UTP Semantics for Safety-Critical Java

University of York

hiJaC Workshop

15 November 2011

Outline

Outline

UTP

- *Unifying Theories of Programming*
- Verified Software Initiative
- who put the "T" in "VSTTE"?
  - Verified Software: Theories, Tools, and Experiments
- C. A. R. Hoare, He Jifeng: *Unifying Theories of Programming*, Prentice-Hall, 1998
- long-term research agenda in a nutshell:
  - researchers have proposed programming theories
  - practitioners have proposed pragmatic programming paradigms
  - how do we understand the relationship between all of these?
- for history, see
  - Eric C. R. Hehner: Retrospective and Prospective for Unifying Theories of Programming, *UTP 2006*: 1–17
- UTP gives three principal ways to study relationships
  1. by computational paradigm
  2. by level of abstraction
  3. by method of presentation

1. Computational Paradigms

- programming languages are numbered in the thousands
- general-purpose languages
  - C++, Java, Python, Perl, ...
  - Peter Landin: The Next 700 Programming Languages
- domain-specific programming languages
  - HTML, Logo, Verilog and VHDL, Mathematica, SQL, regular expressions, YACC grammars, EMF, ...
- group them together by computational paradigm
  - structured, object-oriented, functional, logical, ...
- identify common concepts
- deal separately with additions and variations
- two fundamental scientific principles used in UTP
  1. simplicity of presentation
  2. separation of concerns

2. Abstraction

- orthogonal to classification by paradigm
- individual paradigm treated at different levels of abstraction
  1. *highest:* requirements capture and analysis
  2. *high:* architectural description, from requirements to solution
  3. *intermediate:* component definition, contracts for interfaces
  4. *low:* programming language, full behavioural description
  5. *lowest:* platform specific, technology of implementation
- UTP offers ways of linking these elvels
- *refinement calculi* translate between levels
- guarantee of correctness from requirements to code

3. Presentation

- classify by the method of presentation of language definition
- three scientific methods
  1. *denotational semantics*
     - each syntactic phrase is given a mathematical denotation
     - specification is just a set of denotations
     - simple correctness criterion
  2. *algebraic semantics*
     - no direct meaning for the language
     - equalities between different programs with the same meaning
     - most useful for engineers
  3. *operational semantics*
     - programs defined by how they execute
     - abstract mathematical machine
     - guide for compilation, debugging, testing, . . .
- comprehensive account of programming theory needs all three

UTP

- UTP uses all three ways of classifying programming theories
- *example:* process algebras ACP, CCS, and CSP
- first description at the most abstract level
- no regard to practical implementation programming language
- study how different presentations affect the language
  - *algebraic:* for ACP
  - *operational:* for CCS
  - *denotational:* for CSP
- study differences
- study mutual embeddings
- derive each by mathematical definition, calculation, and proof

UTP Research Agenda

- ultimate goal:
  - cover all the interesting paradigms of computing
  - declarative and procedural, hardware and software.
- theoretical foundation for software engineering
- study the variety of existing programming languages
- identify the major components of programming languages
- select theories for new, perhaps special-purpose languages
- *the theory supermarket*
- shop for exactly those features you need
- you can be confident that the theories plug-and-play

UTP Theory
= Alphabet + Signature + Healthiness Conditions

UTP Example: Alphabet

- *alphabet:* the set of observational variables
- *example:* simple theory to model the behaviour of a gas with regard to varying temperature and pressure
- *Boyle's law*
  "for a fixed amount of an ideal gas kept at a fixed temperature k, p (pressure) and V (volume) are inversely proportional (while one doubles, the other halves)"
- alphabet: three mathematical variables: k, p, and V
- model observations correspond to real-world observations
- the model-based agenda:
  - the variables k, p, and V are *shared* with the real world
- we must specify the alphabet for every predicate we use
- suppose P is a predicate, then $\alpha(P)$ is its alphabet

UTP Example: Signature

- syntax used to denote objects of the theory
- requirement: constant temperature
- to animate Boyle's law we need two operations:
  1. change the pressure
  2. change the volume
- this is the *signature*

UTP Example: Healthiness Conditions

- healthiness conditions:
  - a way of determining membership of a theory
- we are interested only in gases that obey Boyle's law
- this states that $p * V = k$ must be *invariant*
- healthiness determines the correct states of the system
- we need both static and dynamic invariants
- $p * V = k$ is a *static* invariant: it applies to a *state*
- but we also we require k to be constant
- start in the state (k, p, V), where $p * V = k$
- transit to the state (k', p', V'), where $p' * V' = k'$
- we must have that $k' = k$
- this is a *dynamic* invariant: it applies to a *relation*

Healthiness Conditions I

- some healthiness conditions can be defined using functions
- suppose $\alpha(\phi) = \{p, V, k\}$
- define $\mathbf{B}(\phi) = (\exists\, k \bullet \phi) \wedge (k = p * V)$
- regardless of whether ϕ is healthy or not, $\mathbf{B}(\phi)$ certainly is
- example:

$$\phi = (p = 10) \wedge (V = 5) \wedge (k = 100)$$

$$\begin{aligned}\mathbf{B}(\phi) &= (\exists\, k \bullet \phi) \wedge (k = p * V) \\ &= (\exists\, k \bullet (p = 10) \wedge (V = 5) \wedge (k = 100)) \wedge (k = p * V) \\ &= (p = 10) \wedge (V = 5) \wedge (k = p * V) \\ &= (p = 10) \wedge (V = 5) \wedge (k = 50)\end{aligned}$$

- notice that $\mathbf{B}(\mathbf{B}(\phi)) = \mathbf{B}(\phi)$
- *idempotence*: taking the medicine twice leaves you healthy
- this give us a simple test for healthiness: $\phi = \mathbf{B}(\phi)$
- fixed point of idempotent function

Healthiness Conditions II

- an unhealthy observation:

  $\phi = (p = 10) \land (V = 5) \land (k = 100)$

- another observation: pressure is between 10 and 20Pa

  $\psi = (p \in 10 \,..\, 20) \land (V = 5)$

- fact: $\phi \Rightarrow \psi$
- another fact: $\mathbf{B}(\phi) \Rightarrow \mathbf{B}(\psi)$
- this means that $\mathbf{B}$ is *monotonic*
- the best heathiness conditions are
  - *monotonic idempotent functions*
- very important mathematical properties
- complete lattices, Galois connections

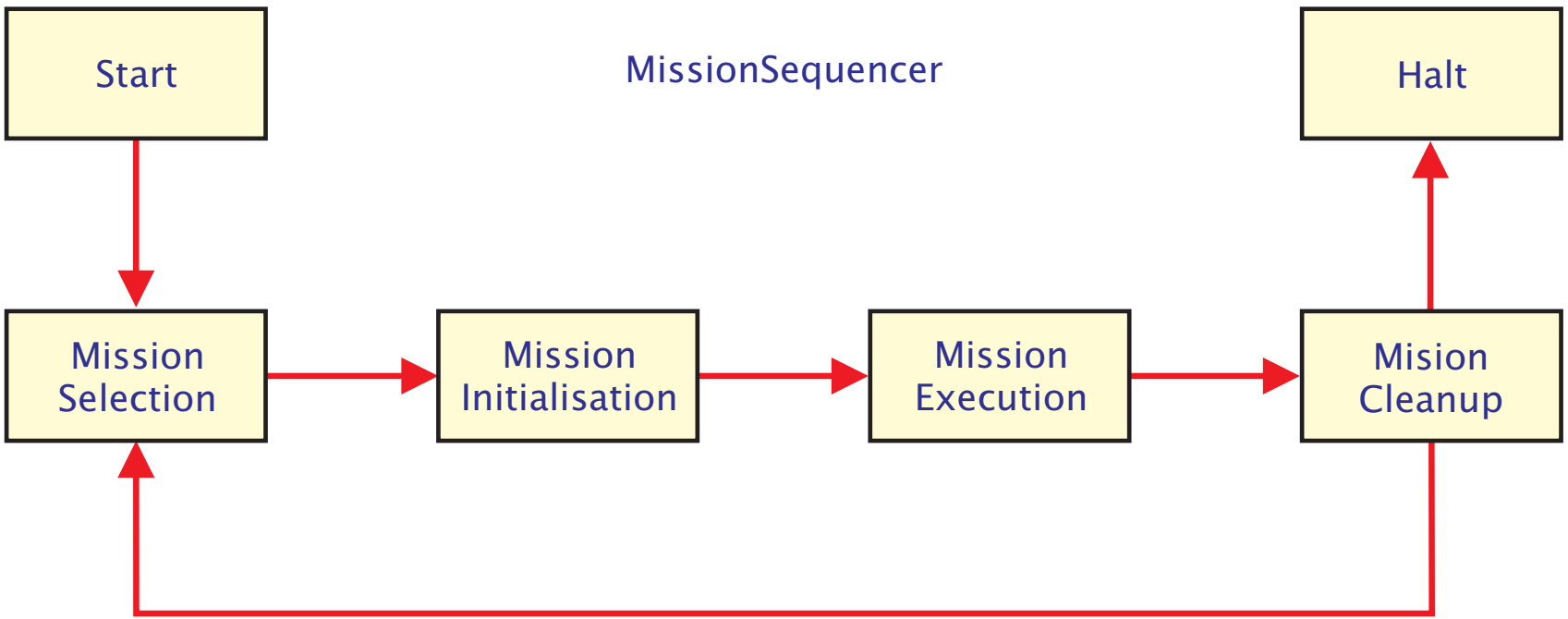
Outline

SCJ-Circus

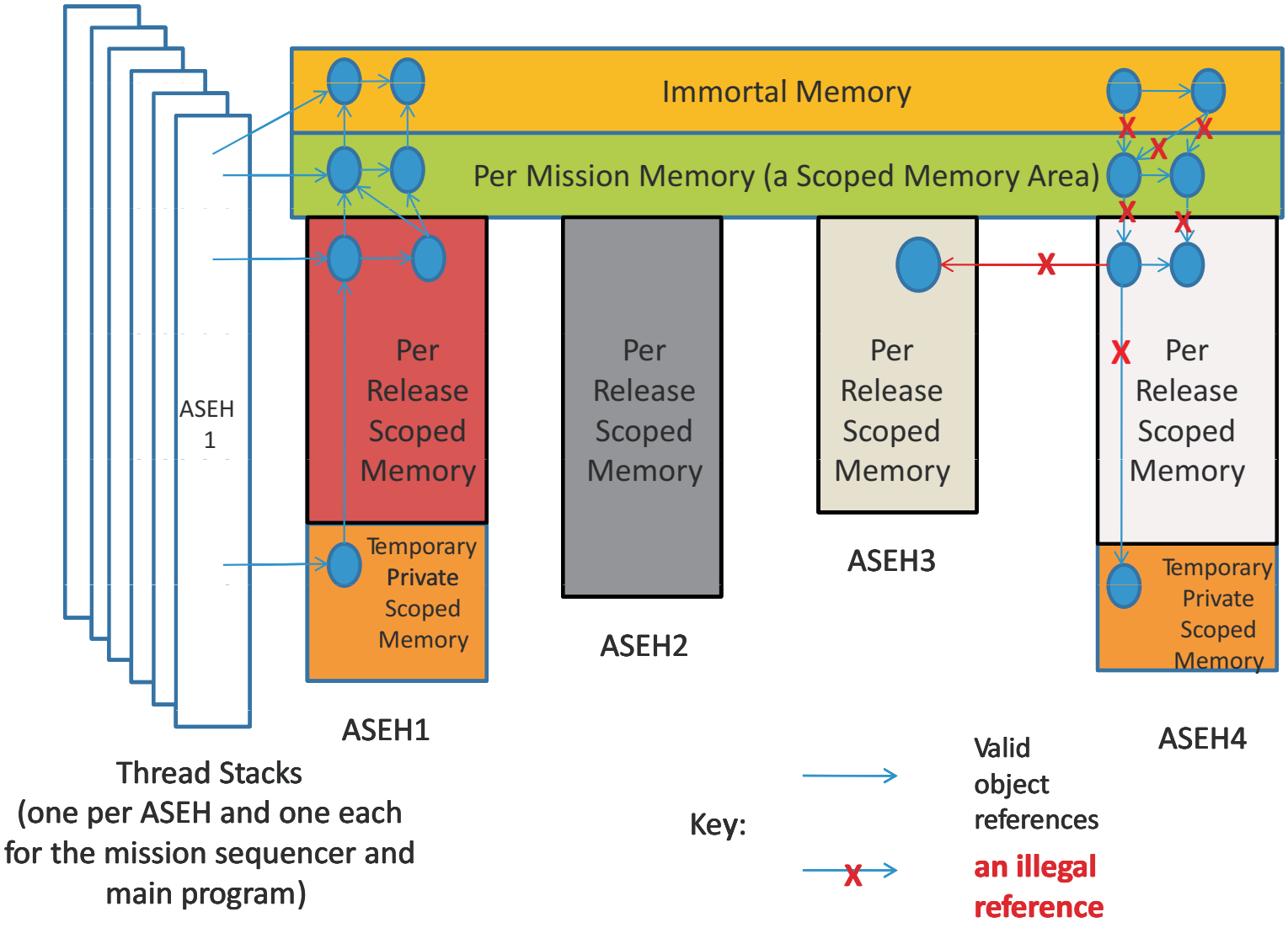
- built from the following items in the theory shopping-cart:
  1. *Circus* designs: nondeterministic imperative programming with specification statements (based on Z)
  2. *Circus* reactive processes: concurrency, communication, and shared variables (based on CSP)
  3. *OhCircus*: OO, with encapsulation, classes, and inheritance
  4. *CircusTime*: discrete real-time
  5. the SCJ memory model
- the UTP agenda is far from complete:
  - some of these theories need to be brought to maturity
  - some need to be linked together using Galois connections

Outline

Application structure



Scoped memory area



◂ SCJ Memory Areas

SCJ Memory Model

- newest *Circus* theory
- main goal: support disciplined dynamic memory management
- safety-critical systems usually forbid dynamic memory
  - manual techniques are error-prone (e.g., `malloc` in C)
  - automated garbage collection (Java) too complex to certify
- SCJ takes a different approach:
  - replace Java's garbage-collected heap-memory model by memory divided into *scoped* memories and *immortal* memory
- many, possibly nested, scoped-memory areas,
- single immortal memory
- rules used to determine legitimacy of reference assignment
- avoid dangling references
- rule violation is a runtime exception
- careful SCJ programmer must think where to create objects
- balance runtime exception-freedom against memory efficiency
- automated techniques needed to assist

SCJ Memory Model

- UTP model validates rules for static analysis techniques
- ensures absence of null references and illegal-assignment errors
- mission starts in initialisation phase
- objects may be allocated in mission or immortal memory
- no dynamic creation of ASEHs
- initialisation followed by execution phase
  - ASEHs are started
- initial ASEH memory area is scoped
  - entered when ASEH is released
  - exited when it completes
- all the area's objects are collected on exit
- no sharing with other ASEHs
- ASEH may enter into nested private memory areas

UTP SCJ Memory Model

- built from a theory of object references
- linked to a structural model of memory areas
- pointers and hierarchical addressing created by data types with recursive records
- three observational variables:
  1. A: set of hierarchical addresses
     - describes all the legal addresses that could be constructed
     - all non-empty sequences of labels
  2. V: partial function from addresses to values
     - maps addresses of primitive (non-object) attributes to values
     - $A \setminus \mathrm{dom}\ V$ describes acceptable addresses that yield objects (non-primitive values)
  3. S: equivalence relation on addresses
     - relates addresses that share a common location
- twelve healthiness conditions
- for example, A is prefix closed
- if $a.b.c$ is a valid address, then so is $a.b$, etc

Healthiness conditions

1. objects only ever *added* to immortal memory
2. all the references in the program stack are resident in the immortal memory
3. all the references in the immortal memory are resident in the immortal memory
4. all the references in the sequencer stack are resident in either the immortal or the mission memory
5. all the references in the mission memory are resident in either the immortal or the mission memory
6. the immortal, mission, per-release, and temporary private memory areas are all mutually disjoint

Outline

COMPASS

- Comprehensive Modelling for Advanced Systems of Systems
- EU FP7 project: October 2011 – September 2014
- Newcastle, York, Aarhus, UFPE, Bremen, Atego, Insiel, B&O
- advance Systems-of-Systems (SoS) Engineering
  1. develop the first formal modelling language specifically for SoS
  2. provide advanced model-based methods and tools
  3. evaluate using benchmarks and industrial case studies
     - accident response
     - audio/video/home-automation ecosystem
  4. link to industrial architectural description frameworks
  5. build tools for model-checking, proof, and simulation
  6. develop an open platform to integrate existing and new tools
- help to plan and roadmap the EU's future research agenda
- COMPASS Modelling Language (CML) is UTP-based
  - VDM, *Circus*, *CircusTime*, *OhCircus*, *Circus*-DF, *TravellingCircus*
  - tools based on Overture, FDR2, Isabelle/HOL, Z3

Outline

Conclusions and future work

- first formalisation of the SCJ memory model
- proof that SCJ is *memory safe*
- formalisation essential for reasoning by refinement

Future work

- connections to other theories
- further extensions to *Circus*
- refinement laws and strategies