Embedded S/W Development Using PTII

Modeling Extensions, Data Representation, Compilation

Zoltan Kemenczy, Sean Simmons
Research in Motion Limited

5th Biennial Ptolemy Miniconference
Berkeley, CA, May 9, 2003

Contents

- Motivation and Observations
- Example Model
- Modeling Extensions
- Compilation

Motivation

Goal: Minimize impact of application changes and target changes

Goal: Reuse test vectors/harness

Change Development Strategy

FROM

1. Simulate/Test Key Algorithms Only in High Level Language tool
2. Implement "by hand":
   - Algorithms
   - Application control flow and task structure
   - Port tests
   - Select data representation
   - Select overflow and precision loss methods
3. Iterate 1&2 until satisfactory test result are achieved

TO

1. Simulate/Test Entire System in High Level Language tool
2. Refine HLL simulations by specifying:
   - Data path representation
   - Overflow and precision loss methods.
3. Compile result for target once all data types are "concrete"

Observations

Two basic types of code "leaf" & "structure"

"Leaf" (Actors)

- Target specific
- Basic algorithms (+, *, FIR)
- Optimization to take advantage of target specific facilities – e.g. dual MAC, ACS
- Compilation "difficult"

"Structure"(MoC/Connections)

- Application specific
- Control/data flow
- Optimizations for memory use/re-use (registers, queues), schedules
- Compilation "easy"

Iteration

- 80%-20% rule
- Place one level of iteration in atomic actors – can make use of target H/W looping
- Use array data types for specifying implicit iteration
- Block processing approach
- Scalars are degenerate arrays

Observations - Example

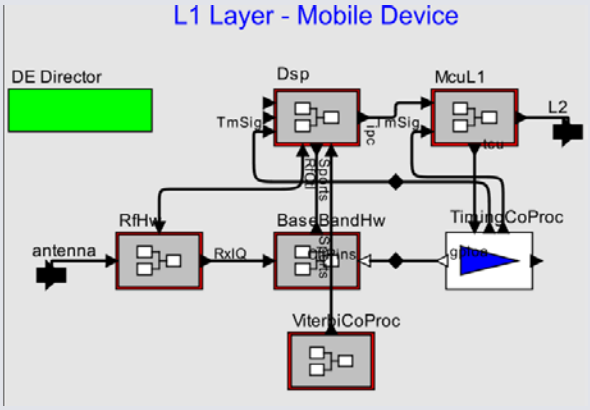
"Leaf"

```
void Abs(int n, S15 *in, S15 *out)
   {
 int i;

 for (i = 0; i < n; i++) {
   out[i] = in[i] >= 0 ? in[i] : -in[i];
 }
}
```

"Structure"

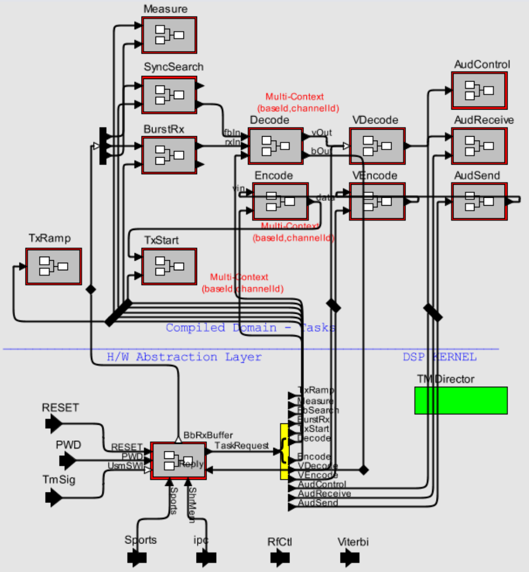
```
void Rx() {
  int n = ReadAvail();
  if (n > MIN_SAMPLES) {
    S15 buf[MAX_SAMPLES];
    S15 buf2[MAX_SAMPLES];

    DetectReset();
    ResampleRead(n , buf);
    RunAfc(n, buf);
    DownConvert(n, buf);
    Detect(n, buf, buf2);
    ....
  }
}
```

Example Model



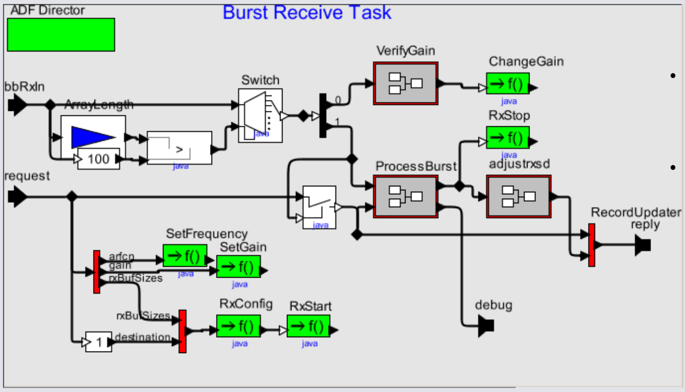
- GSM/GPRS Physical layer Simulation
- Dsp, McuL1 - compilation targets
- H/W Blocks - simulation only
- Component of encompassing test harness
- Typical variants: (S/W X H/W) Versions

Example Model – The DSP



- Low-latency (left side) tasks triggered by timing signals
- Data-flow driven lower rate/priority tasks (e.g. 1 Decode / 4 Bursts)
- Test-paths designed in (e.g MCU may request a Vdecode(data))
- TaskRequest – Union of Records
- H.A.L Object(s) and TMDirector hand-coded

Example Model – A DSP Task



- “request” initiates HAL control
- “bbRxIn” HAL events schedule task
- “reply” generated when finished

Example – Summary

Component	Simulation	Target	Goals
MCU and DSP	TM, ADF, SDF, FSM	Compiled	•Single model / feature set •Model "run" in different targets (PTII, Target Simulation, Device)
PTII "Library"	n/a	Hand-Coded	•Actors •Schedulers •Type/Token/Port handling
HAL	DE	Hand-Coded	•HAL Minimal but complete •Handle different H/W platforms, versions •Simulation true to HAL API
Timing, Coding, Cipher	DE	H/W	
Radio, Baseband, Comm.	DE	H/W	

Contents

- Motivation and Observations
- Example Model
- Modeling Extensions
  - Asynchronous Data-Flow MoC
  - MultiInstanceComposite
  - ObjectMethod
  - New Types
  - Mixing Built-In and User Types
- Compilation

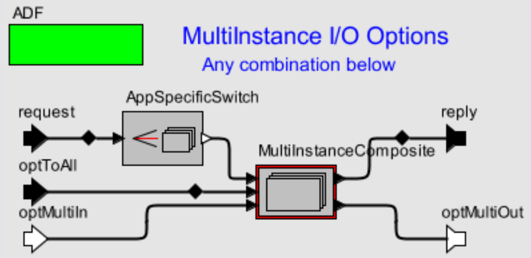
Extensions: Asynchronous Data Flow MoC

- Data flow driven like SDF
- Dynamic schedule like DE
- No notion of time (local or global) like SDF
- No global event queue like SDF
  - Local queues on each port
- Loops are allowed
  - Requires use of a “register” actor
  - Same idea as “zero delay” in DE or Z^{-1} in SDF.
- Port rates computed like SDF
  - Represent maximum number of tokens produced when fired
  - Used to compute queue sizes for compilation
- Uses fixed firing order
  - Uses prefire to evaluate actor’s readiness
  - Repeatedly fires actors in sequence until all actors prefire methods returns false

---

Extensions - MultiInstanceComposite

- Same CompositeActor (MoML class), multiple (object) instances
- Contributed as a form of HOC
- Use DE or ADF MoC, contain Modal Models - typically
- Examples:
  - Objects (6 or more instances) representing tracked base-stations
  - Logical channels within protocol stack layers
  - Easy conversion of single-channel I/O actor to multi-channel (e.g. multi-channel FIR, same or different parameters e.g.“({{taps}})[instance]”



Extensions - ObjectMethod

- HAL objects:
  - are contained within TM-domain composites (s/w running on a processor),
  - control their associated h/w,
  - process h/w signals and send (event) tokens to other actors (tasks).
- The ObjectMethod actor:
  - is used to access HAL objects from classes nested in TM-domain composites,
  - is a form of a "tunnelling relation" to an opaque (HAL) actor instance with SDF semantics: all inputs must be present to fire, rate one, output (if defined by the target method) also rate one, immediately available.
- Example: Base-band ADC Samples Receiver HAL object. Used from Measure, Synchronize, Burst Receive classes ("tasks").
- ObjectMethod actor safely configured using object instance reference (ObjectToken) that also yields object class ("API").
- Directors within HAL simulation blocks (DE) are <u>also</u> fired (by ObjectMethod) after each invoke.

Extensions – "User" Types

- Fixed-length ArrayType – multi-dimensional ArrayType with *known* dimensions '{N,M,K,…}'. Size = N x M x K x … - 'rectangular'. <u>Linear storage</u>, inner (last) dimension first. PTII arrays and matrices map onto this type when their dimensions are known.

- Variable-length ArrayType – Like above, but tokens of this type have a variable outermost dimension '{1..N, M, K,…}.

  Fundamental I/O type for our block-processing actors. Supports actor buffer-size calculations that reflect I/O rates

- EnumType – a set of identifiers.

- UnionType – a set of Type elements, each associated with an identifier. Identifier set is an EnumType. Represents data that share target storage.

  Exactly specifies Types that are expected to pass on a relation. Provides a solution to the problem of passing different RecordTypes over a relation, e.g. Request/Reply interactions with actors without losing any fields.

Extensions – Variable-Length Array

- A multi-dimensional array type with known dimensions {I,J,…} that removes the PTII scalar-, array- and matrix-type distinction
- Tokens of this type have a variable outer-dimension: i=1..I
- Linearized (single) index and multi-dimensional (i,j,k,…) index access
- Target memory layout is along innermost (last) dimension index.
- Type Lattice: Uknown < Vlarray < Array < General
  - Because Array is unsized (∞ size), any Vlarray may be converted to an Array with the same number of dimensions {{…{elemType}…}}. (We avoid this though to preserve dimension information)
- Conversion/Compatibility (cf. ptolemy.data.type.Type, TypeLattice)
  - Vlarray(elem,{I,J,K,…}) can be converted to Vlarray(elem,{L,M,N…}) if I < L and J=M, K=N,… and element types are compatible
  - ScalarType is equiv. to Vlarray(ScalarType,{1,1,…}) (# of dim as needed)
- LUB:
  - Vlarray(max(leftDim[0],rightDim[0]), dim[1], dim[2],…) where dim[i] must be same for left and right i=1..dim.length (compatible)

Extensions - EnumType

- Set of identifiers, some possibly associated with specified integer values, others "unknown"
- Tokens of this type have one of the identifiers from the set as a value
- Type Lattice: Unknown < EnumType < General.
- Conversion/Compatibility:
  - (In the following, typeLS = type.labelSet(), typeVS(labelSet) = type.valueSet(labelSet))
  - argLS $\subseteq$ thisLS
  - argVS($\cap$(thisLS,argLS)) = thisVS($\cap$(thisLS,argLS)), where "unknown" = any
  - A StringToken is convertible if $\in$ thisLS
  - An IntToken is convertible if $\in$ thisVS
- Compare:
  - Equal: leftLS = rightLS $\wedge$ right.isCompatible(left)
  - Less: $\neg$Equal $\wedge$ right.isCompatible(left)
  - …
- LUB:
  - $\cup$(enumArgs) if enumArgs compatible.

Extensions - UnionType

- Set of Types each associated with an identifier ("label")
- PTII expression parser entry:
  - union("name", {id1=token1, id2=token2, … }) - registers a named union
  - union("name", {idx=tokenx}) - creates the "idx" member with tokenx value, tokenx type must equal the type of name.idx
  - union({id1=token1, id2=token2, … },{idx=tokenx}) - creates type and token
- Type Compatibility:
  - In the following, typeLS = type.labelSet() , typeTS(labelset) = type.typeSet(labelset)
  - argLS $\subseteq$ thisLS $\wedge$ argTS() = thisTS(argLS)
  - Note 1 – we chose type set equality (more stringent) not element-by-element compatibility.
  - Note 2– {idx=tokenx} (RecordToken) is compatible with union({idx=tokenx})
- Type Compare:
  - Equal: leftLS = rightLS $\wedge$ leftTS() = rightTS()
  - Less: $\neg$Equal $\wedge$ right.isCompatible(left)
  - …
- LUB (note: cls = $\bigcap$(leftLS,rightLS)):
  - $\bigcup$(leftTS, rightTS) if leftTS(cls) = rightTS(cls), else General

---

Extensions: Built-In and User Types

- Each user type is on a separate branch between Unknown and General on the Type lattice ([1] Ch. 12).
- Consequences:
  - LUB(any *known* built-in type, user type) = General.
  - LUB(user type 1, user type 2) = General.
  - => Type information is lost.
- But precise type information is essential for compilation domain actors ☹
- User Actor Extensions (mixing and preserving types):
  - User actors (dealing with 'mixed' input types) must have an *empty* type constraint list to avoid output port types evaluated to 'General'.
  - Hence user actors use type functions:
    - ***Output port type = f (input port types, output port)***
  - (Default type functions are incorporated into actor base classes. Methods are provided to override default type function results.)
  - Do not mix built-in and user types on different relations connected to an input multi-port since this also yields 'General'

Contents

- Motivation and Observations
- Example Model
- Modeling Extensions
- Compilation
  - Strategy
  - Target Actor
  - Data Representation

Compilation Strategy - Approach

- Use PTII as much as possible
  - Type resolution
  - Introspection
- Target environment support
  - Tokens, types, ports, parameters, schedulers, startup code.
  - Multiple target environments can be supported in one model.
- Target atomic actors
  - Only support create(), initialize(), prefire(), and fire().
  - Can be specialized based on port/parameter types, parameter values, target.
- Hook compilation process into "top level" actor's initialize method to determine:
  - Target data representation selected based on target description and port/parameter types
  - Target actor specialization based on port/parameter types and parameter values
  - Maximum inter-actor queue sizes can be determined based on schedule information
  - Static schedule information
- Compiled output produced
  - "Dynamic" uses JNI to interact with target simulator
  - "Static" exports a target memory image in source form.

Compilation Strategy - Continued

- Assume no garbage collector for tokens
  - Tokens still immutable
  - Store output tokens in a circular buffer of token instances attached to the output port.
  - Each connected input port has a private read pointer on the corresponding output port's circular buffer.
- Limited Support for "run-time" data type polymorphism
  - Export type information as part of the compilation process for actors that need it.
  - Enables writing single implementation of actors like RecordAssembler
  - Type information can answer following about tokens/ports:
    - Size – in target words
    - Length – total # of elements
    - Dimension length
    - Number of dimensions
    - Array element type
    - Record member type
    - Record member offset
    - Is scalar, fixed length array, variable length array, record, union, …

Compilation - Target Actors

Atomic

- PTII "Front End"
  - Handles type resolution issues
  - Handles specialization issues
  - Uses proxy strategy to integrate back end into PTII environment.
- Target Specific "Back End"
  - create(), initialize(), prefire() and fire() code.
  - Java version as "reference"
- Test cases for Java reference are reused for other target back ends.

Composite with Director

- Support for TM, ADF, FSM, and SDF.
- Implemented as part of the target environment
- Composites without directors are removed during compilation.
- Same "interface" as atomic actors: create(), initialize(), prefire(), and fire().
- Compile-out some actors like: BusAssembler/Disassembler, ZeroDelay, SampleDelay, some RecordAssembler/ Disassemblers

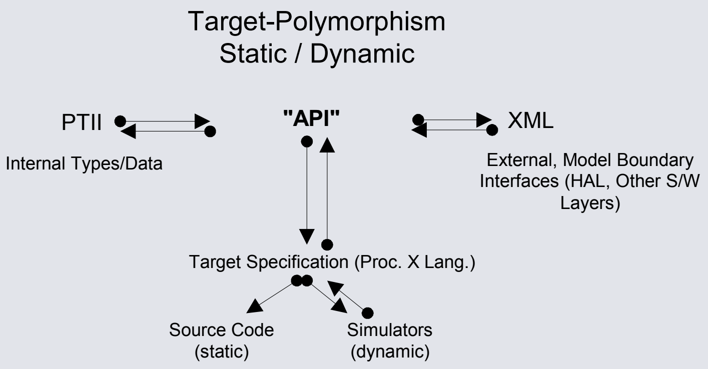
Compilation - Target Actor Trade-offs

- Granularity of atomic actors
  - Use application to guide development
  - E.g. Butterfly actor vs FFT actor.
- Specialization of atomic actors
  - Development time vs. runtime overhead.
  - Different targets can make different trade-offs
  - E.g. In add actor test overflow mode at runtime or create multiple specializations of add actor, one for each overflow mode. Use of a template strategy can help here.
- Appropriate array dimension handling
  - "Vector actors" "linearize" multi-dimensional arrays.
    Works well for element-by-element operations like add, multiply, etc.
  - Actor loop overheads vs. explicit dimension reduction/aggregation actors (and associated data copying)
  - E.g. Max actor with two dimensional input which is to act over "columns". Can create specialized actor implementation that contains double loop, or can explicitly convert two dimensional input array to a sequence of one dimensional arrays and then collect the scalar results back into a one dimensional output array.

Compilation – Actor Specialization

- Example: ALU (vectorized abs, add, subtract, multiply, negate…)
  - PTII/java. Specialized based on operation category/operand count
    - ALUBinary (Add,Subtract,Multiply…), ALUUnary (Abs, Negate,…)
      ALUUnaryWithParameter (Scale, Shift,…)
  - C: Additional specialization based on operation, port and parameter types:
    - ALUBinaryS1_15MultS1_15, ALUAddSW16
  - DSP Asm: Additional specialization based on rounding and overflow.
- Specialization logic part of PTII actor java code, queried by compiler, used for dynamic/static actor linking with target composite.

“API” – Data/Interface Specification



Target-Polymorphism
Static / Dynamic

PTII

Internal Types/Data

"API"

XML

External, Model Boundary
Interfaces (HAL, Other S/W
Layers)

Target Specification (Proc. X Lang.)

Source Code
(static)

Simulators
(dynamic)

“API” – cont’d

- API: abstract (target-independent) type, class, and instance data specification. Used for:
  - Model boundary interfaces (external s/w layers, HAL) – hand-coded
  - PTII types / tokens within the compilation domain (reflect)
- “Target” specification resolves abstract API attributes to target attributes (available integral type size and alignment properties, memory word-size, endianness) during compilation.
  - Sizeof, offsetof queries
- Exports to target source code (“static” compilation)
- PTII <-> target memory translations (“dynamic” compilation using target simulators loaded by PTII)

“API” – XML Elements

- <target> - list of applicable target specifications
- <include> - specification nesting, class-path relative

Scalars

- <int> - width, signed, value
- <real> - width, fractionalWidth, exponentWidth, signed, value
- <complex> - (real, imag) of <real> type, value
- <string> - traditional | hashed, value
- <enum> - (<member>)* - names only, value

Aggregates

- <array> - element type, dimensions – “fixed-length” arrays
- <vlarray> - (outer length, <array>) – “variable-length” arrays
- <struct> - (<member>)*
- <union> - ([selector,](<member>)*)
- <function> - (<inputs>, <outputs>)
- <class> - (all of the above)

Testing Approach

- Ptolemy-embedded: PTII model with contained “Composite-Under-Test” automatically iterated over set of target environments (PTII, C-Simulation, Asm-Simulation).
- Using jython to:
  - create test PTII “configuration”
  - load moml test models containing unit-under-test
  - compute test cases based on test variable sets (set1 X set2 X …)
  - set model test case parameters
  - run
  - report
- Device-embedded target environment: “Composite-Under-Test” linked with a target test shell to run in device under PTII model control (from a host, input/output ports “tunnelled” over comm. link)

Conclusions

- We've made good progress
- There's lot more to be done
- We must be crazy ☺