From Programs to Systems – The Evolution of IST

Scientific Computing
– Defence Applications

Foundations - Alan Turing, Kurt Gödel

Convergence between Computing and Telecommunications

Embedded Systems: Computing + Physicality

Information Systems: Commercial Applications

The Internet of Things: Convergence between Embedded Systems and the Internet

Graphic Interfaces, Mouse

95% of chips are embedded

Evolution driven by exponential progress in technology and explosion of applications


Cloud Computing

Multi-core Systems

WEB – Information Society
From Programs to Systems –– Foundations

- I/O values
- Terminating
- Deterministic
- Platform-independent behavior
- Theory of computation
- I/O streams of values
- Non-terminating
- Non-deterministic
- Platform-dependent behavior
- No unified theoretical framework
From Programs to Systems

SW+HW

Real-Time Control

Resources
Buildings
Transport
Communications
Health
System Design – New Trends

Future computing systems will break with traditional computing systems such as desktop computers and servers in various ways. It is hard to jointly meet technical requirements such as:

- **Reactivity**: responding within known and guaranteed delay
  Ex: flight controller

- **Autonomy**: provide continuous service without human intervention
  Ex: no manual start, optimal power management

- **Safety & Security**: resilience in any case of failures and attacks

- **Scalability**: at runtime or evolutionary growth (linear performance increase with resources)
  Ex: reconfiguration, scalable services

...and also take into account economic requirements for optimal cost/quality

These trends challenge our capacity to build systems of guaranteed trustworthiness at acceptable costs
We master – at a high cost – two types of systems which are difficult to integrate:

- Trustworthy systems of low complexity
  - Flight controller, smart card
- Complex « best effort » systems
  - Telecommunication systems, web-based applications

We need:

- Affordable trustworthy systems
  - Ex: transport, health, energy management
- Successful integration of mixed-criticality systems of systems
  - Internet of Things
  - Intelligent Transport Systems
  - Smart Grids
  - « Ambient Intelligence»
Trustworthiness

Breaking with Old Ideas

Rigorous System Design

- The VLSI Design Paradigm
- Principles for Rigorous System Design
- Component-based Design
- Semantic Coherency
- Correctness-by-construction
- Putting RiSD into Practice

Discussion
Trustworthiness – The Concept

Need for trust is gaining awareness

- Software vendors such as Microsoft and Sun Microsystems primarily focus on security
- Many argue that trustworthiness is a much broader concept that should be viewed as a holistic system property: “… software trustworthiness covers correctness, safety, availability, reliability, performance, security and privacy” from “Software 2015: A National Software Strategy to ensure U.S. Security and Competitiveness”
- Confidence in software is based on both the artifact itself and on the humans who deliver it. Many approaches focus on computational trust models in different domains like sociology and psychology.

We need a rigorous technical definition of trustworthiness identifying key fundamental issues to be addressed and methods for enhancing trustworthiness of existing and future systems.

In absence of adequate technical frameworks, the study of social and psychological aspects of trust is premature and has limited added value.
Trustworthiness – Definition

Assurance that the designed system can be trusted that it will perform as expected despite:

- HW failures
- SW Design Errors
- Physical Environment Disturbance
- Erroneous or Malevolent Action

Trustworthiness is a global property that must be addressed throughout the computing environment in the design process!
System design is the process leading to an implementation meeting given requirements.

Different from pure SW or pure HW design!

- Requirements
- Application SW
- System Model
- Execution Platform Models
- Physical and Human Environment Model
- Execution Platform
- DSE
Trustworthy but also Productive Design

Efficiency of the design process

Skills

Components

Tools
Trustworthy but also Optimized Design

- Optimal use of the available resources of the execution platform through design space exploration to resolve choices such as
  - reducing parallelism (through mapping on the same processor)
  - reducing non-determinism (through scheduling)
  - fixing parameters (quality, frequency, voltage)

- Optimized design requires the use of adequate programming models
  Don’t program multimedia applications in plain C!
- Trustworthiness

- Breaking with Old Ideas

- Rigorous System Design
  - The VLSI Design Paradigm
  - Principles for Rigorous System Design
  - Component-based Design
  - Semantic Coherency
  - Correctness-by-construction
  - Putting RiSD into Practice

- Discussion
Breaking with Old Ideas – The V-model

Breaking with Old Ideas – The V-model

The V-model of traditional Systems Engineering process

- assumes that system development is top-down from a set of requirements. But
  - systems are never designed from scratch; they are built by modifying incrementally existing systems and component reuse
  - requirements cannot be completely formalized e.g. security properties, QoS properties
  - there are no techniques for breaking down global requirements into local requirements for system constituents

- separates development and validation in two distinct interacting flows by focusing on correctness-by-checking e.g. testing, simulation, verification. But correctness-by-checking
  - is a relative judgment “Are we building the system right?”
  - would be an answer to the question “Are we building the right system?” if trustworthiness properties could be formalized and checked efficiently
  - is of limited help for achieving optimized solutions characterized by quantitative properties
Breaking with Old Ideas –- Top-down Development

Top-down development and validation
Requirements should be written in a formal language easy to understand and use by engineers. But
- for some types of requirements we do not have adequate languages e.g. to describe security properties,
- even if some languages are expressive enough, their effective use by engineers seems to be problematic e.g. SUGAR

Requirements should be sound. But
- checking soundness, even for decidable requirements specification languages, is problematic due to intrinsic complexity of decision algorithms

Requirements should be complete that is they tightly characterize the system’s behavior. But
- there is no criterion characterizing completeness for declarative languages – writing requirements may be an endless game!
Models should be

- faithful e.g. whatever property is satisfied for the model holds for the real system
- generated automatically from system descriptions

Furthermore, establishing correctness of some critical requirements and QoS requirements needs models which represent the behavior of an application software executing on a given hardware platform.

Obstacles to overcome:

- Interaction between application software and the underlying execution platform (hardware-dependent software and hardware) is intricate and hard to understand
- Hardware execution platforms have “unpredictable” behavior
  - Variability due to manufacturing errors or aging
  - Varying execution times due to layering, caches, speculative execution
- Detailed models are intrinsically complex
Checking correctness with respect to given requirements

- Ad hoc techniques e.g. by simulation allow error detection only
- Testing of real or virtual prototypes
- Formal verification allows exhaustive validation, but
  - Monolithic verification is limited to small or medium size systems and to specific of properties
  - Attempts to apply compositional verification to composite systems and thus cope with complexity of monolithic verification, have failed e.g. assume/guarantee techniques
- Trustworthiness
- Breaking with Old Ideas
- Rigorous System Design
  - The VLSI Design Paradigm
  - Principles for Rigorous System Design
  - Component-based Design
  - Semantic Coherency
  - Correctness-by-construction
  - Putting RiSD into Practice
- Discussion
The reasons of success

- **Powerful abstractions**
  
  transistor level $\rightarrow$ gate level $\rightarrow$ RTL level $\rightarrow$ IP block level

- **Component-based design**
  
  Limited and well-defined number of components

- **Unified Execution model**
  
  Synchronous execution

- **Synthesis**
  
  Scalable methods and tools

- **Correctness-by-construction**
  
  Extensive use of Architectures
The LSI Design Paradigm – Transposition to System Design

Difficulties to be overcome

- **Powerful abstractions**
  We need abstractions relating an application SW to the its implementation on a given execution platform

- **Component-based design**
  We need a Common Component Model for mixed HW/SW systems

- **Unified Execution model**
  We need a model encompassing synchronous and asynchronous execution

- **Synthesis**
  Replace synthesis by scalable model transformations

- **Correctness-by-construction**
  - We need theory for inferring global properties of a system from properties of its constituents
  - We need an all encompassing concept of architecture relating various architecture styles e.g. Data-flow, Event-driven, Time-Triggered, Distributed
- Trustworthiness
- Breaking with Old Ideas
- Rigorous System Design
  - The VLSI Design Paradigm
  - Principles for Rigorous System Design
    - Component-based Design
    - Semantic Coherency
    - Correctness-by-construction
    - Putting RiSD into Practice
- Discussion
Rigorous System Design – Principles

WHAT
- Requirements
- Application SW
- System Model

HOW
- Execution Platform
- Physical and Human Environment Model
- Execution Platform Models

Separation of concerns
Rigorous System Design – Principles

Component-based Design

Physical and Human Environment Model

Execution Platform Models

Execution Platform

DSE
Rigorous System Design – Principles

Semantically Coherent Model-based Design

Physical and Human Environment Model

Execution Platform Models

Execution Platform

HOST LANGUAGE
Rigorous System Design – Principles

Correct-by-construction Design

Physical and Human Environment Model

Execution Platform Models

DSE

Execution Platform Models

HOST LANGUAGE

Pr1

Pr2

Pr3

VI

VI

VI

Execution Platform
Component-based Design

- Build complex systems by composing components confers numerous advantages such as productivity and correctness.

- Component composition orchestrates interactions between components. It lies at the heart of the parallel computing challenge.

- We need theory and methods for building faithful models for mixed SW/HW systems as the composition of heterogeneous components.
Component-based Design – Heterogeneity

Synchronous components (HW, Multimedia application SW)
- Execution is a sequence of non interruptible steps

Asynchronous components (General purpose application SW)
- No predefined execution step

Open problem: Theory for consistently composing synchronous and asynchronous components e.g. GALS
Component-based Design – Heterogeneity

Two basic protocols

- **Rendezvous**: atomic symmetric synchronization
- **Broadcast**: asymmetric synchronization triggered by a Sender

- Any interaction mechanism can be expressed as the hierarchically structured combination of rendezvous and broadcast
- Existing formalisms and theories are not expressive enough
  - variety of low-level coordination mechanisms including semaphores, monitors, message passing, function call
  - usually point-to-point interaction either by rendezvous or by broadcast
Component-based Design – Heterogeneity

Thread-based programming

Actor-based programming

Software Engineering

Systems Engineering
Component-based Design – Heterogeneity (Example)
Build a component $C$ satisfying a given property $P$, from
- $C_0$ a set of **atomic** components described by their behavior
- $GL = \{gl_1, …, gl_i, …\}$ a set of **glue operators** on components

- Glue operators are coordination mechanisms such as such as protocols, schedulers, buses
- We need a unified composition paradigm for describing and analyzing the coordination between components in terms of tangible, well-founded and organized concepts
Component-based Design – The Concept of Glue

Glue is a first class entity independent from behavior that can be decomposed and composed

1. Incrementality

2. Flattening
- Trustworthiness

- Breaking with Old Ideas

- Rigorous System Design
  - The VLSI Design Paradigm
  - Principles for Rigorous System Design
  - Component-based Design
  - Semantic Coherency
  - Correctness-by-construction
  - Putting RiSD into Practice

- Discussion
System designers deal with a large variety of languages, with different characteristics, each highlighting different dimensions of a system

Consequences:
- Using semantically unrelated formalisms e.g. for programming, HW description and simulation, breaks continuity of the design flow and jeopardizes its coherency
- System development is often decoupled from validation and evaluation.
Semantic Coherency – Host Language

To ensure global consistency of the design flow we need to express the semantics of the various languages in terms of an all encompassing **host language**.

- **DSL**
  - Data-flow
  - Synchronous
  - Event-driven
  - Asynchronous MP

- **Phys. Systems**
  - Mod. Langu.
  - Matlab
  - Modelica

- **HDL**
  - Verilog
  - SystemC
  - TLM
  - IP-XACT

- **Modeling Languages**
  - UML
  - SysML
  - AADL

**Host Language H**
- **Common Component Model**
- **Expressive**
- **Simple and Elegant**
Structured Operational Semantics for L is implemented by an Engine which cyclically executes a two-phase protocol:

1. Monitors components and determines enabled connections

2. Chooses one enabled connection and executes the corresponding interaction – the latter may modify the states of the involved components
Semantic Coherency – Host Language

SW written in a language L

Engine for L (SOS for L)

SW written in Host Language H

Engine for L written in H

Engine for H (SOS for H)

EMBEDDING
- Trustworthiness
- Breaking with Old Ideas
- Rigorous System Design
  - The VLSI Design Paradigm
  - Principles for Rigorous System Design
  - Component-based Design
  - Semantic Coherency
  - Correctness-by-construction
  - Putting RiSD into Practice
- Discussion
Correctness-by-construction – Architectures

System developers extensively use libraries of reference architectures ensuring both functional and non functional properties e.g.
- Fault-tolerant architectures
- Resource management and QoS control
- Time-triggered architectures
- Security architectures
- Adaptive Architectures

Architectures
- depict design principles, paradigms that can be understood by all, allow thinking on a higher plane and avoiding low-level mistakes
- are a means for ensuring global properties characterizing the coordination between components – correctness for free
- Using architectures is key to ensuring trustworthiness and optimization in networks, OS, middleware, HW devices etc.
- Architectures can be formalized as glue operators
Correctness-by-construction – Property Preservation

Deadlock-free Routing Protocol

Deadlock-free Components

Deadlock-free Routing Algorithm

Deadlock-free Composite Component
Correctness-by-construction – Property Enforcement

Architecture for Mutual Exclusion

Components
Automatic Implementation

Problem: Given an application software and an execution platform generate trustworthy and optimal implementations

Application Software is written in high level languages supporting abstractions such as

- Atomicity of primitives and interactions between components – in particular multiparty interaction
- A logical notion of time assuming zero-time actions and synchrony of execution with respect to the physical environment

The generated implementation should be

- functionally equivalent to the application software (correctness)
- generated automatically for a given mapping associating
  - Processes of the ASW → processors of the platform
  - Data of the ASW → memories of the platform
  - Interactions → execution paths or protocols
Correctness-by-construction – Composability

An architecture ensuring a given property can be obtained as the combination of a set of architectures ensuring basic properties.

For example, security architectures are obtained by composition of architectures ensuring

- Antivirus protection
- Intrusion Detection System, Intrusion Protection System
- Sampling
- Monitoring
- Watermarking
- Embedded cryptography
- Integrity checking

We need theory for combining basic architectures and their characteristic properties to obtain an architecture meeting a given global property.
Correctness-by-construction – Composability

Mutual Exclusion Protocol

Scheduling Algorithm

Mutual Exclusion Protocol

Scheduling Algorithm
- Trustworthiness
- Breaking with Old Ideas
- Rigorous System Design
  - The VLSI Design Paradigm
  - Principles for Rigorous System Design
  - Component-based Design
  - Semantic Coherency
  - Correctness-by-construction
- Putting RiSD into Practice
- Discussion
Putting RSD into Practice – Basic Concepts of BIP

Layered component model

Priorities (schedulers)

Interactions (protocols)

Composition operation parameterized by glue IN12, PR12
Putting RSD into Practice – Using BIP as a Host Language

BIP is a component framework including
1. a language for hierarchical construction of composite components

2. associated tools
   - Translators from domain specific languages
   - Run-time Engines
   - Validation and analysis tools
   - Code generation tools

\[
\text{workP} \prec \text{workC}
\]

\[
c := p
\]

\[
p := f(p)
\]

\[
talkP \quad p
\]

\[
\text{work} \quad \text{talkP}
\]

\[
\text{ConsumerProducer}
\]

\[
c := f(c)
\]

\[
talkC \quad c
\]

\[
\text{workC} \quad \text{talkC}
\]

\[
\text{com} \quad \text{p}
\]

Producer

Consumer
Putting RSD into Practice – System Design Flow in BIP

- Application SW
- Embedding
  - Application SW model in BIP
- Code Generation
- Implementation
- Platform
- HW Infrastructure
- Mapping
- Transformation
  - System model in BIP
  - S2S
- Analysis & Validation
- Protocols
- System model in S/R-BIP
Putting RSD into Practice – Integrating HW constraints

- Application SW
- HW Architecture
- System Model

- Processor
- Bus
- Memory
- Scheduler
- Arbiter
- Interface
- Memory
Putting RSD into Practice – Integrating HW constraints

The **frontend** transforms networks of processes written in DOL into BIP models.

The **backend** applies a series of source-to-source correct-by-construction transformations over the BIP software model.

The generated system model includes can be validated and analyzed.
The MJPEG decoder

- reads a sequence of frames and displays the decompressed frames
- is described as a process network with five processes and nine communication channels
A simplified Multi-Processor ARM (MPARM)

- Five identical tiles and a Shared Memory connected via a Shared Bus.
- Tiles contain a CPU connected to its Local Memory via a Local Bus.
- CPU frequency: 200 Mhz.
- Access times: 2 CPU cycles for local memory
  6 CPU cycles for shared memory
### MJPEG decoder — Mapping

#### Process mapping table

<table>
<thead>
<tr>
<th>Mapping</th>
<th>ARM1</th>
<th>ARM2</th>
<th>ARM3</th>
<th>ARM4</th>
<th>ARM5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mapping1</td>
<td>all</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mapping2</td>
<td>SS, SF, IQ</td>
<td>MF, MS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mapping3</td>
<td>SS, SF</td>
<td>IQ, MF, MS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mapping4</td>
<td>SS, SF</td>
<td>IQ</td>
<td>MF, MS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mapping5</td>
<td>SS, MS</td>
<td>SF</td>
<td>IQ</td>
<td>MF</td>
<td></td>
</tr>
<tr>
<td>Mapping6</td>
<td>SS</td>
<td>SF</td>
<td>IQ</td>
<td>MF</td>
<td>MS</td>
</tr>
<tr>
<td>Mapping7</td>
<td>SS, SF</td>
<td>IQ</td>
<td>MF, MS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mapping8</td>
<td>SS</td>
<td>SF</td>
<td>IQ</td>
<td>MF</td>
<td>MS</td>
</tr>
</tbody>
</table>

#### SW-Channel mapping table

<table>
<thead>
<tr>
<th>Mapping</th>
<th>Shared</th>
<th>LM1</th>
<th>LM2</th>
<th>LM3</th>
<th>LM4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mapping1</td>
<td>all</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mapping2</td>
<td>C6, C7</td>
<td>C1, C2, C3, C4, C5</td>
<td></td>
<td>C8, C9</td>
<td></td>
</tr>
<tr>
<td>Mapping3</td>
<td>C3, C4, C5, C6</td>
<td>C1, C2</td>
<td></td>
<td>C7, C8, C9</td>
<td></td>
</tr>
<tr>
<td>Mapping4</td>
<td>C3, C4, C5, C6, C7</td>
<td>C1, C2</td>
<td></td>
<td>C8, C9</td>
<td></td>
</tr>
<tr>
<td>Mapping5</td>
<td>all</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mapping6</td>
<td>all</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mapping7</td>
<td>C6, C7</td>
<td>C1, C2, C3, C4, C5</td>
<td></td>
<td>C8, C9</td>
<td></td>
</tr>
<tr>
<td>Mapping8</td>
<td>C1, C2</td>
<td>C3, C4, C5, C6</td>
<td>C7</td>
<td>C8, C9</td>
<td></td>
</tr>
</tbody>
</table>
Mapping (1) gives the worst computation time as all processes are mapped to a single processor.

The communication overhead is reduced if we distribute sw-channels to the local memories of the processors.
As more channels are mapped to the local memory, the shared bus contention is reduced. However, this might increase the local memory contention, as is evident for mapping (8).
Putting RSD into Practice – Distributed Implementation

Distributed Implementation

SW model

I1

I2

I3

Distributed Mutual Exclusion Protocol

Interaction Protocol for I1

Interaction Protocol for I2

Interaction Protocol for I3

Distributed Execution Engine

Interface

Interface

Interface

Interface

Interface
Putting RSD into Practice – Distributed Implementation

Conflict Resolution Protocol

Partitioning of Interactions

Dining Philo. CRP

Interaction Prot. α1 α2

Interaction Prot. α3 α4

Partitioning of Components

Dining Philo. Conflict Resolution Protocol

Component 1  129.2.2.1  Core1
Component 2  129.2.2.1  Core3
Component 3  129.2.2.1  Core2
Trustworthiness

Breaking with Old Ideas

Rigorous System Design
- The VLSI Design Paradigm
- Principles for Rigorous System Design
- Component-based Design
- Semantic Coherency
- Correctness-by-construction
- Putting RiSD into Practice

Discussion
New trends break with traditional Computing Systems Engineering - Goodbye to desktop applications and their ilk

Too much of research in software engineering, systems, formal methods, etc. never made it in practice because it assumed a "design from scratch" approach and correctness-by-checking

Formal methods
- can only partially contribute to enhancing trustworthiness and optimality
- are limited to systems and properties that can be formalized and checked efficiently e.g. functional properties of SW components
Conclusion – Rigorous System Design

New approach for the design of trustworthy and optimized systems

- Endeavors unification through formalization of design as a process
  - for deriving trustworthy and optimal implementations from an application software and models of its execution platform and physical environment
  - which is semantically sound, incremental, scalable and accountable

- Opens the way for moving from ad hoc and empirical design techniques to a well-founded design discipline
The BIP component framework has been developed for more than 10 years, with Rigorous Design in mind

Main achievements:

- Translation of several DSL (Simulink, Lustre, DOL, nesC) into BIP
- Rigorous design flow based on source-to-source transformations proven correct-by-construction, in particular for:
  - taking into account HW resources
  - generating distributed implementations for several platforms
  - code optimization
- Run-times for centralized execution/simulation, distributed execution, real-time execution
- Validation and analysis tools:
  - Incremental checking for Deadlock-freedom: D-Finder tool
  - Statistical Model Checking
- Successful application in many industrial projects:
  - software componentization for robotic systems (DALA Space Robot for Astrium)
  - programming multi-core systems (P2012 for ST Microelectronics, MPPA for Kalray)
  - complex systems modeling (AFDX and IMA for Airbus)
Thank You