nanjing U., UCLA). Eurosys 2019

Projects on Verified Concurrent Systems

- “GPS: Navigating Weak Memory with Ghosts, Protocols, and Separation”, Aaron Turon et al. (MPI-SWS), OOPSLA 2014
- “Automated and modular refinement reasoning for concurrent programs”, Chris Hawblitzel et al (MSR)., CAV 2015
- “Verifying Read-Copy-Update in a Logic for Weak Memory”, Joseph Tassarotti et al. (MPI-SWS, CMU), PLDI 2015
- “Proving the correct execution of concurrent services in zero-knowledge”, Srinath Setty et al. (MSR), OSDI 2018
- “Verifying Concurrent, Crash-safe Systems with Perennial”, Tej Chajed et al. (MIT), SOSP 2019

Projects on Verified Secure Systems

- “RockSalt: Better, Faster, Stronger SFI for the x86”, Greg Morrisett et al. (Harvard), PLDI 2012
- “Verifying Security Invariants in ExpressOS”, Haohui Mai et al. (UIUC), ASPLOS 2013
- “Implementing TLS with Verified Cryptographic Security”, Karthikeyan Bhargavan et al. (INRIA, MSR), Oakland 2013
- “Ironclad Apps: End-to-End Security via Automated Full-System Verification”, Chris Hawblitzel et al. (MSR, Cornell, …), OSDI 2014
- “Proving confidentiality in a file system using DiskSec”, Atalay Ileri et al. (MIT), OSDI 2018

Possible Presentations on Provers and Model Checkers

- NuPrl,
- TLA+
- ACL2
- Coq
- Dafny
- Ivy
- Chalice
- Isabelle/HOL
- Verdi
- Z3
- Boogie
- SPIN
- MaceMC, MoDist
- …

Your Participation

- Read all assigned chapters/papers and participate in discussions
- There will be ”programming” assignments
- Present survey on some class of systems or a tutorial on some technique or tool for formally verifying systems
  - E.g., verifying concurrent systems, modular verification, …
  - May become standard part of future version of this course
- Do a non-trivial formal verification task
  - Verify some ”system” (possibly part of your own research project)
  - Or develop some tool for system verification

First Assignment

- Read Chapters 1-4 from Specifying Systems
- Create a TLA+ spec that generates all and only prime numbers in order starting at 2
  - Desired behavior: $[p = 2] \rightarrow [p = 3] \rightarrow [p = 5] \rightarrow \ldots$
- Challenge: create a TLA+ spec for distributed consensus
  - Agreement: if two processes decide, they decide the same value
  - Validity: if a process decides a value, the value has been proposed by some process
  - Hint: *specify*, not *implement*
- Think about what you'd like to prepare and present

Specifying Systems (using TLA+)

Based on Leslie Lamport’s book “Specifying Systems”

Definition: *State*

- Definition: A *state* is an assignment of values to (*all*) variables
- TLA+ notation: $[var_1 = value_1, var_2 = value_2, \cdots]$
  - Meaning: a state in which var_1 has value $value_1, \cdots$
  - Order is immaterial
- Example: $[hr = 3]$
  - Meaning: a state in which $hr = 3$
    - The values of other variables are not specified
  - There can be many infinitely many states in which $hr = 3$
    - e.g. $[hr = 3.\ temp = 62]$, $[hr = 3.\ temp = 68]$, …
  - *Models* perhaps the hour hand being 3 on some hour clock HC

Definition: *Behavior*

- Definition 1: A *behavior* is a function of time to state

Computer systems can be thought of as executing in steps, so

- Definition 2: A *behavior* is a sequence of states
- Notation: $state_1 \rightarrow state_2 \rightarrow state_3 \rightarrow \cdots$
- Example: $[hr = 11] \rightarrow [hr = 12] \rightarrow [hr = 1]$

Definition: *Step*

- Definition: A *step* consists of two consecutive states in a behavior
- aka *transition*
- Notation: $state_1 \rightarrow state_2$
- Example: $[hr = 3] \rightarrow [hr = 4]$

Definition: *Specification*

- A *specification* is a set of all possible behaviors
- Consists of two parts
    1. Set of all possible *initial states*
    2. A "*next-state*" relation that describes the ways a state may change in a step
        - i.e., the set of all possible pairs of states

Set of Initial States

- Example: HCini $\triangleq hr \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$
    - Or, informally, HCini $\triangleq hr \in \{1, \cdots, 12\}$
    - HCini is simply a name given to the predicate
- A set of states can often be succinctly described by a predicate
    - Example: HCini $\triangleq hr \in \mathbb{N} \land 1 \leq hr \land hr \leq 12$
- Note again that these describe not 12 but an infinite set of states

Definition: *Next-State Relation*

- A *next-state relation* is a relation between pairs of successive states
    - $\{(state_1^{pre}, state_1^{post}), (state_2^{pre}, state_2^{post}), \cdots\}$
- Example:
    - HCnxt $\triangleq$ $\{ ([hr = 11], [hr = 12]), ([hr = 12], [hr = 1]), \cdots \}$

Definition: *Action*

- A next-state relation can often be more succinctly described by a predicate
- Definition 1: an *action* is a predicate over a pair of states
- Example: HCnxt $\triangleq hr' = hr \ \% \ 12 + 1$ (% is the “modulo” operator)
  - or, $\text{HCnxt}_2 \triangleq hr' = \text{IF } hr = 12 \text{ THEN } 1 \text{ ELSE } hr + 1$
  - But note that $\text{HCnxt}_2 \not\equiv$ HCnxt
- hr' is the value of hr in the new state; hr is the value in the old state
- Definition 2: an *action* is a predicate containing both primed and unprimed variables
- An ordinary predicate and does not have to be of the form “*x*’ = f(*x*)”
  - Example: HCnxt $\triangleq hr' - hr = 1 \bmod 12$

Steps versus *Actions* versus *Execution*

- A *step* is a pair of states
- An *action* $\mathcal{A}$ is a predicate over steps
- We call a step that satisfies $\mathcal{A}$ an $\mathcal{A}$ step
  - Example: a step that satisfies HCnxt is an *HCnxt step*
- We sometimes informally say that HCnxt is *executed*

Example specification: hour clock (in complete isolation)

Module HourClock

- Variable hr
- HCini $\triangleq hr \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$
- HCnxt $\triangleq hr' = hr \bmod 12 + 1$
- HC $\triangleq$ HCini $\land$ $\square$HCnxt

Temporal logic formula $\square$P means that predicate P *always* holds
(thus HCnxt is *invariant* in HC)

Note:

1. All three statements are definitions, but the last one happens to constitute the full specification of the hour clock)
2. There is no conventional naming in TLA+, so pick names that are descriptive

Definition: *Stuttering steps*

- Clocks are usually part of a larger system
- They have more state variables than just the hour hand of the clock
- State changes must allow for hour hand not to change
  - Example: $[hr = 3.\ \mathrm{t}emp = 62] \rightarrow [hr = 3.\ \mathrm{t}emp = 63]$
- This is called a *stuttering step* of the clock
  - i.e., $hr' = hr$

Final specification: hardware clock

Module HourClock

- Variable hr
- HCini $\triangleq hr \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$
- HCnxt $\triangleq hr' = hr \bmod 12 + 1$
- HC $\triangleq$ HCini $\land \square$(HCnxt $\lor$ $(hr' = hr)$)

The latter can be abbreviated using the following TLA+ notation

$$\text{HC} \triangleq \text{HCini} \land \square[\text{HCnxt}]_{hr}$$

($[\text{HCnxt}]_{hr}$ is pronounced ”square HCnxt sub hr”)

Definition: *theorem*

- Definition: in TLA+, a *theorem* of a specification is a temporal formula that holds over every behavior of the specification
- Example: HC ⇒ ☐ $hr \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$
  - That is, HC ⇒ ☐ HCini
- Proof: by induction on #steps

A note on variables and types

- Variables in TLA+ are untyped
- However, if one can prove SPEC $\Rightarrow \Box\, v \in S$ for some variable v and constant set S, then one can call S the type of v in SPEC
- Example: the type of hr in HC is $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$
- It is useful to specify the types in a specification
- Example: HCtypeInvariant $\triangleq hr \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$
- Note, in this case HCtypeInvariant $\equiv$ HCini

A note on states and behaviors

- Recall
  - A state is an assignment of values to variables
  - A behavior is a sequence of states
- Thus
  - $[hr = 13]$ is still a state, and so is $[hr = "blue"]$
  - $[hr = 4] \rightarrow [hr = 3]$ is still a behavior
- However, they are not in specification HC

HourClock Specification in Dafny

```
class HourClock {
    var hr: nat

    method nxt()
        modifies this
        ensures hr == old(hr) % 12 + 1

    constructor(ihr: nat)
        requires 1 <= ihr <= 12
    {
        hr := ihr;
    }
}
```

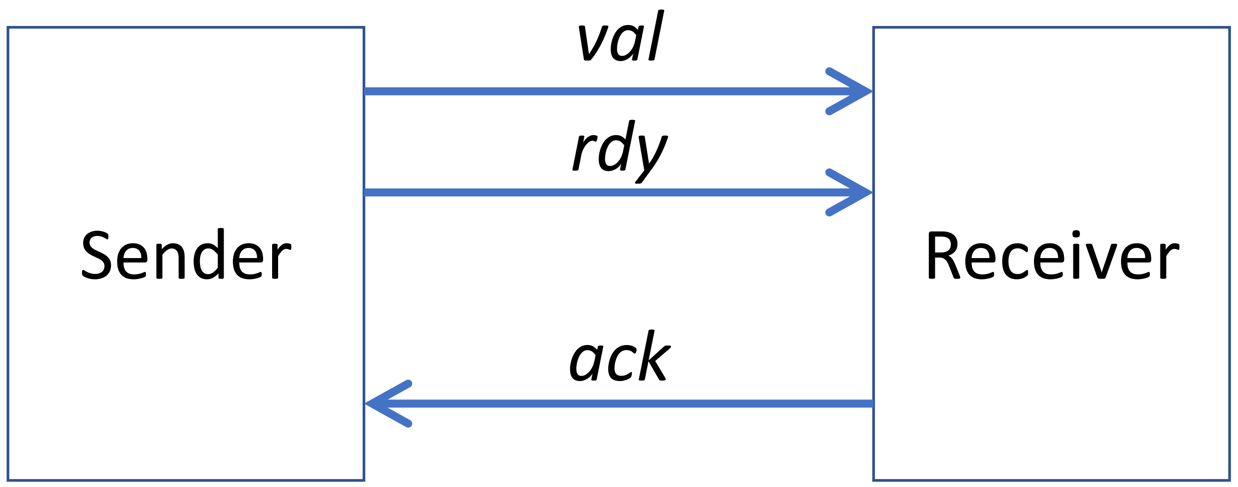
HourClock Implementation in Dafny

```
class {:autocontracts} HourClock {
    var hr: nat

    predicate Valid() { 1 <= hr <= 12 } // class invariant

    method nxt()
        modifies this
        ensures hr == old(hr) % 12 + 1
    {
        hr := hr % 12 + 1;
    }
}
```

Asynchronous FIFO Channel Specification



$$
\begin{aligned}
Send \triangleq\ & \wedge\ rdy = ack \\
& \wedge\ val' \in Data \\
& \wedge\ rdy' = 1 - rdy \\
& \wedge\ ack' = ack
\end{aligned}
\qquad
\begin{aligned}
Rcv \triangleq\ & \wedge\ rdy \neq ack \\
& \wedge\ ack' = 1 - ack \\
& \wedge\ val' = val \\
& \wedge\ rdy' = rdy
\end{aligned}
$$

Asynchronous FIFO Channel Specification

$$TypeInvariant \triangleq \begin{array}{l} \land\ val \in Data \\ \land\ rdy \in \{0, 1\} \\ \land\ ack \in \{0, 1\} \end{array}$$

$$Init \triangleq \begin{array}{l} \land\ val \in Data \\ \land\ rdy \in \{0, 1\} \\ \land\ ack = rdy \end{array}$$

$$Send \triangleq \begin{array}{l} \land\ rdy = ack \\ \land\ val' \in Data \\ \land\ rdy' = 1 - rdy \\ \land\ ack' = ack \end{array}$$

$$Rcv \triangleq \begin{array}{l} \land\ rdy \neq ack \\ \land\ ack' = 1 - ack \\ \land\ val' = val \\ \land\ rdy' = rdy \end{array}$$

$$Next \triangleq Send \lor Recv$$

$$Spec \triangleq \mathrm{Init} \land \Box[\mathrm{Next}]_{\langle rdy, ack, val \rangle}$$

Asynchronous FIFO Channel Specification

introducing *operators with arguments*

$$\begin{aligned}
\mathit{Send} \triangleq\ & \land\ \mathit{rdy} = \mathit{ack} \\
& \land\ \mathit{val}' \in \mathit{Data} \\
& \land\ \mathit{rdy}' = 1 - \mathit{rdy} \\
& \land\ \mathit{ack}' = \mathit{ack}
\end{aligned}
\quad \Rightarrow \quad
\begin{aligned}
\mathit{Send}(d) \triangleq\ & \land\ \mathit{rdy} = \mathit{ack} \\
& \land\ \mathit{val}' = d \\
& \land\ \mathit{rdy}' = 1 - \mathit{rdy} \\
& \land\ \mathit{ack}' = \mathit{ack}
\end{aligned}$$

$$\begin{aligned}
\mathit{Next} \triangleq\ & \lor\ \mathit{Send} \\
& \lor\ \mathit{Recv}
\end{aligned}
\quad \Rightarrow \quad
\begin{aligned}
\mathit{Next} \triangleq\ & \lor\ \exists d \in \mathit{Data} : \mathit{Send}(d) \\
& \lor\ \mathit{Recv}
\end{aligned}$$

Asynchronous FIFO Channel Specification
introducing *records*

$TypeInvariant \triangleq chan \in [val: Data, rdy: \{0,1\}, ack: \{0,1\}]$

$Init \triangleq chan.val \in Data \land chan.rdy \in \{0, 1\} \land chan.ack = chan.rdy$

$Send(d) \triangleq chan.rdy = chan.ack \land chan' =$
$[\, val \mapsto d, rdy \mapsto 1 - chan.rdy, ack \mapsto chan.ack \,]$

$Recv \triangleq chan.rdy \neq chan.ack \land chan' =$
$[\, val \mapsto chan.val, rdy \mapsto chan.rdy, ack \mapsto 1 - chan.ack \,]$

$Next \triangleq \exists d \in Data: Send(d) \lor Recv$

$Spec \triangleq Init \land \Box[Next]_{chan}$

First Assignment

- Read Chapters 1-4 from Specifying Systems
- Create a TLA+ spec that generates all and only prime numbers in order starting at 2
    - Desired behavior: $[p = 2] \rightarrow [p = 3] \rightarrow [p = 5] \rightarrow \ldots$
- Challenge: create a TLA+ spec for distributed consensus
    - Agreement: if two processes decide, they decide the same value
    - Validity: if a process decides a value, the value has been proposed by some process
    - Hint: *specify*, not *implement*
- Think about what you'd like to prepare and present