

ARTIST OVERVIEW

Introduction

The following figure provides an overview of the ARTIST workflow. Software and physical system models may be defined or extracted from state-space models, non-UML declarative models, UML declarative models, and other declarative model documents. These models may be composed to form complete executable products within a generic modeling language representation (UML).

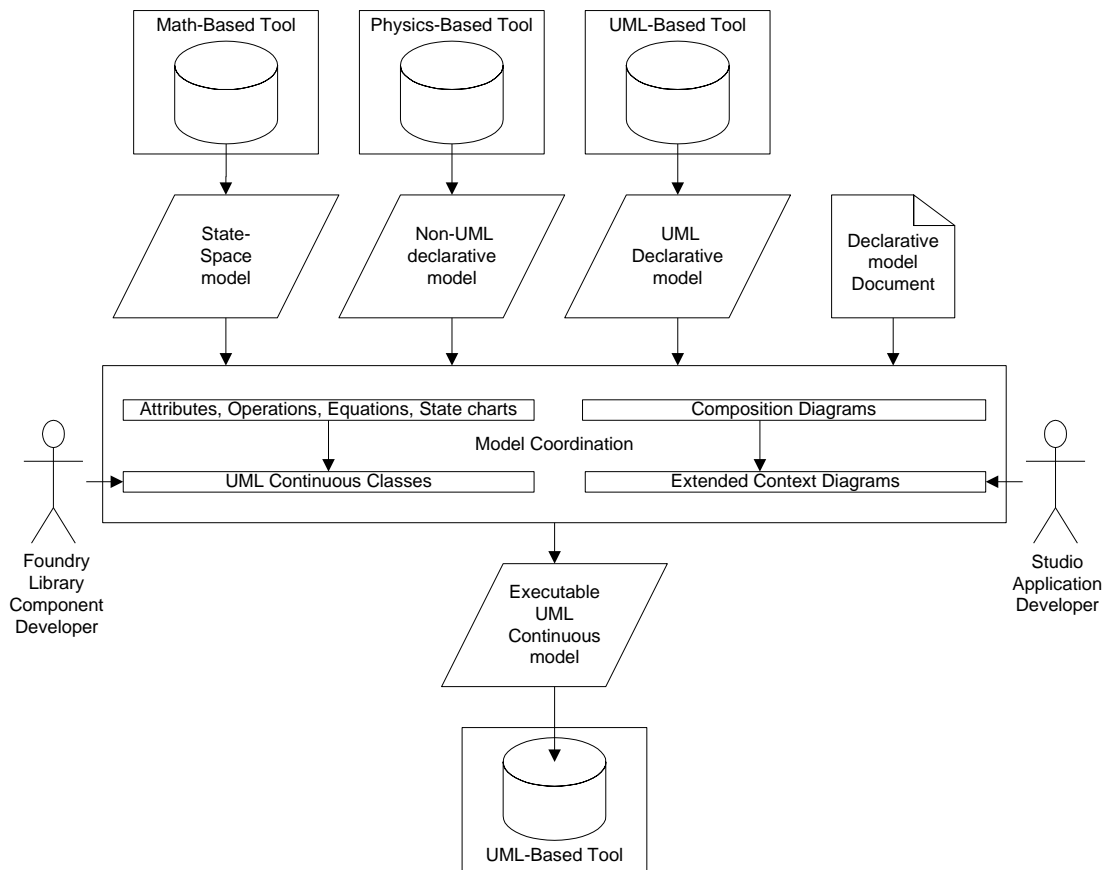


Figure 1 Artist workflow overview

The generic modeling language representation of a complete product enables software programmers to mix, within a single object-oriented class or package, different aspects of a single product element, including algorithmic aspects (control and digital signal processing), network communication aspects, IO software aspects (sensor and actuator device drivers), physical interaction aspects (dynamics and conservation laws), simulation aspects, and documentation aspects. As a result, the user is relieved from representing the different aspects of the software and systems elements in different tools and repositories. Hence, offering a unified view of the overall embedded product within a single modeling environment.

Consider for example the generic language model of a radar sensor in an adaptive cruise control product (example: BMW's Series 7 automobile). The radar sensor object is a publisher on the CAN/TTP bus (communications), it handles the RTOS timer interrupts (data acquisition system), it processes radar signals to compute distance to leading vehicle (supervisory control algorithm), it is a part of the active throttle and ABS control system (low level control), and is part of a physical system that includes the vehicle (transmission data, acceleration data, steering data, etc.), the lead vehicle and road dynamics (physical).

ARTIST enables the user to represent all the aspects of the above-mentioned radar sensor within a single class or package and within a single modeling tool. Today, the different aspects must be represented using different distinct languages within different modeling tools.

This document provides an overview of the features, and works through several examples to illustrate how users can design and integrate software and simulation models for software validation, design optimization, advanced controls, and parameter estimation.

ARTIST includes a collection of engines and tools for modeling large, complex, and heterogeneous physical systems within the UML modeling environment. ARTIST does not impose yet another modeling language on the end user.

ARTIST also includes a product integration environment in the form of an extended context diagram editor and a variety of model generators. This environment is used for composition of discrete software and continuous physical systems. A communication layer is under development intended to support resource allocation transparency, communication, and distributed simulation.

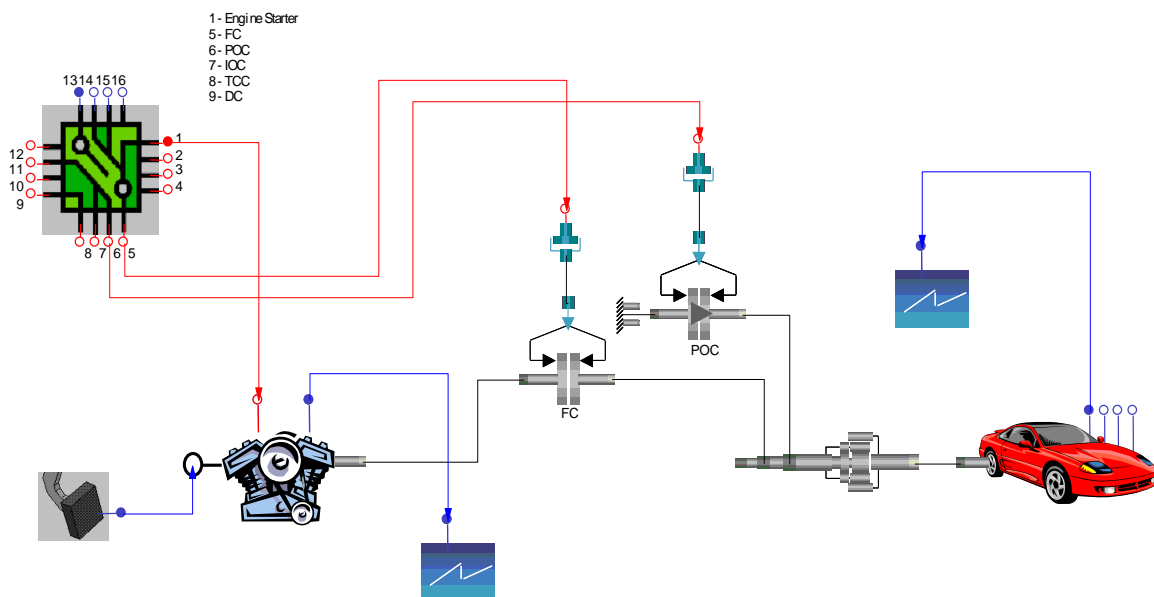


Figure 2 Extended context diagram (simple vehicle)

ARTIST is suited for multi-domain modeling of complex products involving software communicating with automotive, aerospace, mechanical, electrical, and chemical processes. ARTIST enables engineers to assemble discrete software and continuous system components in a natural way. Real world components like fuel injection controllers, gears, clutches, and engines are inter-connected through discrete data and events, and physics-based connections. Sub-systems can be aggregated and encapsulated simplifying the construction and maintenance of large complex systems.

Physical system models are described by continuous differential, algebraic, and discontinuous equations. Equations do not have to be manipulated manually to isolate any variable. Third-party libraries can be integrated into equation calculations.

Reuse is a key requirement for software developers. ARTIST models can be reused in any context or application. A gearbox model can be reused in any transmission. A transmission model can be used for system or software parameter estimation, software validation, Monte Carlo simulation, model-reference control, or parameter optimization. All object-oriented reuse concepts including inheritance, aggregation, and encapsulation can be applied to system and software models. System models sharing common equations, state machines, and / or connectivity requirements can derive from the same base model. A clutch, one-way clutch, and band-brake may share common equations, a common state machine, and common connectivity defined in a base clutch class.

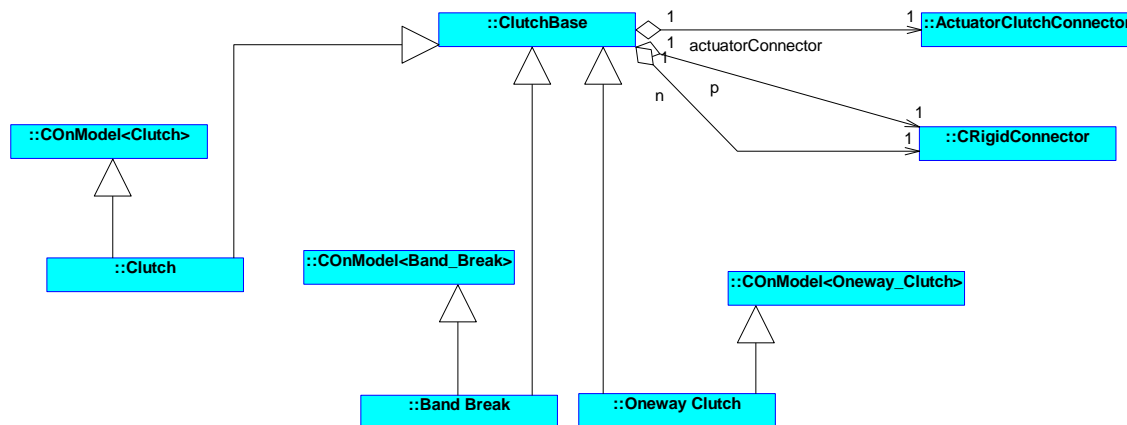


Figure 3 System modeling using OO reuse (clutch inheritance, connector aggregation)

The ability to build complex composite models from simple ones not only keeps software complexity manageable, but also provides a pragmatic development and validation path. Simple models can be tested, validated and parameterized from lab data. The validation of complex models is greatly simplified because the constituent components are already proven. In addition, model parameterization data may be available for the constituent parts of a new component long before a working prototype is available. Calibration can begin as early as the prototype stage of a design.

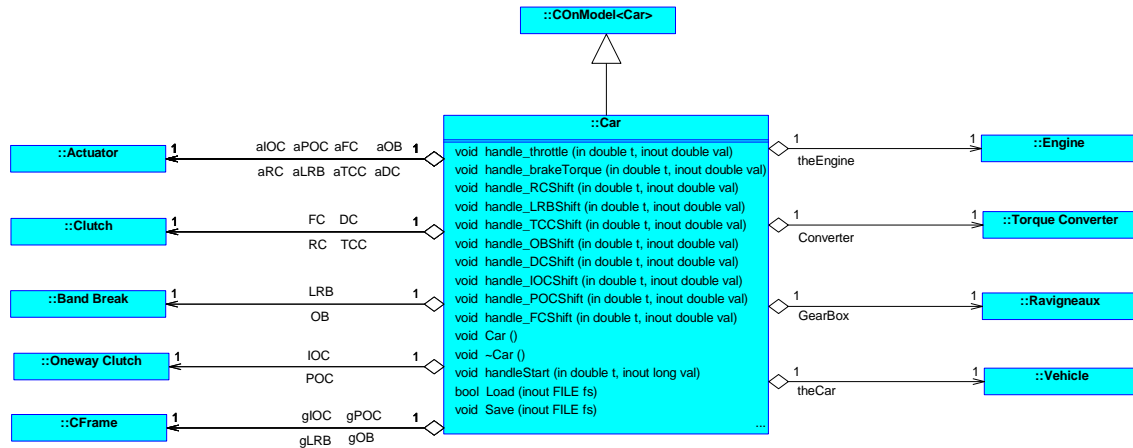


Figure 4 Composite system model

Capabilities of the UML modeling environment such as code generation, state machine simulation, and automatically generated test vectors can be applied to discrete software and physical system models as well as software components. Further, features of the debugging environment for software components can also be used for debugging system components.

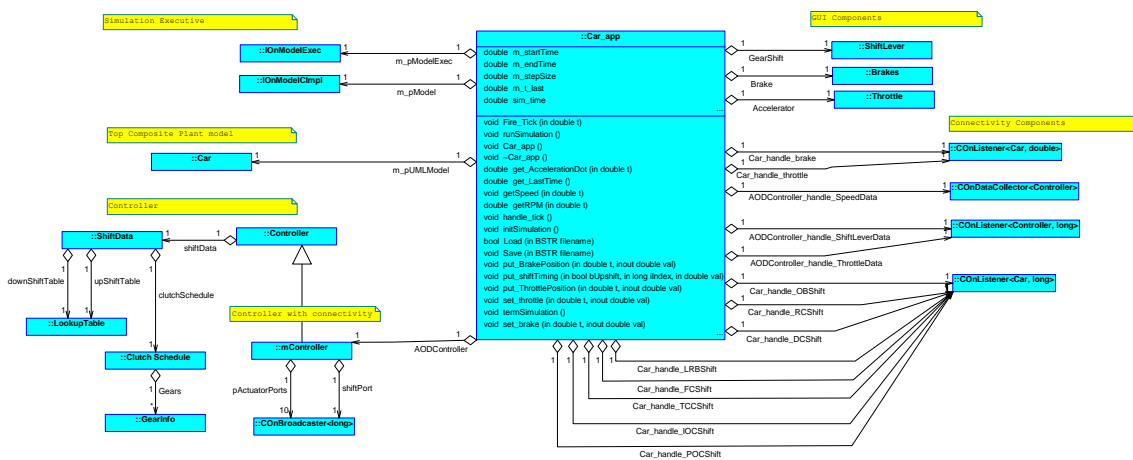


Figure 5 Application test harness composition

Feature overview

ARTIST supports both high level modeling by composition and detailed library component modeling by equations. Models of standard components are available in model libraries. A composite model can be built graphically by dropping symbols representing standard models and their connectors on an extended context diagram, and drawing connections between the models. The composite model can be constructed at run-time or design time (an aggregate UML model). The ARTIST extended context diagram editor is available as an ActiveX control for easy integration with most software development environments.

Composite model components can be distributed across multiple computing nodes, and the inter-node connections are transformed automatically to run efficiently in the distributed environment. This distributed computing capability provides scalability for complex HIL applications without sacrificing accuracy.

A typical composition diagram is illustrated in the following figure.

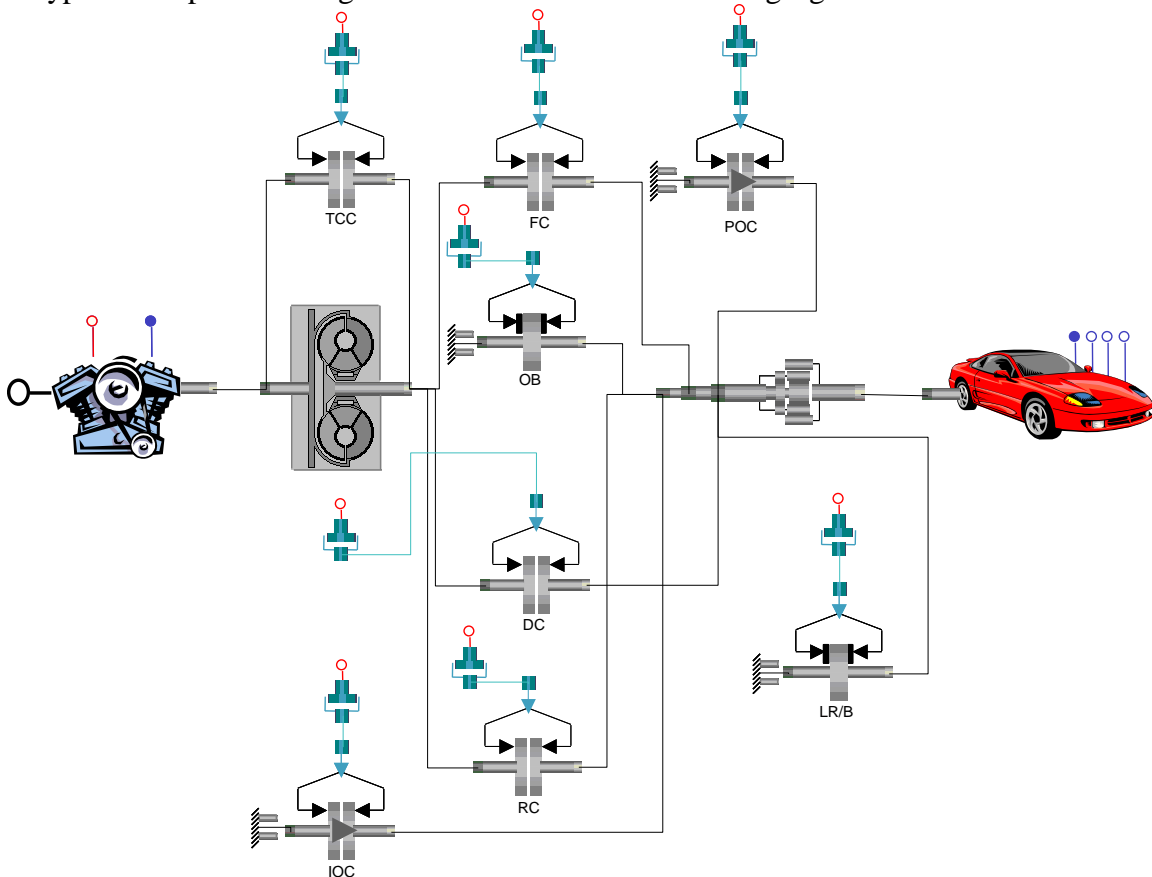


Figure 6 Example automatic transmission composition

Developers do not have to leave the problem domain during standard or composition model development. The system can be broken into a set of connected components: an engine, a series of clutches, a gearbox, and a vehicle. Connectivity is intuitive. In the case of sensors and actuators particularly, object often have many aspects. Any number of discrete and continuous aspects may be integrated into a single class or package.

The class representation of the composite car model of Figure 6 is as follows:

```
class car : public ARTISTModel
{
    attributes:
        Engine* engine;
        OnewayClutch* IOC;
        ...
    operations:
        initModels();
        initConnections();
}
```

The aggregated models become attributes (or associations), and the sub model initialization and connection initialization are captured in simple code generated initialization functions.

Connections specify the interaction between models, and are represented graphically as lines. There are 3 types of connectors defined in ARTIST:

- Physics-based (non-directional)
- Event – invoke an event handler function on the object
- Data acquisition – read a collection of data typically from a system model.

The following figure illustrates the 3 types of connections. Black lines represent bi-directional physics-based connections. Red lines represent directed event connections. Blue lines represent data acquisition connections.

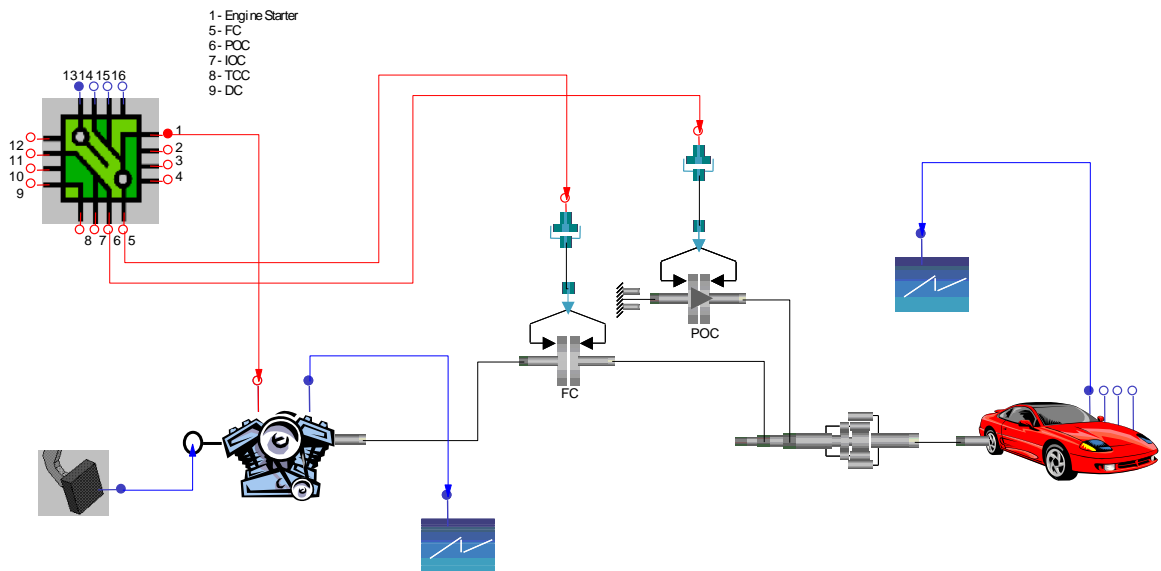


Figure 7 Simplified automatic transmission problem (neutral and first gear only)

Integrating multiple controllers and associated plant models into a common test harness per the following figure enables the user to perform coordinated testing of interacting control logic components. Each control logic component is defined independent of its target hardware.

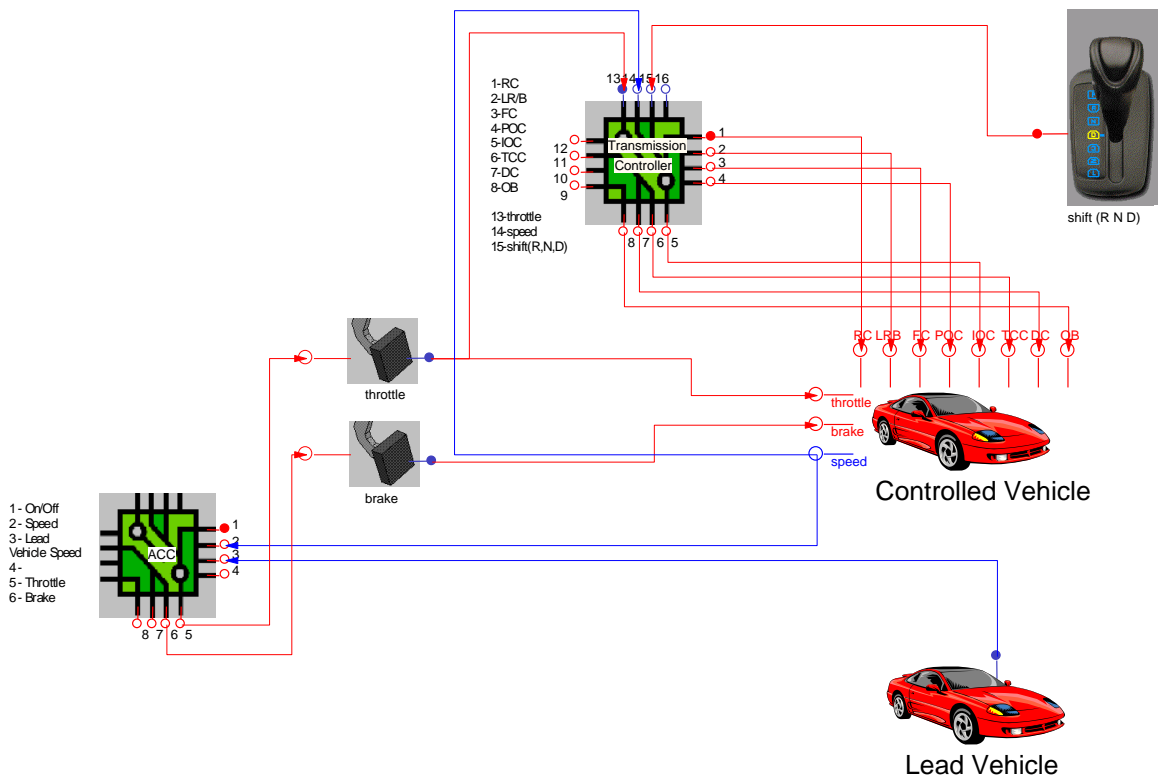


Figure 8 Multiple controller test harness

An expanded view of the transmission controller above appears as follows:

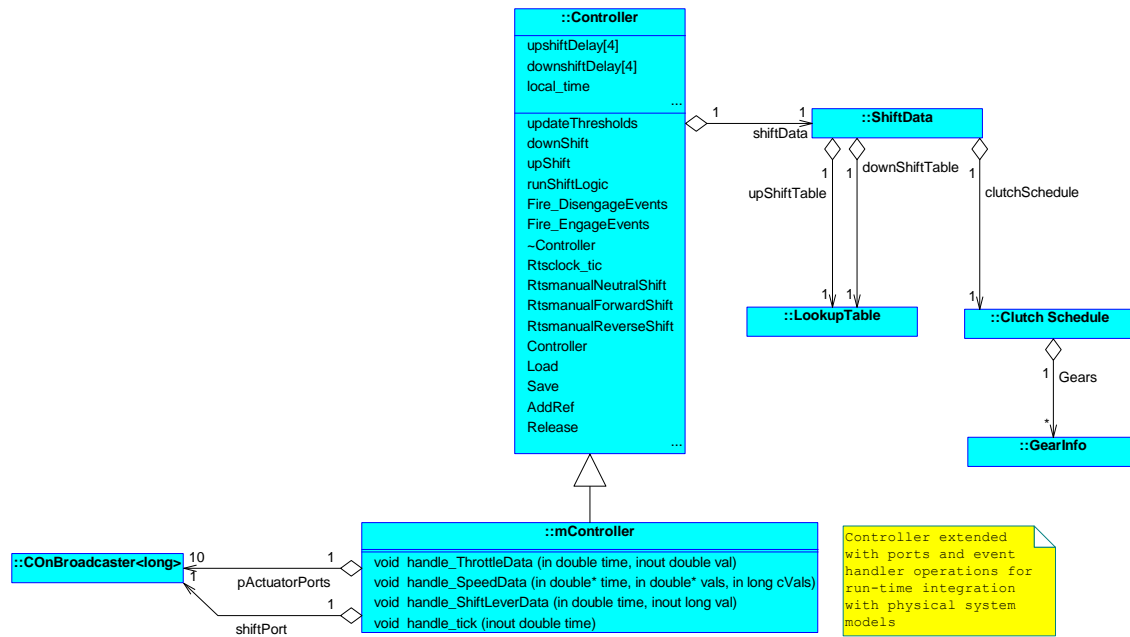


Figure 9 Transmission controller

The expanded view of the adaptive cruise controller is as follows:

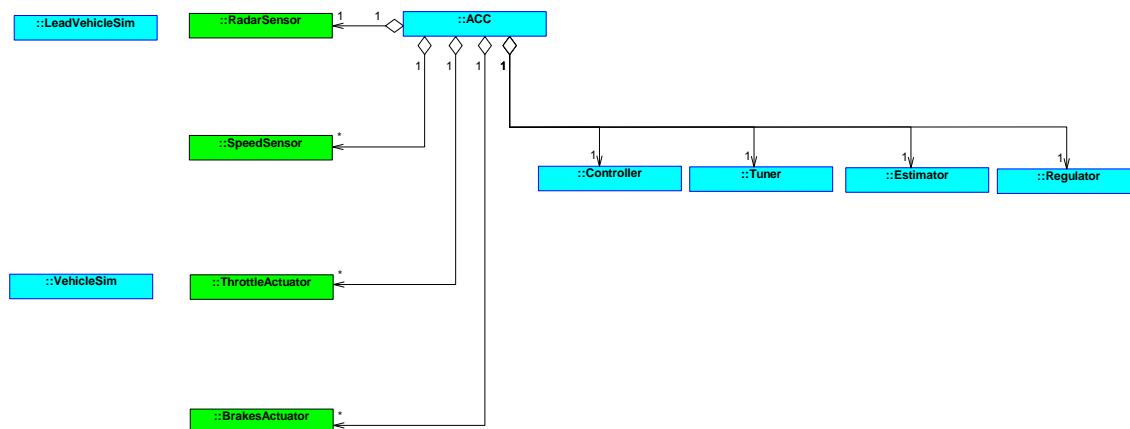


Figure 10 Adaptive cruise controller

In the automotive examples illustrated in the above figures, there are two types of physics-based connectors required: rigid shaft connectors for the drive train components, and hydraulic pressure connectors for the clutch actuators.

Physics-based connections impose physics-based algebraic constraints between objects. A rigid shaft connection would define torque and acceleration constraints ($\sum T_i = 0$, $A_i = A_{i+1} = \dots A_n$). An electrical connection would define voltage and current constraints (V_i

$= V_{i+1} = \dots V_n, \sum I_i = 0$). A simple view of the connector models used in the components of the car is illustrated below:

```
class RigidConnector : public ARTISTConnector
{
    attributes:
        double torque;
        double velocity;
        double acceleration;

    operations:
        sum_torque();
        velocity_equal();
        acceleration_equal();
}

class HydraulicConnector : public ARTISTConnector
{
    attributes:
        double pressure;

    operations:
        sum_pressure();
}
```

Modeling features

Any system model in ARTIST can have:

- Connectors
- Variables
- Parameters
- Equations
- State machines
- Aggregated models
- Other software functionality

Connectors

A clutch model will have a hydraulic actuation connection, and 2 rigid shaft connections. The beginnings of the clutch model appear as follows:

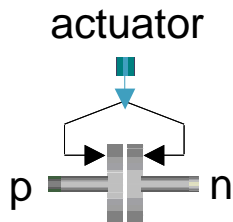
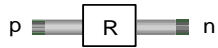


Figure 11 Connector example

```
class ClutchBase : ARTISTModel
{
    attributes:
        // connectors:
        RigidConnector* p;
        RigidConnector* n;
        HydraulicConnector* actuator;
}
```

```
operations:
    initModels();
```

A resistor will have two electrical pin connectors. A resistor model would appear as follows:



```
class resistor : ARTISTModel

attributes:
    // connectors:
    ElectricalConnector* p;
    ElectricalConnector* n;

operations:
    initModels();
```

Variables

The equations for the clutch must be expressed in terms of variables, parameters, and the variables of member models and connectors. Variables are typically defined as values that change quickly during the simulation; whose values are updated by a simulation engine. Some variables used in a clutch model might be as follows:

```
double relativeVelocity;
double displacement;
double cBreakfreeDisp;
double ccBreakfreeDisp;
```

Parameters

Parameters are typically defined as values that are either fixed, or change slowly during the simulation, where said changes are triggered by discrete events. Parameters in ARTIST models can be manipulated during simulation by external signals (from sampled sensor signals, or controller actuation signals), or an ARTIST optimization engine.

The parameters for the clutch would be as follows:

```
double torqueRatio;
double damping;
double area;
double radius;
double pressureMax;
```

Parameters for the resistor would be as follows:

```
double R;
```

The distinction between variable and parameters is artificial, and is provided as a convenience to the user. The variable / parameter designation may change depending on the application of a model.

Equations

One or many equations may be defined in a class operation. There are 3 types of equations defined in ARTIST:

- Continuous – equations that define the continuous behavior of an object in a given state. The calculation of viscous friction is a continuous equation.
- Discontinuous – equations that define conditional behavior using continuous math. Discontinuous equations are used for triggering state events, which trigger state transitions. Discontinuous equations like $(x > 100)$ are defined using continuous math $(x - 100)$ where any positive value indicates a true result.
- Reset – equations that define the continuous behaviour of an object during state transition. Reset equations are typically used to simplify complex behavior, or apply physics-based constraints that would apply during state transition. A bouncing ball could have equations defining the compression of the ball on impact with another object. A simplification is to define a reset equation that resets the velocity variable ($\text{velocity} = -1.0 * \text{stiffness} * \text{old}(\text{velocity})$).

The equations for the resistor are as follows:

$$\begin{aligned} p.V - n.V &= p.I * R \\ p.I &= n.I \end{aligned}$$

State machines

ARTIST uses standard UML state machines for defining the physical states of hybrid models.

A clutch has 3 physical states, in which different friction equations apply. A state machine is added to the clutch model, and the common and state-dependent behavior equations are added.

The state machine for the clutch, as defined in UML, is as follows:

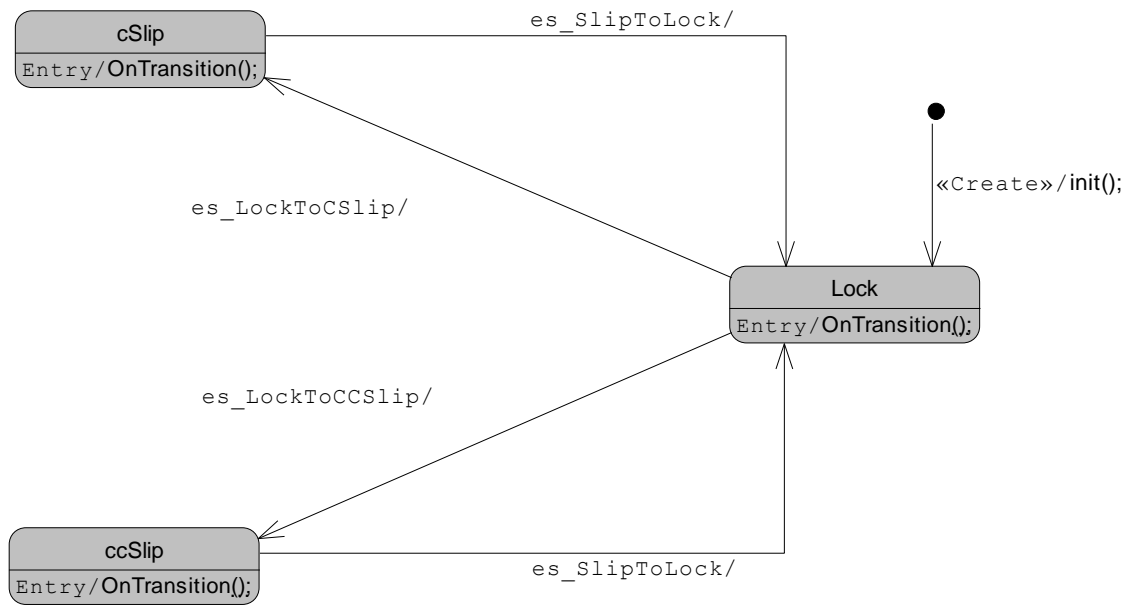


Figure 12 State machine example (clutch)

The equations for the continuous behavior of the clutch in its different states are defined as follows:

```

// common behavior:
r_clockwiseBreakfreeDisp();
r_counterclockwiseBreakfreeDisp();
r_forceBalance();
r_relativeVelocity();

// ccSlip continuous behavior:
r_ccSlipDisplacement();
r_ccSlipFrictionTorque();

// cSlip continuous behaviour:
r_cSlipDisplacement();
r_cSlipFrictionTorque();

// lock continuous behaviour:
r_StickFrictionTorque();

```

The discontinuous and reset equations required for triggering state transitions, and reset variables during transitions are as follows:

Event	From state	To state	Equation	Equation type
es_SlipToLock	ccSlip	Lock	z_velocity_pos();	discontinuous
			s_ccLockDisplacement();	reset
es_SlipToLock	cSlip	Lock	z_velocity_neg();	discontinuous
			s_cLockDisplacement();	reset
es_LockToCCSlip	Lock	ccSlip	z_velocity_neg(); // AND	discontinuous
			z_disp_neg_slip();	discontinuous
es_LockToCSlip	Lock	cSlip	z_velocity_pos(); // AND	discontinuous
			z_disp_pos_slip();	discontinuous

The resulting clutch model is expanded to the following:

```
class ClutchBase : ARTISTModel

attributes:
// connectors:
  RigidConnector* p;
  RigidConnector* n;
  HydraulicConnector* actuator;

// variables:
  double relativeVelocity;
  double displacement;
  double cBreakfreeDisp;
  double ccBreakfreeDisp;

// parameters:
  double torqueRatio;
  double damping;
  double area;
  double radius;
  double pressureMax;

operations:
  initModels();

// common behavior:
  r_clockwiseBreakfreeDisp();
  r_counterclockwiseBreakfreeDisp();
  r_cSlipDisplacement();
  r_forceBalance();
  r_relativeVelocity();

// ccSlip continuous behavior:
  r_ccSlipFrictionTorque();

// cSlip continuous behaviour:
  r_cSlipFrictionTorque();

// lock continuous behaviour:
  r_StickFrictionTorque();

// discontinuity equations:
  z_velocity_pos();
  z_velocity_neg();
  s_ccLockDisplacement();
  s_cLockDisplacement();
  z_disp_neg_slip();
  z_disp_pos_slip();
```

Operations are named arbitrarily, and no causality information is required from the modeler.

ARTIST creates an executable physical model from the above collection of attributes and operations.

The state machine for a bouncing ball could be as simple as having 1 state (InAir) and one transition (bounce).

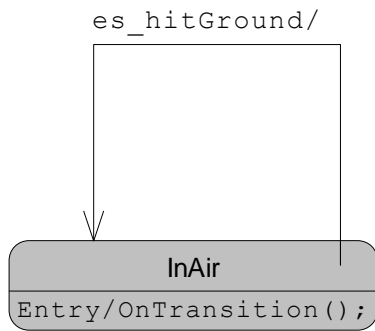


Figure 13 State machine example (bouncing ball)

The resulting model for a bouncing ball would appear as follows:

```

class BouncingBall : public ARTISTModel
{
    attributes:
        double height;
        double velocity;

    parameters:
        double gravity;
        double bounce_ratio;

    operations:
        // continuous equations:
        r_height();           der(height) - velocity;
        r_velocity();         der(velocity) - gravity;

        discontinuous equations:
        z_height_neg();       0.0 - height;

        reset equations:
        s_velocity();         velocity + bounce_ratio * old(velocity);
}
  
```

The discontinuous and reset equations required for triggering state transitions, and reset variables during transitions are as follows:

Event	From state	To state	Equation	Equation type
es_hitGround	InAir	InAir	z_height_neg();	discontinuous
			s_velocity();	reset

Simulation environment

ARTIST models can be written in C++, or any COM or .NET compliant language. The ARTIST engines and tools are all COM and .NET components. Application components that integrate system and software models, optimization, or parameter estimation can be written in any COM or .NET compliant language including C#, C++, Visual Basic, and VB Script.

All aspects of a component including discrete software, communication, and physical and electrical behaviour may be modeled in a single class or package. Consider the radar sensor example from the introduction.

All ARTIST system models, including standard and composite models, are arbitrary classes defined in UML. The models can therefore be included in non-executable UML constructs like object sequence diagrams. ARTIST provides test code generation tools based on object sequence diagrams. Additionally, test scenarios can be easily written manually using rapid development languages like Visual Basic.

The ARTIST optimization engines can be used to optimize control or system parameterization. All standard and composite system models support snapshot save and restore capabilities. The same capabilities can be easily added to software models as well, thus enabling Monte Carlo simulations and constrained optimizations to be run efficiently from a variety of initial conditions against a complete product or application.

All system models can respond to external events, parameterization changes, and can be queried for variable values at any time during simulation. Software components can intuitively interact directly with system sub-models, rather than with a global model, or an external simulation tool's API. A transmission controller for example can send events directly to the various clutch actuator models, and can collect data from any models.

When system and software models are written in the same language (typical case), the features of the language IDE (like Microsoft Visual Studio) can be used to debug both.

Model diagnostic features are underway such as querying for low-level behaviour defined on class instances and operations from the global behavioural model of a system.

Monte Carlo co-design example

An example co-design activity for a transmission controller is to vary the delay between disengaging the departing clutch and engaging the oncoming clutch, and monitor the clutch energy dissipation, and vehicle jerk. This form of testing enables the designer to quantitatively address qualitative requirements like maximize clutch life while minimizing vehicle jerk during shifts.

The simple transmission controller is enhanced with time delay states between shifts. The duration in the time delay states is a tunable parameter on the controller.

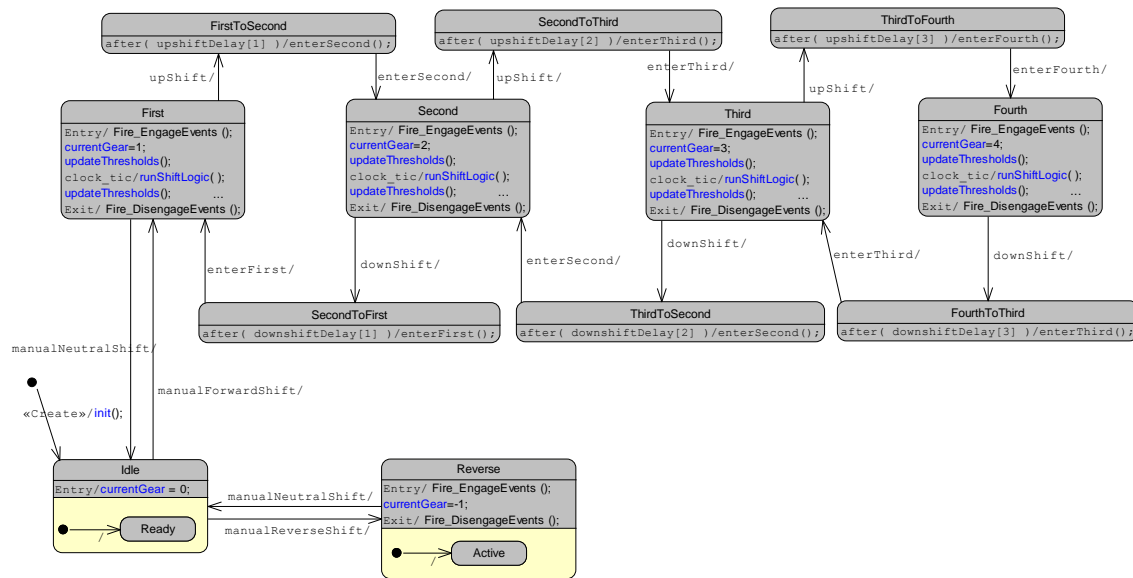


Figure 14 Transmission controller state machine (with tunable shift delays)

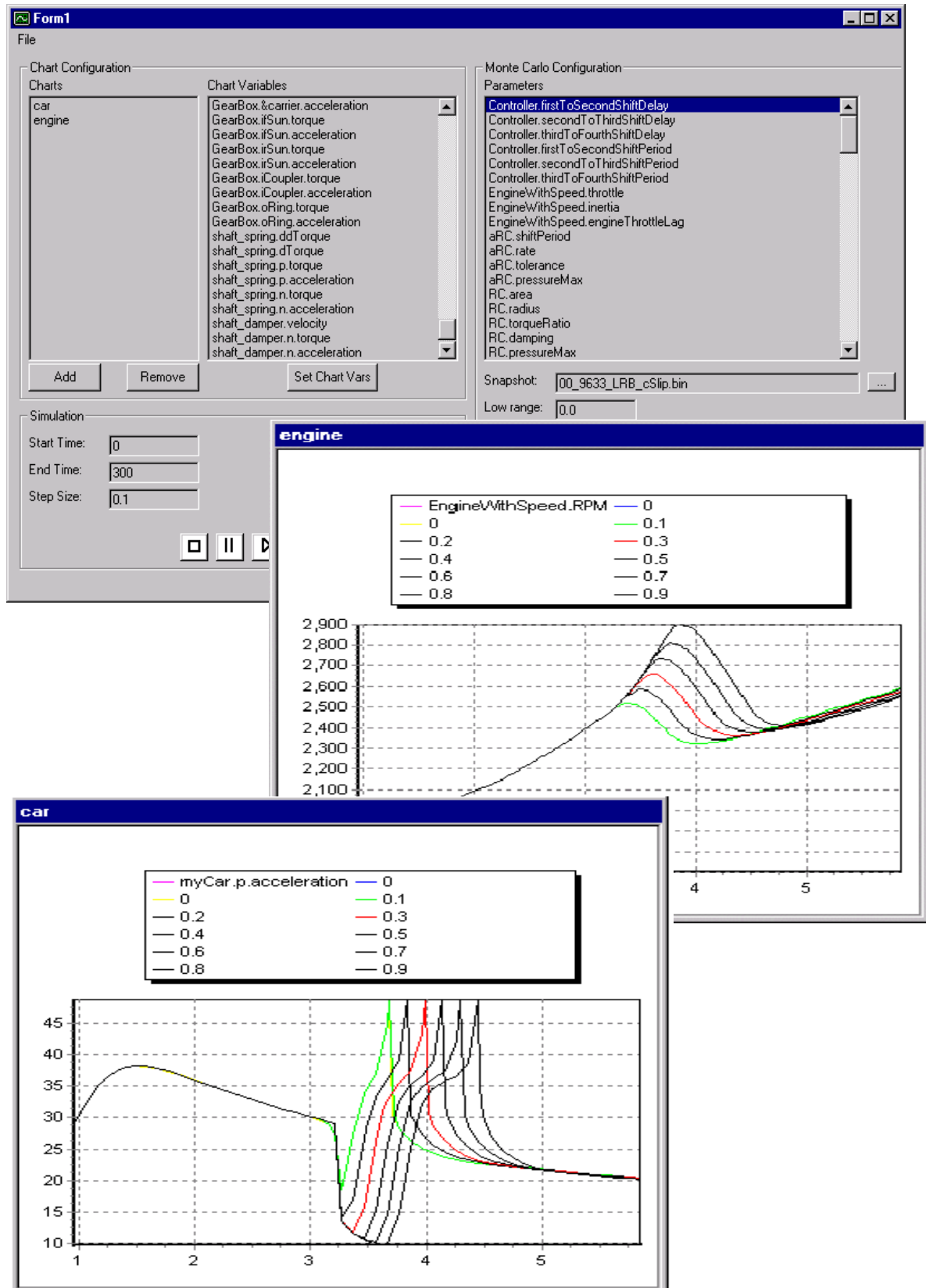
The Visual Basic script to execute the simple Monte Carlo simulation against the test harness (th) is as follows:

```

' iterate over increments
For inc = 0 To MCIncrements
    ' add a series (line) to each output chart
    For Each aChart In m_Charts
        aChart.addSeries CStr(val)
    Next
    ' load the starting point snapshot
    th.Load MCSnapshot
    ' update the parameter value
    SetMCPParameter inc, val
    simTime = MCstartTime
    MCEndTime = MCstartTime + MCRuntime
    ' the following loop is a simple clock master stepping through time
    While simTime < MCEndTime And Not bMCPause
        ' issue clock tick to test harness, which in turn issues ticks to all components
        ' including the model executive
        th.tick simTime
        ' update the charts with the latest data
        For Each aChart In m_Charts
            aChart.tick simTime
        Next
        ' advance time
        simTime = simTime + stepSize
        DoEvents
    Wend
Next inc

```

The result screenshots are shown in the following figures:



Model execution

ARTIST provides simulation management classes and model base or helper classes that minimize the requirements of end-user defined models to equation-based operations, attributes, and state machines.

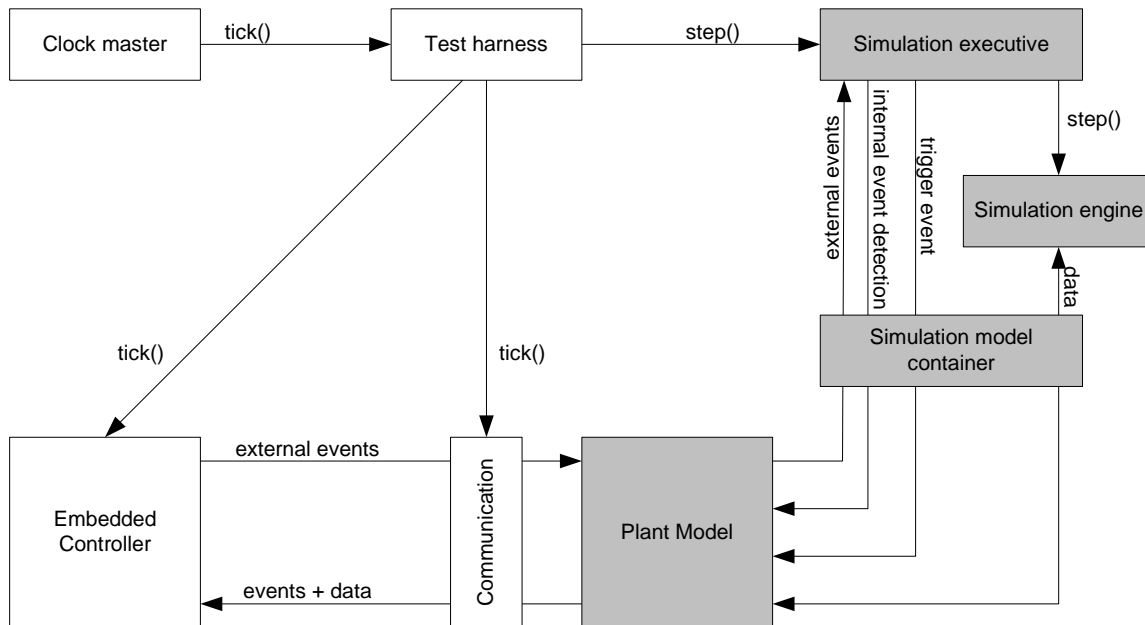


Figure 15 Simple test harness package implemented by ARTIST

The above figure illustrates a simple test harness package implemented by ARTIST test harness generation tools.

The plant model may be a user-defined model, or a composite model. Simulation model container components provide necessary additional model functionality to achieve an executable model. A simulation executive monitors events, and manages the event queue. A simulation engine, interacts with the plant model via the standard interfaces provided by the simulation model container, and updates variable values as simulation steps through time.

Model initialization involves organizing and validating equations, variables, and parameters. Numerical and automatic differentiation methods are used to determine equation to variable dependencies. Graph analysis is used to validate equations structurally. This analysis includes detection of hidden constraints, variable index calculations, and sorting of equations and variables into blocks for efficient calculation. Algebraic analysis is used to validate equations numerically, and determine consistent initial values for variables.

The model executive coordinates model execution. The simulation engine is stepped through time, while maintaining the specified level of accuracy. Internal event detection is invoked during each step. Internal and external events are posted to the event queue. The model executive processes the event queue. Where state changes are required in the plant model, the appropriate transition events are triggered on the plant model (or any component therein). Where parameter changes are required in the plant model, the parameter values are updated, and algebraic analysis is used where necessary to determine consistent initial variable values.

The internal event detection algorithms employed handle multiple zero crossing cases, whereby always selecting the first zero crossing. The event initialization algorithm includes the triggering discontinuity equation to constrain the re-initialization, and provide chatter-proof transitions.

The ARTIST engines inherently handle high index problems. High index problems can however be avoided or minimized through modeling, where the easily recognized signature of high index problems is addressed during modeling. Standard ARTIST libraries are written in such a way as to minimize the index of resulting applications. User-written models can be reduced to index 1 by highlighting which equations cause high index, and automatically differentiating the model to avoid the high index problem.

The test harness interacts with the discrete components (controller, sampled sensors and actuators), and the system components via the model executive, by issuing clock ticks each time the clock master advances time.

Controllers generate both state events and parameter events. Engaging a clutch actuator is an example of a state event. This event is queued by the model executive, and processed along with other internally generated events. Changing throttle position from a cruise controller is an example of a parameter event. This event is queued by the model executive, and again processed along with other internal or external events. In the case of parameter events, processing savings are realized by estimating the error introduced by the parameter change, and not re-initializing the model variables if the error is considered to be within the capacity of the solver to handle the error automatically.

Model integration

The following figure illustrates how systems may be integrated using ARTIST.

ARTIST solution: 1 language, 1 tool, 1 repository, 1 coordinated solution, inter-connections (always possible) based on simple port-to-port communication

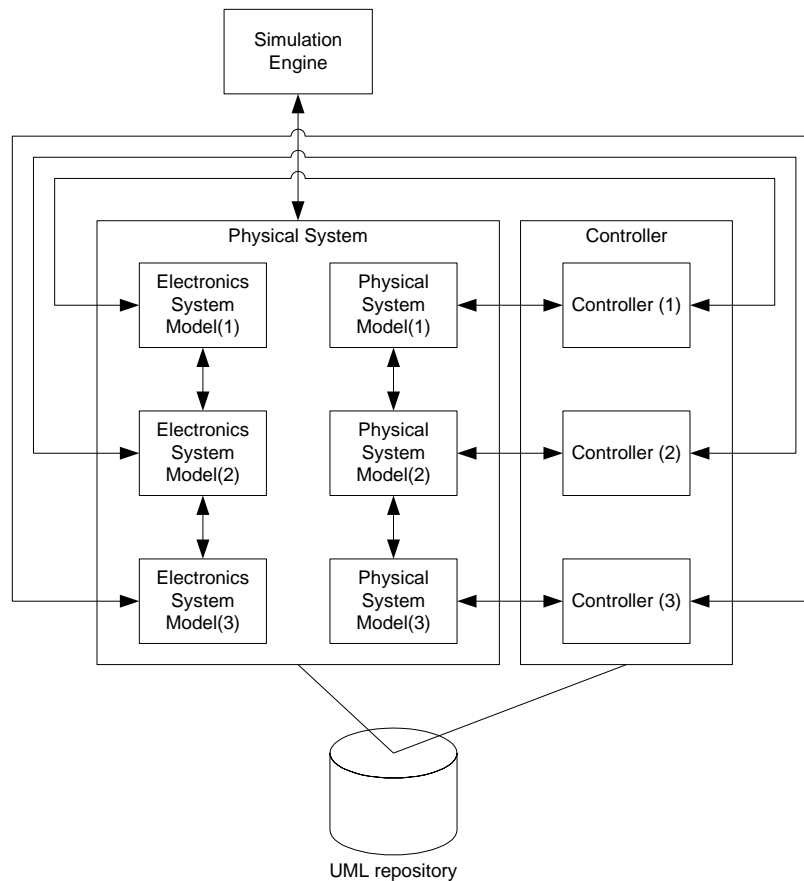


Figure 16 Product composition using ARTIST

Complete systems potentially including physical and electronic systems, and all associated controllers can be composed into a single test harness. Integrated test harnesses enable users to perform complete controller and physical system concurrent design, from prototype to deployment.

Performance Information

The following ARTIST performance tests were performed on a 700 Mhz Pentium III.

The problem includes an engine, transmission with 8 clutches, a ravigneaux gearbox, and a vehicle (as depicted in Figure 6).

The integration test environment is per the following figure.

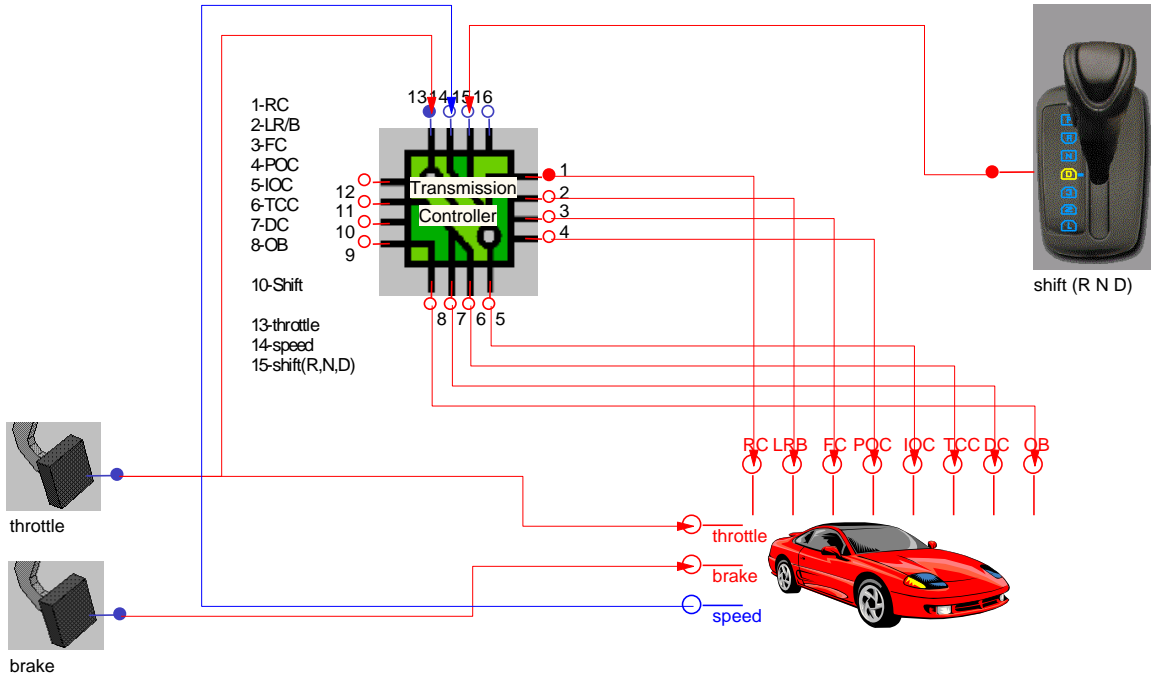
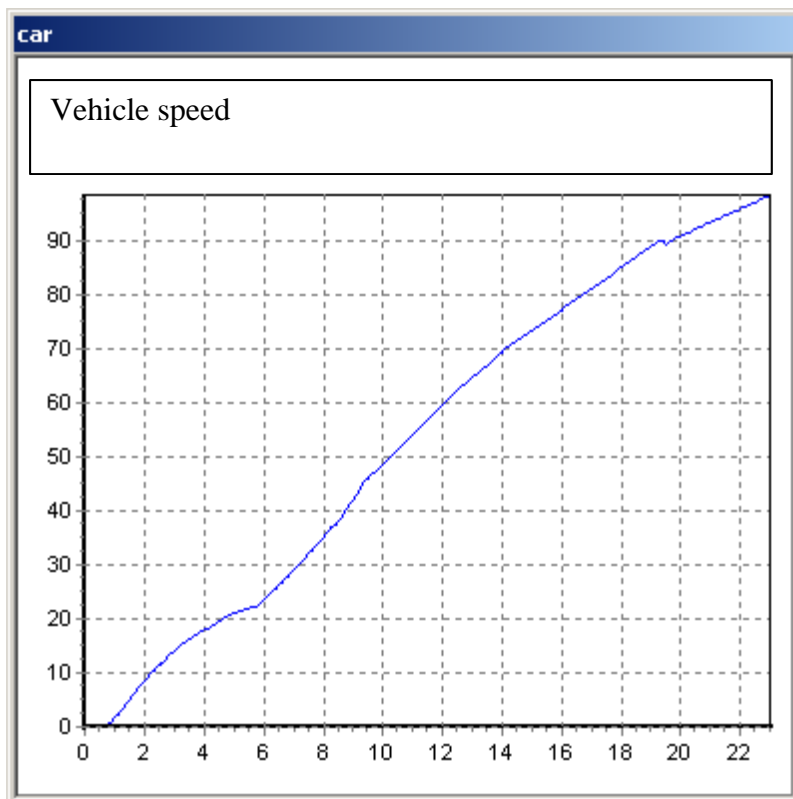
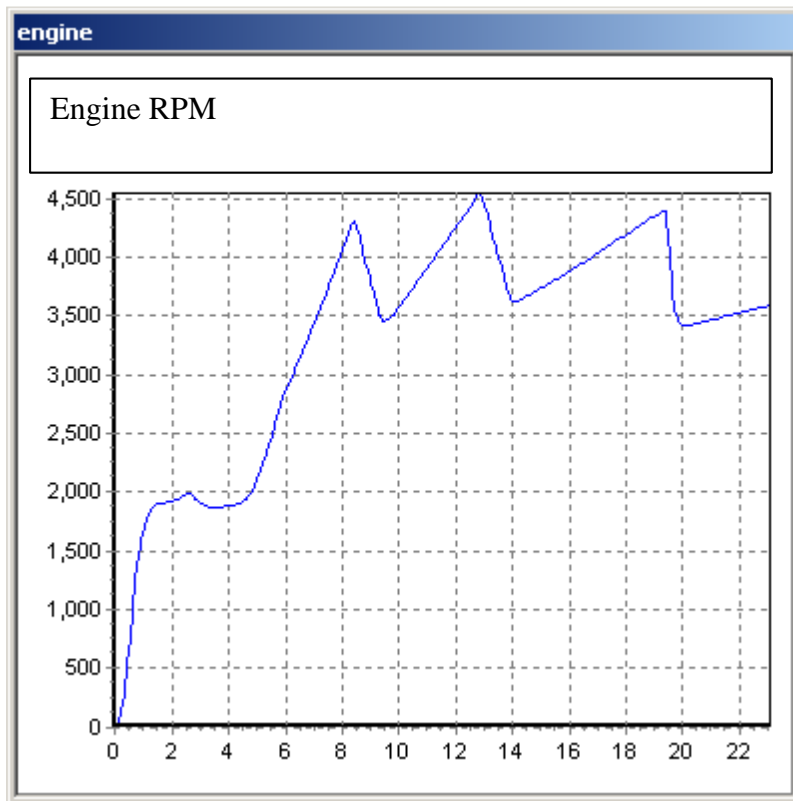


Figure 17 Test harness for performance example

Number of models: 46

Number of state machines: 18

The following charts illustrate recorded engine RPM and vehicle speed, where throttle is set to 80% at 5 seconds.



Recorded performance statistics for transitions from neutral to fourth gear are as follows:

Figure 18 Continuous system Meta models

The above figure illustrates the stereotyping of UML elements used to define UML continuous executable physical system models.

The continuous model stereotypes illustrated in Figure 18 are used to functionally classify UML class elements in preparation for the generation of executable UML physical system models. Continuous declarative specifications extracted from non-UML models are represented within UML using these stereotypes.

Primitive attributes like scalars, arrays, matrices, and collections may be stereotyped as Parameters and Variables. A parameter is typically a constant or slow changing value used in continuous behaviour equations. A Variable is a dynamically changing value used in continuous behaviour equations.

A variable's value can be updated by a simulation engine that satisfies a set of continuous constraints. A variable's value may be used to evaluate a continuous constraint, and may be used as an input to a discrete operation.

A parameter's value can be constant, or calculated as a result of a discrete operation. A parameter's value may be an input to a discrete operation, the result of a discrete operation, or an input to a set of continuous constraints used to initialize constrained variable values.

User-defined types (structures) may be stereotyped as Continuous System models. A Continuous System may be comprised of Variables, Parameters, and other Continuous System objects.

Physics-based operations that reference at least one Variable and optionally Parameters to depict physical system behaviour may be stereotyped as Discontinuity Equation operations, Continuous Dynamic Equation operations, or Continuous Reset Operations. Physics-based operations may have a one-to-one relationship with a physics-based equation, however one operation may implement 1..n physics-based equations. For simplicity and clarity, unless specified otherwise, this document treats one physics-based operation as one equation.

Discontinuity Equation operations evaluate a condition using continuous math. The condition $X > 100.0$ is expressed in canonical form $X - 100.0$, where the condition is considered true when the result is any positive number. Discontinuity Equation operations are typically used in state event triggers. A solver would monitor the equation for a positive zero crossing to detect the event, and accurately determine the event time.

Continuous Reset Equation operations evaluate a constraint during a state transition. The bounce transition of a ball may for example reset the velocity $V = -0.85 * old(V)$.

Continuous Dynamic Equation operations evaluate continuous behaviour. Continuous behaviour of a resistor for example could be expressed by the equation $V = I * R$.

A Connectable Hybrid System model is used to model the behaviour of dynamic systems that have 1 or more states. Connectable Hybrid System models may be comprised of Variables, Parameters, and Continuous System models, Discontinuity Equation operations (for internal event detection), Continuous Reset Equation operations (for defining transition constraint behaviour), and Continuous Dynamic Equation operations (for defining continuous behaviour in each state), a Continuous State Machine (for hybrid continuous behaviour), other Connectable Hybrid System models (for composition), message handler operations (for handling external discrete events), and Continuous Connectors (for constraint-based plug and play integration with other models). Examples of Continuous Hybrid System models are clutches, transistors, resistors, and hydraulic servo actuators.

The constraint behaviour of inter-connected connector models is defined by one and only one Continuous Connector model. Continuous Dynamic Equation operations in Continuous Connector models may support a scalable number of equations based on the number of inter-connected connector models. A Continuous Connector may be comprised of Variables, Parameters, and Continuous System models, scalable and non-scalable Continuous Dynamic Equation operations.

A Continuous Constraint Connection is an entity used to indirectly define physics-based constraint connections between two instances of a Continuous Connector class. The collection of all Continuous Constraint Connection models that share at least one common connector indirectly define the physics-based constraint behaviour between inter-connected connector models.

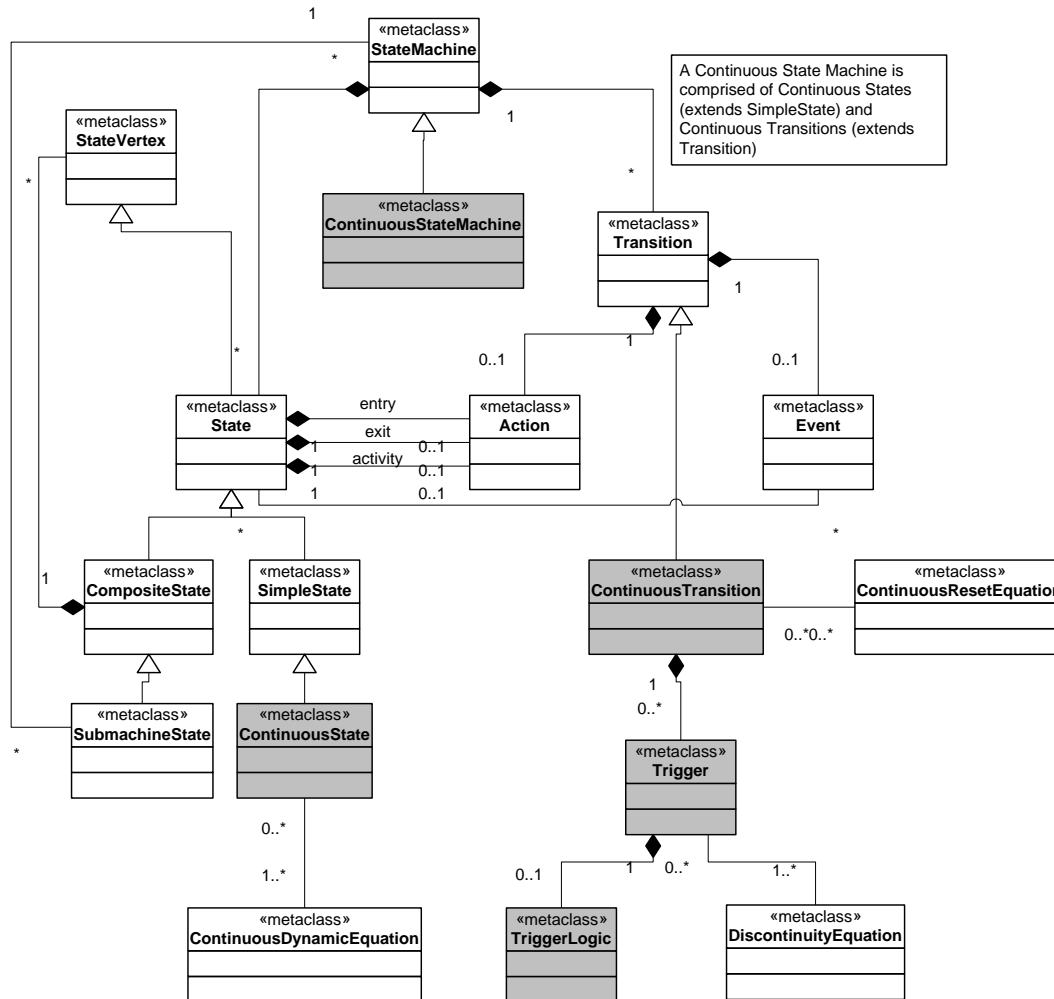


Figure 19 State machine Meta models

The above figure illustrates the stereotypes and associations added to extend the standard UML state chart to support continuous hybrid modeling. Some standard UML state machine elements are not shown for simplicity and clarity. Refer to the OMG-UML specification (Figure 2-24 'State Machines – Main' and Figure 2-25 'State Machines – Events' in September 2001 OMG-UML v1.4) for a complete state machine Meta model.

The Continuous State Machine may be comprised of Continuous States and Continuous Transitions.

Continuous States extend Simple States with a collection of Continuous Dynamic Equation operations. The operation mapping supports continuous behaviour specification for each state. The number of equations (implemented via Continuous Dynamic Equation operations) must be equal in all states for a model to be valid.

Continuous Transitions extend basic Transitions with a collection of zero or more Continuous Reset Equation operations, and a collection of zero or more state event

Triggers. Triggers may be comprised of 1 or more Discontinuity Equation operations. The Continuous Reset Equation Operation mapping supports additional continuity constraint behaviour during transition initialization, modeling the behaviour of the transition itself. The bounce transition behaviour of a ball could be modeled using a Continuous Reset Operation $Velocity = -0.85 * old(Velocity)$, thus modeling velocity reversal and energy loss of the bounce itself.