Precise Specification Methods

Kaisa Sere and Luigia Petre
Distributed Systems Laboratory
IT-Department, Faculty of Technology, Åbo Akademi University
Turku, Finland
Starting point

• Software-based systems
  • Embedded in multiple fields
    • Financial, commercial, medical, nuclear, etc
  • More and more complex
• We are software dependent
  • We need systematic methods of construction
  • We need dedicated verification methods
What are precise specification methods?

• Usually called FORMAL METHODS
  • Fault-avoidance techniques for constructing trustworthy systems
  • Consist of a specification language and a construction and/or verification method
  • Mathematical basis

• Specification language
  • Based on logics and mathematics (has SEMANTICS!!)
  • Allows the devising of models for the system to be built by using a specific syntax

• Construction/verification method can allow
  • The specification of the required properties for verifying that
    • A model correctly captures a set of requirements
    • A more detailed model correctly develops a less detailed model of the same system
  • Various analysis types based on the devised models
Some formal methods

- CSP (Communicating Sequential Processes)
- CCS (Calculus of Communicating Systems)
  - \( \pi \)-calculus
- Ambient calculus
- Action systems
- UNITY
- Z
- B
Action systems

- For modeling distributed systems
- Imperative specification language
  - State-based formalism
  - Extension of the “guarded commands” language of Dijkstra
  - Can model parallel computing
  - Based on the refinement calculus of Back
  - Semantics: wp-calculus ("weakest preconditions")
Why action systems?

• Introduced in 1983, by Back and Kurki-Suonio
• Brings a new perspective for the modeling of distributed systems
  • Until then it was CSP and CCS – process algebras
• Extended in very many directions
  • Hardware modeling → BlueSpec, MIT
  • Hybrid systems
  • Critical systems
Action system appearance

\[A = \left[ \begin{array}{c}
\text{var } x, y, z, v, w \\
\text{init}
\end{array} \right.
\begin{array}{c}
do A_i \od\end{array}
\begin{array}{c}S_i\end{array}\]

\(A_i = g_i \Rightarrow S_i\)

\(A_i\) can be chosen for execution when \(g_i\) holds; then \(A_i\) is called enabled.
The fundamental feature

• Refinement quality: $A$, $B$
  • A high level specification $A$ of a program is transformed by a sequence of correctness-preserving steps into an executable and efficient program $B$ that satisfies the original specification
  ◦ $A$ – action system; beh ($A$) (behavior of $A$): the set of state sequences that correspond to all the possible executions of $A$
• Algorithmic, data, superposition
• Algorithmic: just input-output behavior kept
• Data: also the reactive behavior (internal)
  • Abstraction relation $R$ between the new and the old data
  • $R$ models what needs to be respected by the refined version
• Superposition: adding things
Algorithmic refinement

- Degree of parallelism can be increased, while correctness is kept
- *Total correctness* preserved
  - Appropriate correctness notion for parallel algorithms
  - Parallel programs differ from the sequential ones in that
    - They are executed by cooperation of processes (in parallel)
    - They are intended to terminate
    - Only their final results are of interest
Stepwise refinement of action systems

• Special rules for introducing parallelism
  • Transform sequential statements directly into iterative constructs
  • Combine iterative sub-statements into a single iterative construct
• Change variables in sub-statements
  • Change the program state representation from centralized to distributed
Refining atomicity

• Starting point
  • Large action, possibly needing synchronization and participation of many processes over an extended time

• Replaced by
  • A number of smaller actions achieving the same effect as the larger action but exhibiting more parallelism among themselves
    • Less synchronization of processes, more overlapping of (parallel) execution
    • Less time

• Boils down to
  • Transforming the control structure: moving actions of an inner loop to the level of the outer loop
Data refinement

• Algorithmic refinement
  • does not guarantee that the behavior of $B$ is the same as the behavior of $A$ in every step of the execution

• Data refinement also preserves the reactive (internal) behavior
  • Abstraction relation $R$ between data in $A$ and data in $B$
  • $R$ acts as an invariant capturing the reactive behavior to be kept
  • Besides preserving the correctness, all changes have to also ensure $R$

• Specific rules
  • $R$ is established by the initializations in $A$ and $B$
  • Each action $A$ in $A$ is data refined by its corresponding action $B$ in $B$ using $R$
  • For each action $A$, when $A$ is enabled, then either $B$ or an extra action $H$ in $B$ is enabled whenever $R$ holds
  • Every extra action $H$ in $B$ is a stuttering action (acts as skip for the global variables)
  • Every extra action $H$ in $B$ terminates when executed in isolation
  • The environment preserves $R$
Trace refinement

- Theoretical expression of data refinement for reactive systems that
  - May / may not terminate
  - Atomic actions of the reactive systems need not terminate themselves
- Trace
  - $s$ – behavior; $s = \langle (a_0, u_0), (a_1, u_1), \ldots \rangle$
  - *Stuttering step*: an occurrence of 2 consecutive steps in $s$ with the same
    global component
  - $Tr(s)$ is the behavior $s$, removing all finite stuttering and keeping only the
    global component in each state
- Trace refinement
  - $\mathcal{A} \subseteq \mathcal{C} \iff \forall t \in \text{beh}(\mathcal{C}). \exists s \in \text{beh}(\mathcal{A}). s \leq t$, where $s \leq t$ (s approximates t)
    if
    - Either $s$ is aborting and $tr(s)$ is a prefix of $tr(t)$
    - Or $tr(s) = tr(t)$ and neither $s$ nor $t$ are aborting
  - Difficult to use; instead simulation methods are used
  - Simulation method: constructs an abstract behavior which approximates
    a given concrete behavior
Superposition refinement

- $S, S': S \subseteq S'$ if
  - Whenever $S$ is guaranteed to terminate, $S'$ is also guaranteed to terminate
  - Any possible outcome of $S'$ for some initial state is also a possible outcome of $S$ for the same initial state

- $S'$ can add data and behavior to $S$, without interfering with the behavior of $S$
  - Keep behavior of $S$
  - No influence on the behavior of $S$
  - No taking over the behavior of $S$

- In practice
  - Strengthening guards
  - Adding non-influencing and non-taking over actions
Action systems in use

- Middleware supporting the functioning of a network
  - MIDAS
- Energy-aware computing
## Specification levels

<table>
<thead>
<tr>
<th>Role</th>
<th>Specification</th>
<th>Represents</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>Requirements</td>
<td></td>
</tr>
<tr>
<td>Application developer</td>
<td>Level 1 Spec</td>
<td>Application</td>
</tr>
<tr>
<td>Network manager</td>
<td>Level 2 Spec</td>
<td>Middleware</td>
</tr>
<tr>
<td>Programmer</td>
<td>Implementation</td>
<td>Code</td>
</tr>
</tbody>
</table>
MIDAS

• a middleware language
• resource-centric
• assists the network manager to treat
  • resource accessibility
  • replicated and homonym resources
  • mobility of resources
  • node failure and recovery
• based on action systems
Resources in MIDAS

- data, code, or computation units
- locations of resources
  - network: \((V, \text{Edges})\)
  - nodes \(\alpha\) in the network where data can reside or where computation can take place
  - implicitly: stationary devices
- Is it necessary?
MIDAS resources

- **Data**: \((v, \text{loc}, \text{VAL}, \text{val})\)
  - Data unit: variable
  - \(v.\text{loc}, v.\text{val}, \alpha.\text{var}\)

- **Code**: \((a, \text{loc}, A)\)
  - Code unit: action
  - \(A = g \rightarrow A.\text{body}; a.\text{loc}, A.\text{loc}, \alpha.\text{action}\)

- **Computation unit**:
  - \(CU = [\text{exp } y; \text{var } x; \text{imp } z;\)
    - \(\text{do } [\prod_{i \in I} A_i \text{ od} \]
  - non-deterministic execution
  - atomic actions
Modularity

\[
CU_1 = [\exp y_1; \var x_1; \imp z_1; \\
\quad \text{do } []_{i \in I} A_i \text{ od} \\
]
\]

\[
CU_2 = [\exp y_2; \var x_2; \imp z_2; \\
\quad \text{do } []_{j \in J} A_j \text{ od} \\
]
\]

\[
CU_1 \parallel CU_2 = [\exp y_1 \cup y_2; \var x_1 \cup x_2; \\
\quad \imp z_1 \cup z_2 \setminus (y_1 \cup y_2); \\
\quad \text{do } []_{i \in I \cup J} A_i \text{ od} \\
]\]
Resource tools

• Resource accessibility
  • A.cell, gd(A) = lg(A) \land g
• Replicated and homonym resources
  • |R.loc|>1, |H.loc|=1
  • Updating mechanisms distinct
  • Not more than one replica at one location
• Mobility – of all resources
• Node failure and maintenance
  • V=V_{act} \cup V_{maint} \cup V_{failed}
  • location guard: every involved location \in V_{act}
Networks and Energy

- Networks are everywhere
  - need them to work continually and at full capacity
  - i.e.: reliable and available
- Networks are not static anymore
  - devices are battery-powered
  - devices have power-hungry features
  - devices need to plug to electricity
Energy (power)

- non-functional property
- addressed together with reliability, security, availability
  - by a network manager
  - at the middleware level
Energy-aware resources

- Energy: consumed or requested
- Data and storing data
  - Consumes no energy (assumption)
- Code
  - $P$ – amount of power that code requests for being executed
  - $(a, \text{loc}, A, P)$
  - $A.P$
Distinct types of energy aware resources

• Code needs hardware to execute on
• Hardware needs power supply
  • From battery or electric socket
• Software and hardware resources
  • Energy requesters/consumers
  • Latter have battery and electric socket connection
• Electrical sockets
  • Energy suppliers
Electrical socket

- $ES = \left[ \exp (E, \text{loc}, \text{Boolean}, T) \right]$  
- Socket is free to charge  
  - $E.val = T$  
- Socket already charges a device  
  - $E.val = F$  
- Homonym $E$ allowed, replicated $E$ not  
  - $|E.loc| = 1$
Hardware resource

\[ HW = [\text{exp}\{(B,,,0), (charge,,,F)\}]; \]
\[ \text{imp } E; \]
\[ \text{do charging::} \]
\[ \text{charge.val } \land \text{B.val } < \text{BMAX } \rightarrow \]
\[ \text{B.val } := \text{B.val } + \text{E}_{\text{unit}} \]
\[ \text{[] plug:: plug.loc=} E.\text{loc } \rightarrow \]
\[ \text{charge.val } := \neg \text{charge.val}; \]
\[ \text{E.val } := \neg \text{E.val} \]
\[ \text{[] skip:: skip} \]
\[ \text{[] act} \]
\[ \text{od} \]
\[ ]\]
Hardware resource, cont.

• act:: if charge.val then A

  else

    gd(A) ∧ B.val > min + A.P →
    A.body; B.val := B.val – A.P

  fi

• Assumptions

  • min >= skip.P + plug.P + charging.P
  • A.P < BMAX
Connecting the resources

- Data and code cannot exist unless stored on hardware
  - \[[\text{imp } B, \text{ charge, } z; \text{ do } A_i \text{ od}] || \text{HW}\]
  - we check there is a local hardware:
    \{B, \text{ charge}\} \subseteq (A_i.\text{loc}).\text{var} →

- Hardware consumes energy when executing code:
  - if \text{charge.val} then \text{A}_i
  - else
    - \text{gd}(\text{A}_i) \land \text{B.val} > \text{min} + \text{A}_i.\text{P} →
      - \text{A}_i.\text{body}; \text{B.val} := \text{B.val} – \text{A}_i.\text{P}
  - fi
Resources and locations

• Nodes $\alpha$ in (V, Edges) are hardware devices
  • Can be mobile…
• Location: any spatial coordinate where computation, communication can happen
  • E.g.: all physical space in a cellular network
  • Location is now fixed!
• (V, Edges) is a reference system for mobile resources
  • Infrastructure network
Example

• Suppose you walk in Vigeland park
• You pass a restaurant
  • Get the daily menu and price as SMS on the phone
• You pass through the statues
  • Can get a SMS or sound file or a video about that statue
  • Enough memory to get the info?
  • Enough power to watch the video?
Example (cont.)

• If not enough power, check back the restaurant
  • They have power supplies (available?)
• With the phone charged, you can check the statue video
  • And the statue itself
  • Get other videos
2 types of networks

- Electricity network
  - Provides energy
  - Contains the electrical sockets or *electricity resources*
  - Typically stationary
- Resource network
  - Consumes energy
  - Software resources and hardware devices
  - Can be mobile
- Both networks are location aware
Unitary approach in MIDAS

• We model various types of resources
  • Data, code, hardware, electricity
• Resource locations interrogated
  • Data, code, remaining power
• Electricity resources interrogated
  • Data, code, devices co-located with them
  • Useful for context-aware computing (avoid power supplies that are crowded)
Formalism

• Conservative extensions
  • Action systems $\rightarrow$ MIDAS $\rightarrow$ energy-aware MIDAS
  $\Rightarrow$ We can reuse techniques
    • Priorities of code/hardware
    • B-method tools

• Energy-aware specifications refine the energy-transparent ones
Conclusions

• We proposed a modeling framework for energy-aware resources
  • Allows for measuring energy
  • Suggests SW/HW model distinction
  • Gives a new approach for locations
    • Suggests definition of location clusters

• Further work
  • Strategies for specifying code or hardware consuming the least amount of energy
Some formal methods

- CSP (Communicating Sequential Processes)
- CCS (Calculus of Communicating Systems)
  - $\pi$-calculus
- Ambient calculus
- Action systems
- UNITY
- Z
- B
CSP

- For modeling concurrent systems
- CSP system = a set of processes
  - Each process has its own interface of \textit{events} and typed \textit{channels}
  - \textit{Events}: for process synchronization
  - \textit{Channels}: for data communication among channels
- Operational semantics
  - Transition rules for each process type
- Denotational semantics
  - Traces (special sequences of events)
Constructing CSP systems

- CSP specifications
  - Describe the allowed behavior of the processes
- CSP implementations
  - Describe the actual behavior of the processes
- Trace-based refinement
  - Ensure that every trace of an implementation is also a trace of the corresponding specification
CSP Extensions

• For hybrid systems and for time-dependent systems
  • Timed CSP: continuous model of time
  • Tock – regularly marks the passing of time
    • Tools

• Software tools
  • Model checker FDR
  • Animator ProBE
CCS

- Also for modeling concurrent systems
- CCS system: parallel composition of processes
  - They can synchronize and communicate by using pre-established links
- $\pi$-calculus
  - CCS extension
  - Links can communicate values AND links
    - First technique for modeling mobility
CCS and $\pi$-calculus: semantics and constructing systems

- Operational semantics
  - Labeled transition systems
  - Many variants

- Constructing systems
  - We can develop abstract and concrete models
  - Bisimilarity (weak and strong)
    - Behavioral equivalence
    - Ensures that 2 models correspond to each other
Extensions and software for $\pi$-calculus

- Many extensions
  - Exp: OO based on a type system in terms of sorts
- Software tools
  - Model checker Mobility WorkBench
    - Verifies also bisimilarity properties
  - Programming languages based on $\pi$-calculus
    - Pict, Nomadic Pict, Piccola
- Semantic model for concurrent calculus
Ambient calculus

- For modeling computations over Internet
- Ambient
  - Administrative domain in Internet
  - Place where various computations take place
  - Can contain sub-ambients or be contained in a parent-ambient
  - Sibling ambients can communicate locally
  - Mobility – most important feature
    - Need to cross barriers
    - Stepwise
Ambient calculus: semantics and analysis methods

• Operational semantics
• Can express both $\pi$-calculus as well as $\lambda$-calculus
• Expressing properties
  • Communication and mobility types
    • Ensure that communication is correct (well-typed)
    • Mobility can be restricted if desired
  • Decidable modal logic
    • Model checking for various process properties
  • (More complex) modal logic for mobility
    • Can express time and space
    • Mobility: change of spatial configurations over time
UNITY

• State-based formalism for modeling distributed systems
• System state
  • Described by variables
  • Modified by conditional assignments
  • An assignment can be chosen for execution only when its guarding condition holds
  • Choosing between several executable assignments is weakly fair
• UNITY system: more programs
  • Homonym variables are shared
  • Assignments can execute synchronously or asynchronously
UNITY: semantics and constructing systems

- Denotational semantics
  - Transition systems
- Reasoning about programs
  - Temporal logic: safety and liveness properties can be expressed and proved wrt the UNITY models
- Structuring techniques
  - Union and superposition refinement
Mobility in UNITY

• Mobility
  • Mobile UNITY: specialization of UNITY
  • Programs have “location” variables
  • Specialized conditional assignments execute as long as their conditions hold
  • centralized “Interactions” section coordinates the programs
    • The section contains the sharing and synchronization conditions
• Dynamic style of computing
Context in UNITY

• Context-UNITY
  • Specialized version of Mobile UNITY
  • Models environment-sensitive applications
  • “Interactions” section replaced by several “Context” sections
    • One per program
    • Contain coordination rules
  • Very flexible systems can be modeled
Z

- Oldest state-based formalism
- Modeling systems based on abstract data types
- Z-System: variables and operations
  - Variables have types, invariants and initializations
  - Operations are expressed as predicates describing the system state before and after the operation
  - Z-Schemas (syntactical structures)
Semantics and analysis in Z

• Semantics based on values
  • Schema S corresponds to $\Theta S$
  • $\Theta S$ – characteristic binding of S
    • Each declared component is associated with its current value

• Proofs and refinements
  • Based on set theory and logics
Extensions of Z

- Timed refinement calculus
  - Z extended with total continuous functions of time
- Object-Z
  - Enhances the structuring of Z specifications
  - Class-schema contains all the schemas concerning an abstract data type
  - Refinement of classes with simulation techniques
  - Reference-based semantics
    - Variables refer values instead of directly representing them
    - Extra semantics in terms of transition systems
- Combinations
  - Timed refinement calculus and Object-Z
  - Object-Z and CSP (and their timed extensions)
• State-based formalism for
  • Modeling, refinement, validation, and implementation of software systems
• B specification unit: abstract machine
  • Variables and operations that can evaluate and modify variables
• Invariants:
  • Constrain variables
  • Must be preserved by the operations
Modularity, validation and refinement in B

• Fundamental features of B
• Modularity
  • System: a collection of machines that are independently developed
• Refinement
  • Algorithmic or data refinement
• Validation
  • Discharging the proof “obligations” generated in the refinement process
B supported by software tools

• Validation
  • Highly automated due existing associated tools (theorem provers)
    • Atelier-B and B Toolkit
  • Many proof obligations are discharged automatically by the software
  • The remaining obligations can be interactively proved
  • When all the proof obligations are discharged code can be generated in C or ADA
Semantics and extensions in B

• Semantics
  • Based on an extension of the weakest-precondition calculus, set theory, logics

• Extensions
  • Distributed and reactive systems (Event-B)
  • Complex systems (software and hardware)
Conclusions

• We have plenty of formal tools to choose from
  • They can model anything
  • How to choose?
    • Integrate formal methods? (UML?)
  • Do they scale?
  • Do they have associated tools?
Motto

• The sciences do not try to explain, they hardly even try to interpret, they mainly make models.

John von Neumann
Material

- http://web.abo.fi/~lpetre/

