Programming for the Real-World (Embedded Systems)

SDP Workshop
Nashville, TN
Dec 13, 14 2001
Participants

- Robert Laddaga (leader)
- Paul Robertson
- Scott Smolka
- Mitch Kokar
- David Stewart
- Brian Williams

- Insup Lee
- Kang Shin
- John Reekie
- Rajeev Alur
- With help from:
  - Dave Sharp
  - Calton Pu
Questions

- What can’t we do
- What are the promising technologies
- How can we enhance and extend current technologies
- How can we do technology transfer
- How can we spend $.5B/year
- Specific sellable ideas
- How to build for change
- How to exploit legacy SW
- What are our Basic Assumptions – challenges and threats
Outline

• Notable Progress
• What we can’t do
• What we can’t do (tech speak)
• Promising Technology
Notable Progress

- Use of tools
- Use of real-time operating systems
- Memory/Computation constrained
- Fault tolerant protocols
- Scheduling single processors
- Testing and validation
What we can’t do

- high level sensing
- infer decision crucial info from multiple disparate sources
- Competent transfer of control (human or automated)
- automate our mechatronic miracles (e.g., UCAV, FCS, Space-Vehicles)
- make embedded systems invisible—ultra stable.
- certify embedded systems for less than $\{gag\}.$
- trust our embedded software—especially safety critical and novel systems.
- adapt to changing requirements/environment.
Unsolved Problems

- Uncertainty and model selection
- Hardware/Software co-design
- Testing & Validation
- Meeting non-functional constraints in integrated systems.
- Use of models in integration
- Sensing Environment and Adaptive (resource allocation, dynamic and adaptive)
- Can’t automate what we know how to do
More Unsolved Problems

- Distributed dynamic resource allocation.
- Fault management.
- Handle harsh environments cost effectively.
- Encoding and measuring functional redundancy.
Promising Technologies

• Model based software development
• Non-functional reasoning of embedded SW development
• Model Selection and Estimation
• Temporal Decision Theoretic Embedded Languages
• Self adaptive software
• Synthesis
• Model-based Programming of Embedded Software
Promising Technologies - Format

• What it is
• Research Agenda
• How it helps
Model-based software development

• Begin with informal requirements
• Capture requirements in a model serving as a specification of the system
• Lots of different modeling paradigms (ptolemy, simulink, charon)
• Model refinement and requirements tracing
• Code Generation
• Model based testing and validation
MBSD – Research Agenda

• Closing consistency gap between model and code – preservation of structural features of design in code.
• Translating informal requirements into formal requirements
• Tracing requirements into implementation
• How can we include disparately modeled submodels
• Enriched formalisms supporting non functional aspects
• More efficient testing
• Capture of distributed embedded systems
• Models including uncertainty
• Self adaptive models
MBSD – how it helps

• Certification for lower dollars
  – Streamlining testing
  – Early bug discovery
  – Validation techniques

• Trust of embedded software
  – Improved certification, reliability, understandability

• Invisible – ultra stable
Model Selection & Estimation

• Techniques for simultaneous estimation of model parameters and comparing alternate models
MSE – Research Agenda

• Algorithms, optimization and approximation techniques to allow tractable computation along with realistic dependency assumptions
• Estimation over large distributed spaces
• Integration of multiple model representations – models include constraints, logic, bayes nets, HMM, ODE
• How to seamlessly fold methods for MSE into embedded languages
MSE – how it helps

• Info fusion
  – Integrates vastly distributed information sources

• Detection of incipient states
  – Helps to detect masked states
Temporal Decision theoretic Embedded Languages

• Tracking large numbers of execution trajectories
• Planning using expected values
• Dynamic technique involving On-line:
  – Tracking
  – Projection
  – Execution
  – replanning
TDTEL – Research Agenda

- How to decide which unlikely trajectories to track
- How to project forward consequences of traced trajectories to ensure safety
- On-line model checking
- How to fold TDT Planning and execution into embedded languages
- How do we do TDT Planning at reactive timescale
- How do we concurrently do planning and execution on line.
TDTEL – How it helps

• Automation and adaptation
  – Dynamic planning
  – recovery
Non-functional reasoning of embedded software development

- Bottom up approach to produce reliable components and building blocks – including functional and non-functional description and assurance
- HW/SW Codesign – software redesign and reconfiguration
- Reliable device drivers – reliable interfaces to unreliable hardware
- Component specification – resource allocation under scarcity
- Aspect oriented software development – performance monitoring
- Non functional constraints – imprecise computation, uncertainty, fault tolerance issues
- Low bit rate networking protocols
- Trade-off analysis
- Configurable hardware
- Application level
Self-adaptive Software

• What it is:
  – Monitor detect and repair in response to faults and changes by modifying/resynthesizing program.
  – Feedback/Control-system-like

• Examples:
  – Networks of cooperating air vehicles.
  – Reconfiguration of hardware within vehicles and submarines.
  – Adaptation of control laws for flight surfaces.
  – Adaptation of numerical codes for optimization or simulation.
  – Adaptation of assumptions to track changing conditions during high level sensing (vision, speech).
SAS – Research Agenda

• Investigate ways of ensuring stability.
• Investigate ways of ensuring that the high level goals of the system are met – the set point.
• Investigate how to represent models and monitor models for different classes of systems.
• Investigate ways of performing program synthesis.
• Investigate how to achieve acceptable performance (good enough soon enough, QoS metrics)
• Architectures and design of Self-adaptive software.
• Design languages that incorporate ideas of sensing and adaptation.
Self-Adaptive Software

• What problems it addresses:
  – High level sensing
  – Adaptation
  – Automation

• Why it solves the problems:
  – Divides a complex space into smaller tractable ones.
  – Control systems are inherent engineering artifacts. Embedded systems control physical systems and are inherently control system-like.
Synthesis

• What it is:
  – Automatic code generation from specifications, models, design rules.
Synthesis - Research Agenda

• Dealing with uncertainty and hidden states.
• Automatic generation of monitor code from models.
• Model Fitting.
• How to bring focused synthesis online.
• Integration of offline compilation with online reasoning.
• Dealing with optimality and feasibility.
• Dealing with functional redundancies and contingencies.
• Dealing with dynamically changing components.
• Resource allocation/constraints.
Synthesis

• What problems it addresses:
  – Supports self-adaptive software model-based programming
  – Adaptation
  – Assured low-level components.

• Why it solves problems:
  – Allows software to be generated dynamically (at runtime).
  – Provides for automatic verification.
  – Improves confidence vs. Human coding.
Model-based Programming of Embedded Software

• What it is:
  Embedded languages that:
  – Encode strategic guidance and incorporate models of the environment
  – Use these descriptions to automatically interpret and coordinate environmental interactions.
MPES: Research Agenda

• Seamless extension of embedded languages to:
  – Incorporate rich models of the embedded environment.
  – Shift the role of a program from an imperative to an advisory role
• Fast on-line reasoning for managing interactions, including: State estimation, environment reconfiguration, planning, scheduling, discrete event control and continuous control.
• Automated partitioning of coordination between run-time and compile-time tasks.
• Frameworks for incorporating and reasoning from a rich set of modeling formalisms.
MPES: How it helps

• Simplifies programming for autonomy by
  – Offering a simpler model of interaction between the programmer and the environment
  – By delegating reasoning about interactions to the language’s interpreter/compiler.

• Improves robustness for autonomy by
  – Systematically consider a wider set of possible interactions and responses.
  – Responding to novel events on-line.
  – Employing provably correct algorithms.

• Supports adjustable levels of autonomy by
  – Allowing the programmer to delegate the desired level of control authority within the control program.