

AIRES: Automatic Integration of Reusable Embedded Software

Sponsor: DARPA IXO/MoBIES Program

Kang G. Shin
Real-Time Computing Laboratory
The University of Michigan
Ann Arbor, MI 48109, USA

MoBIES Program Goal



Create customizable frameworks that establish *composability* for:

1. temporal constraints
2. noise constraints
3. synchronization constraints
4. dependability constraints

Composability means that system level properties can be sufficiently and verifiably predicted from subsystem properties.

The Program in a Nutshell



The Hard Problem:

- Integration of embedded SW from **separately constructed parts** satisfying **physical** and computational constraints.
- The result must be:
 - **sufficiently** correct and safe
 - affordable at **small quantities**

The Solution:

Customizable, Reusable Frameworks that resolve cross-cutting physical constraints

Approach:

- Multiple-View Modeling of Physical Constraints
- Model-Based Generation Technology
- Framework Composition

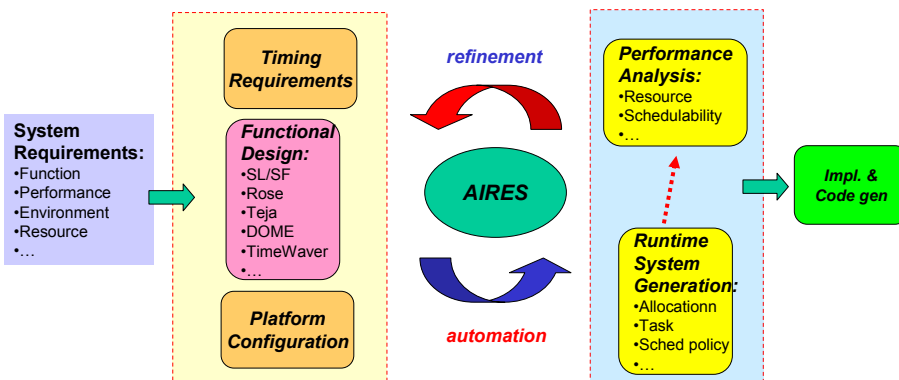
From Janos Sztipanovits' Presentation



AIRES in MoBIES

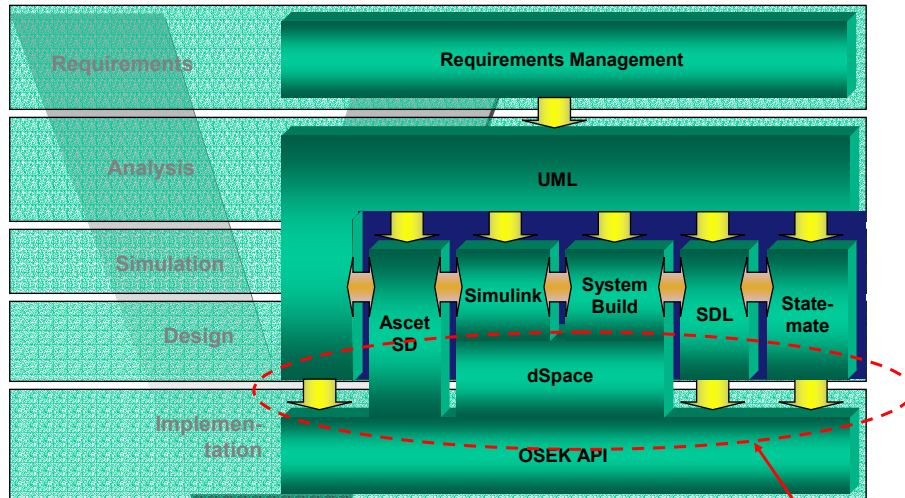


- AIRES addresses model-based generation and framework coupling issues
 - Generate runtime properties
 - Integrate performance/timing model with functional model





Challenge Problems Toolchain Topology



Source: automotive OEP presentation

AIRÉS focus



Research Objectives



- **Automatic mapping** between views subject to non-functional constraints
 - *Build* an application model with reusable components
 - *Construct* runtime model from structural model
 - *Generate* and *allocate* timing attributes for implementation
- **Integration** of timing analysis with functional design model and platform design
 - *Early-phase* performance analysis
 - System+performance modeling in *different* design phases
 - *Performance-aware* system design and refinement



Research Issues Addressed



- **Model integration**
 - Timing/performance model integrated with functional models
 - Integration at the same layer
 - Performance modeling of components
- **Model transformation**
 - Transform a high-level design model (functional description) to a runtime model (implementation description)
 - Multiple constraints transformation
- **System analysis**
 - Timing analysis with abstract design model
 - Distributed timing analysis with different scheduling policies
 - Obtain component performance values
- **Design process improvement**
 - Feedback analysis results to different design stages
 - Recommendations to alter design parameters
 - Performance-aware design



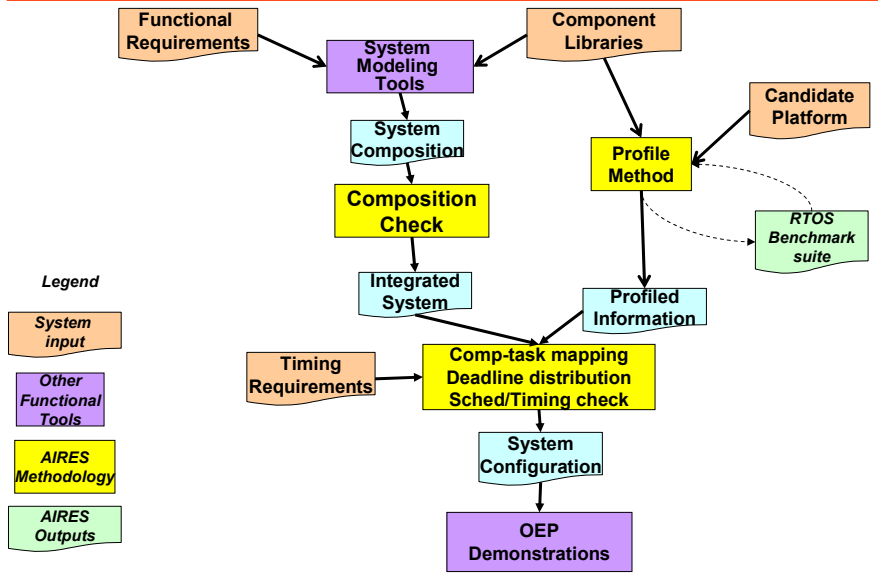
Challenge Problems for Industry



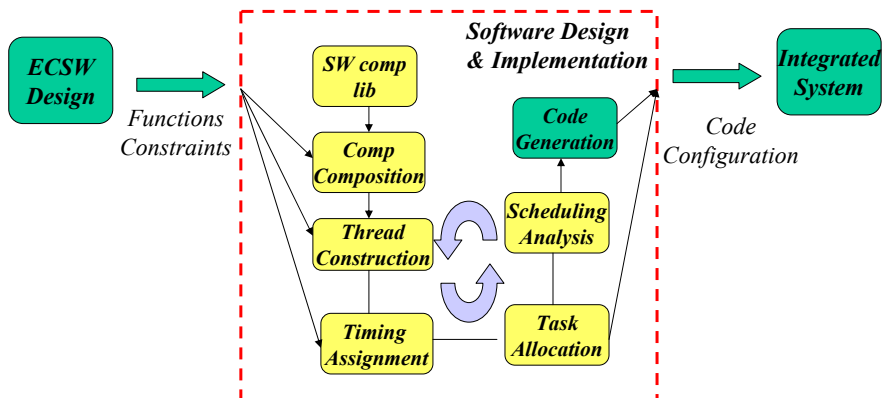
- **Target domain: Embedded Control**
 - Avionics
 - Automotive
 - Machine tool control/robotics
- **Application characteristics**
 - Large number of software components
 - Small number of devices/resources
 - Different from sensor network
- **Avionics apps**
 - Event dependency analysis
 - Schedulability checking
 - Unused and trapped resource
 - Task/processor utilizations
 - Timing constraints checking
 - Frame overruns
 - Queue buildups
- **Automotive apps**
 - Integrating with current industry tools
 - System composition and checking
 - Mapping between views
 - Schedulability and timing constraints check



Approach Overview



Approach Overview: Development process





Model Integration



- **Goal:**
 - Model different aspects of the designed software
 - Support trace-ability of model on error detection
- **Functional model:** graphs
 - Nodes: subsystems/components
 - Edges: dependencies/communications
- **Platform model:** a set of resource bins
 - Computation resource
 - Communication resource
 - Storage resource
- **Performance model** as a set of mathematical equations



Model Integration

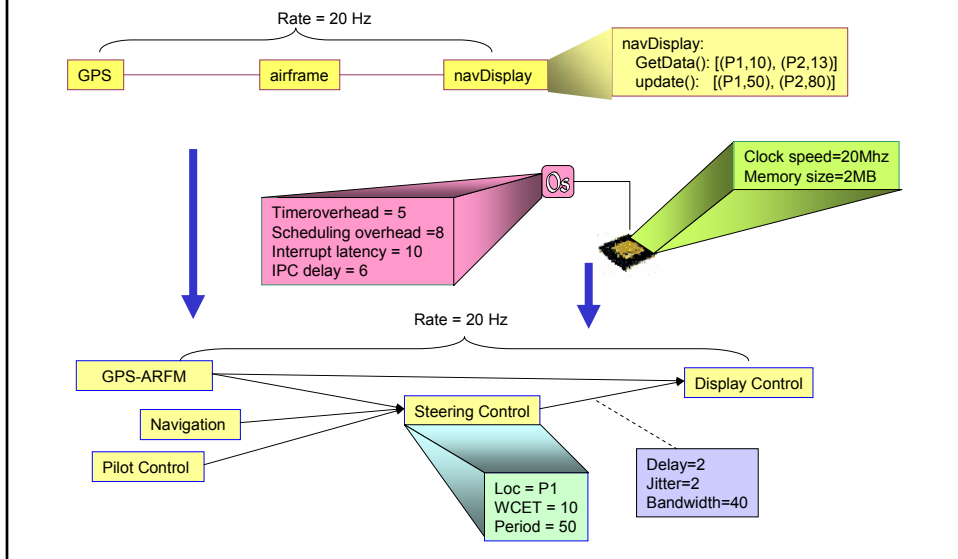
basic approach



- **Meta-model** defines the elements and relations between different views
- **Performance:** using annotations for both functional model and platform model
 - Functional: performance parameters
 - Both components and connections
 - Both requirements and characteristics
 - Platform: capacities and overheads
- **Platform:** container for functional blocks
- **Effects of any change of one aspect on other aspects can be evaluated**
 - Replacing component *A* in functional model results in performance failure (delay increased by Δ)
 - Using elements *A* in platform provides addition *x* more resource with *y* more cost



Model Integration example



Model Transformation



- Support design automation
 - Distributing software components to execution locations
 - Forming OS-level threads/tasks
 - Assigning proper execution timing properties for tasks
- Transform with consideration of system constraints
 - Timing: end-to-end delay
 - Resource: computation, communication, storage, power, etc.
 - Location: I/O-constrained

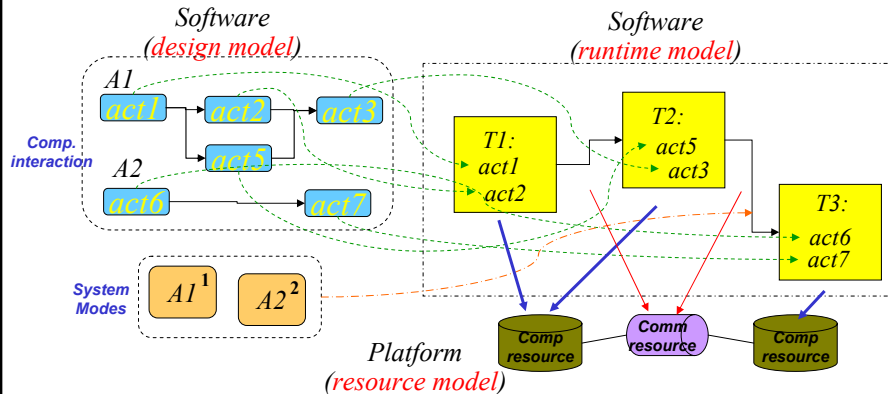


Model Transformation

multi-level system architecture



- System model contains:
 - Software component model
 - Application software model (integration of components)
 - Platform model (OS, HW, NW)
 - Runtime model for implementation (task graph)
- Multi-level modeling



Model Transformation

formal problem statement



- **Problem: Multi-constrained graph transformation problem**
- **Structural model:**
 - $G = \langle Vs, Es \rangle$ where nodes in Vs denote actions, and Es dependencies.
 - A set of end-to-end timing constraints $\{d_i\}$.
- **Runtime model:**
 - $G' = \langle Vr, Er \rangle$ where nodes in Vr denote tasks, and edges inter-task communications. All $v_r \in Vr$ and $e_s \in Es$ are allocated on some resource r_i with timing attributes assigned
- **Platform model:**
 - A set of resource $\{r_i\}$, whose capacity is bounded by $\{c_i\}$

Runtime model construction problem:

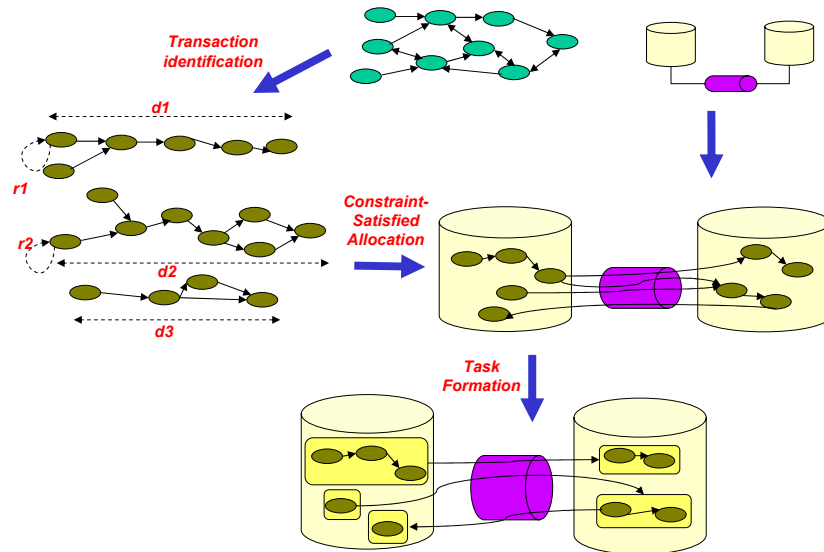
Given G , $\{d_i\}$ and $\{r_i\}$, find a runtime model

G' so that:

- Every node $v_s \in Vs$ is contained in one and only one node $v_r \in Vr$ in G'
- For any $e_s = (v_{s1}, v_{s2}) \in Es$, there exists $e_r = (v_{r1}, v_{r2}) \in Er$
- Every d_i that are satisfiable for G are also met in G'
- No r_i with a usage $u_i > c_i$



Model Transformation approach



Model Transformation algorithm description



- **Given:**
 - Component actions (functions) as basic units in structural model
 - Execution order is defined in structural model according to StateFlow
 - Constraints and partition criteria
 - Platform information
- **Basic idea:**
 - Identify the execution path (transactions) according to system properties (mode, rate, dependencies)
 - Allocate actions in a transaction on the same processor (rate similarity)
 - Partition a system thread that meets all constraints if not fit on a processor (keeping relative action order)
 - Adjust *partial order* of actions from different transactions to meet the end-to-end timing constraints
 - Execution order is altered by altering the priority of actions

Algorithm: *Comp2Task*

Input: structural model G , platform P , constraints C

Output: task graph G'

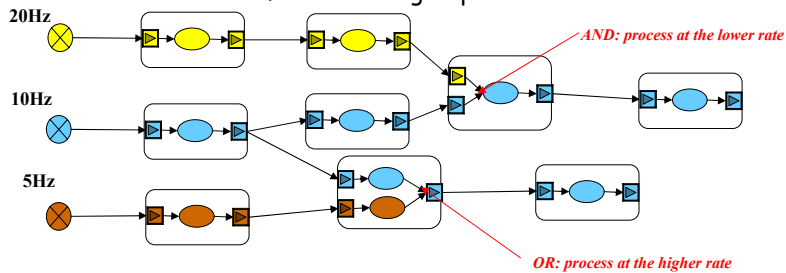
1. Identify the end-to-end transaction ST in G by traversing G from nodes of inputs to nodes of outputs;
2. Allocate v_s in ST to processors in P according to given allocation policy, e.g., first-fit, load-balancing
3. Assign priorities to all partitions according to execution order so that C can be satisfied
4. Group v_s allocated on a processor with the same priority together to form a v_r in G'
5. Create e_r between v_r if e_s between v_s in different partitions exists



Model Transformation transaction identification



- Assumptions:
 - Components in the same transaction are dependent and with same rate
 - Communications between different transactions are asynchronous
- Transactions are determined based on
 - Rate similarity
 - Information dependencies
- Algorithm:
 - Forward trace: initial assignment
 - Backward trace: finalize rate group



Model Transformation component allocation



- Multiple constraint satisfaction problem like a usual task-processor allocation



Model Transformation task formation



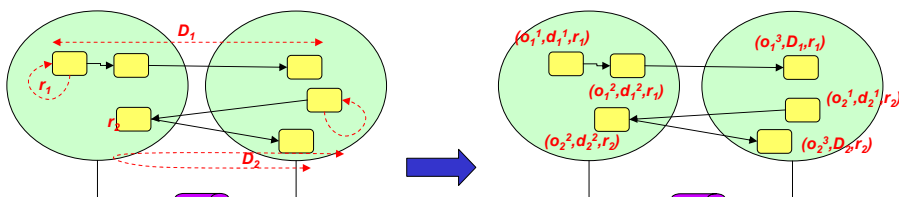
- Generate OS-level tasks: processes/threads
 - Each task contains a sequence of function calls
 - Reduce number of tasks to save system resources (context switch during runtime and TCP entries in memory)
- Dependent task graph is generated by
 - Grouping components with direct dependencies on the same processor to form a task
 - Grouping independent tasks with the same rate
 - Creating links between tasks if components have dependencies



Model Transformation timing assignment



- Assign timing properties for tasks for scheduling
 - Traditional scheduling policies (RM, EDF) can be applied
- Assignment based on deadline distribution
 - Deadline distribution with known communications





Model Transformation deadline distribution



- **Basic idea:**

- Find the longest execution path for each system thread
- Calculate the total execution time along the longest execution path (including network delays)
- Compute laxity for each constraint
- Distribute the laxity according to some policy
 - Proportionally to WCET
 - Equally

Algorithm: Deadline_Distribution

input: Graph T , E2E Constraints $\{C_i\}$,
RateConstraints $\{r_i\}$,
WCET for each node $\{e_i\}$
output: d_i and r_i for each node in T

```

sort  $\{C_i\}$  to  $C_1 \leq \dots \leq C_n$ ;
for  $C_i$  from  $C_n$  to  $C_1$  {
  find longest path  $P$  of thread with  $C_i$ ;
  if  $T_i$  has  $r_i$  and  $r_i \neq C_i$ ,  $e_i = r_i$ ;
  calculate laxity  $S$  along  $P$ ;
  add other nodes  $T_x$  not on  $P$ ;
  distributed  $S$  to  $T_i$  along  $P$ 
}
if exist  $T_i$  subject to multiple  $C_i$ 
  released time  $s_i = \max(s)$ 
  deadline  $d_i = \min(d)$ 
if no overlap or  $\text{overlap} < e_i$ , return fail;
else return  $d_i$  and  $r_i$ 

```



Model Transformation dependency resolution



- **Motivation:** Improve scalability (if # of tasks is large)
- **Approach:** break dependencies using shared buffers

- **Problem statement**

- System A with dependent task set T
- System A' with independent task set T' and a set of shared buffers B
- A transformation $F: A \rightarrow A'$ so that
 - $\forall T_i \in T \Rightarrow \exists T_i' \in T'$
 - $\forall I_{ij}: T_i \rightarrow T_j \Rightarrow b_{ij} \in B$ and $T_i \rightarrow b_{ij}$ and $b_{ij} \rightarrow T_j$
 - End-to-end timing constraints are preserved

- **Issues**

- Polling period derivation
- Overhead reduction



Model Transformation dependency resolution



- Polling period is derived by

- Using equation

$$poll_{T_i} + r_{T_i} \leq d_{T_i} - s_{T_i} \Rightarrow poll_{T_i} \leq (d_{T_i} - s_{T_i} - r_{T_i}) \Rightarrow poll_{T_i} = (d_{T_i} - s_{T_i} - r_{T_i})$$

- Iteratively update

$$exc_{T_i} = poll_{T_i} + r_{T_i} - (d_{T_i} - s_{T_i}), poll_{T_i}^{new} = poll_{T_i} - \frac{exc_{T_i}}{2}$$

- Overhead reduction

- Maximize polling period \rightarrow reduce polling frequency
- Minimize # of tasks \rightarrow reduce # of context switch
- Increase size/# of buffers \rightarrow reduce shared resource contention



System Analysis



- Schedulability and timing analysis is required to verify the correctness after transformation

- Interferences among different transactions are ignored during the transformation

- Required analyses include

- Overall system analysis:
 - schedulable/non-schedulable
 - Meet/miss deadlines by how much
- Detailed runtime information
 - Response times and jitters for tasks and transactions
 - Resource consumptions: how much of each resource by which task/transaction/system software



System Analysis approach



- Analysis is based on busy period analysis
- For each processor:
 - HKL algorithm
- For distributed end-to-end task chain
 - Synchronization protocol: direct synchronize or phase modification



System Analysis obtaining performance info



- Performance information of individual system component can be obtained through measurement
- Profiling can be used for application components
 - Profiled on some candidate platform
 - Info can be converted to normalized ones based on virtual resource with virtual service rate (scalar)
 - Normalized info can be stored with components for early design analysis
- Underlying system (OS, middleware) can be measured using benchmarks



System Analysis

OS overhead measurement

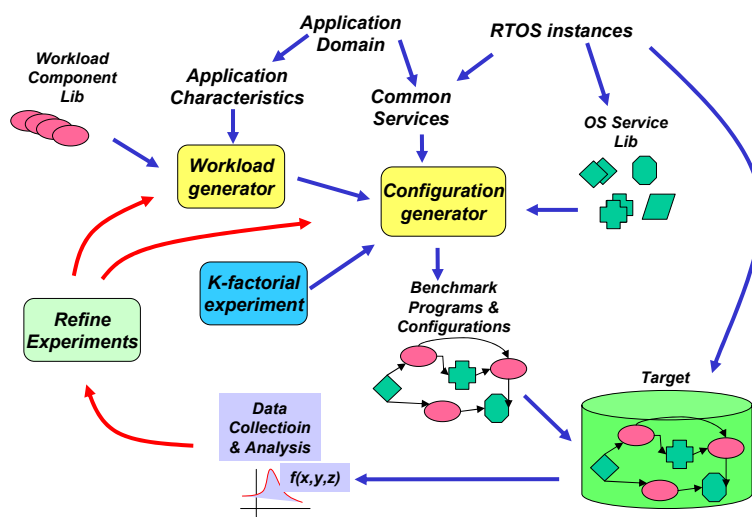


- Measurement method: end-to-end measurements
 - Synthetic workload: workload generation of interested domain
 - Microbenchmark: measurement generation for interested services
- Service dependencies can be detected
- Measured performance is OS-configuration independent, but is hardware-OS dependent



System Analysis

OS overhead measurement





System Analysis iterative measurement



- Detect interference between OS services
- Separate the services into groups based on service dependencies
- Change configurations to measure service performance with consideration of interferences
- Performance derivation:

$$\begin{pmatrix} m_1 \\ m_2 \\ \dots \\ m_n \end{pmatrix} = \begin{pmatrix} f_{s1} \\ f_{s2} \\ \dots \\ f_{sn} \end{pmatrix} (p_1, p_2, \dots, p_k) \Rightarrow PF = \begin{cases} pf_{s1} = f_{s1}(p_1, p_2, \dots, p_k) & \text{service } s_1 \\ pf_{s2} = f_{s2}(p_1, p_2, \dots, p_k) & \text{service } s_2 \\ \dots & \dots \\ pf_{sn} = f_{sn}(p_1, p_2, \dots, p_k) & \text{service } s_n \end{cases}$$

- Application of OS performance

e.g., given application design (known workload+required services), utilization is:

$$o_{ij} = f_{sj}(p_1, p_2, \dots, p_k), \quad U = \sum (e_i + \sum o_{ij}) / per_i$$



System Analysis RTOS benchmark suite



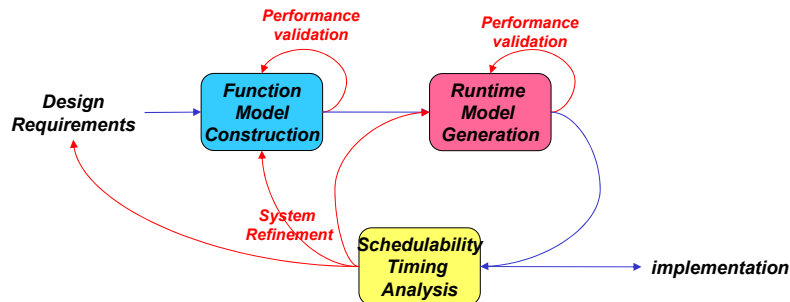
- A set of pre-implemented program library
 - OS specific: QNX, RTLinux, OSEKWorks
- A generator to construct benchmark experiments
 - User need to specify interested services and parameters, e.g., workloads, other services
- Data collection and analysis is still a manual process



Design Process Improvement



- Analysis becomes model-based instead of simulation-based
 - Enable early design-phase analysis
- Close-loop design automation is possible
 - Auto design refinement can be supported



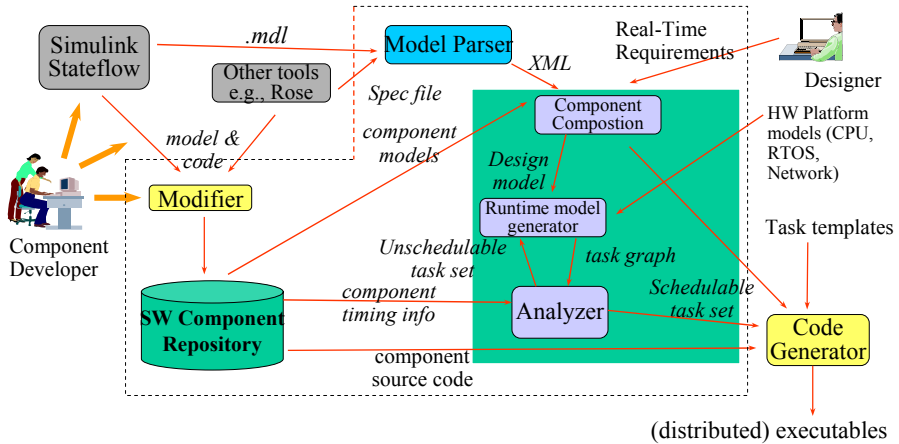
Design Process Improvement system refinement



- Refinement approaches:
 - **Adjust scheduling policies** (priority, scheduling algorithm, etc)
 - Cheapest, no additional cost
 - **Adjust timing assignments** (period, deadline, etc.)
 - Moderate cost, subject to controller design
 - Could lower control quality (under-sampling) or over-consume resource (over-sampling)
 - **Change software model** (component, interaction, etc.)
 - Expensive, subject to constraints such as code certification
 - Involve human being, take longer cycle
 - **Change platform configuration** (hardware, OS, etc)
 - Expensive, introducing additional cost
- Different approaches are suitable for different refinements



Integration in AIRES Toolkit



References



- <http://kabru.eecs.umich.edu/aires>