

Overview of ARTIST Architecture (UML Diagrams)

Figure 1a) Continuous Meta Model

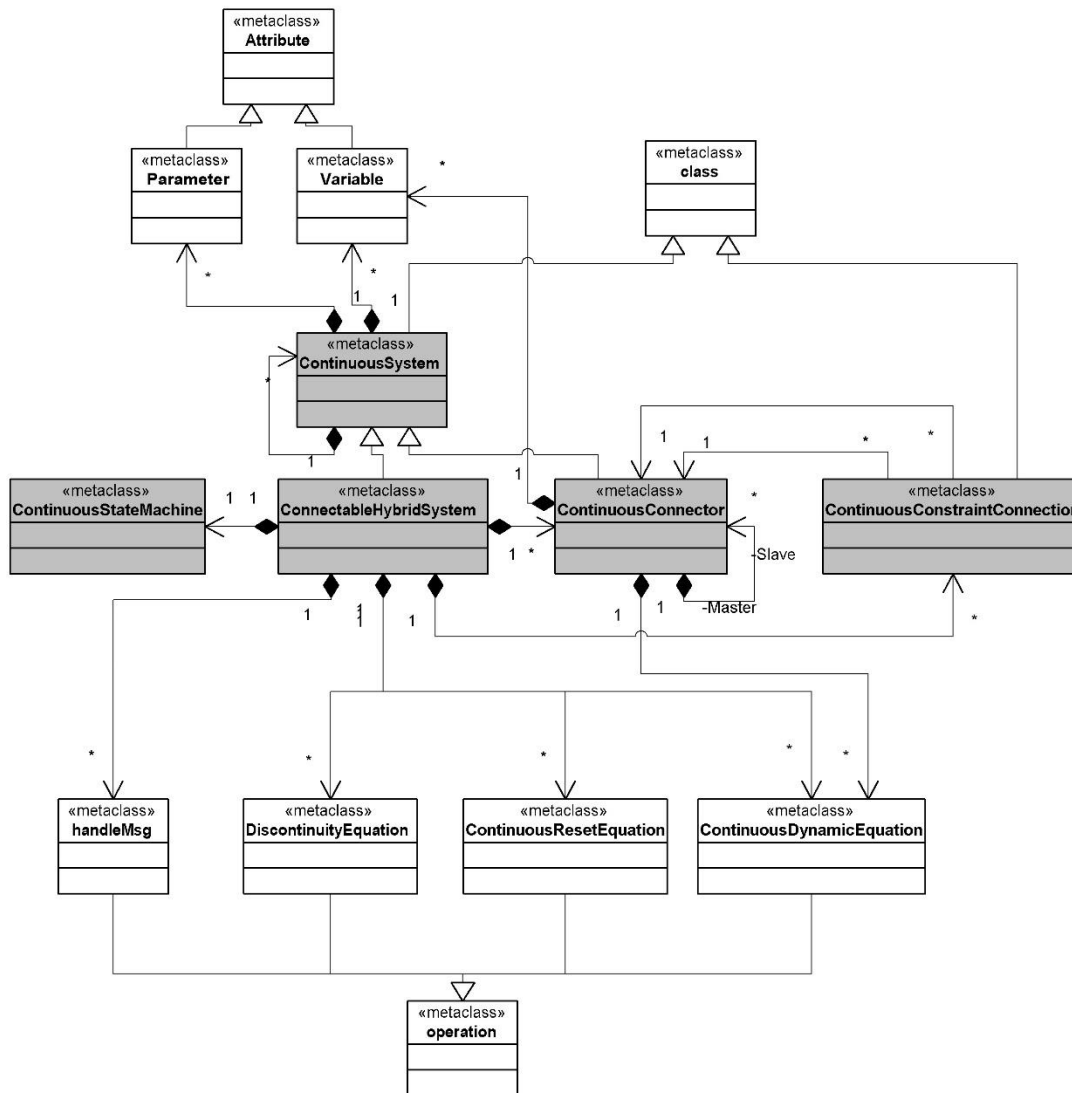


Figure 1a is a UML Meta model diagram illustrating the stereotypes used to define continuous executable physical system models within UML.

Figure 1b) Continuous State Machine Meta Model

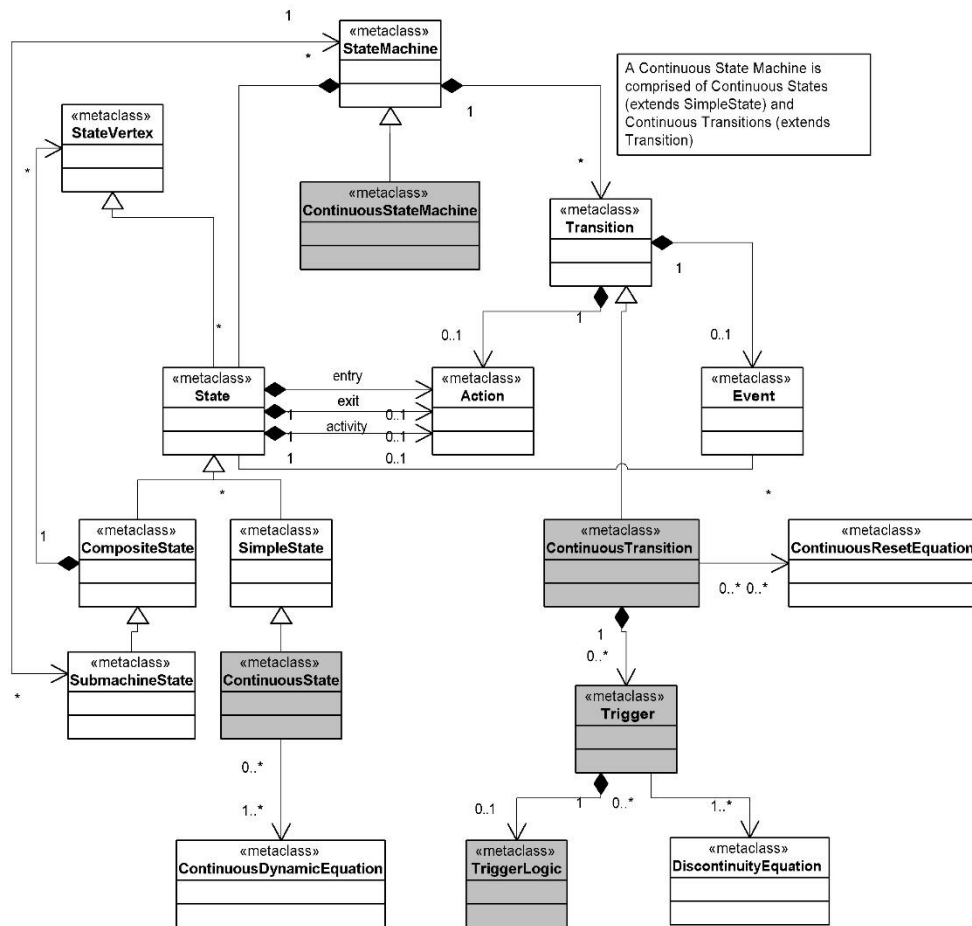


Figure 1b is a UML Meta model diagram illustrating the associations and stereotypes required to extend the UML state chart for continuous hybrid physical system modeling.

Figure 1c) Discrete Meta Model

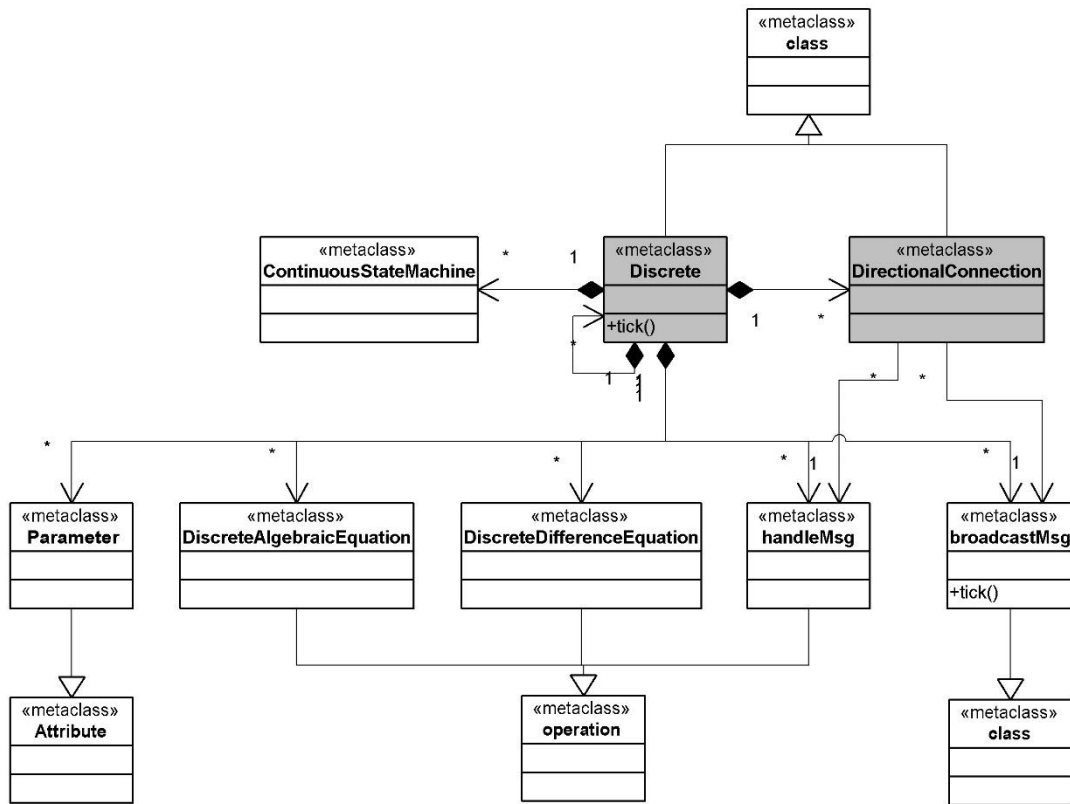


Figure 1c is a UML Meta model diagram illustrating the stereotypes used to define discrete executable models within UML.

Figure 1d) Extended Context Diagram Node Meta Model

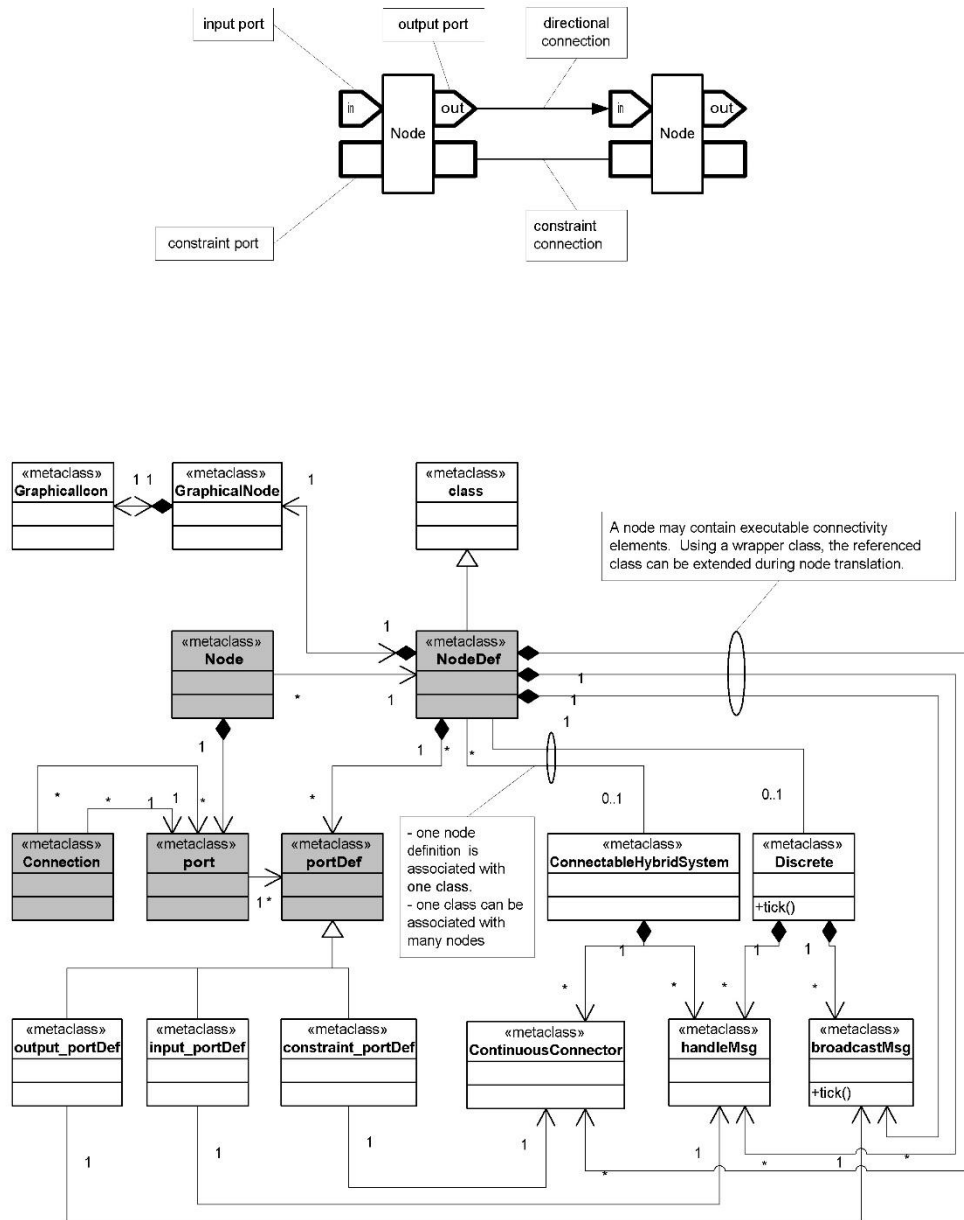


Figure 1d is a UML Meta model diagram illustrating the structure of the node objects used for generating executable physical system composite, and executable applications with integrated physical system models.

Figure 2a) Workflow overview

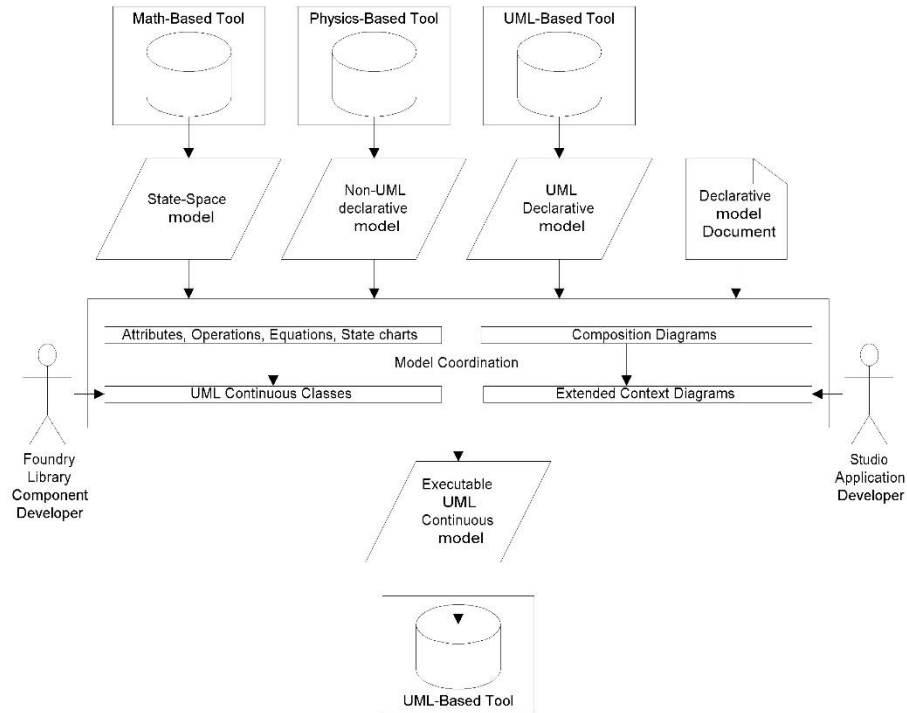
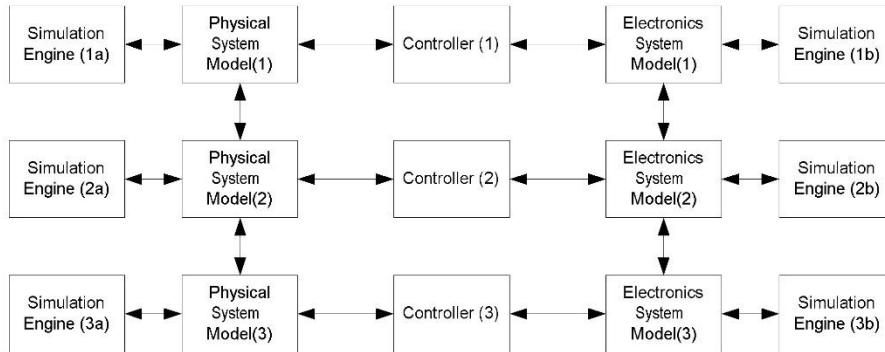


Figure 2a is an overview diagram of the workflow made possible by this patent. The workflow being the ability to import software models from any declarative or state-space format, and compose and generate simulation-based executable test models within standard model-based tools.

Figure 2b) Execution comparison

Today's solution: many languages, many tools, many repositories, many disjoint solutions, inter-connections (if possible) based on external tool-to-tool communication



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

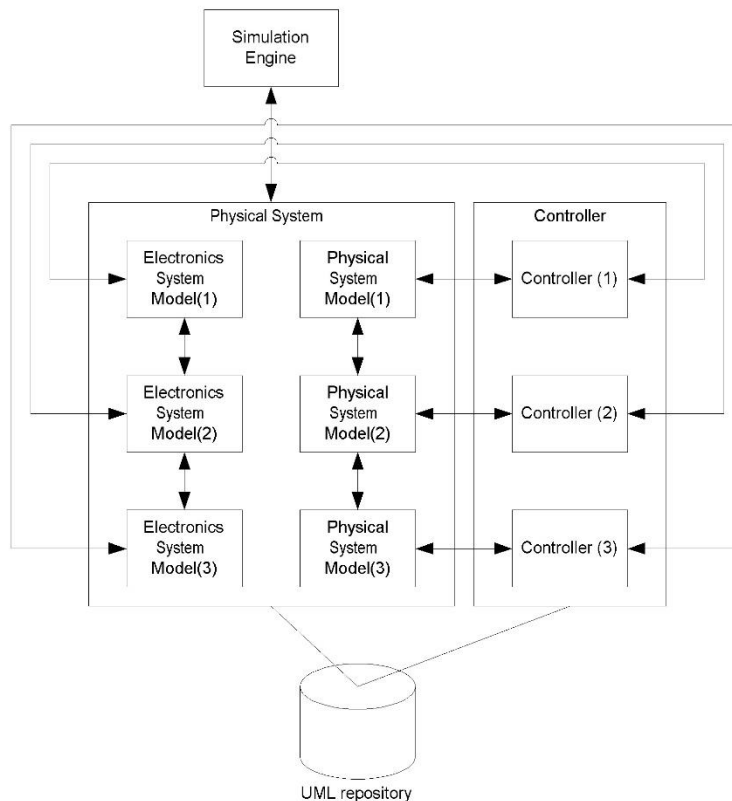


Figure 2b illustrates how the ARTIST coordinated solutions compare to today's solutions.

Figure 2c) Extended context diagram node definition overview

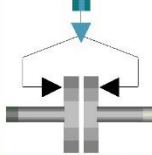
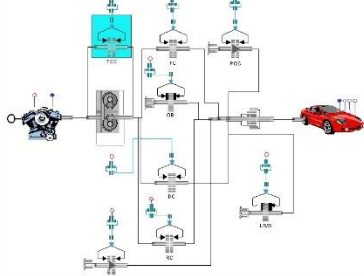
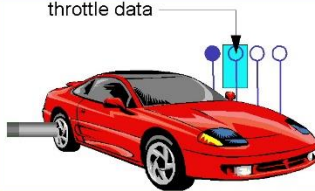
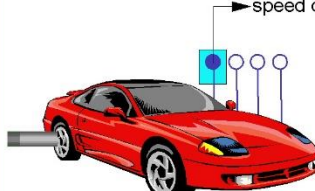

View element	Model element	Example
Node definition	UML Class	
Node (node definition instance)	Object (class instance)	
Input port	Message handler operation	
Output port	Message broadcast object	
Constraint port	Continuous connector	

Figure 2c is a table illustrating the relationship between node view elements and model elements. Node elements are used on extended context diagrams.

Figure 2d) Node definition associated with executable UML plant class

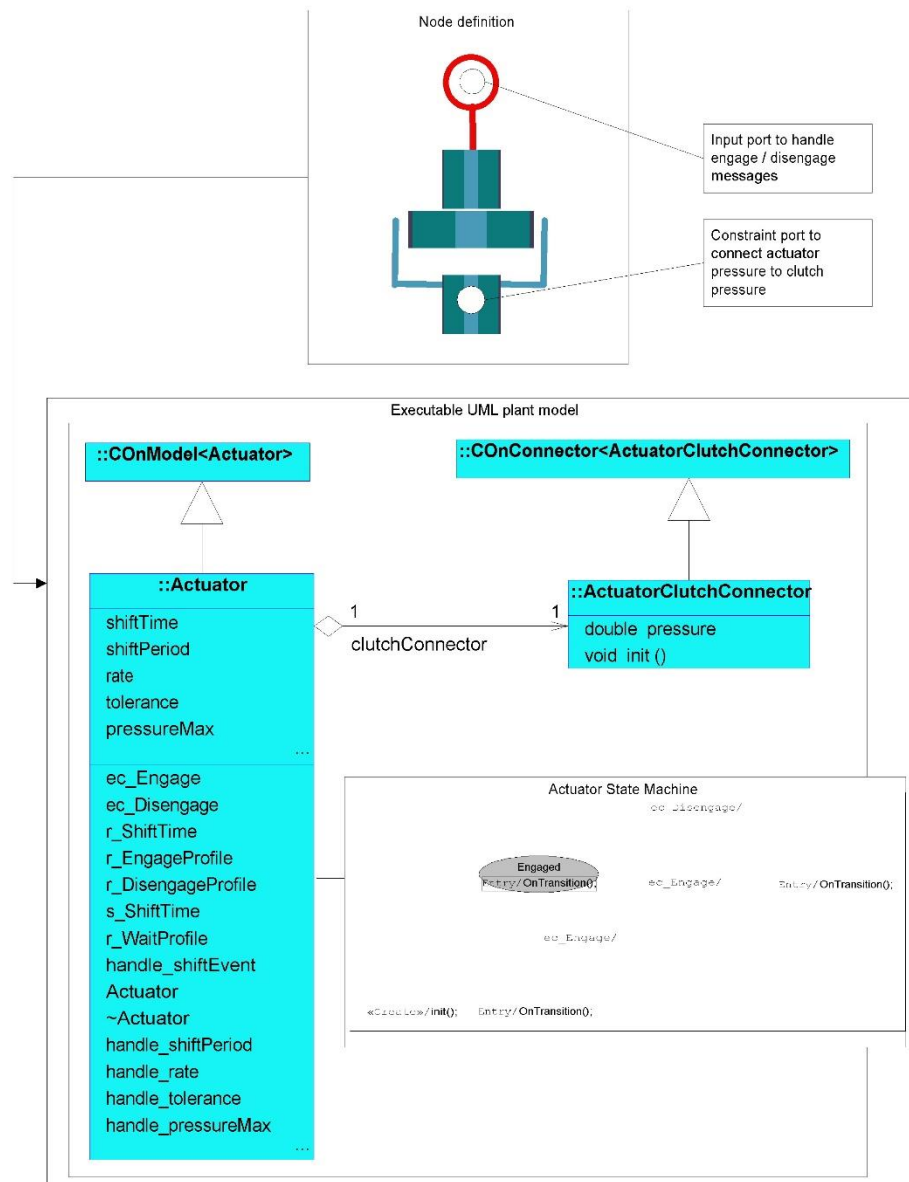
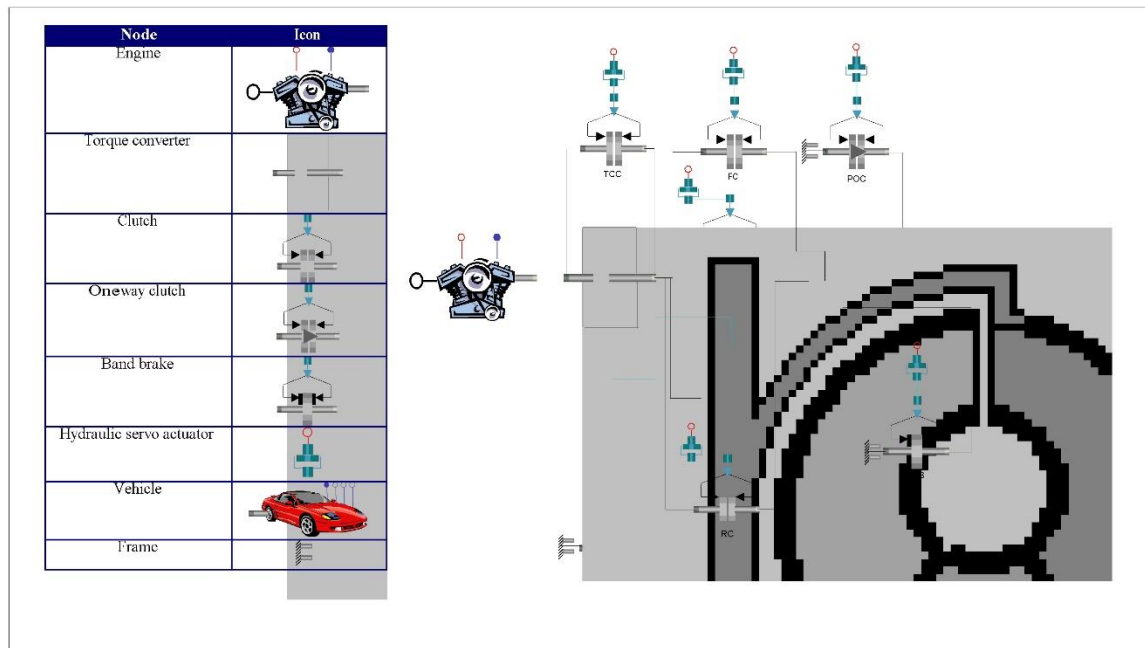


Figure 2d illustrates an example graphical node, and its associated executable continuous physical system model in the UML repository.

Figure 2e) Extended context diagram to executable UML physical system model



Translate to executable UML physical system model

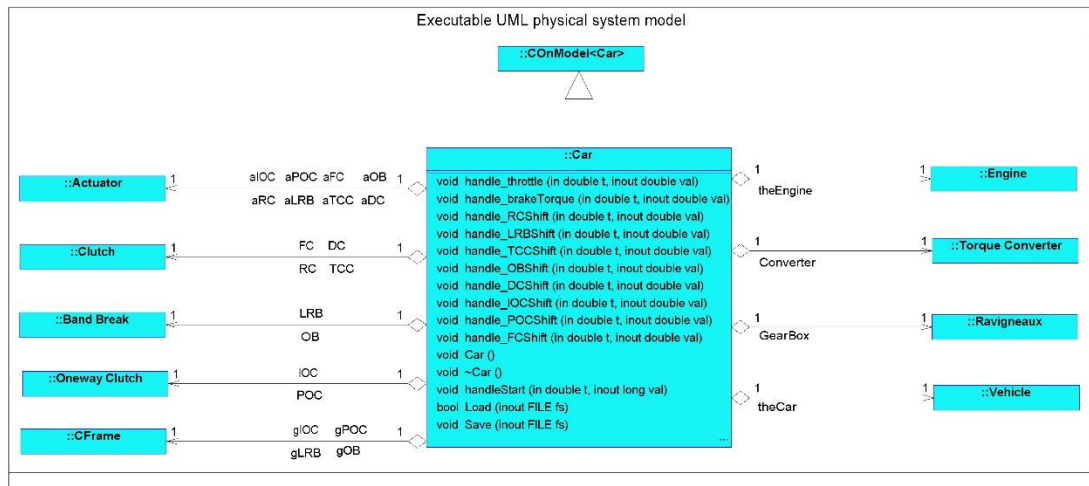
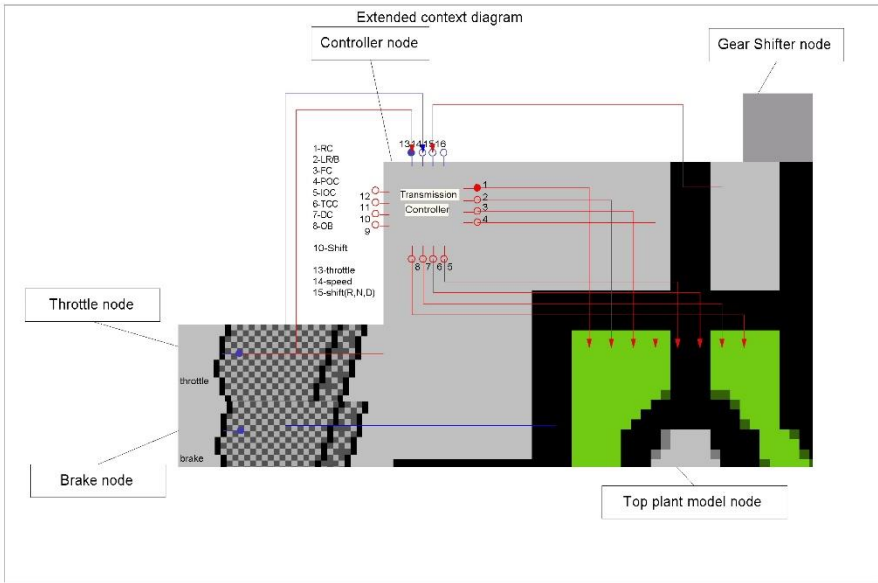


Figure 2e illustrates a composition of drive train components complete with constraint connections in an extended context diagram, and the resulting UML executable package after the nodes are translated.

Figure 2f) Extended system diagram to executable UML application model



Translate composition to executable UML application model

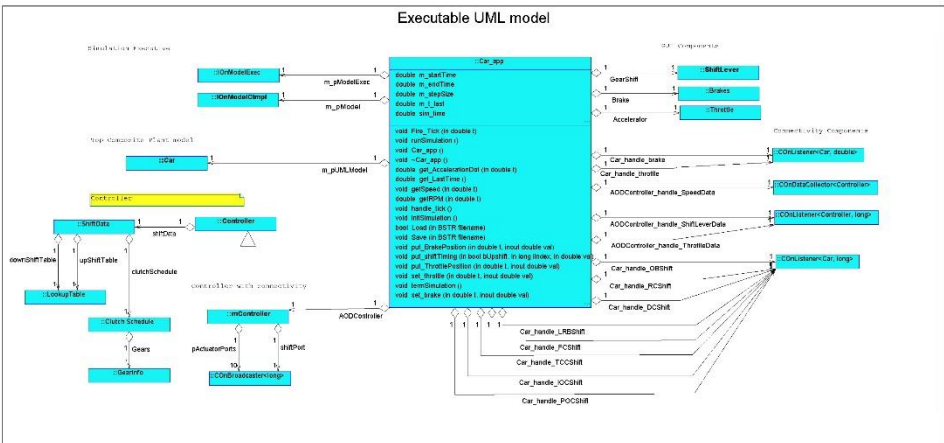


Figure 2f illustrates a composition of discrete and continuous components complete with event and data connections, and the resulting UML executable package after the nodes are translated.

Figure 3a) Generate extended context diagram

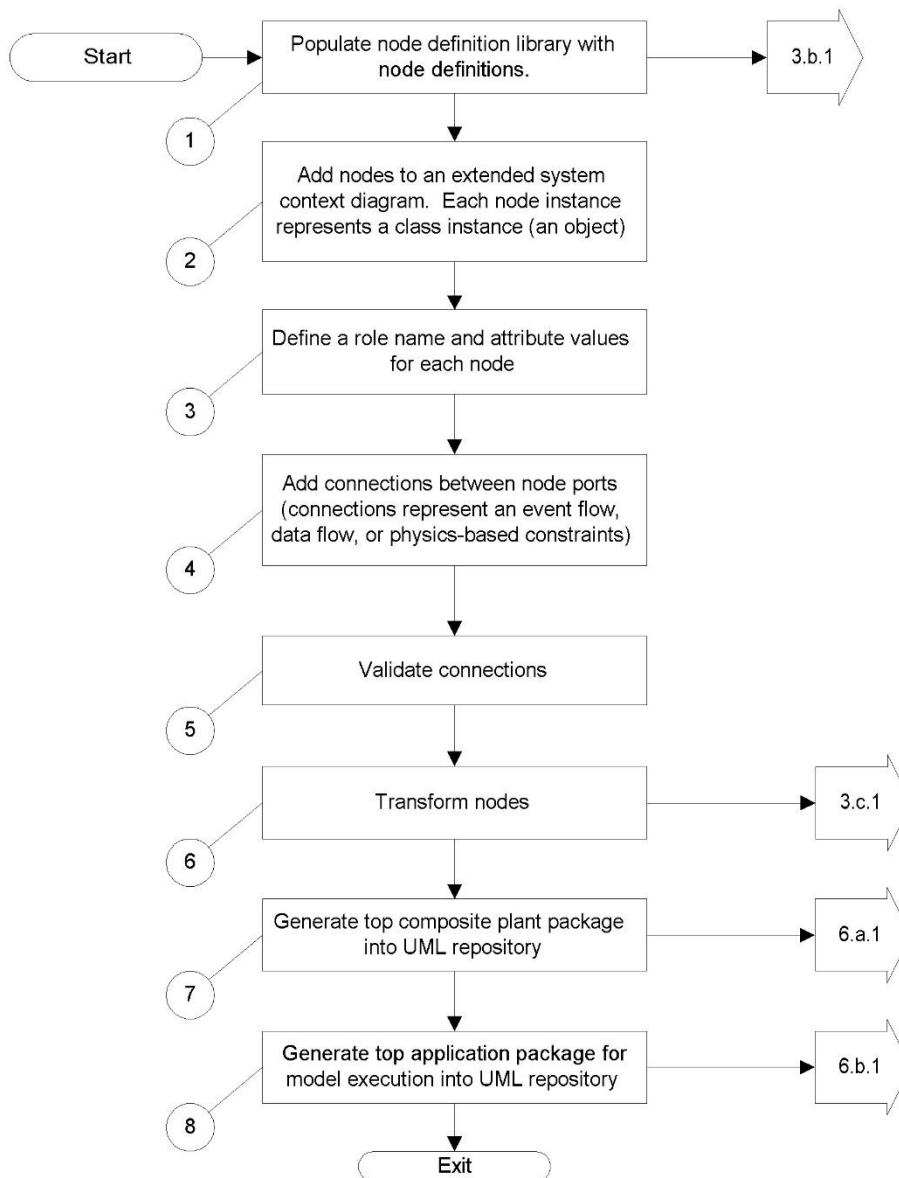


Figure 3a provides an overview of how extended context diagram nodes are connected, and translated to a UML executable package. The extended context diagram may be generated manually, or it may be the result of an import operation.

Figure 3b) Create a node definition (library component)

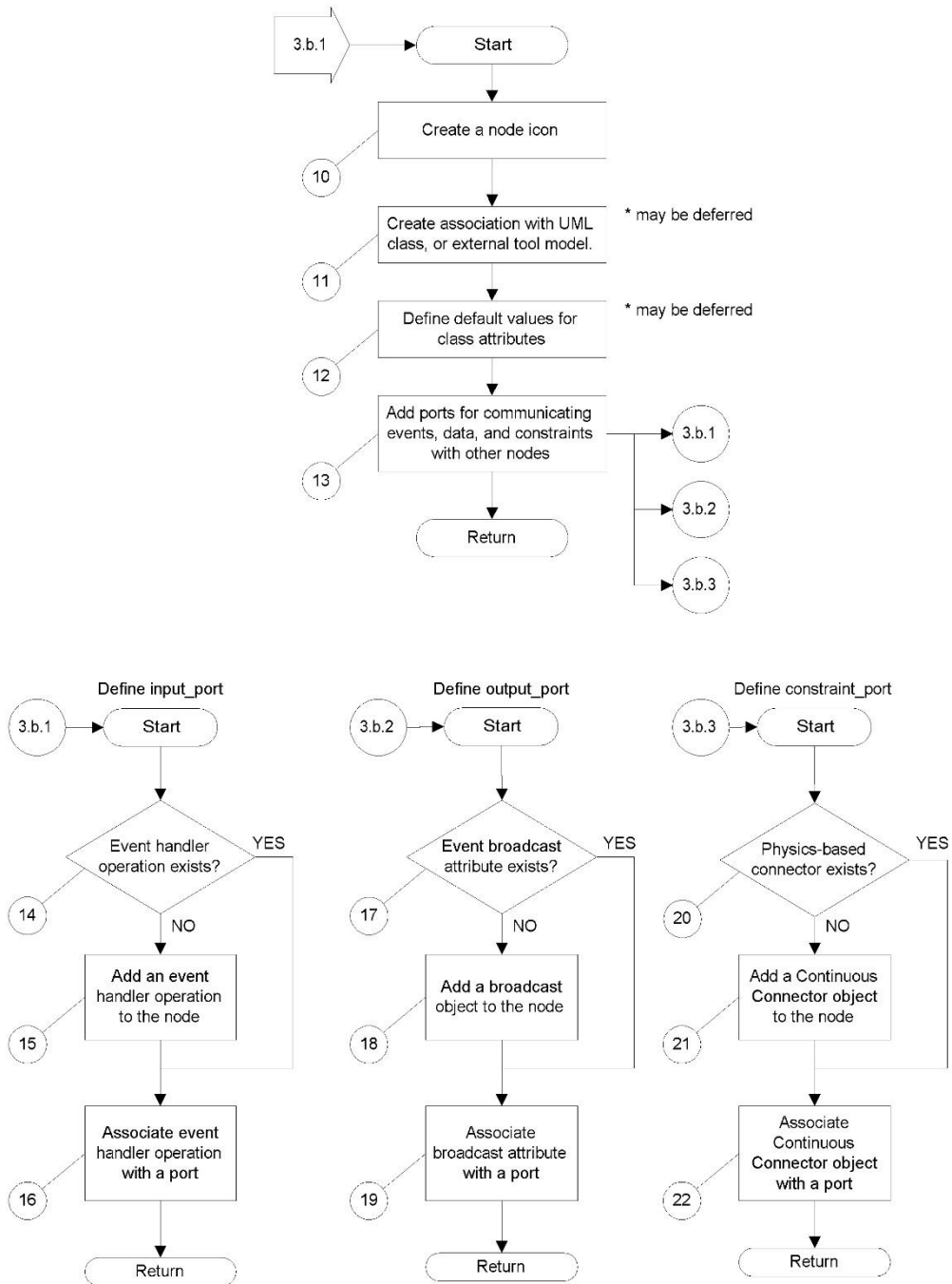


Figure 3b illustrates how a node definition is created for use in extended context diagrams.

Figure 3c) Node transformation

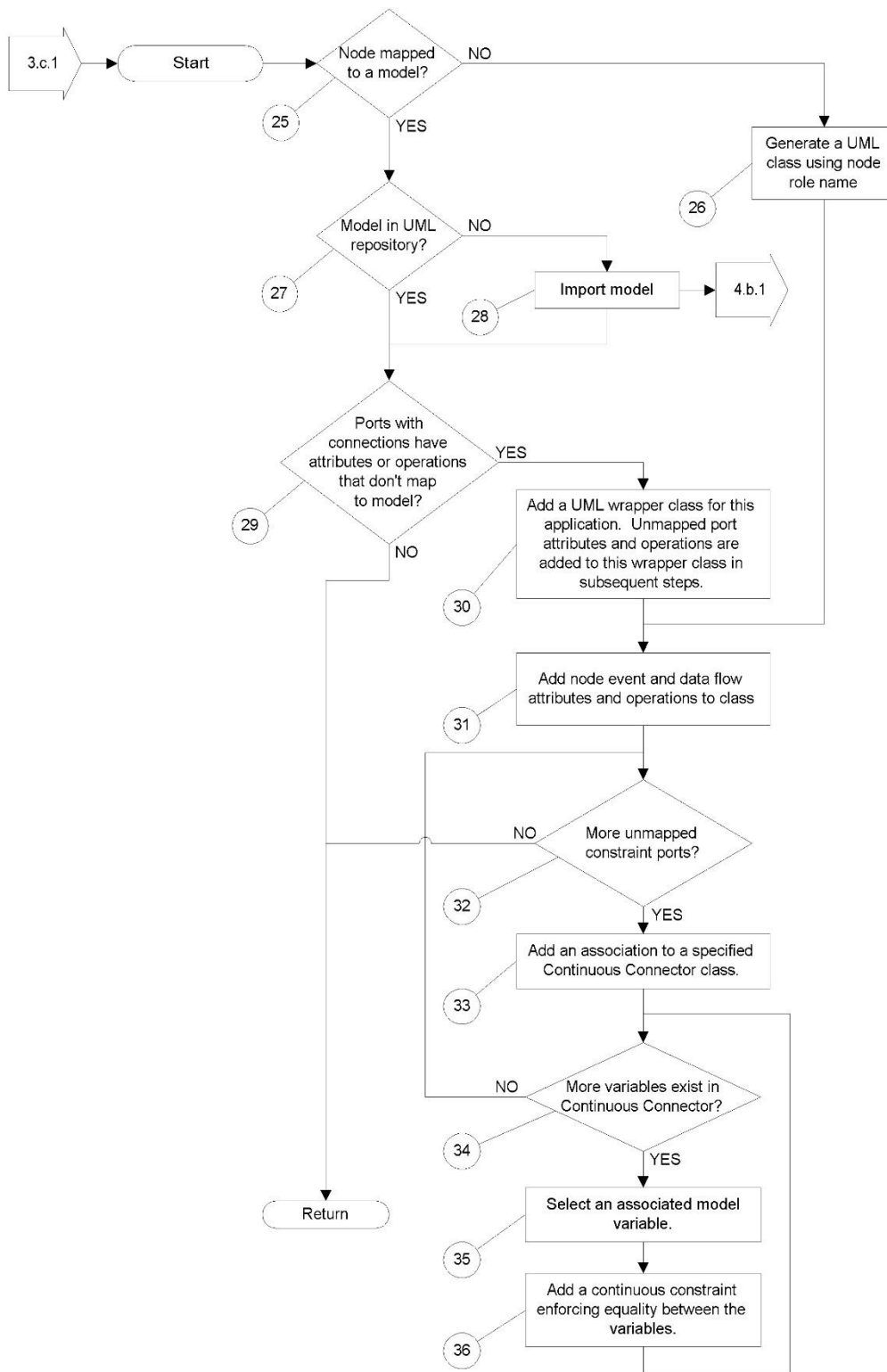


Figure 3c illustrates how a node is transformed into an executable UML model.

Figure 4a) Import models

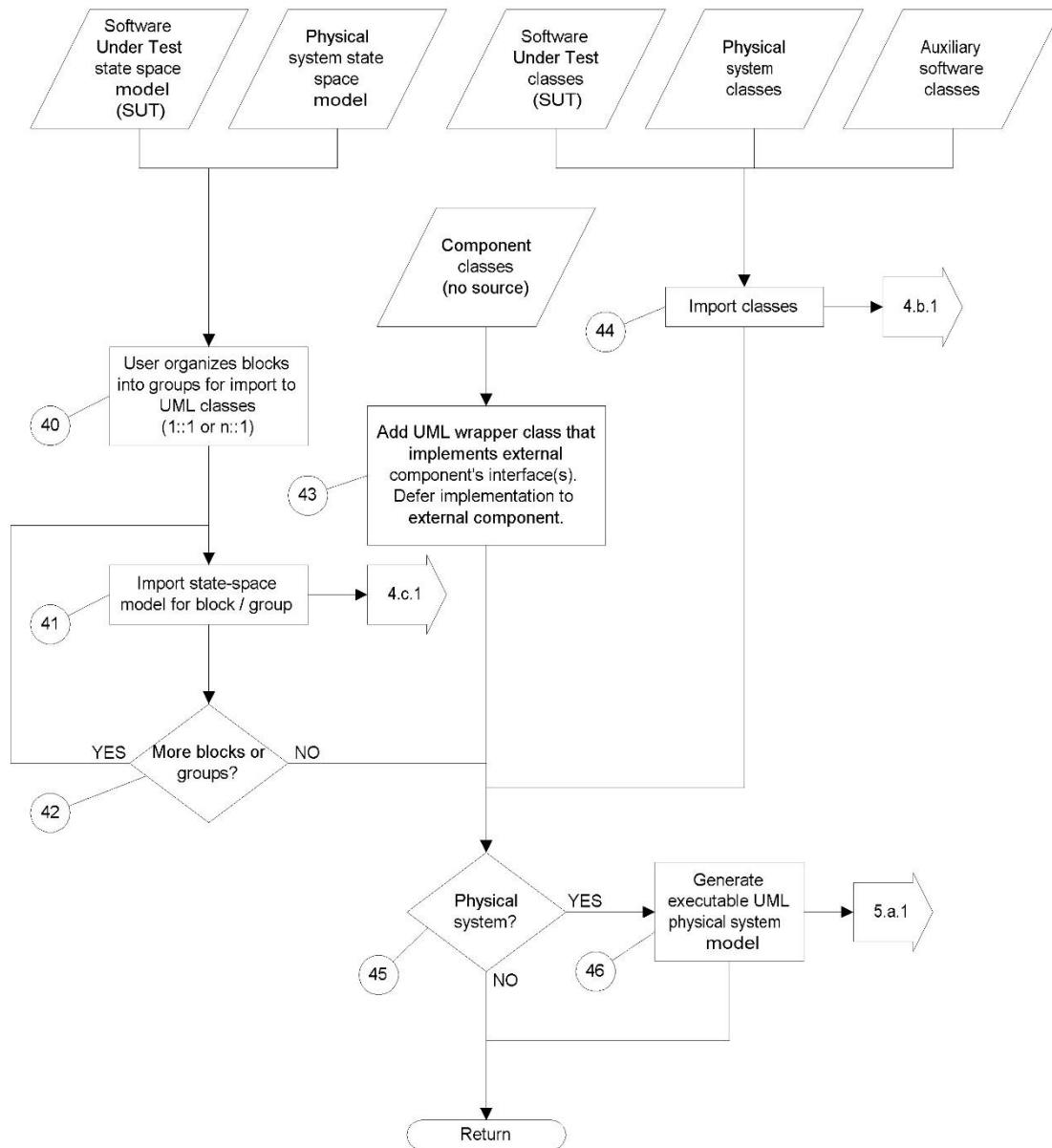


Figure 4a provides an overview of how external software models (including software under test models, physical system models, and auxiliary software models) may be imported.

Figure 4b) Import class

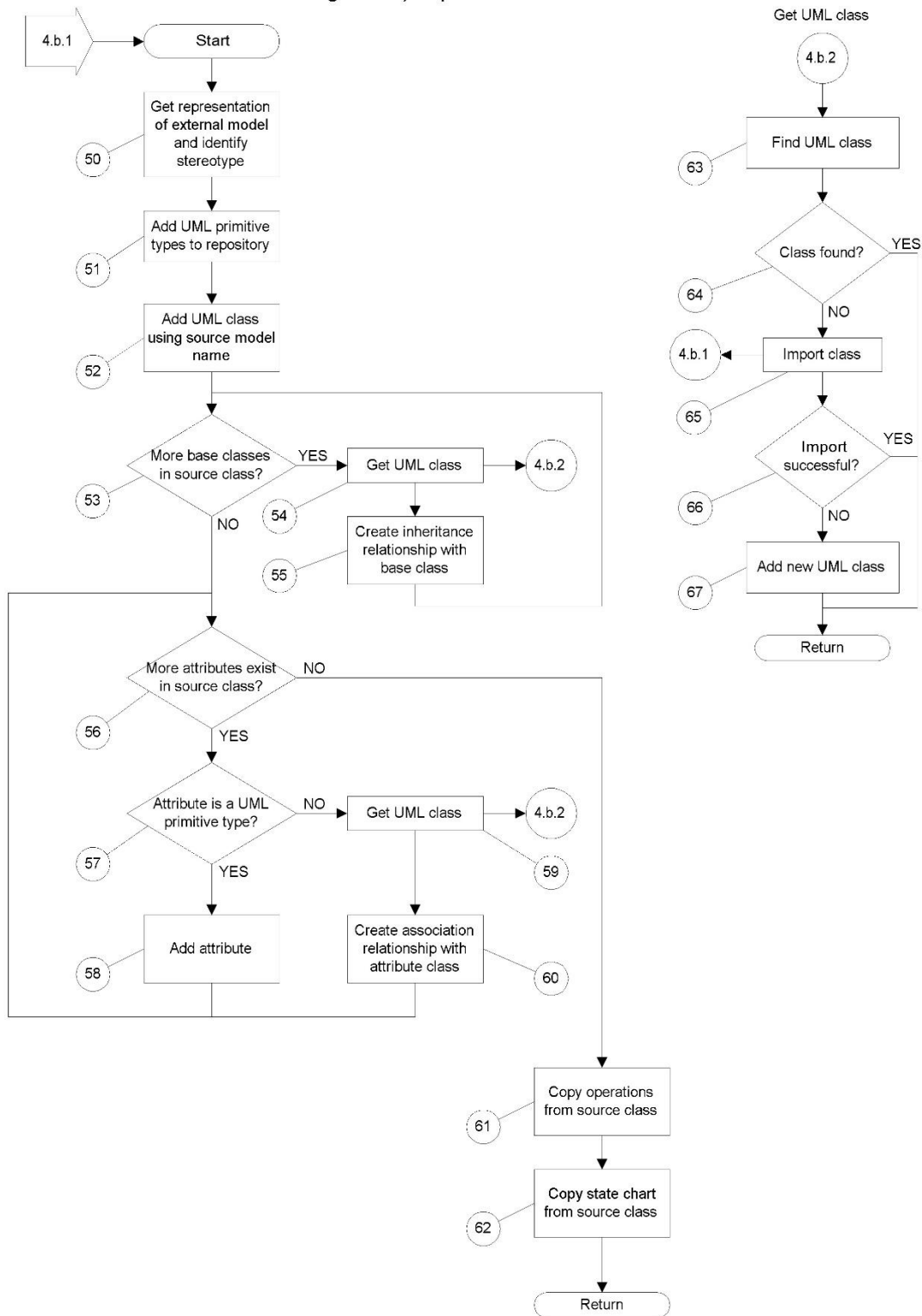


Figure 4b illustrates how models are imported from other repositories to create executable UML models.

Figure 4c) Import state space model

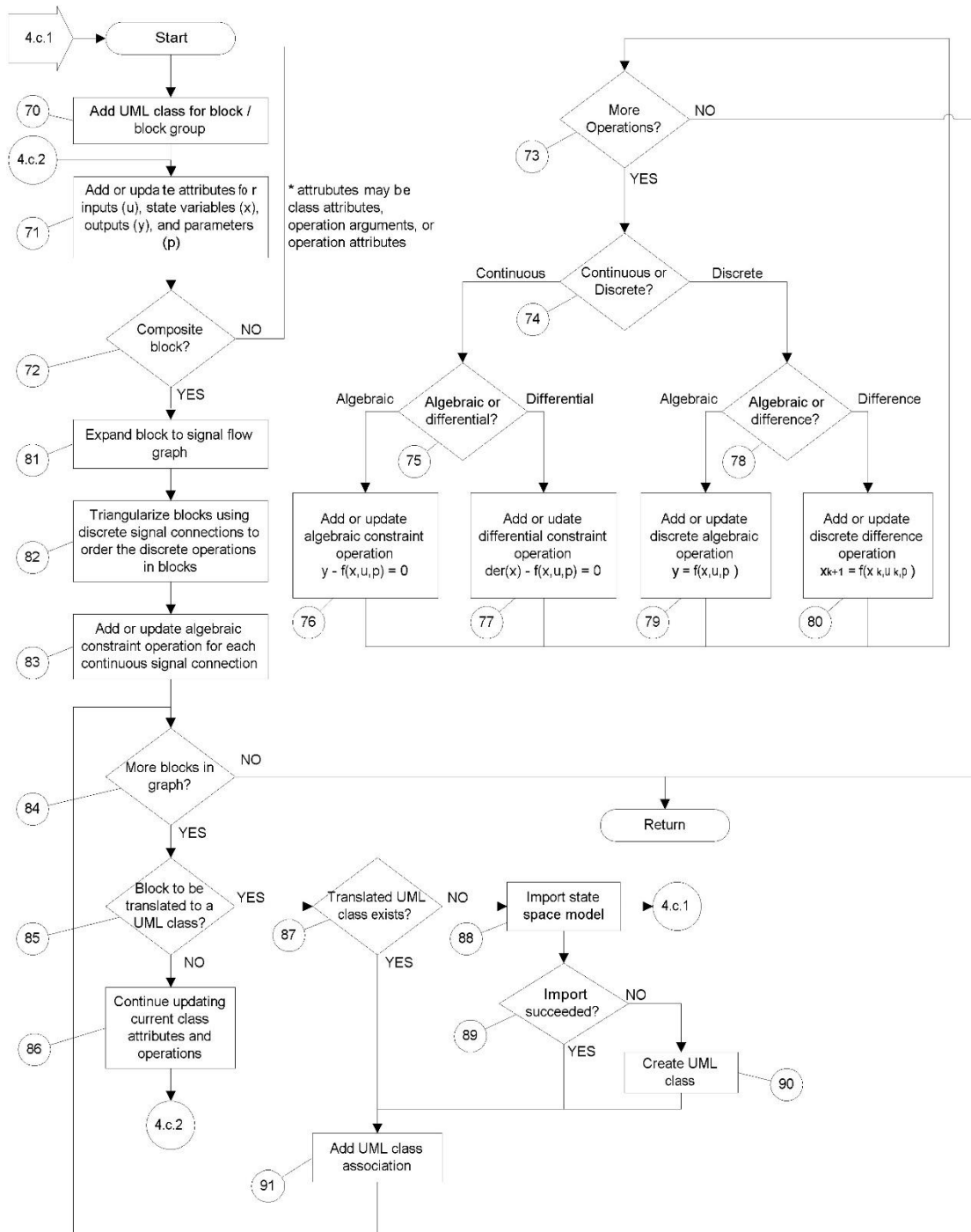


Figure 4c illustrates how state space models are imported and transformed into executable UML models

Figures 5a, 5b, and 5c illustrate how a standard UML class is transformed into an executable UML physical system model.

Figure 5a) Generate executable UML physical system model (1)
(Attribute typing and mapping)

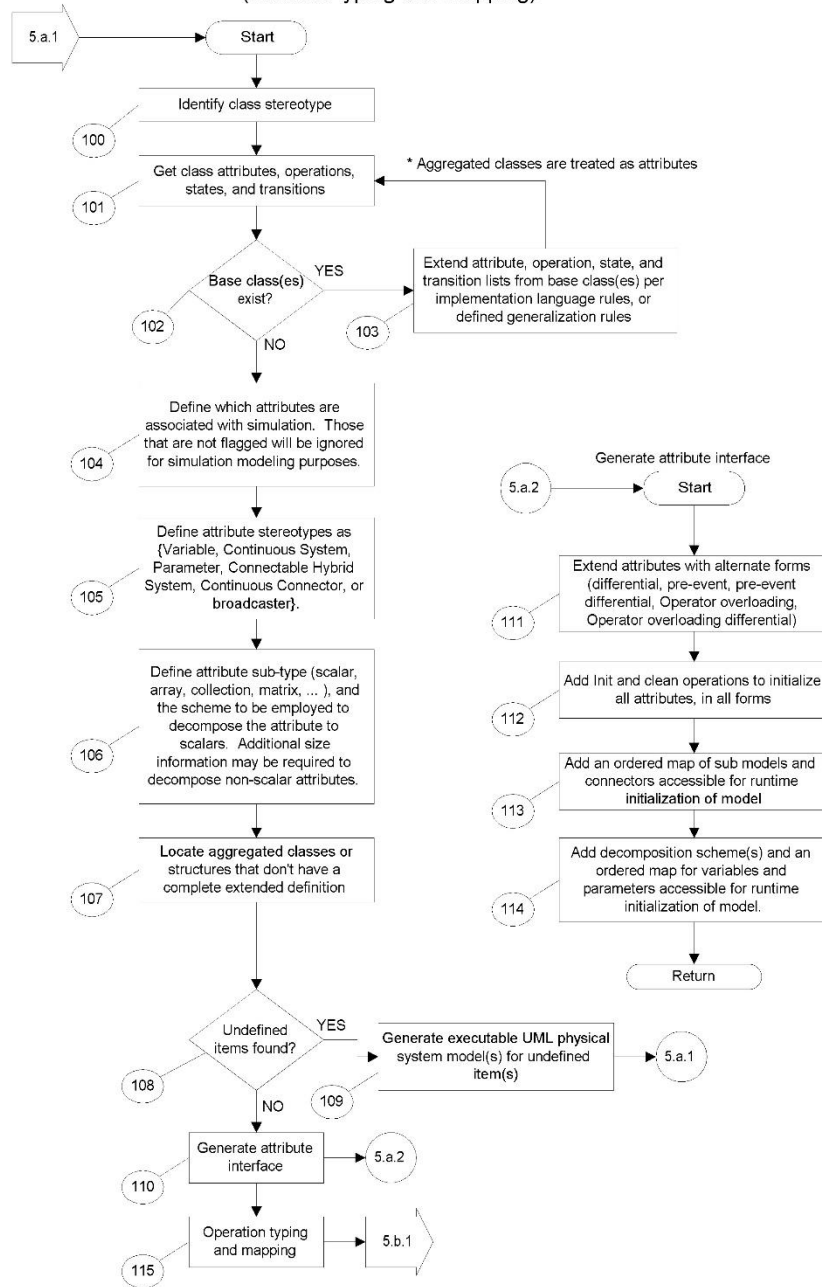


Figure 5a illustrates attribute and association typing and mapping, and the generation of the attribute interface to a simulation engine and executive.

Figure 5b) Generate executable UML physical system model (2)
(Operation typing and mapping)

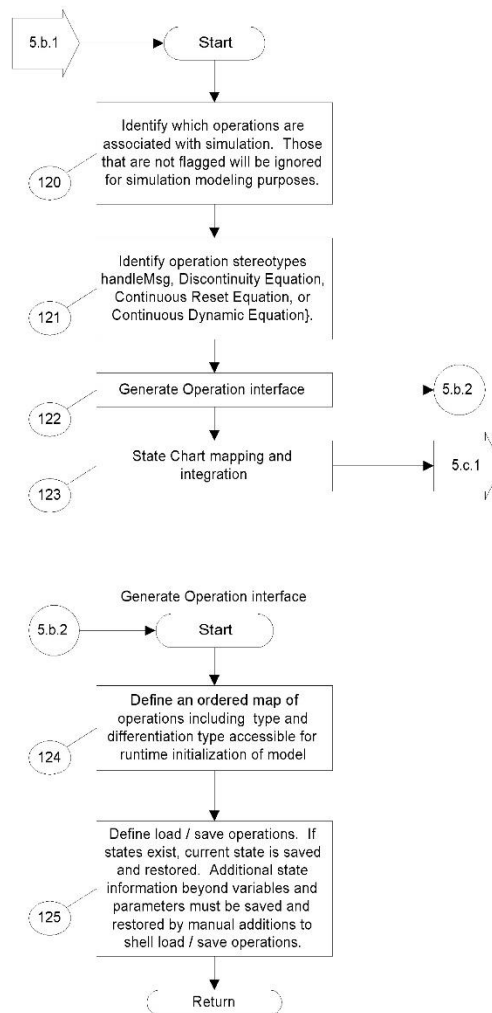


Figure 5b illustrates operation typing and mapping, and the generation of the operation interface to a simulation engine and executive.

Figure 5c) Generate executable UML physical system model (3)
State Chart mapping and integration

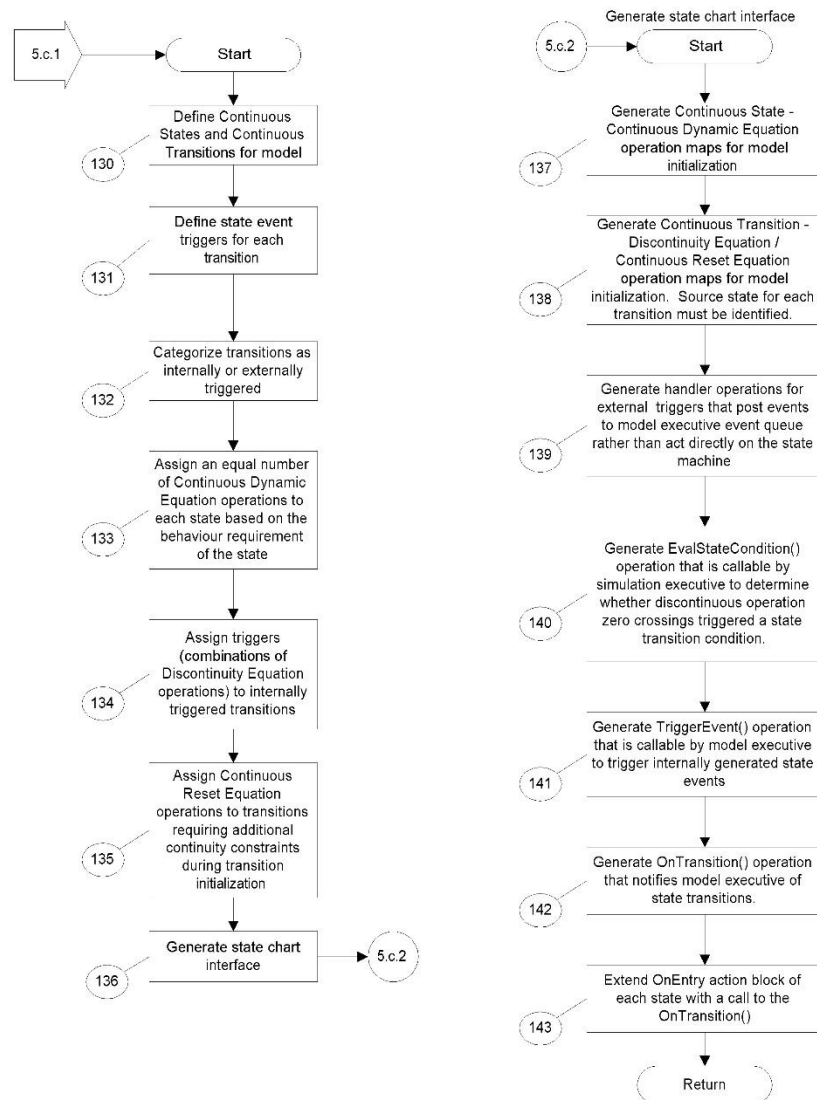


Figure 5c illustrates state machine mapping and integration, and the generation of the state machine interface to a simulation engine and executive.

Figure 6a) Generate executable composite physical system package into UML repository

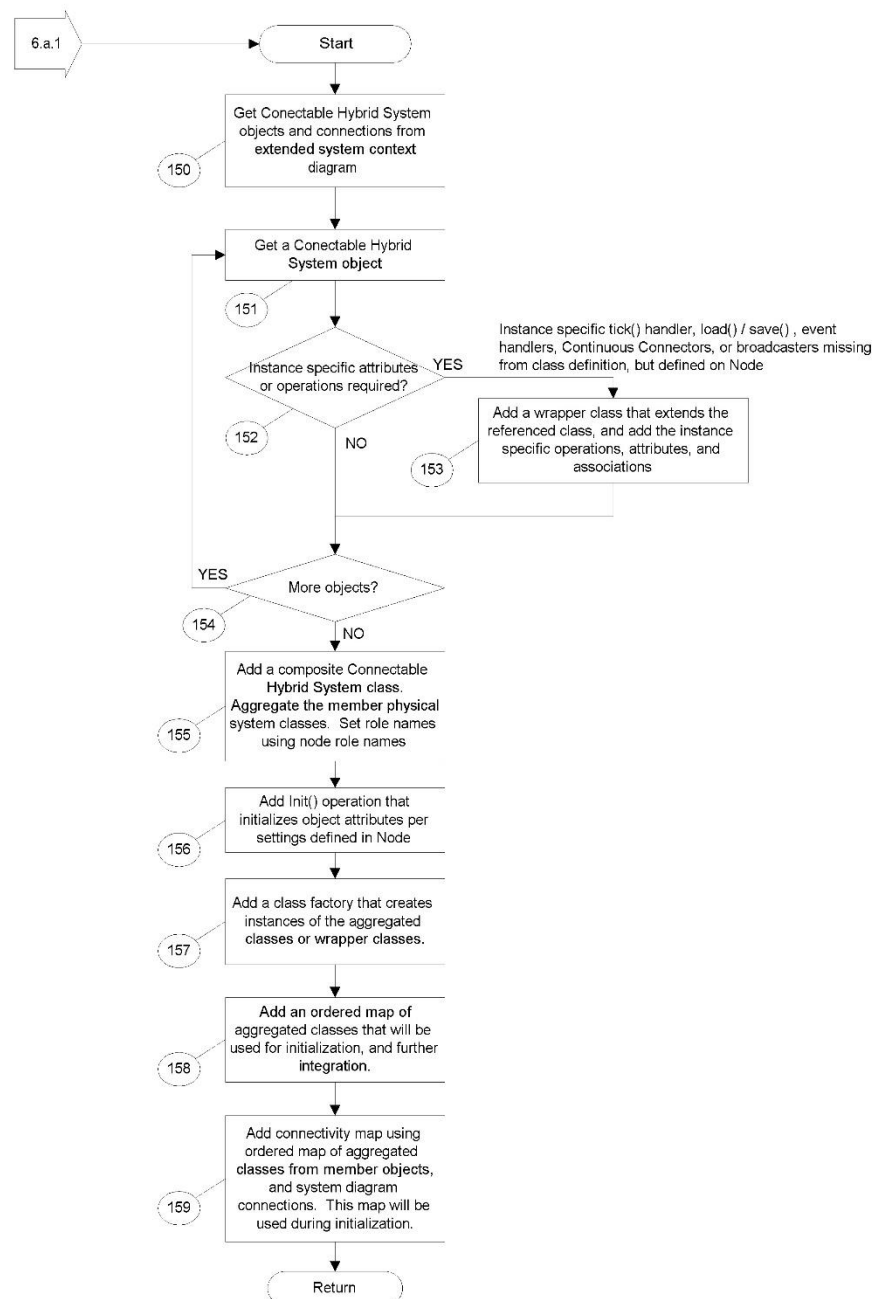


Figure 6a illustrates how an executable composite physical system package can be generated from an extended context diagram.

Figure 6b) Generate executable application package into UML repository

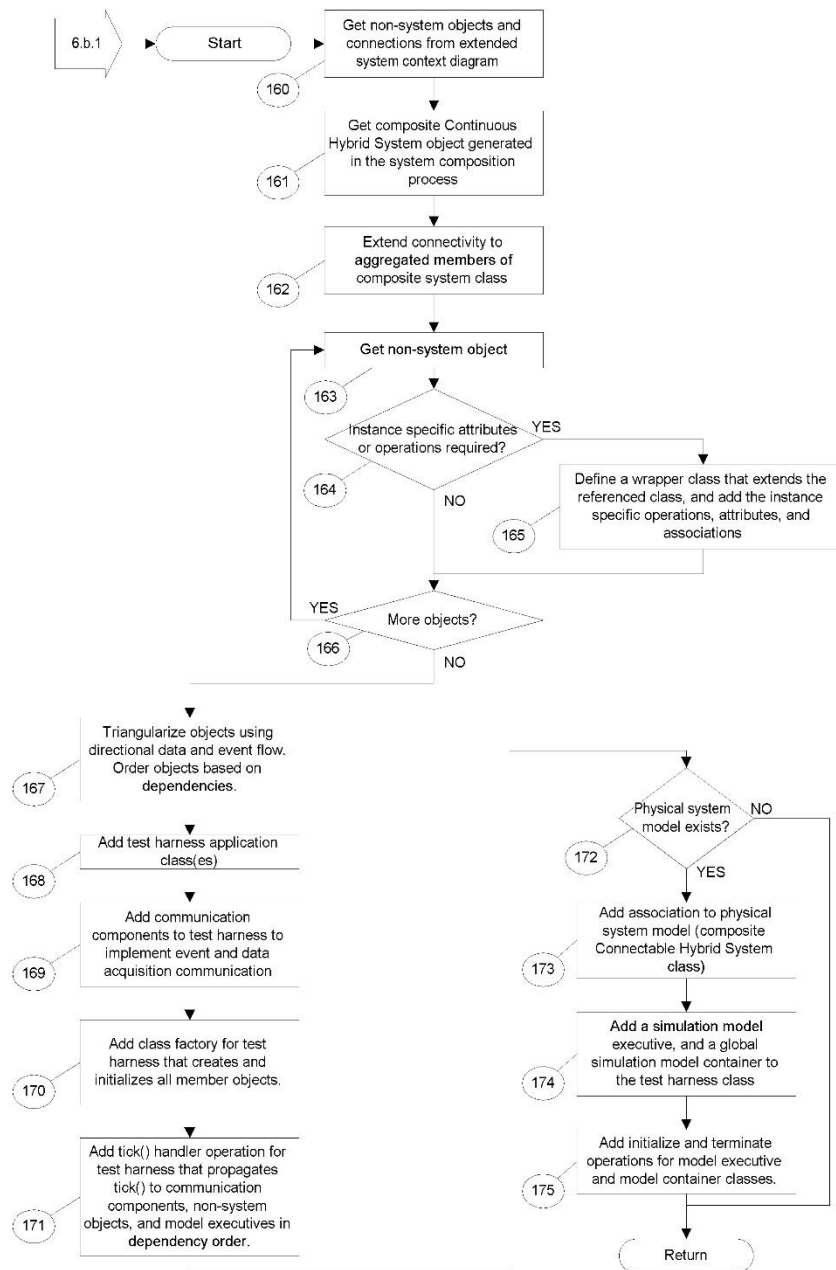


Figure 6b illustrates how an executable application package can be generated from an extended context diagram.

Figure 7a) Host Test Environment

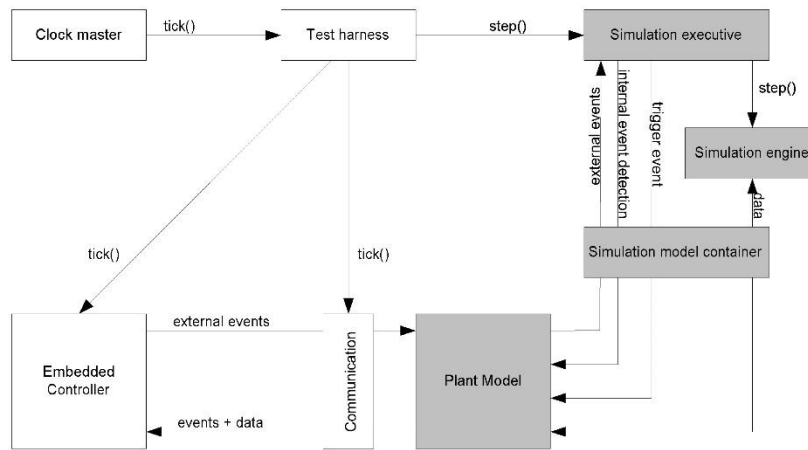


Figure 7b) Target Embedded Control Environment

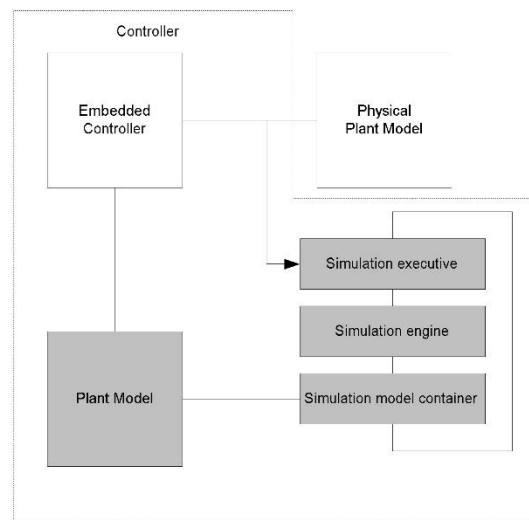


Figure 7a illustrates a sample configuration of how a controller can be integrated with physical system models for software validation and testing.

Figure 7b illustrates a sample configuration of how an advanced embedded controller can be integrated with a physical system model.

Figure 7c) Bidirectional connection initialization

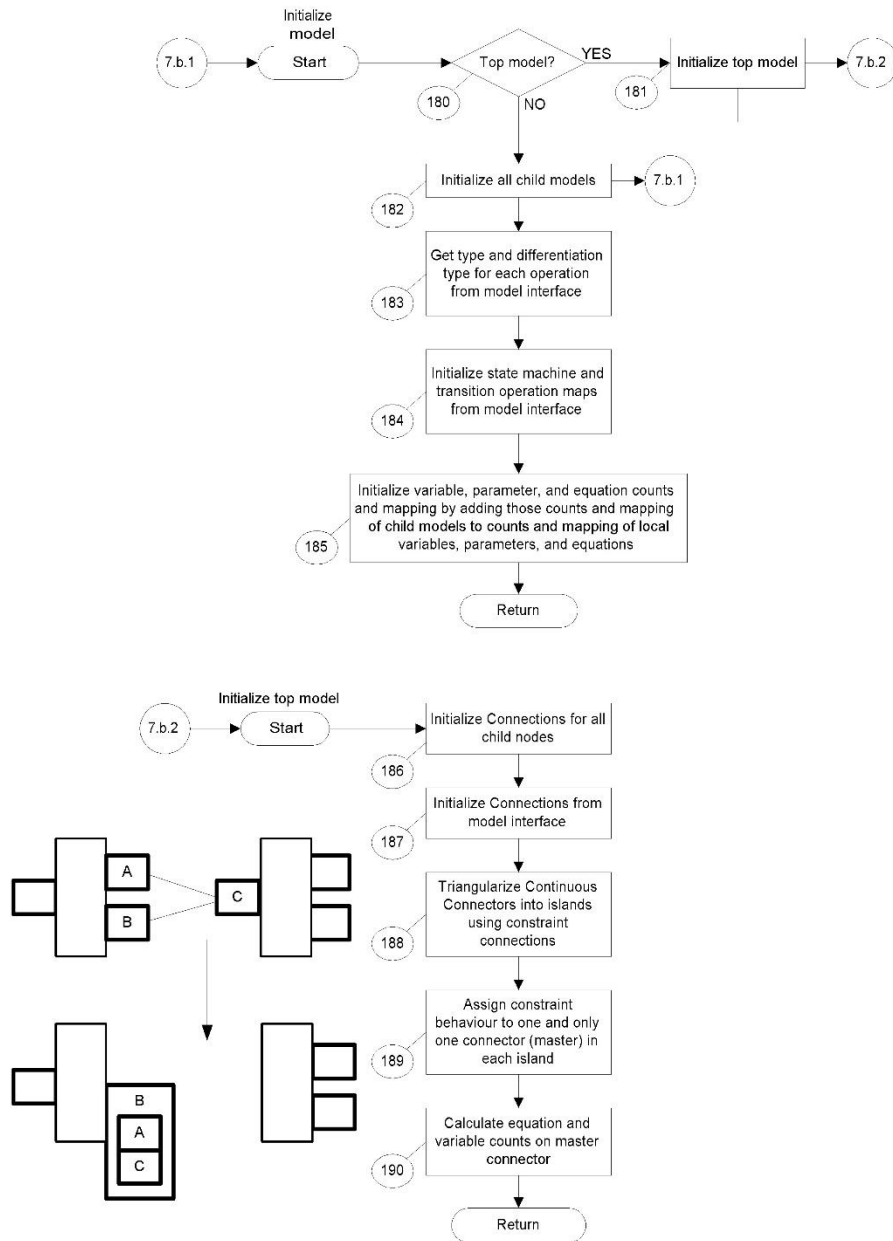


Figure 7c illustrates how bi-directional constraint connections are realized during the runtime initialization of a physical system model.

Figure 7d) Sample step execution

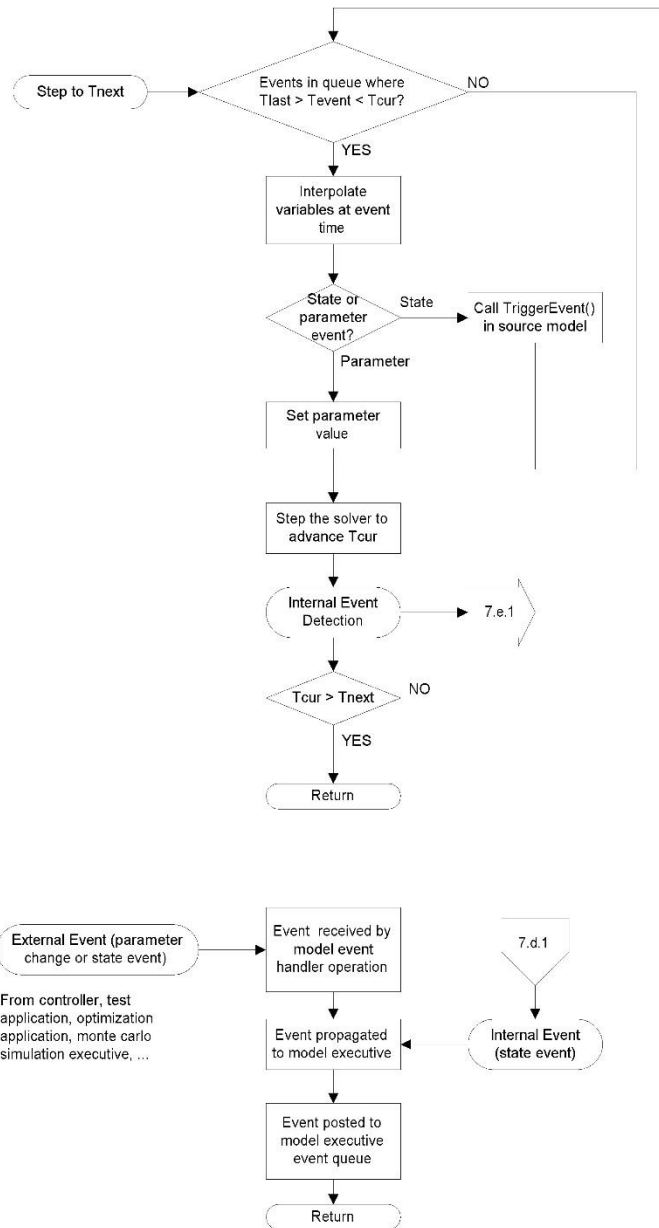


Figure 7d illustrates a sample step execution by a simulation executive.

Figure 7e) Sample internal event detection

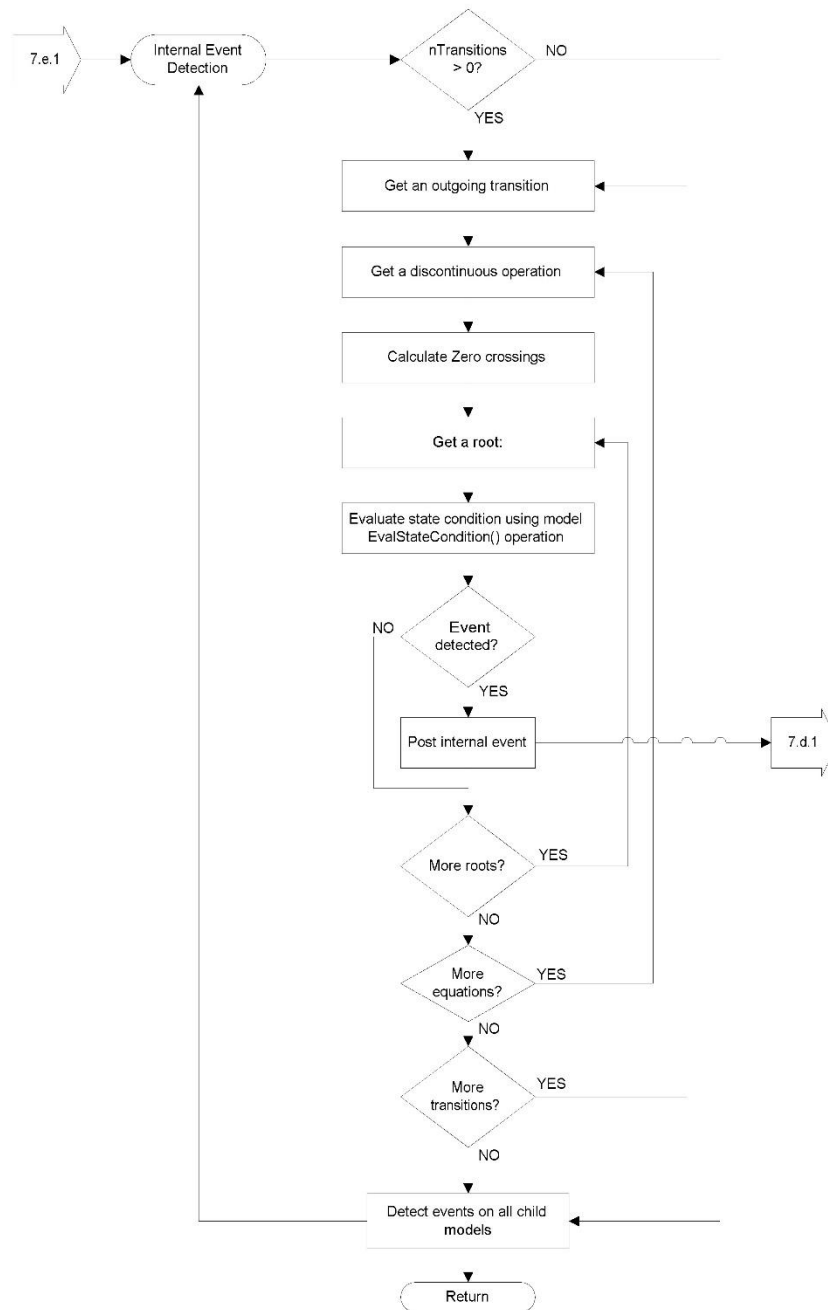


Figure 7e illustrates a sample flow diagram for internal event detection.

Figure 7f) Event handling overview

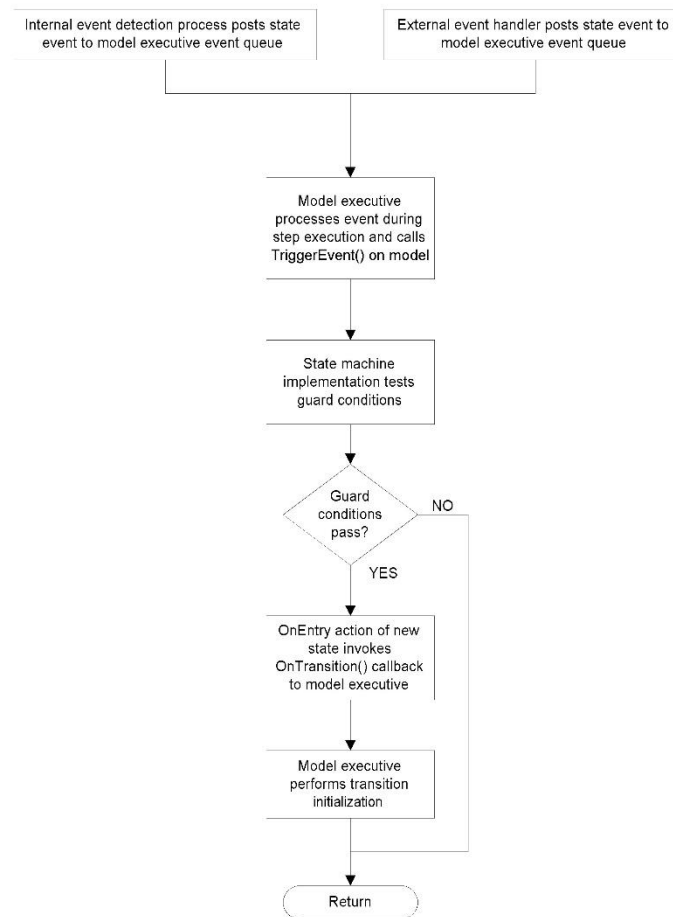


Figure 7f provides an overview of how events are propagated and processed.

Figure 7g) Sample transition initialization

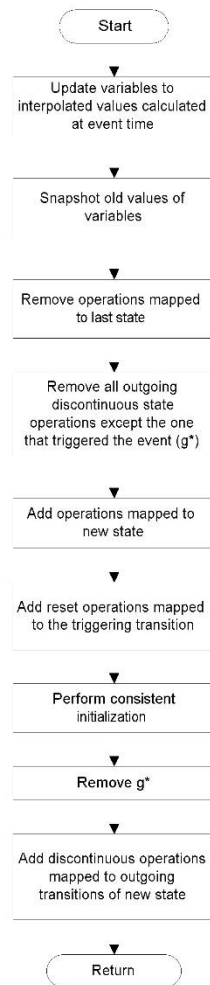


Figure 7g illustrates how standard UML state transitions are handled within a physical system model.

Detailed description

This disclosure describes the import of declarative and state-space physical system models and transformation to executable UML physical system models. Composition of physical system models, and integration of software and physical system models are stored and edited in an intermediate diagram type referred to herein referred to as an extended context diagram. The extended context diagram is similar to an object diagram, but provides additional convenience for generating executable models by supporting event, data, and constraint connectivity between nodes (objects).

Figure 1a illustrates the stereotyping of UML elements used to define UML continuous executable physical system models.

The continuous model stereotypes illustrated in Figure 1a 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. Decomposition and mapping of Parameters and Variables is addressed on each Continuous System model until only primitive UML data types remain. This process is described in detail in figure 5a.

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, the remainder of this disclosure 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 1b 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.

Figure 1c illustrates the stereotyping used to define UML discrete models.

Discrete models are recalculated using a periodic clock tick. Sampled systems like controllers are the most common discrete components. Some physical system modeling is also done using discrete difference models.

Discrete models have directional connections. A Discrete model may be comprised of Parameters, message handling, and message broadcasting operations (in addition to other class defined attributes and operations).

Figure 1d illustrates Node and Node Definition elements used for extended system context diagrams.

Extended system context diagrams can be used to create or extract composite physical system models, and composite applications, where the members may be a mix of continuous and discrete models. The connections on an extended system context diagram include event, data flow, and physics-based constraints.

The Node Definition element is a library component equivalent to a UML class. It may be comprised of a graphical representation, port definitions, executable connectivity elements, and a reference to a Continuous Hybrid System model or a Discrete model (where the referenced component exists in the UML repository), or an external software model (where the referenced model has not yet been imported into the UML repository). In order to support varying connectivity requirements, executable connectivity components can be integrated into the Node Definition, where these are added to the executable model generated during node transformation.

A Node element is a component equivalent to a UML object. A Node is an instance of a Node Definition.

A port is a connection interface for a Node. There are 3 types of ports: input, output, and constraint. Composite ports supporting composite signals (protocols) are not shown for simplicity and clarity. Input ports are associated with class message handler operations. Output ports are associated with class message broadcaster models. Constraint ports are associated with class Continuous Connector models.

Ports may be further sub-typed for asynchronous and periodic message communication.

Figure 2a introduces the workflow of the patent. Import physical system models (State-Space models, Non-UML declarative models, UML Declarative models, and Declarative models) and generate executable UML physical system models.

Figure 2b compares the ARTIST coordinated solutions to today's solutions. ARTIST supports general development of simulation models within one tool. ARTIST enables coordinated development, testing, and validation of software under test and auxiliary software, where physical systems used for unit testing can be inter-connected using plug-and-play connectivity (block diagramming).

Today, simulation models for complex systems are often implemented in domain specific tools. State space tools do not support physics-based plug-and-play connectivity; so physics-based inter-connections between physical system models are often not possible. Where inter-connection is possible, it often involves tool API to tool API connections which are cumbersome to create and expensive to maintain. Simulation coordination between tools generally involves sequential modular techniques, which can lead to instability for stiff problems.

Software processes require integrated documentation and solutions, powerful version control, and integrated test environments. ARTIST plus UML tools provide a powerful solution.

Figure 2c introduces the concept of a node, an element used in the extended context diagram used for system and software composition (introduced in figure 2e).

Extended context diagrams are an intermediate documentation type used to define executable UML physical system compositions, and executable physical system + discrete component applications. This intermediate documentation type is used for creating and importing composition diagrams from external tools.

A node definition is a library component analogous to a shape definition in a shape library. A node definition is associated with, and therefore analogous to a UML class. The example shown is a clutch model with 3 physics-based constraint ports (1 – hydraulic servo pressure connection, 2,3 – rigid mechanical connections on the clutch shafts).

A node is analogous to a shape on a composition diagram. A node is an instance of a node definition, and therefore analogous to an object. The example is a clutch instance in a transmission extended context diagram.

An input port is associated with a message handler operation. The example shows an input port on a vehicle node that receives throttle position data.

An output port is associated with a message broadcast component. The example shows an output port on a vehicle node that transmits vehicle speed data.

A constraint port is associated with a type of Continuous Connector. The example shows the drivetrain shaft connection to the drive wheels of a vehicle node. The connector would define acceleration, torque, and velocity constraints between the vehicle model and the drivetrain model.

Figure 2d introduces the relationship between a node definition, and an executable UML continuous physical system model. The example illustrates a clutch actuator node associated with a clutch actuator UML class.

The node has two ports. The input port connects to the source of engage / disengage messages. The `ec_Engage()` and `ec_Disengage()` operations on Actuator class are mapped to this port to handle these messages. The constraint port connects to a clutch model and defines a constraint that the actuation pressure be equal between the models. The `clutchConnector` association on the Actuator class is mapped to the constraint port to implement the constraint.

Figure 2e illustrates an application of the extended context diagram. Seven node definitions have been created in a library. Each node definition has an associated continuous model. The continuous model could be a previously created executable UML continuous class, or simply a reference to an external continuous model.

The node composition and node associations are used to transform this extended context diagram into an executable UML composite continuous package.

Figure 2f introduces another application of the extended context diagram. The example shows a mix of discrete and continuous components connected by event and data flow messages. The throttle, brake, and gear shifter nodes are mapped to discrete UI components (auxiliary software). The transmission controller node (software under test) is mapped to a discrete transmission controller.

The throttle node output is connected to the throttle inputs on the vehicle model, and the transmission controller. The connections are mapped to connectivity components in the executable model.

The vehicle node is mapped to the composite vehicle with drivetrain executable continuous class generated in figure 2e.

Figure 3a illustrates the process of generating an extended context diagram, and generating an executable model from the diagram.

The extended system context diagram is a powerful intermediate document type for importing composition diagrams depicting data flow, event flow, and / or physics-based constraint connections from external tools.

Nodes (similar to Capsules) have ports for defining data flow, event flow, and physics-based constraint connectivity. A class can have many node representations to address different connectivity requirements.

The sequence of operations outlines the process of defining nodes, adding them to the diagram, and inter-connecting them. The first step (1) involves populating a library with node definitions. Instances of nodes (each representing an object) can then be added to an extended context diagram (2). The role name of each node in the composition must be defined, along with non-default initial conditions (3). Add connections between ports to graphically define event flow connections, data flow connections, and physics-based constraints (4). Validate event and data flow connections using underlying data types. Validate physics-based constraint connections by ensuring all connectors involved in a connection are instances of the same connector class (5). Transform nodes by importing models where necessary, and generating associations to the resulting classes (6). The top composite plant package may be generated in UML by creating a composite class, adding associations to referenced physical models, and adding connections (7). The top application test package may be generated in UML by creating a composite class, adding the top composite plant package, the software under test, and referenced auxiliary software (8).

The process of generating the extended context diagram illustrated in figure 3a could be manual or automated. The automated process would typically involve importing a model composition diagram from another tool, and propagating as much information about the composition and node associations as could be gathered from the other tool.

Figure 3b illustrates the process of creating a node definition. The process involves creating an icon, adding sufficient connectivity ports for the application, and associating the node with a UML class or an external tool model, and the node's ports with executable connectivity elements.

Connectivity requirements cannot reasonably be predicted for all applications of a class. The node definition therefore supports the internal definition of executable connectivity elements including message receivers and transmitters, and associations with Continuous Connector classes. The node translation process will extend the referenced class with these connectivity elements via a wrapper class.

The process begins with the creation of a node icon (10). The icon may be any graphical representation that includes graphical representation of ports (connectors supporting connections with other nodes). Create an association with a UML class, or a model defined in an external tool (11). The definition of this association may be deferred and added to the node definition at a later time. Define default values for initialization of the

model (12). Add ports to support the definition of communication for events, data, and the definition of constraints (13). Three port types are depicted {input, output, and constraint ports}. When defining an input port, the input handler (data or event handler operation) may or may not already exist on the referenced model (14). If the handler does not exist, add an event handler operation to the node (15). Associate the event handler operation with a port defined on the icon (16). When defining an output port, the mechanism or attribute for broadcasting events or data to external classes may or may not already exist (17) on the associated model. If the broadcast mechanism does not exist, add a broadcast attribute or association to the node (18). Associate the broadcast mechanism with a port (19). When defining a physics-based constraint port, a physics-based connector may or may not already exist (20) on the associated model. If the connector does not exist, add a connector to the node (21). Associate the connector with a port (22).

Figure 3c illustrates node transformation to an executable model.

Node transformation may be invoked while some nodes still have no association to a model. Therefore, if a node is not mapped to a model (decision block 25), a skeleton UML class is generated using the node's role name (26). The model may have been imported previously. If the model can be found in the UML repository (decision block 27), import is not necessary; otherwise, the import model process is invoked (reference 4.b.1). Ports may have connections that don't map to attributes, operations, or associations of a model. It is impossible to foresee all applications of a software model. Therefore, to support unforeseen connectivity requirements, it is possible to define ports that don't map to attributes or operations of the referenced model. If ports exist that don't map to model elements (decision block 29), a UML wrapper class is generated for this application of the model (30). The wrapper class aggregates or derives from the referenced model. Unmapped port attributes and operations are added to this wrapper class in subsequent steps. Node event and data flow attributes and operations are added to the wrapper class (30), or the empty class created in (26). These include event and data broadcast and receive mechanisms. Unmapped constraint ports not only require the addition of Continuous Connectors, but they also require the addition of constraints to balance the addition of variables (each constraint connector will have one or more member variables). A loop which will be broken by decision block (32) when there are no more unmapped constraint ports, adds associations to specified continuous connector classes (33), and adds constraints for variables in a second inner loop which will be broken by decision block (34) when no more variables exist on the continuous connector. For each connector variable, a variable in the referenced model must be selected (35), and a constraint must be added equating the two variables. When all unmapped connectors have been added (if any), the process exits.

Figure 4a illustrates how object-oriented and state-space models may be imported to UML classes. The categories of models depicted are software under test (SUT), and physical system and auxiliary models that make up the environment of the software under test.

State space models (compositions of procedural models with directed data flow) can be imported where each block is translated one-to-one to a class, and each connection is translated to a constraint or data flow. This is generally undesirable for real-time performance, readability, and practicality reasons. Therefore, the user may organize blocks into groups (40), where each group is imported to a UML class (41) (reference 4.c.1). This process continues until all blocks / groups are imported (decision block 42).

Component models that are available in binary form only (third party library components, COM components) are imported indirectly via wrapper classes (43). The wrapper exposes the interfaces of the external class, and defers the implementation to the external class.

Object-oriented models are imported along with associated models, and more general models (44) (reference 4.b.1).

If a model represents declarative or procedural specifications of physical constraint (decision block 45), said declarative and procedural specifications are extracted and an executable physical system model is generated (46).

Figure 4b illustrates how external classes are imported to UML classes.

The first step (50) is to identify the class with a stereotype as defined in figures 1a-c. Next, primitive types (enums, type definitions, interface definitions, etc.) are added to the UML repository (51). A class is added using the source model name (52). If the source class is derived from more general classes (decision block 53), the more general classes are retrieved (54) (reference 4.b.2), and an inheritance relationship is established with the more general class (55). While more attributes (or associations to other classes) exist in the source model (decision block 56), attributes are added directly to class if primitive (57, 58), or the associated UML class is retrieved (59) and an association relationship is created with the attribute class (60). Operations are copied from the source class (61), and any state and transition specification is copied to a standard UML state chart (62). UML classes are retrieved (reference 4.b.2) by attempting to locate the class in the repository (63), and if not found (decision block 64), the class is imported (65) (reference 4.b.1). If the import is not successful (decision block 66), a new UML class is added (67).

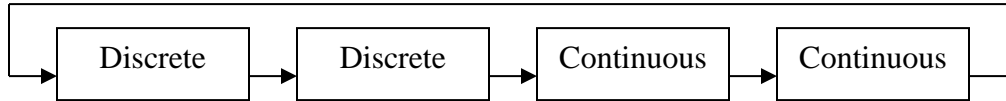
Figure 4c illustrates the import of a state space model into a UML class.

There are two fundamental classifications of attributes and operations: continuous and discrete.

A simulation solver updates the value of a set of continuous attributes. The solver follows a time-based trajectory satisfying the constraints defined by a corresponding set

of continuous equations. When the set of continuous constraints is satisfied, the set of continuous attributes is said to be consistent.

A set of discrete operations updates the value of a set of discrete attributes directly. On each clock tick, the set of discrete operations are invoked to update the discrete attribute values.



The above sketch illustrates 4 types of connections that can be found in state space models:

Discrete → Discrete

A discrete attribute (result of a discrete operation) may be used as an input to another discrete operation. Execution order is important (discrete operation execution should follow dependency order).

Discrete → Continuous

A discrete attribute (result of a discrete operation) may be used as an input to a continuous operation. Consider the example of a continuous operation defined by $A * X1 + X2$ where A is updated from a discrete attribute, and X1 and X2 are continuous attributes. If X1 and X2 are consistent for a given value of A, X1 and X2 will be inconsistent when the value of A is modified. Updating the value of A is a time-stamped parameter change event, and the solver must handle this event by updating the values of X1 and X2 to make the set of continuous attributes consistent again.

Continuous → Continuous

A continuous attribute X2 defined to be equal to another continuous attribute X1 simply defines another constraint ($X2 = X1$). This constraint could be implemented through variable reduction (replace all references to X2 with references to X1), or through an additional constraint equation.

Continuous → Discrete

A continuous attribute may be used as an input to a discrete operation. The value of the continuous attribute must be determined at time t_K .

State-space models may define a mix of continuous and discrete attributes and operations. The attributes and operations are separated based on the two classifications, and connections are handled based on the source and target classification.

The first step in the import process is to add a UML class for the source block or block group (70).

Attributes are added or updated for block inputs (u), state variables (x), parameters (p), and outputs (y) (71). The container into which the said attribute additions or updates are inserted could be the UML class from 70, or an operation (body and / or function

prototype) from reference 4.c.2. Said attributes are stereotyped as follows: {u, x, y} are stereotyped as variables, and { p } are stereotyped as parameters.

If the block or is not composed from other blocks (a leaf block), its operations are analyzed (starting from decision block 73) (decision block 72).

Operation type classification is determined as continuous or discrete (decision block 74). If the classification is not apparent from the source model, the user must provide this information. Continuous operation type is determined as algebraic or differential (decision block 75). If algebraic, a Continuous Dynamic Equation operation is added or updated (76). The explicit declarative specification in the source model may be implemented as explicit $y = f(x,u,p)$, or implicit $y - f(x,u,p)$. Placing constraints into a common constraint operation is an inline implementation optimization. Eliminating algebraic constraints through variable elimination is problem definition optimization. If differential operation type, a Continuous Dynamic Equation operation is added or updated (77). The explicit declarative specification in the source model may be implemented as explicit $\text{der}(x) = f(x,u,p)$ or implicit $\text{der}(x) - f(x,u,p)$. Discrete operation type is determined as algebraic or difference (decision block 78). If discrete algebraic, add or update a Discrete Algebraic Equation operation (79). If difference operation, add or update a Discrete Difference Equation operation (80).

If block is a composition of blocks (decision block 72), expand block to signal flow graph (81). A signal flow graph is comprised of member blocks, and connections. The discrete to discrete signal connections may be used to determine dependency of discrete operations, whereby defining execution order (82). Ordering information available in source model may be used if available.

Continuous to continuous signal connections are interpreted as continuous constraints. These constraints may be implemented by adding or updating an algebraic constraint operation (83). These algebraic constraints may alternatively be used for variable reduction.

Discrete to continuous signal connections represent procedural specifications of the controlled physical device, and may be implemented as parameter change events. Continuous to discrete signal connections may be handled as data acquisition connections where data is extracted for time t_k .

While there are more blocks in the graph (decision block 84), the remaining blocks will be processed. There are 2 ways blocks can be imported, inline along with other blocks, or as a distinct class. When the user groups blocks for import, they define which blocks are to be imported inline, and which are to be imported to their own class. If a block is not to be translated to a UML class (decision block 85), continue updating the current class's operations and attributes (86) following reference 4.c.2. If the block is to be translated to a UML class (decision block 85), the repository is searched to determine if the class has been imported previously. If the translated UML class does not exist (decision block 87), import the state space model (88) (reference 4.c.1). If the import

failed (decision block 89), a skeleton UML class is created in its place (90). An association is added to the referenced UML class (91).

Figure 5a illustrates the attribute typing and mapping process for generating an executable UML physical system model from a UML class.

The first step involves identifying the stereotype of the class per the stereotypes introduced in figure 1a. Some or all stereotype assignments may be complete depending on the source of the UML class (100). Lists of class attributes, operations, states and transitions are compiled (101). Associations are treated as attributes for the purpose of compiling the lists. If the referenced class is derived from more general classes (decision block 102), the attribute, operation, state and transition lists are expanded from the base class(es) per implementation language rules, or defined generalization rules (103). Overloaded operations for example may be taken from the most derived class.

When the attribute, operation, state and transitions lists are complete, attributes associated with simulation are flagged (104). For those attributes associated with simulation, stereotypes are defined (105) per figures 1a-c. A class may contain attributes that are not associated with continuous behaviour. Discrete control, continuous system, and auxiliary software may be freely mixed in the same class. Those attributes that are not associated with continuous behaviour will be ignored.

Mechanisms must be defined for decomposing each attribute to primitive Variable and Parameter scalar elements (106). Standard mechanisms exist for arrays, collections, matrices, user-defined types etc., and custom mechanisms may be added. Reference to sizing information may be required to decompose non-scalar attributes. Examples include array and matrix dimension attributes.

The decomposition mechanism, and Variable vs. Parameter stereotyping must be defined for all attributes. This information is herein referred to as extended definition. The list of attributes is examined for aggregated classes or structures that don't have a complete extended definition (107). If any are found (decision block 108), generate executable physical system model(s) for said undefined items (reference 5.a.1). When all items have complete extended definition, generate the attribute interface (110) (reference 5.a.2).

The attribute interface is generated using the above stereotype and decomposition information, to support communication of variable and parameter data from a simulation executive or simulation engine. A simulation engine updates the values of variables. Parameter value changes are events that must be handled by a simulation executive. The attribute interface is used to define variables and parameters, and to read and write their values.

Attributes are extended with alternate forms (differential, pre-event, pre-event differential, and operator overloading) (111). With these alternate forms, a variable X has a differential form $\text{der}(X)$, a pre-event form $\text{old}(X)$ generally used in Continuous

Reset Equation operations, a pre-event differential form $\text{old}(\text{der}(X))$, and an operator overloaded form $\text{ad}(X)$ generally used to calculate partial derivatives through operator overloaded automatic differentiation. Init and clean operations are added to initialize and cleanup all attributes in all forms (112).

An ordered map of Connectable Hybrid System sub models and Continuous Connector models is added accessible for runtime initialization of model (113). The decomposition scheme(s) and an ordered map of Variables and Parameters are added accessible for runtime initialization of model (114).

This alternative to writing continuous models in terms of work vectors provided by a simulation engine makes it possible to take any UML class and transform it into an executable continuous model. The process continues with operation typing and mapping (115) (reference 5.b.1).

Figure 5b illustrates the operation typing and mapping aspect of generating an executable UML model from a UML class.

The first step involves identifying which operations are associated with simulation (120). Operations are identified with stereotypes (per figures 1a-c) (121). Operations not associated with continuous behaviour are ignored. The operation interface is generated (122) (reference 5.b.2).

Define an ordered map of operations including type and differentiation type accessible for runtime initialization of model (124). Define load / save operations (125). Define load / save operations. If states exist, current state is saved and restored. Additional state information beyond variables and parameters must be saved and restored by manual additions to shell load / save operations. Load and save operations are generated automatically for all continuous models. These operations are added to support the load and save of simulation snapshots. The attribute interface is used to load and save all parameter and variable values. The load and save operations enable additional user-defined state information to be loaded and saved.

The process continues with state chart mapping and integration (123) (reference 5.c.1).

Figure 5c illustrates the mapping and integration of standard UML state charts into hybrid models.

Physical discrete models of a controlled physical device may be represented as states in a UML state chart, and SDL state machine, or a standard state machine implementation.

Define Continuous State and Continuous Transition stereotypes (per figure 1b) for model (130).

Define state event triggers for each transition (131). One or more associations to the class's Discontinuity Equation operations define a state event trigger. When all associated Discontinuity Equation operations are satisfied, internal event detection logic will detect that the state event trigger is satisfied, and will trigger the transition.

Transitions are categorized for internal or external triggering (132). A transition with no state event triggers cannot be internally triggered. Externally triggered transitions further define the procedural specifications for a model, and may be implemented by a state event handler operation that will post the event to the event queue rather than acting directly on the state machine implementation. This enables events to be processed on the event queue in the correct order, and with validation.

Each Continuous State is assigned an equal number of continuous equations, defined in Continuous Dynamic Equation operations (133). The selection of equations is dependent on the required behaviour for the state. Equations may be common to many states.

Assign reset equations (defined in Continuous Reset Equation operations) to transitions requiring additional continuity constraints during transition initialization (135).

Generate the state chart interface (136) (reference 5.c.2).

Generate Continuous State to Continuous Dynamic Equation operation maps accessible for model initialization (137). Generate Continuous Transition to Discontinuity Equation and Continuous Reset Equation operation maps accessible for model initialization (138). For each transition, the source state must be identified. The target state need not be identified, and often cannot be identified in cases of complex transitions.

Generate state event handler operations based on the external event procedural specification (139). State event handler operations will post state events to the event queue.

Internally triggered transitions require a set of discontinuous operations (state event triggers) for internal event detection. The sets of discontinuous operations associated with each state event trigger are encapsulated into an EvalStateCondition() operation. When zero crossings are detected in Discontinuity Equation operations, a simulation executive may use this procedural implementation to determine if a state event should be posted to the event queue.

The event queue requires a mechanism to trigger events. There are several options for the triggering mechanism for state transition. The mechanisms for triggering the state transitions are encapsulated in a TriggerEvent() operation (141).

As noted above, the target state is unknown to the solver and model executive for several good reasons. Transitions may be complex and guard conditions may require validation. An OnTransition() operation is added to provide notification back to the simulation

container model that a state transition has taken place (142). The OnEntry action block for each Continuous State is extended with a call to the OnTransition() operation (143).

With the composition of maps and operations added per the above, all transitions are synchronized with the event queue, internal transitions are detected, a standard triggering mechanism exists back to the state machine, and the continuous model can be initialized with the appropriate operations as each state becomes active.

Figure 6a illustrates the process for generating an executable composite physical system package in the UML repository.

Get the Connectable Hybrid System objects and connections from the extended context diagram (150). An extended context diagram may be composed from software under test objects (SUT), auxiliary software objects (AUX), and Connectable Hybrid System objects. Connections may include event, data, and constraint connections.

For each Connectable Hybrid System object (151) (decision block 154), determine if instance specific attributes or operations are required (decision block 152). These additional model elements may be defined on the node. If additional requirements exist, a wrapper class is added that extends the referenced model, and the instance specific attributes, operations, and associations are added (153).

A new composite Connectable Hybrid System class is added to the repository (155). Member physical system classes are aggregated. Role names are assigned per node role names.

Initialization and cleanup operations are added to initialize object attributes per settings defined in the node (156). A class factory is added that creates instances of the aggregated classes or wrapper classes (157). An ordered map of aggregated classes is added accessible for model initialization, and accessible for further model integration (158). Add a connectivity map defining leaf connection implementation using ordered maps of from the composite system class and its members (159). This map will be used to initialize the leaf connections of the composite model during initialization.

A composite Connectable Hybrid System model is functionally equivalent to any leaf Connectable Hybrid System model. While the above steps outline how a composite model may be built by aggregating other Connectable Hybrid System models, composite models may also be inline optimized by extracting the declarative specifications from the member models to build a self-contained composite model. Additionally, adding Continuous Connectors, and algebraic constraints via Continuous Dynamic Equation operations enables physics-based connectivity for the composite model in larger physical systems.

Figure 6b illustrates the process for generating an executable application package in the UML repository.

Get non-system objects and connections from the extended system context diagram (160). This would typically involve software under test (SUT), and auxiliary software.

Get the composite Continuous Hybrid System objects generated in figure 6a (162). The continuous models (unless there was only 1 continuous model) have effectively been removed from the extended context diagram by the process of generating a top composite physical system package. Since many of the Connectable Hybrid System models aggregated into the composite system model have other data and event connections in the extended context diagram, the connectivity to these models is extended. The members of the composite system model will be read from the interface maps, and will be exposed for connectivity in the application class.

For each non-system object (163) (decision block 166), if instance specific operations or attributes are required, a wrapper class is defined that extends the referenced class. The required instance specific attributes, operations, and associations are added to the wrapper class (165). Instance specific procedural requirements not directly supported by the referenced class may include a clock tick handler, load and save operations for snapshot support, event handlers, or broadcaster attributes missing from class definition, or defined on the node.

The default mechanism for ordering procedural execution is to extract the dependencies for the discrete components (167). Alternative approaches may be used, particularly if the order was originally defined in an external tool.

Test harness or application class(es) are added to the repository (168). Communication components are added to the test harness to implement event and data acquisition communication (169). A class factory is added to the test harness that creates and initializes all member objects (170).

Add clock tick handler operation for test harness that propagates clock ticks to communication components, non-system objects, and model executives in dependency order (171).

If a physical system model exists (decision block 172), add an association to the top composite Connectable Hybrid System model (173), add a simulation model executive, add a global simulation model container are added to the test harness (174), and add initialize and terminate operations for simulation model executive and model container classes (175).

A simulation model executive will manages the simulation event queue, the simulation engine and the physical system models, and the clock interface to the test harness. A simulation model container provides a standard interface to models for the model executive and the simulation engine. The simulation model container is layered on top of the physical system implementation classes.

Figure 7a illustrates the result of the process illustrated in figures 6a-b. During execution, a clock master will drive all continuous and discrete components with a series of clock ticks. The ticks are forwarded to all discrete and where necessary all communication components. The clock master advances simulation by stepping the simulation executive to T_{next} on each clock tick. The simulation executive manages the event queue, processes events, monitors for events, and steps the simulation engine.

Test case specification

Test scenarios involve testing a systems response to specified stimulation, starting from specified initial conditions.

Software Under Test (SUT), auxiliary software, and the physical system can all be implemented using the same language. Test scripts written in this language directly access any class in the test harness. Parameter changes on SUT or AUX software are handled by those components, generally without complexity because they are typically discrete software components. Parameter changes and external events may be applied to physical system models because these changes are automatically routed to and handled by the simulation event queue. Variables and parameters of all models can be read directly from each component model.

All physical mode transitions are implemented in standard UML state machines; so all events can be monitored, and even subscribed for from the model executive that processes the event queue.

Initial conditions for test cases can be recorded by running the test harness from known consistent initial conditions, to the conditions of interest, and recording snapshots of not only the physical system and simulation environment, but also of the SUT and AUX software. The required support for snapshot capability has been highlighted throughout this disclosure to support this feature.

The test case may therefore be designed and implemented per the following steps:

- (1) Define and implement required external stimulus by injecting events and data to various components of the test harness.
- (2) Run the test harness from known initial conditions to required test conditions (possibly interactively with the end user through faceplates and charts), and snapshot the state of the test harness when defined conditions are achieved.
- (3) Run the test harness from defined initial conditions by loading snapshots, and inject stimulus as defined by test requirements.
- (4) Record information of interest, SUT response, AUX response, physical system response, or information derived from these responses through

- calculations or comparisons to data derived from alternate sources (lab data, field test data, alternative modeling approaches).
- (5) Identify pass / fail criteria.

Test cases may be defined that manipulate SUT or physical system parameterization to achieve a desired result, or response comparable to another data source. These cases may employ an optimization engine for parameter selection, a cost function measuring the result, and the test harness running from a specified start time to a specified end time, starting from specified initial conditions.

Figure 7b is very similar to figure 7a except the physical system model, and simulation components are running inside the controller itself for the purpose of advanced diagnostics and control. In this application, the simulation software is downloaded to the embedded controller and executed for the purpose of controlling a physical device.

Figure 7c illustrates how models are initialized at runtime, and in particular, how executable continuous models are realized, and composed within a UML environment.

Physics based connections must be realized after all connections have been made. This is would appear to make plug and play components realized at design time unachievable.

State space models require that physics-based connections be decomposed for the entire model before implementation can begin. If any changes are made to the model connectivity, in most cases, the design process and entire implementation must be regenerated manually.

Modeling languages like Modelica address physics-based connectivity during symbolic translation from the Modelica mathematical modeling language.

The approach of this disclosure is to define scalable Continuous Connector models that scale automatically based on the number of connectors linked directly or indirectly to each other.

Model initialization tests each model to determine if it is the top model (decision block 180). If it is the top model, additional initialization steps supporting connectivity are required (181) (reference 7.b.2). All child models are initialized (182) (reference 7.b.1). The count, type and differentiation type for each operation is retrieved from the model interface (183). The state machine state and transition maps are read from the model interface (184). Initialize variable, parameter, and equation counts and mapping by adding those counts and mapping of child models to counts and mapping of local variables, parameters, and equations (185).

Initialization of the top model begins with initialization of leaf connections for all child models (186). The leaf connection information is retrieved from the model interface

(187). The leaf connections are triangularized into islands where member connections share at least one Continuous Connector (188). The constraint behaviour for each island is assigned to one and only one member connector (189). The number of variables and constraints defined by the selected connector is calculated based on the number of connectors in the island (190).

Hierarchical modeling

This hierarchical approach to modeling is significant for several reasons.

The variables, parameters, and equations of child models are added to the parent, effectively making them local to the parent. The parent model can be written in terms of the variables, parameters, and equations of children. This is the foundation of the physics-based connectivity.

If an electrical connector has 2 variables voltage (V) and current (I), a resistor could be expressed in terms of 2 of these connectors as follows:

attributes:

ElectricalConnector* p;

ElectricalConnector* n;

double Resistance;

operations:

(1) *res = p->V - p->I * Resistance - n->V;

(2) *res = p->I - n->I;

In this simple example, all variables referenced in the 2 equations defining the resistor are expressed in terms of variables of child models.

The connectors themselves rely on the same hierarchical feature. When one and only one connector in a connected island is selected to define the constraint behaviour of the island (here forward referred to as the master), the other connectors in the island are dynamically assigned as children. Extending the above example, if 4 ElectricalConnector are found to be directly or indirectly connected, and thus are placed in the same island, 3 are assigned as members of the one selected to exhibit the constraint behaviour of the island. The variable count on this master connector is $3 \text{ children} * 2 + 2 \text{ local} = 8$. The equation count and procedural implementation can be expressed in terms of the number of children.

Therefore, models can be expressed in terms of their children, and indirectly in terms of their neighbours through their connectors.

Consistent initialization

When the model structure is initialized per the above process, the top model has a defined equation and variable count. If there is a mismatch, this must be reported to the design engineer.

Equation to variable dependencies may be determined through numerical and automatic differential techniques. Graph analysis may be used to detect hidden constraints, and equation – variable assignments that cannot be resolved. Algebraic analysis may be used to determine a consistent initialization for the complete set of variables, satisfying not only the problem constraints, but also said hidden constraints detected during graph analysis.

Problems detected through graph and algebraic analysis areas can be highlighted to the end user. Special tools are not required to present problem areas, particularly the relationship between source model equations and detected structural or numerical inconsistencies, because the source model is not symbolically transformed.

Figure 7d illustrates a sample step execution. On each step, events are processed if necessary, the solver is stepped, internal event detection is performed, and the process loops until the target time T_{NEXT} is met. If state transitions are processed, TriggerEvent() is triggered on the model sourcing the event.

When external events are fired, the events are routed to the event queue. Internal events are routed to the event queue in essentially the same way. External events are often parameter changes. Parameters are the simulation items that can change at discrete intervals with the external discrete components. Handling parameter changes and state events in the event queue enables complete integration of discrete and continuous systems.

Figure 7e illustrates a sample internal event detection algorithm. Of general interest is how the EvalStateCondition() operation added to the state machine interface for hybrid models is integrated into the event detection logic. Zero crossings in discontinuous operations are tested against the logic encapsulated in the model's EvalStateCondition() operation.

Robust solvers must know which discontinuous operation triggered the event to properly avoid chattering at transition points. The scheme illustrated in figure 7e achieves this necessary goal without introducing scripting languages, or awkward simulation specific requirements on the model designer.

Figure 7f provides an overview of the complete path of a state event from detection to initialization.

All events must be queued because other events may also be detected in the same solver step that will occur first. Sometimes events will never occur when other events are processed first.

When an event is processed, the `TriggerEvent()` trigger mechanism is used. This triggers the transition logic on the state machine implementation. Guard conditions may still block the transition so transition cannot be assumed. Also, to simplify the interface to complex state machines, the target state is not mapped, and is not known to the simulation model container.

The `OnEntry` action block in each state calls `OnTransition()`, which is a callback to the simulation executive to initiate transition initialization.

Figure 7g provides a sample transition initialization. The operations of the old state, the new state, the triggered transition, and the new outgoing transitions are all determined based on the operations maps constructed in figure 5c. The snapshot of old values, and the capability to reference the old values in operations is made possible by the attribute mapping and attribute form augmentation performed in figure 5a. Operation access in general was made possible by the operation maps constructed in figure 5b.