VANDERBILT UNIVERSITY

INSTITUTE FOR SOFTWARE INTEGRATED SYSTEMS

Institute for Software-Integrated Systems

Technical Report

TR#:	**ISIS-15-104**
Title:	**CyPhyML Language in the META Toolchain**
Authors:	**Sandeep Neema, Jason Scott and Ted Bapty**

This research is supported by the Defense Advanced Research Project Agency (DARPA)'s AVM META program under award #HR0011-13-C-0041.

Table of Contents

Table of Figures iii

1. Motivation and Design Rationale 1

1.1. Activities in META Design Flow 2

1.2. Heterogeneity of CPS Components 3

1.3. Approach to Model Integration 5

2. CyPhy Language Taxonomy 6

2.1. Components 6

2.1.1. Component Metadata 8

2.1.2. Component Interfaces 8

2.1.3. Domain Models 9

2.2. Design Spaces 10

2.3. Design Evaluation (Test Benches) 14

2.3.1. Dynamics Test Benches 16

2.3.2. FEA Test Benches 16

2.3.3. Verification Test Bench 17

2.3.4. Parametric Exploration Test Bench 17

2.4. Extending the CyPhy Language 18

2.4.1. Components 18

2.4.2. Connectors 18

3. CyPhy Language Formalization 18

3.1. Semantic Backplane 18

3.1.1. MetaModeling Languages 20

3.1.2. Metamodels 20

3.1.3. Metamodeling Tools 22

3.2. Formal Specification of CyPhy 22

3.2.1. Structural Semantics 24

3.2.2. Denotational Semantics 26

3.2.3. Semantics of Signal Port Connections 28

3.3. Formalization of language integration 29

3.3.1. Integration of structure 30

3.3.2. Bond Graph integration 31

3.3.3. Modelica integration 33

3.3.4. SignalFlow integration 34

3.3.5. Power Port Units 35

Table of Figures

Figure 1: META Language Suite 3
Figure 2: Cyber-Physical Components 4
Figure 3: CyPhy as a Model Integration Language 5
Figure 4: Example CyPhy Component 6
Figure 5: Example Structural Aspect of a CyPhy Component Model 7
Figure 6: Example Dynamics Aspect of a CyPhy Component Model 8
Figure 7: Example Design Space Alternative 11
Figure 8: Example Constraints 12
Figure 9: Design Space Exploration. Before and After Constraint Application 12
Figure 10: Example Test Bench 15
Figure 11: Semantic Backplane 19
Figure 12: CyPhy MetaModel 23
Figure 13: Semantic Anchoring Framework 26

1. Motivation and Design Rationale

The Cyber Physical Modeling Language, referred to as CyPhyML or CyPhy, is a Domain Specific Modeling Language designed for modeling, evaluation, and synthesis of Cyber-Physical Systems (CPS). The development of CyPhy was driven by the goals of AVM program towards accelerating design and synthesis of complex CPS. The design of CyPhy is tightly coupled with the design methodology implemented by the META project in support of the AVM program goals. We briefly revisit the key principles of the META design methodology (discussed comprehensively in the Executive Summary and Program Overview sections) to elucidate the requirements for CyPhy.

- Component-Based Design enables design cycle compression by reuse of existing technology and knowledge, encapsulated in integratable and customizable components that can be rapidly used in a design. Components in CPS are heterogeneous, span multiple domains (physical – thermal, mechanical, electrical, fluid, and computational – software, computing platforms), and require multiple models to represent the behavior, geometry, and interfaces, at multiple levels of abstractions. Enabling component-based design for CPS requires a modeling language that allows *multi-domain, multi-model representation of CPS components*.
- Design Space Exploration entails a design methodology that allows a combinatorial design consisting of many architecture and component options. The META design methodology enables a designer to systematically engineer a flexible and comprehensive design space for sub-systems and system that can be explored for satisfying product-specific requirements. Furthermore, the design spaces for subsystems and systems are assets that encapsulate design knowledge, which can be reused in a context different for which it was originally created. A suite of computational methods that trade accuracy with computation time for exploring the large design spaces, allow progressive exploration and reduction of the design space, iteratively converging over to solutions of interest given a set of requirements. Enabling design space exploration requires a modeling language that can methodically *represent design choices at multiple levels of design hierarchy - systems, subsystems, and components*.
- Executable Requirements allow tracking system design through quantitative metrics and ensure that the system design continues to meet requirements as it undergoes evolution and revisions. Enabling this capability requires modeling language constructs that allows capturing requirements in a form that can be automatically evaluated, including the *context in which the requirement must be evaluated, the procedure and the method to evaluate the requirement, and mechanism to extract metrics of interest from the evaluation*.

While these core tenets of the META design methodology shaped the conceptual design of CyPhy, there were additional challenges and considerations that drove the implementation of CyPhy discussed below.

1.1. Activities in META Design Flow

CyPhy has to support a range of design and analysis activities entailed by the META design methodology as follows (see Figure 1).

1. Initial *Architecture design* involves modeling and rapid Exploration of early design space sketched out with the system requirements. These activities in design flow require that the modeling language includes concepts for modeling system design space and constraints, enable representing the key architectural variants that can broadly support the customer requirements. The early architecture exploration also requires low compute intensity methods that can allow examination of a potentially very large set of design options. The modeling language needs to support modeling low-resolution components to enable coarse grain exploration.
2. The Integrated *Multi-Physics and Cyber Design* stage expands upon the broadly identified architecture, and refines them with integrated design of physical and cyber components and conducts relevant tradeoffs. These activities in addition to modeling of the design space and constraints require dynamics modeling for progressively refined performance simulation of the system. This phase also requires support for computational modeling to analyze the behavior and interaction of software with physical components. Geometry and geometry-driven analyses are central to the physical nature of CPS, and consequently the modeling language suite needs to support CAD and derivative analysis such as thermal, FEA, and allow evaluation of designs for manufacturability.
3. The *Detailed Design* stage involves further refinement, and analysis, of designs leading towards production, which shifts the emphasis of modeling capabilities from design space to domain model elaboration.

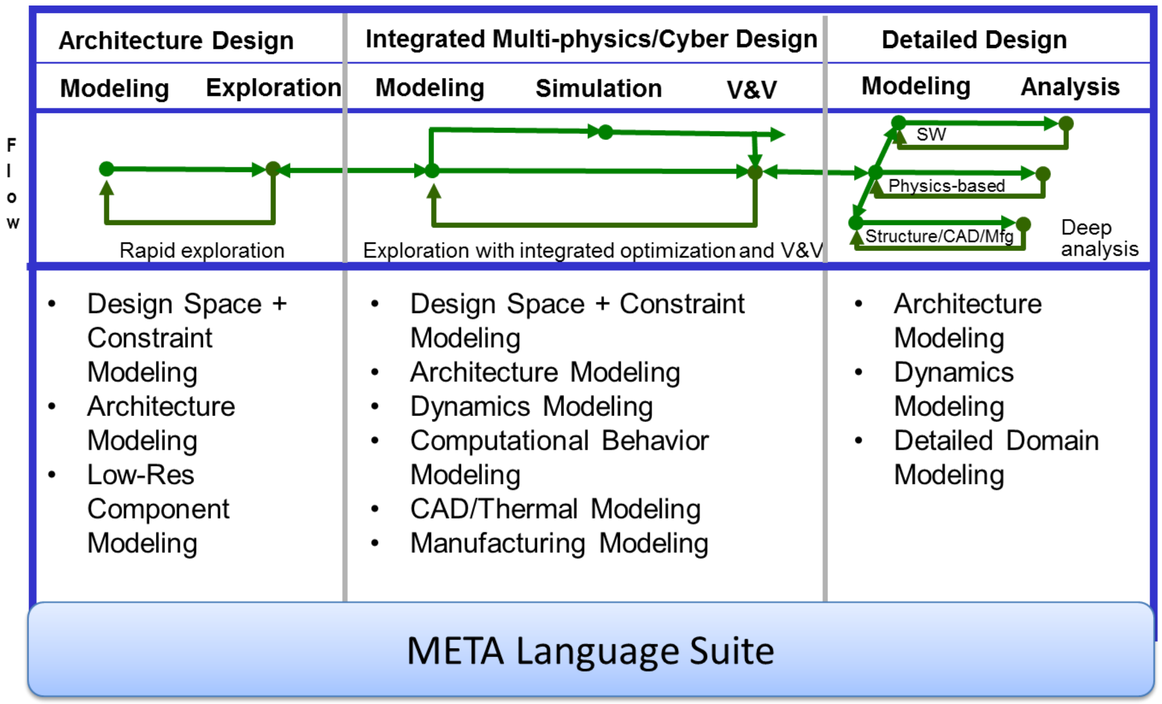


Figure 1: META Language Suite

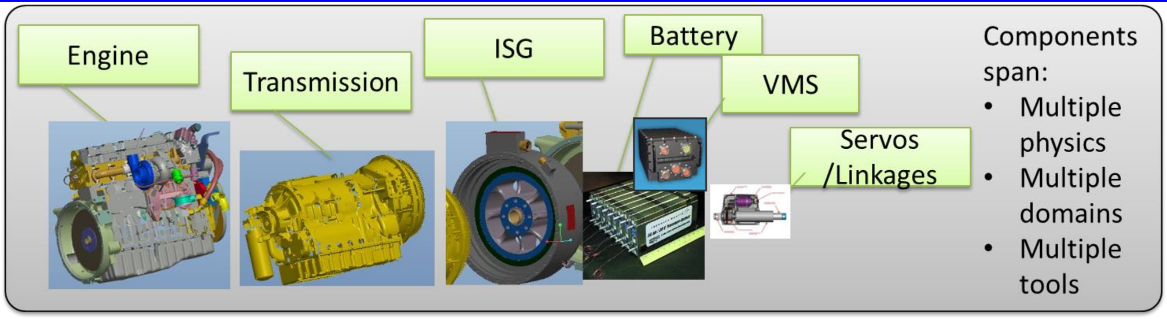
1.2. Heterogeneity of CPS Components

In addition to the diversity of the design and modeling activities, the modeling language needs to facilitate representation of a diverse range of cyber physical components.

The components that constitute a typical cyber physical system such as a military ground vehicle span a broad range from commodity physical components such as nuts and bolts, to large complex dynamical components like Engines, Transmissions, Sensors, Actuators, and Controllers. These components can broadly be categorized as:

1. Physical – components that consist purely of electro-hydro-mechanical elements with little or no programmability. Examples of such components include transmissions, differentials, gears, clutches, starter-generators, servos, among others. These can be further categorized as: <u>functional</u> – implementing a function in the design, or <u>interconnect</u> – that act as facilitator for physical energy flow or provide linkages such as nuts, bolts, pipes, and tubes.
2. Cyber – components are software components that require a computer processor to run, and implement some function such as the Vehicle Management Software, or controller algorithms implemented in software.
3. Cyber-Physical – components that cross-cut cyber and physical domains, such that they are physical and implement some function, however, contain deeply embedded

computing and communication functions that enable configuration and control of the designed function. Modern combustion engines are a good example of cyber-physical components, in that they include programmable controllers that will interface with rest of the vehicle management system over communication buses (such as CAN and TT/FlexRay), and allow optimizing torque delivery by controlling air/fuel mixture and valve timing for optimal combustion.



- Physical
  - Functional: implements some function in the design
  - Interconnect: acts as the facilitators for physical and information flows
- Cyber
  - Software that implements some function
  - Requires a physical platform to run/to communicate
- Cyber-Physical
  - Physical with deeply embedded computing and communication

Figure 2: Cyber-Physical Components

The examples depicted in Figure 2 above also illustrate the fact that CPS components span across different energy and physics domains. A combustion engine, for example, turns chemical energy into rotational mechanical energy; a battery delivers electrical energy from stored chemical energy, while an integrated starter generator (ISG) delivers mechanical rotational energy from electrical energy.

Moreover, the components depicted in Figure 2 require multiple models to describe and analyze their behavior. A combustion engine has a CAD model which represents the physical geometry including mass distribution, center-of-gravity, a dynamics model will describe the performance of the engine as a function of the driver and torque demand, a thermal model will describes heat generation, distribution, and dissipation as a function of the driver and torque demand.

Furthermore, these different models are often developed in different domain tools i.e. ProE/CREO or SolidWorks tools are used for CAD modeling, while Dymola and Simulink might be used for modeling dynamics. Often these models constitute an asset base of different engineering organizations, and have been developed with significant time and resource investment.

These motivating and constraining factors had a strong impact on the design of CyPhy. CyPhy had to be designed to represent components that are “Heterogeneous, Multi-Physics, and Multi-Model”, in such a way that it could leverage and integrate existing model assets created in domain tools.

1.3. Approach to Model Integration

The need to integrate multiple models, at different abstraction levels, and often created in different tools required a systematic approach to model integration. CyPhy was engineered as a Model Integration Language, which in essence is a “thin” wrapping language that wraps the domain models and exports only the key interface and parameters that are relevant for integration (see Figure 3). The wrapping maintains the link to the domain model – to allow integration in the domain tool. The integration language itself has a very small set of native modeling constructs by design. The native constructs of the language includes concepts such as hierarchical ported modules and interconnects, structured design spaces, and includes a variety of meta-model composition operators which enables systematic integration across different domain modeling languages.

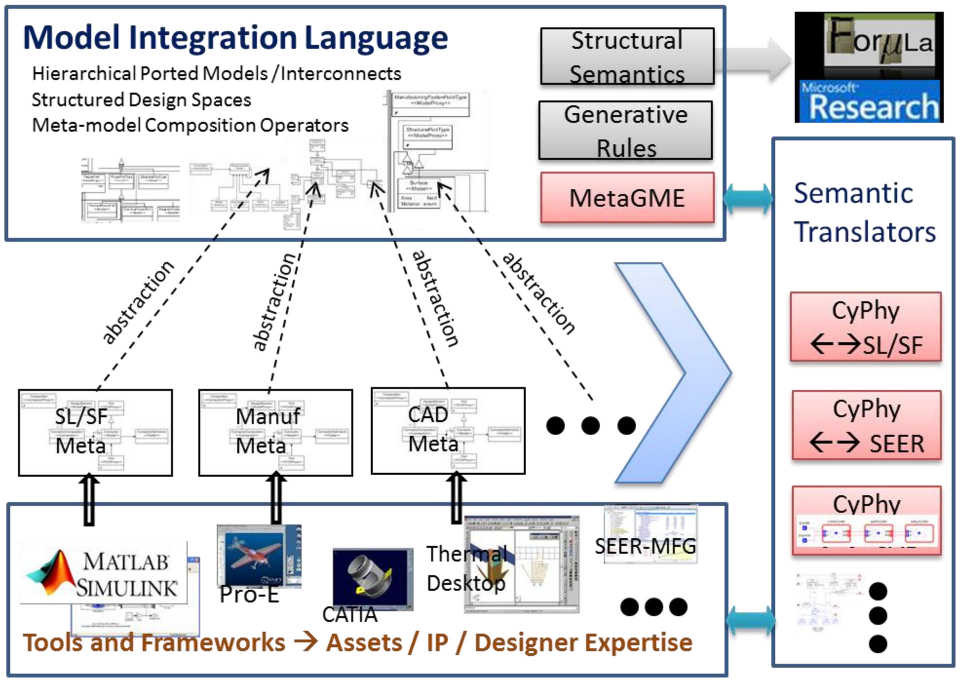


Figure 3: CyPhy as a Model Integration Language

The integration is done by abstracting the key properties and interfaces from the domain models that are relevant for integration across domains. These constitute the key variability’s, or design parameters that must be reasoned about in a multi-domain context. For example, when modeling system architecture the detailed and exact geometry may not be important, however, the key concepts of relevance are the join interfaces, surfaces, and constraints with which components must be physically attached to each other. A linkage of the abstractions and modeling concepts automatically enables the projection from architecture models back into the domain models.

The Tools and Frameworks depicted in the figure above are rich engineering infrastructures that have been developed with significant investment, and have accumulated a large volume of Design Assets, Intellectual Property, and Designer Expertise. The Model Integration Language approach enables reuse of these assets in the form of a Component Library, and when systems are built using the components, the Model Integration Language approach allows projecting the integrated models back into to the Domain Tools and Frameworks to analyze, visualize and refine the design using domain-specific tools.

A model integration language approach also allows the opportunistic linkage and additions of new design languages on demand, enabling an open language framework that allows for the adaptation of languages to accommodate evolving needs of design flows.

A major challenge in realizing a model integration approach, relates to the heterogeneous semantics of the integrated modeling languages. CyPhy addresses this challenge by formally specifying the semantics of the integrated domain languages, as well as formally specifying the composition semantics. The semantics are specified using FORMULA, a constraint logic programming environment, developed at Microsoft Research. FORMULA allows specification and reasoning over well formed-ness of domain composition.

2. CyPhy Language Taxonomy

With this pretext, now we examine the CyPhy language and its core concepts - the semantics and usage in CPS design. The functional taxonomy of CyPhy spans three key areas: 1) Components, 2) Designs and Design Spaces, and 3) Test Benches.

2.1. Components

Components in CyPhy are the basic units for composing system design. Components are self-contained models representing a physical or software part of the system. As an atomic component, they are not intended to be further subdivided, at least at the level of representation in CyPhy, but are used as a standalone part (see sample CyPhy component representation in Figure 4).

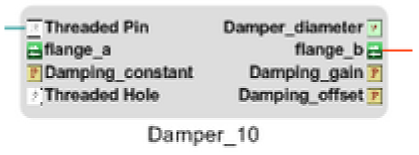


Figure 4: Example CyPhy Component

The component model represents several things about the actual component, including its physical representations and connections, its dynamic behavior, and numerical properties. The figure above shows an architectural view of a Damper component, symbolically represented as a rounded rectangle. We should note that CyPhy does not mandate a specific iconic visualization - rather the tool that implements the CyPhy language (GME in this case; in the future a browser-based version with better visualization is currently in development) as well as the authors of the component model are free to provide and render their own preferred iconic visualization. The figure above also renders the interfaces of the components as port objects that span different aspects: structural (threaded pin & hole), dynamics (flange_a/b), and parameters (damping constant, etc.). These aspects are:

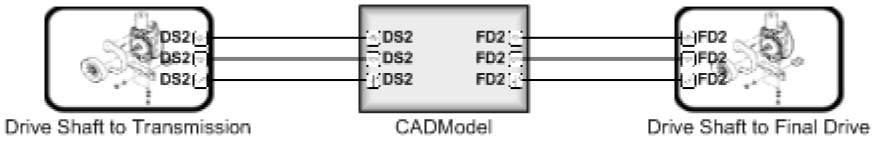


Figure 5: Example Structural Aspect of a CyPhy Component Model

- Physical implementation: The components have a 3D solid representation, and various physical properties, such as mass, center of gravity, etc. As the components will be interconnected into *assemblies*, subsystems and systems, the *interfaces* are carefully defined to permit *composition* of models. The physical properties of the model are shown in the *Structural Aspect* of the model (see Figure 5).
- Dynamics: The component has behaviors in one or more energy domains (e.g. Electrical, Thermal, Mechanical-Rotational, Mechanical-Translational, Hydraulic, etc.). Dynamics is expressed in the Modelica language, which uses a mix of Causal (directional input or output is assigned to each port) and Acausal (power flows either direction based on its context, as in most physical systems). These energy-transfer properties of the model are shown in the *Dynamics Aspect* of the model (see Figure 6).
- Cyber: The software is a critical part of the cyber-physical system design, with many components having a physical, dynamic, *and* software implementation. The Cyber aspect captures the software representation. The Cyber aspect is intended primarily for specifying controller logic for the system. Controllers can be specified in a combination of state diagrams and signal flow. Software is automatically generated from these models.

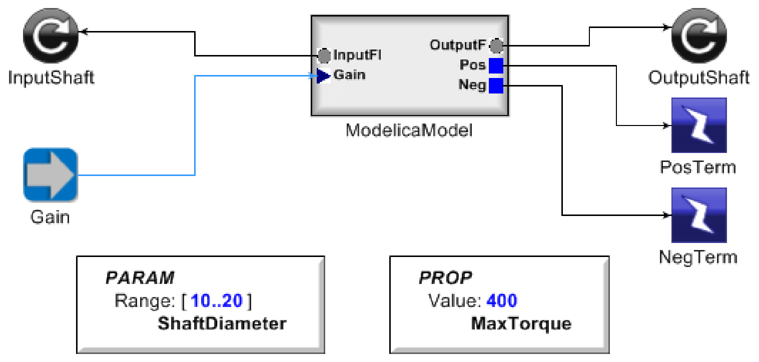


Figure 6: Example Dynamics Aspect of a CyPhy Component Model

2.1.1.Component Metadata

The component metadata captures attributes pertaining to the components physical or behavioral characteristics - for example: mass of an engine, or torque curve. Component metadata can be accessed by model composers and used to evaluate analytical constraints or compute system properties. In CyPhy, we support two variants of component metadata: (1) Property, and (2) Parameter described below.

- Property - A Property is a value provided by or associated with a component, such as material, mass, or an attribute associated with its maturity, e.g. TRL. Properties can be fixed, or could admit an uncertainty bound with a distribution. For parametrically variable components (see parameter description below), properties can be variable and algorithmically associated with parameters. Properties can also be scalar or vector, and have a unit associated with them. CyPhy employs a standard unit system based on NASA’s QUDT unit ontology.
- Parameter - A Parameter is a mechanism to capture component variability, allowing customization for a specific instantiation or use. For example, a variable-length drive-shaft component can be represented with a length parameter that can be configured for a specific use. The parameters are largely similar to properties from a specification standpoint and inherit from a common base class, are associated with a unit, and have a specified default value as well as a validity range. Design space exploration tools use component parameters for tuning or optimization of a design, or adaptation of a component for a specific use.

2.1.2.Component Interfaces

Component interfaces play a very important role in the definition of a component. They define the mechanism through which components connect and compose with other components. In CyPhy, a key design decision is to allow interfaces that aggregate multiple domain interfaces. These aggregated interfaces are referred to as Connectors. While it is preferred that component designers expose only aggregated Connector type of interfaces, the language still allows the flexibility of exposing domain interfaces directly. Component Interfaces also significantly influence the usability of components, as a functional interpretation of the interface can be communicate through an intuitive interface name that can guide the user in correctly connecting the component in a system. The following concepts are used to define component interfaces in CyPhy:

Connector_Interface - Connectors are an aggregation concept that allows aggregation and abstraction of multiple domain connections, providing a single connection to capture all of the constituent domain semantics in a single logical connection.

Power_Interface - Power Interfaces capture the lumped parameter dynamics interface, representing the transfer of power (includes mechanical translation/rotation, multibody movement, electrical, thermal, hydraulic, etc.).

Signal_Interface - Signal Interfaces are causal, non-physical interface that represents information flow within the system. This can be used to represent cyber signals.

Structural_Interface - The structural interface is an abstraction of the solid modeling/CAD connectivity. Structural interfaces contain references to CAD "datum", geometrical objects including planes, axes, points, and coordinate systems. These constituent parts are used to establish fully specified, integrated CAD assemblies.

Parameter_Interface - Parameter interfaces allow values to be passed between connectors.

Property_Interface - Property interfaces allow properties to be shared between components.

2.1.3.Domain Models

Domain models capture model artifacts specific to supported domain tools. The Domain models in CyPhy are a wrapping construct that refers to the domain model in its native representation language and tool. The Domain model exposes the core interface concepts that are used to map into the Component interfaces and metadata. For example, the component power interface maps directly into Modelica power ports. There are a set of domain models that are currently supported, however this is a point of expansion for the language. In transition projects, several new DomainModels have been added. The domain models are:

ModelicaModel - ModelicaModels capture the dynamics behavior in the Modelica language syntax.

BondGraphModel - Bond Graph Models use the bond graph formalism for representing component behavior. This domain model is mostly obsolete.

CyberDomainModel (SignalFlowModel) - Cyber domain models are a computational model that operate upon causal signals. CyPhy relies on a Simulink/Stateflow representation, linked through the ESMoL modeling language as its Cyber domain model representation.

CADModel - CAD models represent the solid model object for use in a CAD context. CADModels are typically Creo or STEP.

ManufacturingModel - The Manufacturing Model contains the information needed by iFAB for manufacturability analysis. These can span two distinct classes: 1) COTS parts - which only require cost, and procurement information associated with it; and 2) Make parts - these are parts that must be custom made by the iFAB (or other) foundry and require associated process information.

CyPhy allows multiple domain models for each domain thereby enabling a multi-fidelity representation of the component. In case there are multiple domain models, they are tagged with fidelity tags. The fidelity tags are kept freeform to allow users, component modelers, and tool developer's flexibility in specification and usage, since there is currently no universally accepted taxonomy of model fidelity.

In summary, components are multi-domain and multi-model, include interfaces for composition, can be parameterized, and have properties for informational and analytical evaluation.

2.2. Design Spaces

Using components and assemblies allows the designer to capture a single design architecture, with a single choice of components. This has several drawbacks:

- Requirements often change during the design process, sometimes necessitating a redesign.
- Component and subsystem behavior is discovered during the design process, and the best choice of architecture and components may not be apparent until late in the design process.
- The design is applicable to a single target use, and can require substantial rework for other applications.

Instead, CyPhy offers the concept of a ***design space***. The design space allows the models to contain multiple alternatives for components and assemblies. Any component or assembly can be substituted for another component or assembly with the same interface.

The editor offers a simple syntax for capturing design options. A design alternative container is created with an interface matching a component and the component is placed inside and wired to the external interfaces (there is a tool to automatically do this). Additional alternative components (or assemblies) are added to the alternative design container.

The semantics of this construct are to choose one of the internal components in place of the alternative container.

The design space is the combination of all options of all alternatives. Consequently, the design space can get very large (i.e. design space size is computed as # Alt1 * # Alt2 * # Alt3 *...). While this is a powerful mechanism to expand the range of designs under consideration, a mechanism is needed to limit the design space to a manageable size. For this purpose, design space constraints can be specified, and used by the Design Space Exploration Tool (DESERT) (see Figure 7).

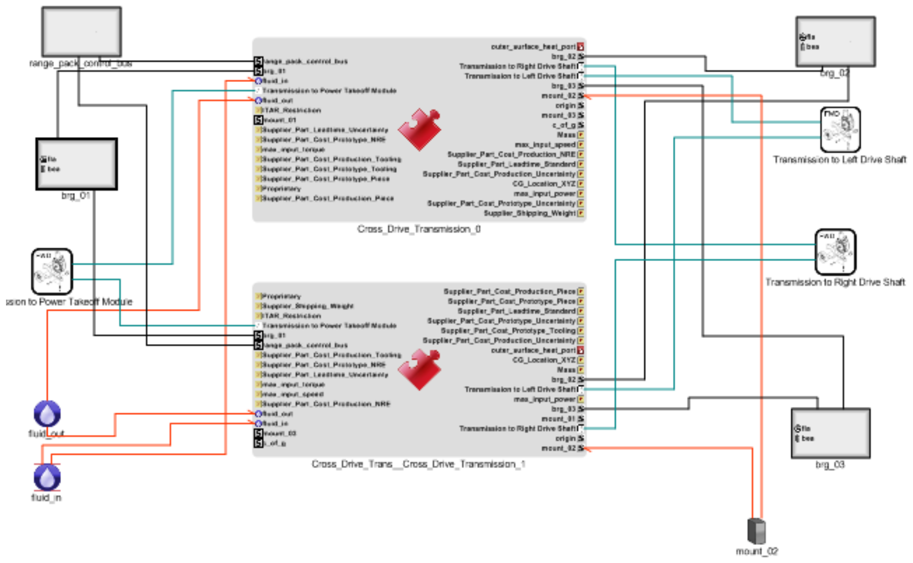

range_pack_control_bus
brg_01
brg_02
brg_03
Transmission to Left Drive Shaft
Transmission to Right Drive Shaft
ssion to Power Takeoff Module
Cross_Drive_Transmission_0
Cross_Drive_Trans__Cross_Drive_Transmission_1
fluid_out
fluid_in
mount_02

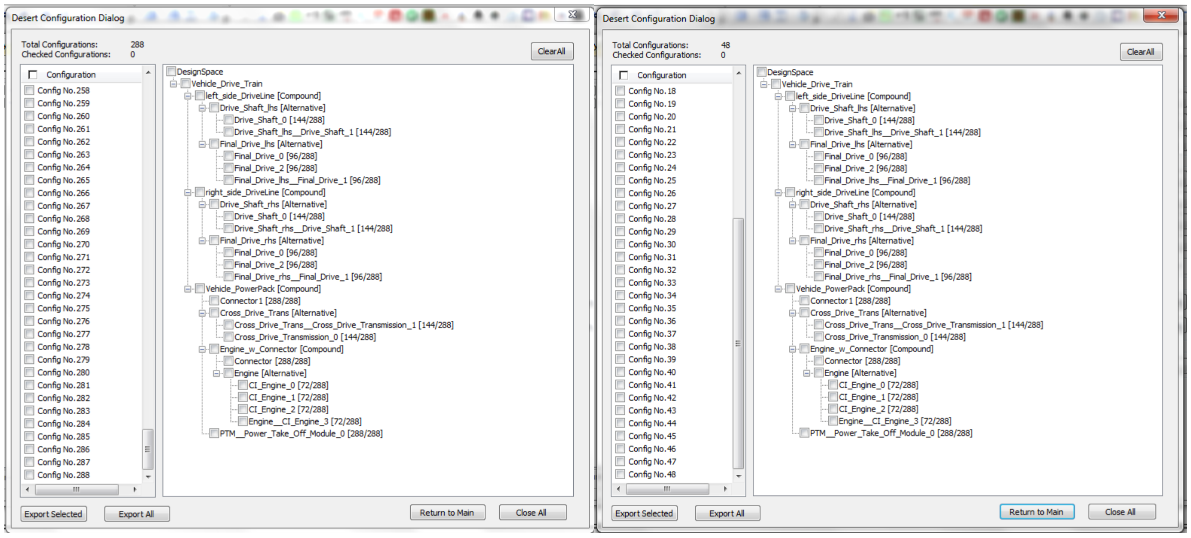
Figure 7: Example Design Space Alternative

Design space constraints are simple, static relationships that can be specified on the properties or identities of components or assemblies in the design alternative space (see Figure 8 for a visual example). Operations on the properties such as total weight and cost, thresholds on a component property (e.g. TRL > 3), or identity (e.g. all wheel types must match – do not mix and match).

Constraint
property
PropertyConstraint
visual
VisualConstraint

Figure 8: Example Constraints

The DESERT Tool uses scalable techniques to apply these constraints to very large design spaces to rapidly prune the design space to a manageable size. Figure 9 below shows the design space for the simple drivetrain. Prior to applying constraints, there are 288 configurations. After, there are 48. Typical design spaces can easily reach 10 billion configurations. After proper constraint application, these design spaces can be reduced to a number in the 1000's.



```
Total Configurations:      288
Checked Configurations:    0

Configuration
Config No.258
Config No.259
Config No.260
Config No.261
Config No.262
Config No.263
Config No.264
Config No.265
Config No.266
Config No.267
Config No.268
Config No.269
Config No.270
Config No.271
Config No.272
Config No.273
Config No.274
Config No.275
Config No.276
Config No.277
Config No.278
Config No.279
Config No.280
Config No.281
Config No.282
Config No.283
Config No.284
Config No.285
Config No.286
Config No.287
Config No.288

DesignSpace
  Vehicle_Drive_Train
    left_side_DriveLine [Compound]
      Drive_Shaft_lhs [Alternative]
        Drive_Shaft_0 [144/288]
        Drive_Shaft_lhs__Drive_Shaft_1 [144/288]
      Final_Drive_lhs [Alternative]
        Final_Drive_0 [96/288]
        Final_Drive_2 [96/288]
        Final_Drive_lhs__Final_Drive_1 [96/288]
    right_side_DriveLine [Compound]
      Drive_Shaft_rhs [Alternative]
        Drive_Shaft_0 [144/288]
        Drive_Shaft_rhs__Drive_Shaft_1 [144/288]
      Final_Drive_rhs [Alternative]
        Final_Drive_0 [96/288]
        Final_Drive_2 [96/288]
        Final_Drive_rhs__Final_Drive_1 [96/288]
    Vehicle_PowerPack [Compound]
      Connector1 [288/288]
      Cross_Drive_Trans [Alternative]
        Cross_Drive_Trans__Cross_Drive_Transmission_1 [144/288]
        Cross_Drive_Transmission_0 [144/288]
      Engine_w_Connector [Compound]
        Connector [288/288]
        Engine [Alternative]
          CI_Engine_0 [72/288]
          CI_Engine_1 [72/288]
          CI_Engine_2 [72/288]
          Engine__CI_Engine_3 [72/288]
      PTM__Power_Take_Off_Module_0 [288/288]
```

```
Total Configurations:      48
Checked Configurations:    0

Configuration
Config No.18
Config No.19
Config No.20
Config No.21
Config No.22
Config No.23
Config No.24
Config No.25
Config No.26
Config No.27
Config No.28
Config No.29
Config No.30
Config No.31
Config No.32
Config No.33
Config No.34
Config No.35
Config No.36
Config No.37
Config No.38
Config No.39
Config No.40
Config No.41
Config No.42
Config No.43
Config No.44
Config No.45
Config No.46
Config No.47
Config No.48

DesignSpace
  Vehicle_Drive_Train
    left_side_DriveLine [Compound]
      Drive_Shaft_lhs [Alternative]
        Drive_Shaft_0 [144/288]
        Drive_Shaft_lhs__Drive_Shaft_1 [144/288]
      Final_Drive_lhs [Alternative]
        Final_Drive_0 [96/288]
        Final_Drive_2 [96/288]
        Final_Drive_lhs__Final_Drive_1 [96/288]
    right_side_DriveLine [Compound]
      Drive_Shaft_rhs [Alternative]
        Drive_Shaft_0 [144/288]
        Drive_Shaft_rhs__Drive_Shaft_1 [144/288]
      Final_Drive_rhs [Alternative]
        Final_Drive_0 [96/288]
        Final_Drive_2 [96/288]
        Final_Drive_rhs__Final_Drive_1 [96/288]
    Vehicle_PowerPack [Compound]
      Connector1 [288/288]
      Cross_Drive_Trans [Alternative]
        Cross_Drive_Trans__Cross_Drive_Transmission_1 [144/288]
        Cross_Drive_Transmission_0 [144/288]
      Engine_w_Connector [Compound]
        Connector [288/288]
        Engine [Alternative]
          CI_Engine_0 [72/288]
          CI_Engine_1 [72/288]
          CI_Engine_2 [72/288]
          Engine__CI_Engine_3 [72/288]
      PTM__Power_Take_Off_Module_0 [288/288]
```

Figure 9: Design Space Exploration. Before and After Constraint Application

Design space creation and exploration is a process of expansion and contraction of the design space. It can be a powerful tool to build adaptable, flexible designs. The design space constructs in CyPhy can be discussed with the following taxonomy.

DesignSpace - A Design Space is a container for creating designs with structural/discrete design options, and captures system topology and hierarchy. Specifically, a Design Container can contain other Design Containers (for Hierarchy), Components, and Connections between components, and any properties/parameters necessary for customizing components.

Compound - A compound, also known as a *Component Assembly*, is a Design Container with a fixed architecture, containing no design flexibility. A Component Assembly is a container for components, other component assemblies, parameters, properties, and connections.

Alternatives - Alternatives are Design Containers that have a “choose one” semantic for all objects in the container. This is useful for representing the need for a component or subsystem within a design, where the decision is to be left to a later design stage.

Optionals - Optionals are Design Containers that capture a component that is not required within a system. Based on constraint application, either the component will be expressed, or all connections will be left unconnected.

Components - Components are instances or references to AVM components in the design container. Components are characterized with an ID, used for tracking component instances across model transformations. Components have associated text description that is typically not interpreted by any model composer.

Constraints - Constraints are logical operators that are used in the DESERT discrete design exploration.

Connections - Connections are linkages between component interfaces (also known as connectors). Connectors can also be present on ComponentAsemblies, DesignContainers, or other test bench components. Connections can be sub-classed into following types:

> Power_Connections - Power Connections are connections between domain power ports in domain models and interface connectors.
>
> Signal_Connections - Signal connections are connections between signal ports.
>
> Structural_Connections - Structural connections are connections between structural interfaces of components.
>
> Parameter_Connections - Parameter and Property Connections are connections between property connectors.

SimpleFormula - SimpleFormulas are mechanisms for a component assembly to use sets of Properties or Parameters to compute Parameters of another component. SimpleFormulas can use only basic arithmetic operators (such as +, -, /, or *).

CustomFormula - CustomFormulas are similar to SimpleFormulas, but allow complex equations to be captured using a relational and arithmetic expression language.

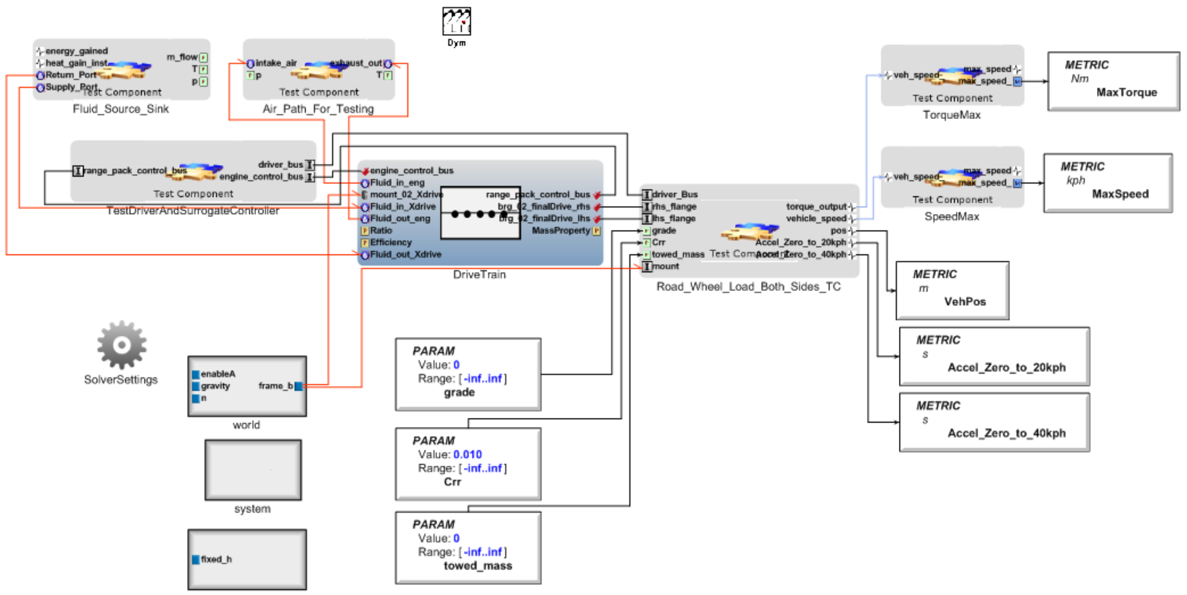
2.3. Design Evaluation (Test Benches)

While application of constraints can eliminate design alternatives based on simple, static properties, much of the system behavior (desirable and undesirable) emerges from the dynamic interaction between components. These interactions occur across and between any and all of the physical domains within the spectrum of the model coverage.

Evaluation of a model configuration can be done vs. requirements imposed on a system design. Requirements are expressed in terms of Metrics that can be computed on the system models. Examples of metrics include: *Speed, Maximum Towing Force, Acceleration time from 0 to 60 MPH*, etc. Requirements are tests on tests on these metrics, e.g. *"the vehicle must accelerate from 0 to 60 MPH in less than 8 seconds"*. Typically, the conditions and scenario will be specified in a requirement that translates to the definition of a CyPhy analysis needed, e.g. Level ground, Pavement, and the scenario of Driver Throttle at 100% (see the example Test Bench in Figure 10).

The system performance evaluation is specified in OpenMETA via a *Test Bench*. A Test Bench is an executable specification of a requirement analysis. The parts of a Test Bench are:

- Test Drivers, reproducing the stimulus to the system
- Wraparound environment, providing the interfaces at the periphery of the system (e.g. the ground interface with the tires, the external air, etc.)
- Metrics evaluation, taking measurements of the system properties and converting into a value of interest (metrics are also tied to requirements, which can convert the metric to a design "score")
- The system under test, either a point design or a design space (in the case of a design space, the Test Bench can be applied over the entire set of feasible designs)



Dym
Fluid_Source_Sink
Air_Path_For_Testing
TestDriverAndSurrogateController
DriveTrain
Road_Wheel_Load_Both_Sides_TC
TorqueMax
SpeedMax
METRIC Nm MaxTorque
METRIC kph MaxSpeed
METRIC m VehPos
METRIC s Accel_Zero_to_20kph
METRIC s Accel_Zero_to_40kph
SolverSettings
world
system
PARAM Value: 0 Range: [-inf..inf] grade
PARAM Value: 0.010 Range: [-inf..inf] Crr
PARAM Value: 0 Range: [-inf..inf] towed_mass

Figure 10: Example Test Bench

The Test Benches are tied to specific workflows. Currently, CyPhy implements Test Benches for:

- Dynamics, using a lumped parameter model executed in the Modelica language. Dynamics cover mechanical, electrical, hydraulic, and thermal domains.
- Structural, using 3D CAD assemblies to evaluate the physical compatibility of parts, locate potential interference, and compute physical properties such as Center of Gravity, Bounding Box, and assembled location of specific points on the system.
- Finite Element, using Finite element techniques to compute stress/strain, thermal propagation, computational fluid dynamics, etc.
- Mobility, using the NATO Reference Mobility Model to predict vehicle mobility based on aggregate system properties.
- Cyber, co-simulating dynamics with a time-based software/processor/network simulation.
- Manufacturability, creating the 3D CAD files, a set of properties of each manufactured join between parts, and an electronic Bill of Materials (BOM). From this design package, iFAB can predict a cost and schedule to manufacture the system.
- Complexity, evaluating the graph-energy complexity of the system based on its component complexity and structure of its connections. The complexity metric will correlate with system cost and schedule.

Test Benches also have a set of limits associated with part minimum/maximum parameters (such as maximum torques on a drive shaft) and design limits associated with an assembly or the use of a part in a system (such as minimum allowed battery charge). The limits are automatically evaluated with each execution of a Test Bench. If limits are exceeded, the analysis results are flagged and the Test Bench result can be ignored or otherwise modified or treated with less trust.

The following CyPhy concepts are used in definition of test benches:

Requirement Link - Requirement Links are a reference to a requirement object stored in a requirements database, such as DOORS. The link indicates that this Test Bench computes metrics that test the system against the Requirement. A Requirement Link can be associated with all the supported CyPhy test bench types.

SystemUnderTest - The System Under Test points to the system to be evaluated when executing the Test Bench. The system can be a single design point, or a design space. All CyPhy Test Bench classes are required to include a SystemUnderTest.

Connections - Connections within a Test Bench model allow interconnections between TestComponents, SystemUnderTest, Metrics, and Parameters.

Metrics - Metrics define outputs of a test bench. Metrics are connected to outputs of a SystemUnderTest, and cause the data to be recorded in the Metrics Output File.

Parameters - Parameters allow a SystemUnderTest parameter to be set within a Test Bench model.

TestComponents - Test Components are special purpose components, similar to standard components, but usable only within a Test Bench model.

2.3.1. Dynamics Test Benches

In addition to the common concepts referred above, Dynamics Test Benches also contain:

SolverSettings - Defines parameters for the Modelica solver for simulation duration, and solver algorithm.

2.3.2. FEA Test Benches

In addition to the common concepts referred above, FEA Test Benches also contain:

AnalysisPoints - AnalysisPoints define points on a system under test that can be exposed external to the Test Bench.

ForceLoad - ForceLoad allow the test bench to specify a force to be applied to the system under test.

PolygonalSurface - PolygonalSurface allow definition of a surface, to be used with a ForceLoad or DisplacementConstraint within a FEA solution.

DisplacementConstraints - DisplacementConstraints allow a surface to be held at a specified location.

2.3.3. Verification Test Bench

In addition to the common concepts referred above, Verification test benches also contain:

VerificationCondition - contains the description of the property to be satisfied by the Verification method.

2.3.4. Parametric Exploration Test Bench

Parametric exploration Test Benches are meta test benches in a sense that they perform parameterized execution of other test benches. The specification involves two primary concepts: 1) Parameter Drivers - that specify how the parameter must be explored, and 2) Test Bench References - the test bench that must be parametrically executed. There are three supported classes of parameter drivers:

Optimizer - An optimizer driver object specifies a goal-seeking parameter driver. The optimizer attributes include the optimizer type and optional code for a custom optimizer. The optimizer contains:

- Design Variable - a port specifying parameter to vary, with min/max attributes
- Objective - a port for the parameter to optimize
- Optimizer Constraint - specification to minimize or maximize the objective

PCC Driver - A PCC driver specifies a probabilistic sampling of parameters. PCC driver attributes include Senstivity Analysis and Uncertainty Propagation controls. The PCC driver contains:

- ParameterDistributionUniform - a parameter to sweep with a uniform distribution, with low and high limits
- ParameterDistributionNormal - a parameter to sweep with a Normal distribution, with mean and standard deviation
- ParameterDistributionLogNormal - a parameter to sweep with a LogNormal distribution, with log scale and shape parameter
- ParameterDistributionBeta - a parameter to sweep with a Beta distribution, with min, max, beta, and alpha

Parameter Study Driver - The parameter study driver specifies a DOE study to compute a surrogate model of the test bench under study. The parameter study has attributes including DOE type (e.g. FullFactorial) and a surrogate type to compute (e.g. Neural Net, Kriging). Parameter Studies contain:

- DesignVariable - a port for the parameters to vary, with min/max range
- Objective - a port for the parameters to explore.

2.4. Extending the CyPhy Language

The CyPhy language is designed for expansion. Expansion is achieved by adding to the language in the following ways:

2.4.1. Components

Components can be extended by adding attributes, domain models, ports, and resources.

- Domain Models: Domain Models capture new behavioral or physical artifacts to be associated with a component. These domain models will potentially contain domain-specific attributes, text or values that capture information about the component.
- Resources: Resources capture pointers to artifacts that are saved in different files.
- Connection Ports: Ports capture domain-specific information or connectivity information relevant to the component.

For example, several artifacts were added to the component for electronics design:

- Schematic Domain Model: this captures the relevant information about the schematic symbol and PCB footprint of a circuit. In the EDA case, EagleCAD was the initial tool supported. The component contained several text attributes for the symbol name, reference designator, chip class, etc.
- Resources: the resource points to an EagleCAD schematic file, containing the data for the electronic part.
- Ports: Schematic pins capture the parts where wires/PCB traces can be connected to the chip. The ports contain the pin number and placement location on the schematic symbol.

2.4.2. Connectors

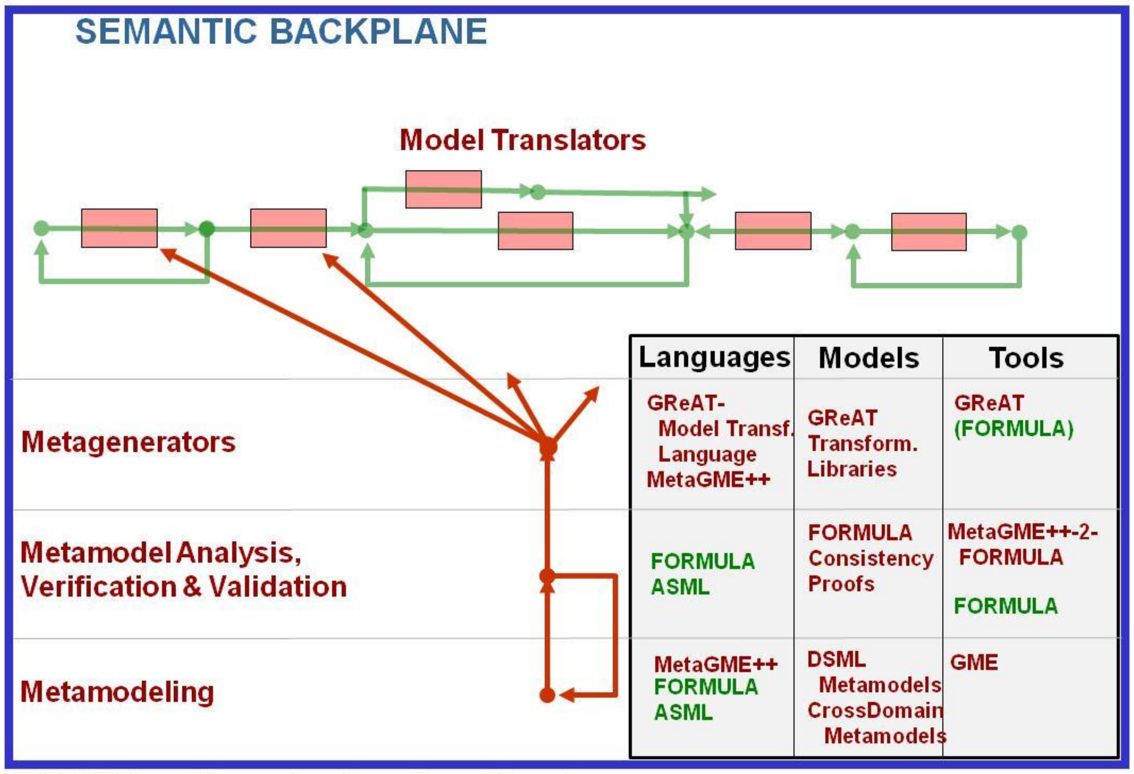
Connectors may require extension, adding domain-specific ports to the internals of the connector. These ports are connected to the domain ports in the component's Domain model. By including the domain ports into the connector, no modifications are required for the component assembly or design space models.

3. CyPhy Language Formalization

3.1. Semantic Backplane

The Semantic Backplane includes modeling languages, models, and tools for the semantic integration of Domain Specific Tool Chain (DSTC) configurations. The semantic integration is performed by the following (see Figure 11):

1. *Metamodeling* - defining structural and behavioral semantics of domain specific modeling languages
2. *Metamodel Analysis and Verification* - composing and relating DSTC-level domain specific modeling languages) and
3. *Metagenerators* - automatically generating model translators from formal specification of relationship between modeling languages



	Languages	Models	Tools
Metagenerators	GReAT-Model Transf. Language MetaGME++	GReAT Transform. Libraries	GReAT (FORMULA)
Metamodel Analysis, Verification & Validation	FORMULA ASML	FORMULA Consistency Proofs	MetaGME++-2-FORMULA FORMULA
Metamodeling	MetaGME++ FORMULA ASML	DSML Metamodels CrossDomain Metamodels	GME

FORMULA: http://research.microsoft.com/formula

Figure 11: Semantic Backplane

Tools and methods developed for the Semantic Backplane are not targeting the general engineering users: these are for a relatively small group of specialized experts responsible for the semantic integrity of the evolving domain specific tool chains.

An essential element of the Vanderbilt MIC tool suite is that most of the Semantic Backplane tools are "metaprogrammable" and used both in the Semantic Backplane and DSTC levels. In the following we summarize the delivered components. Metamodeling provides the formal specification of the semantics of the META modeling language suite.

3.1.1.MetaModeling Languages

1. *MetaGME++:* the mature MIC metamodeling language MetaGME (a variant of UML class diagrams and OCL) extended with generative constructs. MetaGME++ is used as metamodeling language for all MIC metaprogrammable tools. It has well established relationship with various standards, such as MOF, EMF, OWL and others.
2. *FORMULA:* constraint logic programming language developed by Microsoft Research. FORMULA is used as formal language for defining the structural semantics of *MetaGME++* and domain specific modeling languages defined using *MetaGME++*. (MSR and Vanderbilt ISIS collaborate in evolving FORMULA; e.g. current work expands the logic used in FORMULA with metric first order linear temporal logic).
3. *ASML:* a language variant for the Abstract State Machine (ASM) formal framework. We use ASMs as common semantic domain for specifying discrete behavioral semantics of modeling languages. ASML was selected because of its availability in the Visual Studio tool suite (we expect that in the future we migrate to FORMULA as the supporting theory evolves). ASML-based behavioral semantics are operational specifications (as opposed to denotational), therefore they are executable and suitable for generating reference traces.
4. *DE:* lumped parameter differential equations as a common denotational semantic domain for a wide range of continuous time dynamics. We use a syntactic form that can be easily transformed. DE's provide a bridge towards symbolic mathematics tools developed for order reduction. The provided semantics for continuous dynamics is independent from simulation algorithms.

The metamodeling languages listed above are part of the deliverables. We expect that the metamodeling languages will continue to evolve beyond this project as an overall consolidation in the practical use cases for semantics. We are also investigating other alternatives such as *BIP* (developed by Joseph Sifakis – 2008 Turing Award Laurate) for capturing interaction semantics among cyber components.

3.1.2.Metamodels

Metamodels are models of domain specific modeling languages described using metamodeling languages. Their goal is to capture the formal structural and behavioral semantics of modeling languages. The Semantic Backplane includes the *CyPhy Metamodel Library* that integrates semantic aspects of a given configuration of the META DSTC.

Being a model integration language, CyPhy includes a core set of language constructs for model and design space integration as well as an evolving suite of abstracted (sub)languages imported from various META tools. The abstracted sublanguages are the simplest possible well-formed subsets of the domain specific modeling languages of constituent META tools, yet still sufficient for capturing cross-domain interactions (structural and behavioral). Abstracting sublanguages for multi-model integration from bloated and complex domain languages is an important step toward making META DSTC-s practical.

At this point, the *CyPhy Metamodel Library* includes metamodels for the following sublanguages:

1. *ADML (Architecture Design Modeling Language)*: represents hierarchical component architectures and typed interfaces. Precise relationship is being defined between ADML and component modeling sublanguages of various standards or frequently used modeling languages, such as SysML (in progress), AADL (planned) and Simulink/Stateflow. This relationship is defined as model transformation in GReAT (the MIC tool suite graph model transformation specification language) and also, in some cases, in FORMULA.
2. *ADSML (Architecture Design Space Modeling Language):* extends the design modeling languages with constructs for design space modeling, allowing traditional design languages to capture design spaces instead of just point designs. The extensions come in the form of introducing design containers with model structure variability such as Alternatives, Optional, and variable cardinality containment, as well as Parameterization of design elements. Introduction of these design space extensions at all levels within the design hierarchy provides a powerful and compact mechanism of representing very large design spaces.

Beyond the core model and design space integration language elements, CyPhy has been complemented with the following abstracted sublanguages imported from integrated tools:

1. *Modelica Language:* Modelica is a multi-(energy/physics) domain formalism for representing lumped parameter dynamics of physical systems. A Modelica model can represent energy flow across systems in an energy domain neutral manner. Modelica models are also able to represent hybrid dynamics with the aid of if-else and switch constructs and support derivation of causality relation across systems.
2. *Simulink/Stateflow Interface Language:* The cyber aspects, specifically the controller design, are captured using Simulink/Stateflow models. The CyPhy metamodel integrates an abstracted Simulink/Stateflow metamodel, capturing the input, output, and parametric interface of Simulink models and defines associations with CyPhy components and component interfaces.
3. *CAD Constraint ML:* The CAD constraint modeling language represents geometrical constraints (axial alignment, surface placement), between CAD components (linked into CyPhy components) and allows derivation of CAD assemblies with a network of geometric constraints.
4. *Manufacturing (Cost) ML:* The manufacturing language represents manufacturing cost drivers for buy and make parts. These drivers include factors such as parts types, complexity, and counts, join types, complexity, and counts for part assemblies. The Manufacturing ML is integrated within CyPhyML allowing associating manufacturing cost parameters with CyPhy components.

The metamodels above are represented in MetaGME++ and translated for verification and validation to FORMULA.

3.1.3. Metamodeling Tools

1. *Generic Modeling Environment (GME)*: Vanderbilt's metaprogrammable modeling tool is the modeling environment for MetaGME++. Except the newly implemented support for the generative extension of MetaGME, the tool is mature and has been tested in major academic and industrial projects. GME is open source and distributed for research as well as commercial use.
2. *Unified Data Model (UDM)*: is a metaprogrammable API tool that provides API-s to programmatically manipulate domain-specific models built using GME (persisted in GME's native format or conformant XML). UDM is open source, has multiple programming language support (Java, C++, .net, Python), and is mature and tested in various academic and industrial projects.
3. *GReAT:* is a Graphical modeling environment (and associated toolset) for formally defining (modeling) Model Transformations as Graph Rewriting specification over Domain Meta Models. The model transformations defined with GReAT can be interpretively executed for rapid prototyping, or compiled into executable specifications for performance. The formal definition provides opportunities for verifying the transformation, and allows for systematic evolution of the model transformation as the domain metamodels evolve.

3.2. Formal Specification of CyPhy

In this section, we discuss the formalization of the Cyber-Physical Systems Modeling Language. CyPhyML is the composition of several sub-languages, such as a language for describing the composition of CPS components, a language for describing design-spaces with multiple choices, and others. In the following, we discuss only the composition sub-language and by CyPhyML we refer to this language. The GME meta-model of CyPhyML is shown in Figure 12.

Components are the main building blocks of CyPhyML. Components represent physical or computational elements with ports on their interfaces. Component assemblies are used for building composite structures by composing components and other component assemblies. Component assemblies also facilitate encapsulation and port hiding. There are two types of ports in CyPhyML: *acausal power ports* for representing physical interaction points, and *causal signal ports* for representing information flow between components. Both the physical and information flows are interpreted over the continuous time-domain. CyPhyML distinguishes power ports by types, such as electrical power ports, mechanical power ports, hydraulic power ports and thermal power ports.

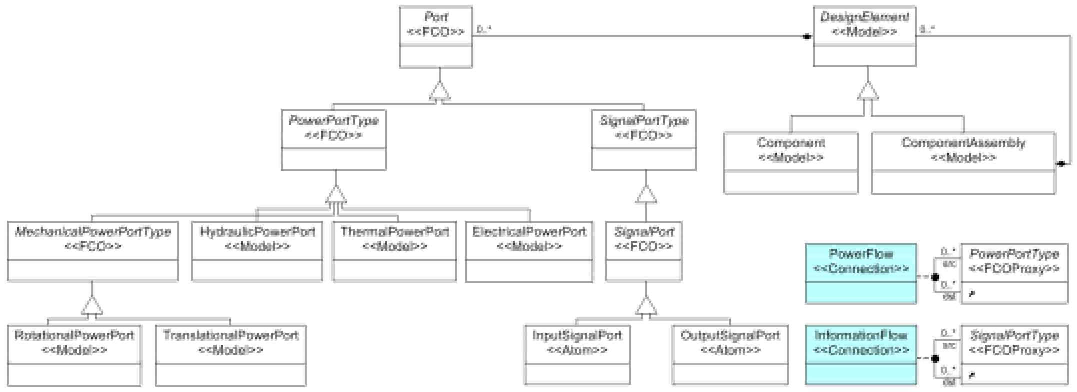


Figure 12: CyPhy MetaModel

Formally, a CyPhyML model M is a tuple $M \equiv \{C, A, P, contain, portOf, E_P, E_S\}$ with the following interpretation:

- C is a set of components,
- A is a set of component assemblies,
- $D = C \cup A$ is the set of design elements,
- P is the union of the following sets of ports: $P_{rotMech}$ is a set of rotational mechanical power ports, $P_{transMech}$ is a set of translational mechanical power ports, $P_{multibody}$ is a set of multi-body power ports, $P_{hydraulic}$ is a set of hydraulic power ports, $P_{thermal}$ is a set of thermal power ports, $P_{electrical}$ is a set of electrical power ports, P_{in} is a set of continuous-time input signal ports, P_{out} is a set of continuous-time output signal ports. Furthermore, P_P is the union of all the power ports and P_S is the union of all the signal ports,
- contain : $D \rightarrow A^*$ is a containment function, whose range is $A^* = A \cup \{root\}$, the set of design elements extended with a special root element root,
- portOf : $P \rightarrow D$ is a port containment function, which uniquely determines the container of any port,
- $E_P \subseteq P_P \times P_P$ is the set of power flow connections between power ports,
- $E_S \subseteq P_S \times P_S$ is the set of information flow connections between signal ports.

We can formalize this language using the following algebraic data types:

```
// Components, component assemblies and design elements
Component ::= new (name: String, ..., id:Integer).
ComponentAssembly ::= new (name: String, ..., id:Integer).
DesignElement ::= Component + ComponentAssembly.
// Components of a component assembly
ComponentAssemblyToCompositionContainment ::=
  (src:ComponentAssembly, dst:DesignElement).
// Power ports
TranslationalPowerPort ::= new (..., id:Integer).
RotationalPowerPort ::= new (..., id:Integer).
```

```
ThermalPowerPort ::= new (..., id:Integer).
HydraulicPowerPort ::= new (..., id:Integer).
ElectricalPowerPort ::= new (..., id:Integer).
// Signal ports
InputSignalPort ::= new (..., id:Integer).
OutputSignalPort ::= new (..., id:Integer).
// Ports of a design element
DesignElementToPortContainment ::= new (src:DesignElement, dst:Port).
// Union types for ports
Port ::= PowerPortType + SignalPortType.
MechanicalPowerPortType ::= TranslationalPowerPort
                          + RotationalPowerPort.
PowerPortType ::= MechanicalPowerPortType + ThermalPowerPort
                + HydraulicPowerPort
                + ElectricalPowerPort.
SignalPortType ::= InputSignalPort + OutputSignalPort.
// Connections of power and signal ports
PowerFlow ::=
  new (name:String,src:PowerPortType,dst:PowerPortType,...).
InformationFlow ::=
  new (name:String,src:SignalPortType,dst:SignalPortType,...).
```

3.2.1.Structural Semantics

Next, we formalize the structural semantics of the language. A CyPhyML model is well-formed if it does not contain any dangling ports, distant connections or invalid port connections, hence it conforms to the domain:

```
conforms
        no dangling(_),
        no distant(_),
        no invalidPowerFlow(_),
        no invalidInformationFlow(_).
```

For this, we need to define a set of auxiliary rules as discussed next. Dangling ports are ports that are not connected to any other ports:

```
dangling ::= (Port).
        dangling(X) :- X is PowerPortType,
        no { P | P is PowerFlow, P.src = X },
        no { P | P is PowerFlow, P.dst = X }.
dangling(X) :- X is SignalPortType,
        no { I | I is InformationFlow, I.src = X },
        no { I | I is InformationFlow, I.dst = X }.
```

A distant connection connects two ports belonging to different components, such that the components have different parents, and neither component is parent of the other one:

```
distant ::= (PowerFlow+InformationFlow).
distant(E) :-E is PowerFlow+InformationFlow,
        DesignElementToPortContainment(PX,E.src),
        DesignElementToPortContainment(PY,E.dst),
        PX != PY,
        ComponentAssemblyToCompositionContainment(PX,PPX),
        ComponentAssemblyToCompositionContainment(PY,PPY),
        PPX != PPY, PPX != PY, PX != PPY.
```

A power flow is valid if it connects power ports of same types:

```
validPowerFlow ::= (PowerFlow).
validPowerFlow(E) :- E is PowerFlow,
        X=E.src, X:TranslationalPowerPort,
        Y=E.dst, Y:TranslationalPowerPort.
validPowerFlow(E) :- E is PowerFlow,
        X=E.src, X:RotationalPowerPort,
        Y=E.dst, Y:RotationalPowerPort.
validPowerFlow(E) :- E is PowerFlow,
        X=E.src, X:ThermalPowerPort,
        Y=E.dst, Y:ThermalPowerPort.
validPowerFlow(E) :- E is PowerFlow,
        X=E.src, X:HydraulicPowerPort,
        Y=E.dst, Y:HydraulicPowerPort.
validPowerFlow(E) :- E is PowerFlow,
        X=E.src, X:ElectricalPowerPort,
        Y=E.dst, Y:ElectricalPowerPort.
```

If a power flow is not valid, it is invalid:

```
invalidPowerFlow ::= (PowerFlow).
invalidPowerFlow(E) :- E is PowerFlow, no validPowerFlow(E).
```

An information flow is invalid if a signal port receives signals from multiple sources, or an input port is the source of an output port:

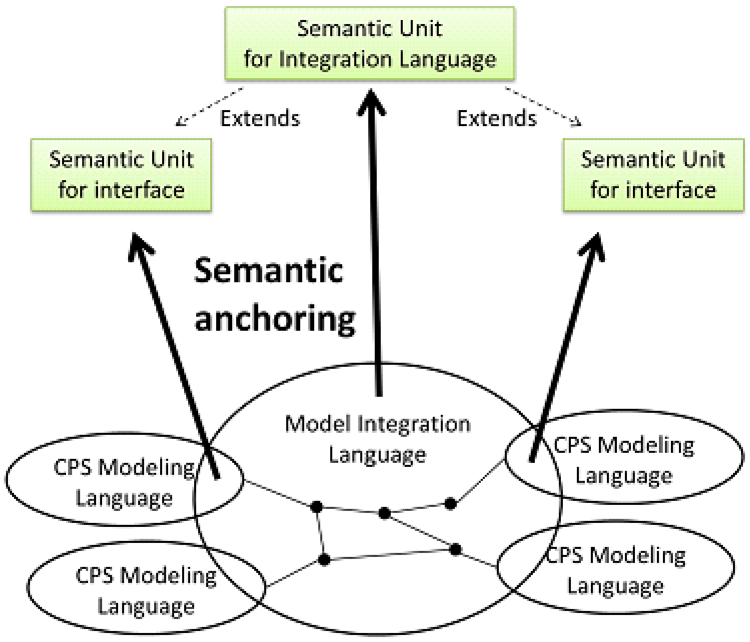
```
invalidInformationFlow ::= (InformationFlow).
invalidInformationFlow(X) :-X is InformationFlow,
          Y is InformationFlow,
          X.dst = Y.dst, X.src != Y.src.
invalidInformationFlow(E) :-E is InformationFlow,
          X = E.src, X:InputSignalPort,
          Y = E.dst, Y:OutputSignalPort.
```

Note that output ports can be connected to output ports.

3.2.2.Denotational Semantics

The denotational semantics of a language is described by a semantic domain and a mapping that maps the syntactic elements of the language to this semantic domain. In this section, we specify a semantic mapping from CyPhyML to the hybrid differential-difference equations semantic unit defined elsewhere.

We use the semantic anchoring framework for the denotational semantic specification of CyPhyML as shown in Figure 13.



Semantic Unit for Integration Language
Extends
Extends
Semantic Unit for interface
Semantic Unit for interface
Semantic anchoring
Model Integration Language
CPS Modeling Language
CPS Modeling Language
CPS Modeling Language
CPS Modeling Language

Figure 13: Semantic Anchoring Framework

Acausal CPS modeling languages distinguish acausal power ports and causal signal ports. In CyPhyML, each power port contributes two variables to the equations, and the denotational semantics of CyPhyML is defined as equations over these variables. Signal ports transmit signals with strict causality. Consequently, if we associate a signal variable with each signal port, the variable of a destination port is enforced to denote the same value as the variable of the corresponding source port. This relationship is one-way: the value of the variable at the destination port cannot affect the source variable along the connection in question.

The semantic function for power ports is mapping power ports to pairs of continuous time variables:

> PP : PowerPort ⟶ cvar, cvar.
> PP [[CyPhyPowerPort]] =
> (cvar("CyPhyML_effort",CyPhyPowerPort.id),
> cvar("CyPhyML_flow",CyPhyPowerPort.id)).

The semantic function of signal ports is mapping signal ports to a continuous time variables:

> SP : SignalPort ⟶ cvar+dvar.
> SP [[CyPhySignalPort]] =
> cvar("CyPhyML_signal",CyPhySignalPort.id).

The semantics of power port connections is defined through their transitive closure. Using fixed-point logic, we can easily express the transitive closure of connections as the least fixed point solution for ConnectedPower. Informally, ConnectedPower(x,y) expresses that power ports x and y are interconnected through one or more power port connections:

```
ConnectedPower ::= (src:CyPhyPowerPort, dst:CyPhyPowerPort).
ConnectedPower(x,y) :-PowerFlow(_,x,y,_,_), x:CyPhyPowerPort,
        y:CyPhyPowerPort;
PowerFlow(_,y,x,_,_), x:CyPhyPowerPort, y:CyPhyPowerPort;
ConnectedPower(x,z), PowerFlow(_,z,y,_,_), y:CyPhyPowerPort;
ConnectedPower(x,z), PowerFlow(_,y,z,_,_), y:CyPhyPowerPort.
```

More precisely, Px = {y | ConnectedP ower(x, y)} is the set of power ports reachable from power port x. The behavioral semantics of CyPhyML power port connections is defined by a pair of equations generalizing the Kirchoff-equations. Their form is the following:

$$\forall x \in CyPhyPowerPort. \left(\sum_{y \in \{y | ConnectedPower(x,y)\}} e_y = 0 \right)$$

$$\forall x, y \left(ConnectedPower(x,y) \rightarrow e_x = e_y \right)$$

We can formalize this the following way:

> P : ConnectedPower ⟶ eq+addend.

```
P [[ConnectedPower]] =
        eq(sum("CyPhyML_powerflow",flow1.id), 0)
        addend(sum("CyPhyML_powerflow",flow1.id), flow1)
        addend(sum("CyPhyML_powerflow",flow1.id), flow2)
        eq(effort1, effort2)
where
        x = ConnectedPower.src, y = ConnectedPower.dst, x != y,
        DesignElementToPortContainment(cx,x), cx:Component,
        DesignElementToPortContainment(cy,y), cy:Component,
        PP [[x]] = (effort1,flow1),
        PP [[y]] = (effort2,flow2).
```

3.2.3.Semantics of Signal Port Connections

A signal connection path (ConnectedSignal) is a directed path along signal connections. We can use fixed-point logic to find the transitive closure by solving for the least fixed point of ConnectedSignal. Informally, ConnectedSignal(x,y) expresses that there is a signal path (chain of connections) from signal port x to signal port y.

```
ConnectedSignal ::= (CyPhySignalPort,CyPhySignalPort).
ConnectedSignal(x,y) :-InformationFlow(_,x,y,_,_),
        x:CyPhySignalPort,
        y:CyPhySignalPort.
ConnectedSignal(x,y) :-ConnectedSignal(x,z),
        InformationFlow(_,z,y,_,_),
        y:CyPhySignalPort.
```

More precisely, Px = {y | ConnectedSignal(x, y)} is the set of signal ports reachable from signal port x. A signal connection (SignalConnection) is a connectedSignal such that its end-points are signal ports of components (therefore leaving out any signal ports that are ports of component assemblies).

```
SignalConnection ::= (src:CyPhySignalPort,dst:CyPhySignalPort).
SignalConnection(x,y) :-ConnectedSignal(x,y),
        DesignElementToPortContainment(cx,x), cx:Component,
        DesignElementToPortContainment(cy,y), cy:Component.
```

The behavioral semantics of CyPhy signal connections is defined as variable assignment. The value of the variables associated with the source and the destination of a signal connection are equal.

$$\forall x, y\left(SignalConnection(x, y) \rightarrow s_x = s_y\right)$$

S : SignalConnection → eq.
S [[SignalConnection]] =
eq(SP [[SignalConnection.dst]], SP [[SignalConnection.src]]).

3.3. Formalization of language integration

So far, we formally defined the semantics of the compositional elements of CyPhyML, but we have not specified how components are integrated into CyPhyML.

In this section, we develop the semantics of the integration of external languages: a bond graph language, the Modelica language, and the Embedded Systems Modeling Language (ESMoL) language. Note that we can easily add other languages to the list following the same steps as presented here. Note also that we include formalization of integration Bond Graphs, as it was integrated in an earlier revision of CyPhyML, but it is no longer supported.

Bond Graphs are multi-domain graphical representations for physical systems describing the structure of energy flows. Here, we consider an extended bond graph language that defines power ports through which a bond graph component interacts with its environment. Each power port is adjacent to exactly one bond; therefore, a power port represents a pair of power variables: the power variables of its unique bond. The bond graph language we consider here also contains output signal ports for measuring efforts and flows at bond graph junctions, and modulated bond graph elements that are controlled by input signals fed to the bond graph through input signal ports. Note that the effort and flow variables of the bond graph language are different from the effort and flow variables of CyPhyML: they denote different entities in different physical domains. The semantics of the languages formalize these differences precisely.

Modelica is an equation-based object-oriented language used for systems modeling and simulation. Modelica supports component-based development through its model and connector concepts. Models are components with internal behavior and a set of ports called connectors. Models are interconnected by connecting their connector interfaces. A connector is a set of variables (input, output, acausal flow or potential, etc.) and the connection of different connectors define relations over their variables. In the following, we discuss the integration of a restricted set of Modelica models in CyPhyML: we consider models that contain connectors that consist of either exactly one input/output variable, or a pair of potential and flow variables.

The Embedded Systems Modeling Language (ESMoL) is a language and tool-suite for modeling and implementing computational systems and hardware platforms. ESMoL consists of several sub-languages for defining platform and software architectures, describing the deployment of software on hardware and specifying the scheduling of execution. In the following, by ESMoL we refer to the state chart variant sub-language of ESMoL that is used for modeling software controllers. This sub-language is based on periodic time-triggered execution semantics, and its components expose periodic discrete-time signal ports on their interface.

3.3.1.Integration of structure

The role of CyPhyML in the integration process is to establish meaningful and valid connections between heterogeneous models. Component integration is an error prone task because of the slight differences between different languages. For instance, during the formalization we found the following discrepancies:

1. power ports have different meaning in different modeling languages,
2. even if the semantics is the same, there are differences in the naming conventions,
3. connecting the signals of ESMoL to the signals of CyPhyML needs a conversion between discrete-time and continuous-time signals.

In order to formalize the integration of external languages, we extend CyPhyML with the semantic interfaces of these languages. Hence, we need language elements for representing models of these heterogeneous languages, their port structures, and the port mapping between the ports and the corresponding CyPhyML ports.

We formalize the models and their containment in CyPhyML as follows:

```
BondGraphModel ::= new (URI:String, id:Integer).
ModelicaModel ::= new (URI:String, id:Integer).
ESMoLModel ::= new (URI:String, id:Integer, sampleTime:Real).
Model ::= BondGraphModel + ModelicaModel + ESMoLModel.
// A relation describing the containment of bond graph models in CyPhyML components
ComponentToBondGraphContainment ::= new (Component => BondGraphModel).
...
```

Note the fields of ESMoLModel: since ESMoL models are periodic discrete-time systems, we need real values describing their period and initial phase in the continuous time world. The interface ports and port mappings are the following:

```
// Bond graph power ports (and similarly for the other languages)
BGPowerPort ::= MechanicalDPort + MechanicalRPort + ...
...
// Port mappings for bond graph power ports (and similarly for other languages)
BGPowerPortMap ::= (src:BGPowerPort,dst:CyPhyPowerPort).
...
// All the power ports in CyPhyML and the integrated languages:
PowerPort ::= CyPhyPowerPort + BGPowerPort + ModelicaPowerPort.
// All the signal ports in CyPhyML and the integrated languages:
SignalPort ::= ElectricalSignalPort
          + BGSignalPort
          + ModelicaSignalPort
          + ESMoLSignalPort.
// List of all ports:
```

```
AllPort ::= PowerPort + SignalPort.
// Mapping from model ports to CyPhyML ports
PortMap ::= BGPowerPortMap
        + BGSignalPortMap
        + ModelicaPowerPortMap
        + ModelicaSignalPortMap
        + SignalFlowSignalPortMap.
```

An integrated model (that is, CyPhyML model integrated with other models) is well-formed if it conforms to the original CyPhyML domain, and its port mappings are valid:

```
conforms no invalidPortMapping.
```

A port mapping is invalid if it connects incompatible ports, or the interconnected ports are not part of the same CyPhyML component:

```
invalidPortMapping :- M is PortMap, no compatible(M).
invalidPortMapping :-M is BGPowerPortMap,
        BondGraphToPortContainment(BondGraph,M.src),
        DesignElementToPortContainment(CyPhyComponent,M.dst),
        no ComponentToBondGraphContainment(CyPhyComponent,BondGraph).
...
// Compatible denotes that port mapping M is valid (i.e., the corresponding ports are
compatible)
compatible ::= (PortMap).
compatible(M) :- M is BGPowerPortMap(X,Y), X:MechanicalRPort,
        Y:RotationalPowerPort.
...
```

3.3.2.Bond Graph integration

The semantics of bond graph power ports are explained by mapping to pairs of continuous-time variables:

```
BGPP : BGPowerPort → cvar, cvar.
BGPP [[BGPowerPort]] =
        (cvar("BondGraph_effort",BGPowerPort.id),
        cvar("BondGraph_flow",BGPowerPort.id)).
```

The semantics of bond graph signal ports is explained by mapping to continuoustime variables:

```
BGSP : BGSignalPort → cvar.
BGSP [[BGSignalPort]] = cvar("BondGraph_signal",port.id).
```

The behavioral semantics of bond graph power port mappings for the hydraulic and thermal domains is the equality of the associated port variables. We can formalize it with the following rules:

```
BGP : BGPowerPortMap → eq+diffEq.
BGP [[BGPowerPortMap]] =
        eq(cyphyEffort, bgEffort)
        eq(cyphyFlow, bgFlow)
where
        bgPort = BGPowerPortMap.src,
        cyphyPort = BGPowerPortMap.dst,
        bgPort : HydraulicPort + ThermalPort,
        PP [[cyphyPort]] = (cyphyEffort, cyphyFlow),
        BGPP [[bgPort]] = (bgEffort, bgFlow).
```

In mechanical translational domain, the effort of CyPhyML power ports denote absolute position and the flow denotes force, whereas for bond graphs the effort is force, and the flow is velocity. In mechanical rotational domain, the effort of CyPhyML power ports denote absolute rotation angle and the flow denotes torque, whereas for bond graphs the effort is torque and the flow is angular velocity. Their interconnection in CyPhyML is formalized by the following equations:

```
BGP [[BGPowerPortMap]] =
        diffEq(cyphyEffort, bgFlow)
        eq(bgEffort, cyphyFlow)
where,
        bgPort = BGPowerPortMap.src,
        cyphyPort = BGPowerPortMap.dst,
        bgPort : MechanicalDPort + MechanicalRPort,
        PP [[cyphyPort]] = (cyphyEffort, cyphyFlow),
        BGPP [[bgPort]] = (bgEffort, bgFlow).
```

For the electrical domain, bond graph electrical power ports denote a pair of physical terminals (electrical pins), while in the CyPhyML language they denote single electrical pins. In both cases, the flow (the current) through the pins is the same; however, there are differences in the interpretation of voltage. In the bond graph case, the effort variable belonging to the electrical power port denotes the difference of the voltages between the two electrical pins. In the CyPhyML case, the effort variable denotes absolute voltage (with respect to an arbitrary ground). The semantics of electrical power port mapping is the equality of the flows and efforts, which means that the negative terminal of the bond graph electrical power port is automatically grounded to the CyPhyML ground:

BGP [[BGPowerPortMap]] =
eq(bgFlow, cyphyFlow)
eq(bgEffort, cyphyEffort)
where
bgPort = BGPowerPortMap.src,
cyphyPort = BGPowerPortMap.dst,
bgPort : ElectricalPort,
PP [[cyphyPort]] = (cyphyEffort, cyphyFlow),
BGPP [[bgPort]] = (bgEffort, bgFlow).

Finally, the denotation of bond graph and CyPhyML signal port mappings is equality of the interconnected port variables:

BGS : BGSignalPortMap → eq.
BGS [[BGSignalPortMap]] =
eq(BGSP [[BGSignalPortMap.src]], SP [[BGSignalPortMap.dst]]).

3.3.3.Modelica integration

The semantics of Modelica power ports are explained by mapping to pairs of continuous-time variables:

MPP : ModelicaPowerPort → cvar,cvar.
MPP [[ModelicaPowerPort]] =
(cvar("Modelica_potential",ModelicaPowerPort.id),
cvar("Modelica_flow",ModelicaPowerPort.id)).

The semantics of Modelica signal ports is explained by mapping to continuous time variables:

MSP : ModelicaSignalPort → cvar.
MSP [[ModelicaSignalPort]] =
cvar("Modelica_signal",ModelicaSignalPort.id).

The semantics of Modelica and CyPhyML power port mappings is equality of the power variables. Formally,

MP : ModelicaPowerPortMap → eq.
MP [[ModelicaPowerPortMap]] =
eq(cyphyEffort, modelicaEffort)
eq(cyphyFlow, modelicaFlow)
where
modelicaPort = ModelicaPowerPortMap.src,
cyphyPort = ModelicaPowerPortMap.dst,

PP [[cyphyPort]] = (cyphyEffort, cyphyFlow),
MPP [[modelicaPort]] = (modelicaEffort, modelicaFlow).

The semantics of Modelica and CyPhyML signal port mappings is equality of the signal variables.

MS : ModelicaSignalPortMap ⟶ eq.
MS [[ModelicaSignalPortMap]] = eq(MSP [[ModelicaSignalPortMap.src]],
SP [[ModelicaSignalPortMap.dst]]).

3.3.4.SignalFlow integration

The semantics of ESMoL signal ports is explained by mapping to discrete-time variables, and the periodicity of the discrete variable is determined by the sample time of its container block.

ESP : ESMoLSignalPort ⟶ dvar, timing.
ESP [[ESMoLSignalPort]] = (Dvar, timing(Dvar, container.sampleTime, 0))
where,
Dvar = dvar("ESMoL_signal", ESMoLSignalPort.id),
BlockToSF_PortContainment(container,ESMoLSignalPort).

While signal ports in signal-flow have discrete-time semantics, signal ports in CyPhyML are continuous-time. Thus, signal-flow output signals are integrated into CyPhyML by means of the hold operator.

$$\forall x, y\left(SignalFlowSignalPortMap(x, y) \rightarrow e_x = hold(e_x)\right)$$

ES : SignalFlowSignalPortMap ⟶ hold+sample+timing.
ES [[SignalFlowSignalPortMap]] = hold(cyphySignal,signalflowSignal)
where,
signalflowPort = SignalFlowSignalPortMap.src,
cyphyPort = SignalFlowSignalPortMap.dst,
signalflowPort : OutSignal,
SP [[cyphyPort]] = cyphySignal,
ESP [[signalflowPort]] = (signalflowSignal,_).

For the opposite direction, we can use the sampling operator. The sample rate of the sampling function is defined by the signal-flow block containing the port.

$$\forall x, y\left(SignalFlowSignalPortMap(x, y) \rightarrow s_x = sample_r(s_y)\right)$$

```
ES [[SignalFlowSignalPortMap]] = sample(signalflowSignal,cyphySignal)
where
        signalflowPort = SignalFlowSignalPortMap.src,
        cyphyPort = SignalFlowSignalPortMap.dst,
        signalflowPort : InSignal,
        SP [[cyphyPort]] = cyphySignal,
        ESP [[signalflowPort]] = (signalflowSignal,_).
```

3.3.5.Power Port Units

Next, we define the physical units for each of the physical power ports. The Units enumeration contains all the supported physical units:

```
Units ::= {
"V", // Voltage
"A", // Ampere
"m", // meter
"N", // Newton
"N.m", // Newton−meter
"m/s", // meter/second
"rad", // radian
"rad/s",// radian/second
"kg/s", // kilogram/second
"Pa", // Pascal
"K", // Kelvin
"W", // Watt
"NA", // Not available
"J/kg", // Joule/kilogram
"Pa,J/kg",
"kg/s,W" // Modelica FlowPort
}.
```

PortUnit assigns two units to each power port: one to its effort variable, and one to its flow variable:

```
PortUnit ::= [port:PowerPort ⇒ effort:Units, flow:Units].
PortUnit(x,"V","A") :- x is ElectricalPowerPort;
        x is ElectricalPin;
        x is ElectricalPort.
PortUnit(x,"m","N") :- x is TranslationalPowerPort;
        x is TranslationalFlange.
PortUnit(x,"N","m/s") :- x is MechanicalDPort.
PortUnit(x,"rad","N.m") :- x is RotationalPowerPort;
        x is RotationalFlange.
PortUnit(x,"N.m","rad/s") :- x is MechanicalRPort.
PortUnit(x,"kg/s","Pa") :- x is HydraulicPowerPort;
        x is FluidPort;
        x is HydraulicPort.
PortUnit(x,"K","W") :- x is ThermalPowerPort;
        x is HeatPort;
        x is ThermalPort.
PortUnit(x,"NA","NA") :- x is MultibodyFramePowerPort.
PortUnit(x,"Pa,J/kg","kg/s,W") :- x is FlowPort.
```

It would be an interesting future work to use these units to verify the consistency of the language, in particular the consistency of the port mappings, where other modeling languages may use different units.