**Proceedings of the ASME 2012 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference
IDETC/CIE 2012
August 12-15, 2012, Chicago, IL, USA**

DETC2012-71378

TOWARDS AUTOMATED EVALUATION OF VEHICLE DYNAMICS IN SYSTEM-LEVEL DESIGNS

**Zsolt Lattmann Adam Nagel Jason Scott
Kevin Smyth Chris vanBuskirk Joseph Porter
Sandeep Neema Ted Bapty Janos Sztipanovits**

Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee 37212
Email: lattmann@isis.vanderbilt.edu

Johanna Ceisel Dimitri Mavris

Aerospace Systems Design Laboratory
Georgia Institute of Technology
Atlanta, Georgia 30332-0150
Email: jceisel@asdl.gatech.edu

ABSTRACT

We describe the use of the Cyber-Physical Modeling Language (CyPhyML) to support trade studies and integration activities in system-level vehicle designs. CyPhyML captures parameterized component behavior using acausal models (i.e. hybrid bond graphs and Modelica) to enable automatic composition and synthesis of simulation models for significant vehicle subsystems. Generated simulations allow us to compare performance between different design alternatives. System behavior and evaluation are specified independently from specifications for design-space alternatives. Test bench models in CyPhyML are given in terms of generic assemblies over the entire design space, so performance can be evaluated for any selected design instance once automated design space exploration is complete. Generated Simulink models are also integrated into a mobility model for interactive 3-D simulation.

INTRODUCTION

The DARPA AVM project aims to reduce the typical specification, design, analysis, construction, and manufacturing time and cost for a new military vehicle design by a factor of five. Two significant challenges that impact development cost and schedule are 1) system-level integration and 2) evaluation of design alternatives. Integration, or the resolution of details between design domains and subsystems in an engineered system is frequently cited as a principal source of schedule and cost overruns [1]. Tackling integration issues early in the design cycle goes a long way towards producing the expected gain. Heterogeneity of components and design concerns complicate integration, leading to innovative solutions to these problems [2]. Second, the number of potential vehicle designs which could be built from a set of existing components is very large, even when considering only macro-component choices such as engines, transmissions, hull, and chassis options. Evaluation of possible candidate designs against the system requirements enables the decision-making necessary to converge to a final design candidate. As for integration, the early pruning of candidate designs dramatically reduces the effort required to complete the end-to-end design and development processes. However, early design candidates usually lack the details necessary to effectively prune the design space, suggesting an iterative approach.

In this work we describe the Cyber-Physical Modeling Language (CyPhyML) and tools that use model-integrated computing techniques to support the design-time integration of numerous aspects of a system-level vehicle design, and the automated exploration of design alternatives. CyPhyML integrates models from existing engineering design and analysis tools to seamlessly support design work across multiple domains.

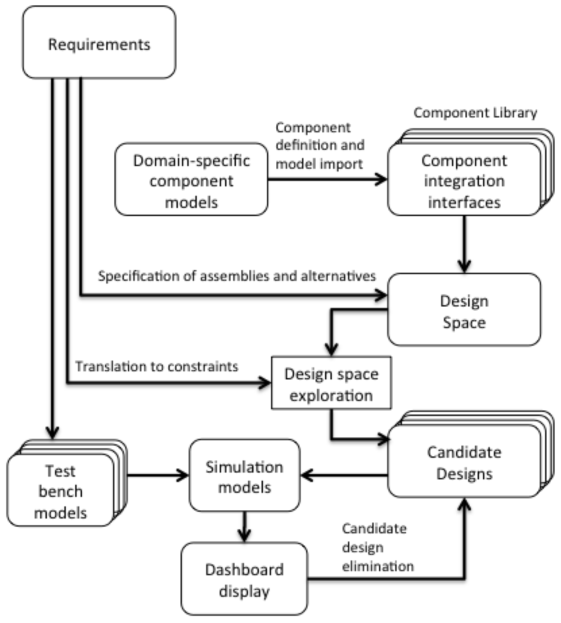

Requirements
Component Library
Component definition and model import
Domain-specific component models
Component integration interfaces
Specification of assemblies and alternatives
Design Space
Translation to constraints
Design space exploration
Candidate Designs
Test bench models
Simulation models
Candidate design elimination
Dashboard display

FIGURE 1. DESIGN SPACE REFINEMENT

CyPhyML supports integration and evaluation efforts using a few key concepts:

1. Model-Integrated Computing (MIC) is the core technology on which CyPhyML and its tools are built [3]. The use of MIC in system-level design moves many of the detailed design decisions normally resolved during lengthy integration (and often redesign) phases forward to the design and analysis stage of a system design, or often even prior to design activities. These details are resolved by an up-front concerted modeling effort which captures concepts and relations from design models in different design and analysis domains, representing each domain in a formal language description known as a metamodel. These metamodel descriptions are then integrated into a single language (in this case, CyPhyML) that precisely represents the relationships between the metamodels for the different domains. Model interpreters automatically import language and analysis artifacts from existing design tools into models in the integrated language, and can also be created to generate required design models. Both CyPhyML and its integrated domain-specific metamodels are defined using the Generic Modeling Environment (GME). GME also provides tools and APIs to develop the model interpreters required to realize the benefits of the MIC approach.
2. Components in CyPhyML correspond (in most cases) to real physical components. CyPhyML aggregates the different domain models associated with each component using carefully designed interface abstractions for each domain, so that larger design models can be assembled automatically using the multi-domain component definitions and descriptions of acceptable interconnections. We say that CyPhyML is *compositional* with respect to well-formed domain models. For the dynamics domain, model designers create behavioral models in a model repository, and specify interface details for those models using CyPhyML. The interface definitions expose strongly-typed signal and power ports that can be used to specify behavioral interactions between system components. Model hierarchy is used to define system assemblies in the design - the behavior of an assembly comes from the behaviors of its individual components, sub-assemblies, and the specified interactions between those components.
3. The design process supported by CyPhyML and its tools performs additional pruning of the design space by generating simulation and analysis models which can be evaluated against test cases to determine suitability and performance of candidate designs. Low-fidelity simulation models are first used to prune the design space (after reduction by structural details), then increasingly detailed (and time-expensive) simulation models are used to iterate and reduce the number of candidate designs which can meet design requirements. The level of fidelity of the models is not statically encoded in the CyPhyML language. Fidelity levels are domain-specific, and are determined by the component designers which must agree on project-specific standards to ensure consistency.
4. CyPhyML allows users to specify test bench models to simulate model behavior or perform detailed model analysis. A test bench provides a context in which the details of a vehicle or particular subsystem can be evaluated. Designers can assess correctness and compare different design configurations with respect to design metrics, which are the key performance parameters for the design.

Figure 1 illustrates the general process of defining a design space to identify and compare design candidates which satisfy the requirements, and then the refinement steps to narrow the set of candidate designs down to a manageable set of alternatives. First, domain-specific component models (such as CAD parts/assemblies, dynamics simulations, etc...) are captured in interface definitions in a component library. System requirements drive the specification of component assemblies and design alternatives. Some requirements are modeled as static constraints during automated design space exploration processes [4]. The static constraints eliminate many designs from a set of feasible design candidates, leaving a smaller set of designs for more detailed evaluation. Behavioral constraints such as performance requirements call for behavioral analysis of designs using simulation or other analysis tools. Since simulation for large models is expensive, structural reduction is a good strategy to restrict the design space before exploring the remaining (smaller) candidate designs using simulation (e.g., as shown in Fig. 1). Repeated simulation and analysis steps driven by test specifications further reduce the set of possible design candidates, while providing metric values to compare between remaining candidates. The summary metrics are displayed in a dashboard to visualize the trade-offs.

We will cover a specific subset of the CyPhyML design flow which relates to the specification and evaluation of the dynamic behavior represented by designs captured in CyPhyML.

BACKGROUND

Model-Integrated Computing

In model-based design, systems are described by models expressed in domain specific modeling languages (DSML). Formally, a DSML is a five-tuple of concrete syntax (C), abstract syntax (A), semantic domain (S) and semantic and syntactic mappings (M_S, and M_C):

$$L = < C, A, S, M_S, M_C > \tag{1}$$

The *concrete syntax C* defines the specific notation used to express models, which may be graphical, textual or mixed. The *abstract syntax A* defines the concepts, relationships, and integrity constraints available in the language. Thus, the abstract syntax determines all the (syntactically) correct "sentences" (in our case: models) that can be built. The integrity constraints, which define well-formedness rules for the models, are frequently called "structural semantics." The *semantic domain S* is usually defined by means of some mathematical formalism in terms of which the meaning of the models is explained. The mapping $M_C : A \rightarrow C$ assigns syntactic constructs to the elements of the abstract syntax. The semantic mapping $M_S : A \rightarrow S$ relates syntactic concepts to those of the semantic domain.

Any DSML requires the precise specification (or modeling) of all five components of the language definition. The languages which are used for defining components of DSMLs are called meta-languages and the concrete, formal specifications of DSMLs are called metamodels [3].

The models and languages we will describe were created using the ISIS Generic Modeling Environment tool (GME). GME allows language designers to create stereotyped UML class diagrams defining metamodels. The metamodels are instantiated into a graphical language, and metamodel class stereotypes and attributes determine how the elements are presented and used by modelers. The GME metamodeling syntax may not be entirely familiar to the reader, but the syntax is well-documented [5].

We build up custom domain-specific modeling languages (DSMLs) in GME by creating metamodels for individual domains, whether describing a component interface language or the language required by an existing analysis tool. These metamodels capture the model structure and relationships to formally represent concepts and relations in each particular domain. Model integration occurs when we use those specific metamodels as sublanguages to build up a larger language to support system integration activities, engineering design flows, and multi-domain analysis. Wrenn et al [4] give a more detailed description of the use of Model-Integrated Computing in the context of CyPhyML.

Acausal Modeling Paradigms

The key to compositional specification of physics is the use of an acausal modeling framework. Causal models such as signal data flows make it difficult to represent component connections that share forces. Acausal models expose power ports, which represent a simultaneous, bidirectional energy exchange. This means that a well-formed model in an acausal framework represents a well-formed set of dynamic equations. For further details see the tutorial article by Willems [6].

Acausal models often must interface with causal models, for example to represent the integration of a controller function into a physical system. A physical modeling formalism must define the interactions between the acausal and causal components in the system. Usually the physical part includes a means for exposing the physical variables in the model as either input or output signals (e.g. through sensors and actuators).

Hybrid Bond Graph Modeling Bond graphs are a physical, domain-independent graphical notation for describing the behavior of components and systems which can be modeled using differential algebraic equations. Bond graphs generically model the energy exchange between different types of energy storage and conversion components, analogous to a circuit diagram in the electrical domain. Junctions in a bond graph correspond to either series (common flow) or parallel (common effort) connections between the primitive components or other junctions. Power ports connect quantities in one component with another, and each includes two variables - an effort and a flow. Bond graphs easily and uniformly represent electrical, rotational, translational, thermal, and other types of power domains. Input signals either control parameters (e.g. modulated effort or flow) or directly influence the system behavior through functions on the physical variables.

The hybrid bond graph language (HBGL) within CyPhyML includes the ability to resolve causality and create a Simulink model from a bond graph model [7].

Modelica Modelica is a general modeling framework for physical systems [8]. Modelica does not strive for the uniformity of representation that bond graphs provide, but provides a library of standard components for each physical modeling domain. Interconnections between components are made using connectors which directly represent physical connections (e.g. attaching a wire to a pin of an electronic device), enabling the compositional definition of system behaviors. For a well-formed model, Modelica compilers translate all of the model subsystems into equations suitable for simulation or analysis. Unlike bond graphs, the Modelica language is an international standard that has well-supported commercial tools.

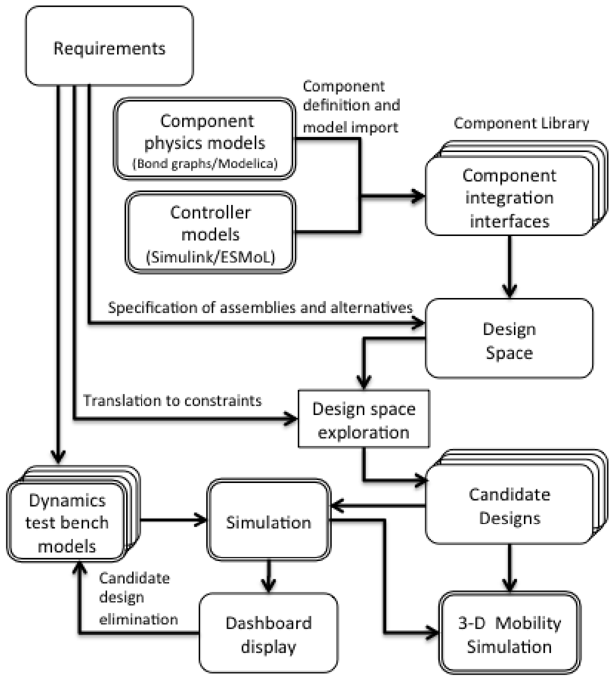

Requirements
Component physics models (Bond graphs/Modelica)
Component definition and model import
Component Library
Component integration interfaces
Controller models (Simulink/ESMoL)
Specification of assemblies and alternatives
Design Space
Translation to constraints
Design space exploration
Dynamics test bench models
Simulation
Candidate Designs
Candidate design elimination
Dashboard display
3-D Mobility Simulation

FIGURE 2. DESIGN FLOW FOR EVALUATING DYNAMICS MODELS

Embedded Systems Modeling Language (ESMoL)

ESMoL is a graphical modeling language created using GME which allows designers to use Simulink diagrams as synchronous software function specifications. In ESMoL the execution of each block is equivalent to a single bounded-time blocking C language call. These specifications are used to create model entities representing ESMoL software component types. ESMoL components have message structures as interfaces, and the type specification includes a map between Simulink signal ports and the fields of the input and output message structures. The messages represent C structures, and the mapping graphically captures the marshaling of Simulink data to those structures.

Once software component types and interfaces have been specified, ESMoL designers instantiate those components into a distributed deployment model. ESMoL allows the separate specification of the logical data flow, the mapping of component instances to hardware, timing information for tasks executing those components, and timing for messages sent over a time-triggered communication bus. Code generated from the models conforms to an API for time-triggered execution. A portable virtual machine implementation of the API allows execution in simulation (via the TrueTime toolkit for Simulink [9] [10]), hardware-in-the-loop, and fully deployed configurations. A more detailed description of ESMoL can be found in [11].

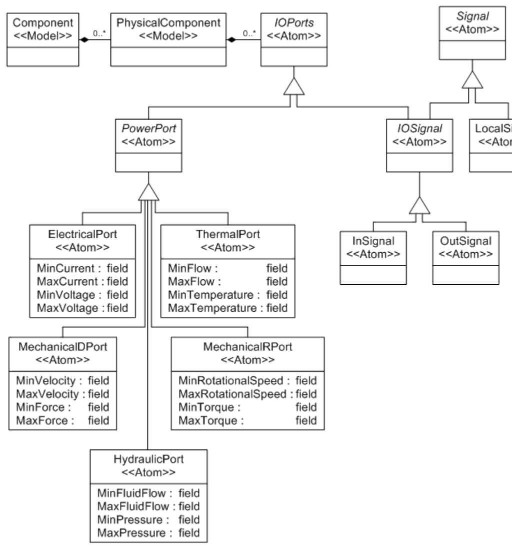


FIGURE 3. PHYSICAL PORT TYPES

BEHAVIOR MODELING

Supported Design Flow

Some design properties and metrics can only be determined through behavioral evaluation, whether it be through simulation or testing a physical prototype. We support automated assembly of dynamic simulations from test bench models defined in CyPhyML together with detailed domain-specific models referenced by the CyPhyML component descriptions. In cases where designs must be refined as part of a design process we support alternative representations of a component so that simulations can be generated for versions of a component having different levels of detail. For example, an ideal continuous-time controller defined in Simulink can be compared in simulation with a much more detailed implementation model of the same controller once it has been discretized, assigned to computing hardware, and transformed into source code. We can also assess controller performance in context using a 3-D mobility simulation for vehicle designs. In the sequel we present a variety of models drawn from different subsystems in a single notional vehicle design. A unified model would have been preferable for illustration, but we are limited by the availability of detailed models for different parts of the design.

Figure 2 depicts the specialization of the generic CyPhyML design evaluation flow to the subset of functionality supporting simulation of vehicle and subsystem dynamics. The component library is built up from acausal physics models and from software and controller models. Candidate designs are evaluated by generating a simulation model for the dynamics. Performance metrics calculated from the simulations can be used to winnow the set of candidate vehicle designs. Individual tests are based on system requirements and captured in CyPhyML test bench models. Iterating the whole evaluation process with successively more detailed dynamics models can significantly reduce the overall simulation and analysis effort by performing the most costly (in compute time) analyses only on small sets of design candidates.

Behavioral Component Interfaces

In the CyPhyML metamodel we have captured key concepts from the physical and computational design domains and combined them using a single integration language. Figure 3 shows the details of the static set of physical port types available in CyPhyML. CyPhyML designs consist of interconnected components and assemblies, where the connections are made between component ports representing physical types, and valid connections must have the same type at both endpoints. For example, the port class *MechanicalRPort* provides a mechanical rotational power interface for its parent component. It must be connected to another mechanical rotational port, which means that rotational power (torque and angular velocity) are shared between the two components. For analysis and simulation, the equations representing the dynamic behavior of two components connected via a power interface are merged to form a single set of equations, and the torque and angular velocity variables associated with the respective power port interfaces are merged in the resulting equation set.

Controller Models

For controllers that can be realized as software, causal modeling is the only option as digital and software systems are inherently causal. In this version of the CyPhyML language, controller components can be specified either as a wrapper around a Simulink component with its input and output signal ports exposed, or as an assembly that specifies the deployment constraints for a controller software component (i.e. the cyber sublanguage). A CyPhyML cyber assembly has components representing computing hardware (sensors, actuators, and processors), and components representing the software realization of a controller function as shown in the example in Fig. 4. Within the controller assembly shown, the *TransHardware* object is also an assembly which includes a single sensor, processor, and actuator. The sensor input signal port and its corresponding logical data connection back to the software function block, the processor scheduling port, the bus connection port, and actuator output signal port (with its corresponding logical data connection input) are exposed on the interface of the hardware assembly. The logical connections are made using connectors associated with *InformationFlow* class (Fig. 5). This type of connector defines constraints between software interface elements and hardware interface elements.

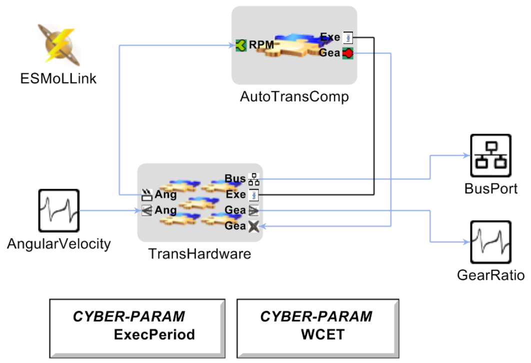


FIGURE 4. CYBER TRANSMISSION EXAMPLE

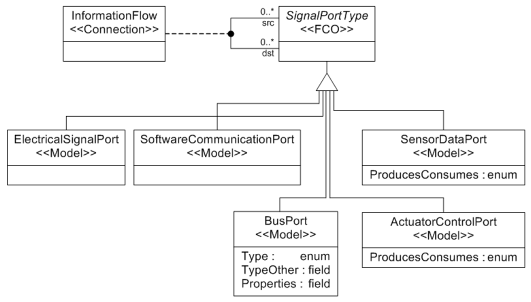
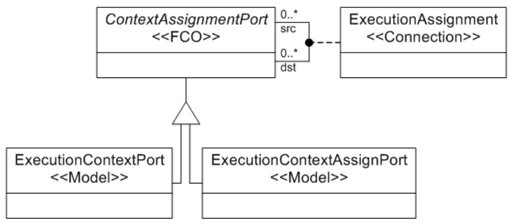


FIGURE 5. CYBER DATA FLOW CONNECTIONS

The *AutoTransComp* object represents the software component which realizes the transmission controller logic. It exposes two data ports (one from the sensor and the other to the actuator), and a scheduling port to indicate which execution context runs this component. In this case we have only a single processor with a single component, so the allocation is straightforward. Semantically, a connection between a processor and a software component represents an allocation constraint – the task which runs the given component must execute in the context defined on that particular processor. Figure 6 describes the language elements defining the connection. Unconnected context ports indicate that the software tasks are available to be automatically allocated, though it must be remembered that data requirements (as given by the *InformationFlow* connections) may force the creation of bus or network data messages if a particular component is not allocated near its data source. The final element of the cyber interface is the addition of *Parameter* objects for the configuration of the controller assembly. In CyPhyML *Parameter* objects appear as ports on the controller assembly model when viewed from a higher level in the design hierarchy. In a CyPhyML test bench model *Parameter* objects are created and attached to those ports, allowing the tester to override default parameter values for a particular test. In the example *ExecPeriod* configures the sampling rate of the computational controller, and *WCET* specifies the worst-case execution time.



ContextAssignmentPort <<FCO>>
0..* src
0..* dst
ExecutionAssignment <<Connection>>
ExecutionContextPort <<Model>>
ExecutionContextAssignPort <<Model>>

FIGURE 6. CYBER ALLOCATION CONSTRAINTS

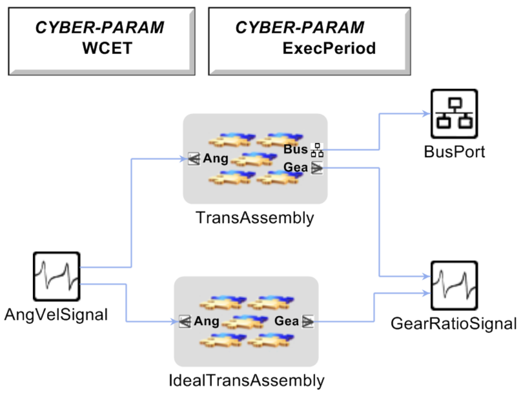


FIGURE 7. CONTROLLER MODEL ALTERNATIVES

The details of the controller implementation are specified outside of the CyPhyML model. The ESMoL language and tools are first used to provide the detailed software component interfaces in the CyPhyML component library. For Fig. 4 the interface message details of the *AutoTransComp* software control block are defined externally in an ESMoL model in the software component library. The key details in the ESMoL model include the full functional specification of the controller (imported from Simulink) and the structure of the data types on the input and output interfaces. The *ESMoLLink* block in the figure indicates that the entire assembly will also be realized in ESMoL. During simulation synthesis the CyPhyML assembly is translated automatically into an ESMoL deployment model, an execution schedule is calculated, and the resulting models and data are used to generate a TrueTime Simulink model of the controller software [10].

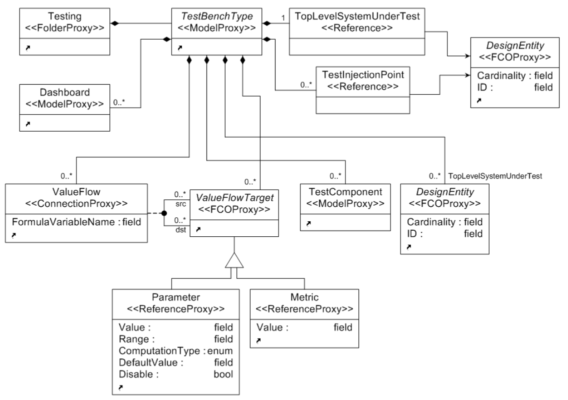


FIGURE 8. TEST BENCH CONCEPTS AND RELATIONS

Component Assemblies, Alternatives, and the Composition of Dynamics

Anywhere in a CyPhyML model that a component can appear we can also substitute a design container. A design container contains other components and specifies the relationship between the included components and the design space. The *ContainerType* attribute class *DesignContainer* indicates its interpretation in the design space – a *Compound* container simply aggregates components and design containers, an *Optional* container indicates that the contents may be excluded, and an *Alternative* container indicates that any one (and only one) of the contained components may be used. The alternative containers are key to the strength of the design space exploration process. For design space pruning we use parameterized component models which share common physical and parametric interfaces. The common interfaces allow the generation tools to operate orthogonally to the design space exploration process, so that any feasible model from the design space can be generated to either CAD or to simulation accordingly. The DESERT design space exploration tool is integrated into the CyPhyML tool suite to support this process [12]. The uniformity of interfaces among component alternatives allows the automated tools to create simulation models in a generic way for any valid combination of components defining a vehicle design.

Alternatives support the design space exploration process, but the alternative structures can also be used to support design processes which require multiple representations over the lifecycle of the design. Specifically, controller design models normally progress through multiple phases - an ideal feedback controller may be designed in the continuous-time domain, then discretized and quantized. The discrete-time form is converted to software with typed data interfaces for deployment on computer hardware. Deployed controller software differs in behavior from its ideal model representation due to discretization, quantization, and timing delays introduced by data movement. We will be able to compare simulation traces of the deployed software with traces from the original ideal controller model to assess the effects of the computation platform on controller stability and performance.

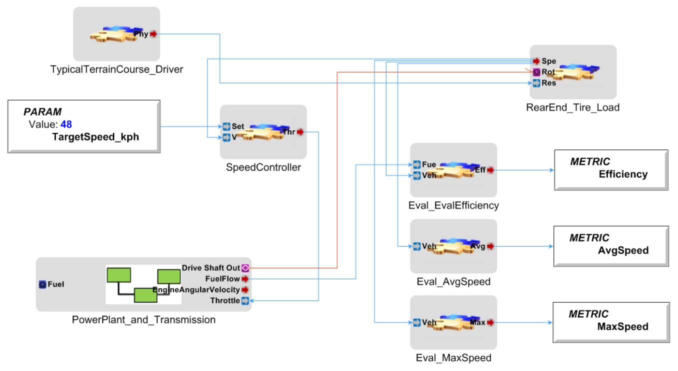


FIGURE 9. TEST BENCH EXAMPLE

Fig. 7 shows an alternative container for the transmission controller block. During design evaluation, the ideal controller (bottom *IdealTransAssembly* alternative) can be synthesized into the design. Once detailed design for the controller software and computing hardware has been made, the implementation alternative (top *TransAssembly* block) can be used during the evaluation.

TEST BENCH MODELS

In CyPhyML each test bench is defined in the context of a complete system or subsystem design. The design container that specifies the test bench context is called the *top level system under test*, often abbreviated SUT (an instance of the *TopLevelSystemUnderTest* reference class in Fig. 8). An object from the design space is included by reference as the single SUT for a particular test bench. The top level may also be the entire vehicle model or a single component. In the test bench example model shown in Fig. 9, the *PowerPlant_and_Transmission* subsystem is the SUT. Subcomponents of the SUT can be called out by reference as *test injection points*. The example model has no separate test injection points, as the interface of the SUT is sufficient for this test. Test injection points allow designers to direct inputs to internal elements of the SUT. The reference class *TestInjectionPoint* in the Fig. 8 specifies its membership in the test bench and its reference association with the *DesignEntity* class, which includes components and design containers. *TestComponent* objects can refer to models external to CyPhyML that specify inputs (Drivers), outputs (Evaluator), or utility functions for the test. Bond graph test bench models generate to Simulink simulations, so the test components can refer to an externally defined Simulink model. In CyPhyML the test component object exposes the signal ports necessary to specify the connection of the test component with the signal ports of the test injection points. A test component itself can also have physical dynamics in order to create a test harness for the SUT. These components can represent physical loads or behavior introduced by the environment. In the example, the object *RearEnd_TireLoad* has its dynamics specified as a bond graph, and it is composed with the SUT block by means of a mechanical power port.

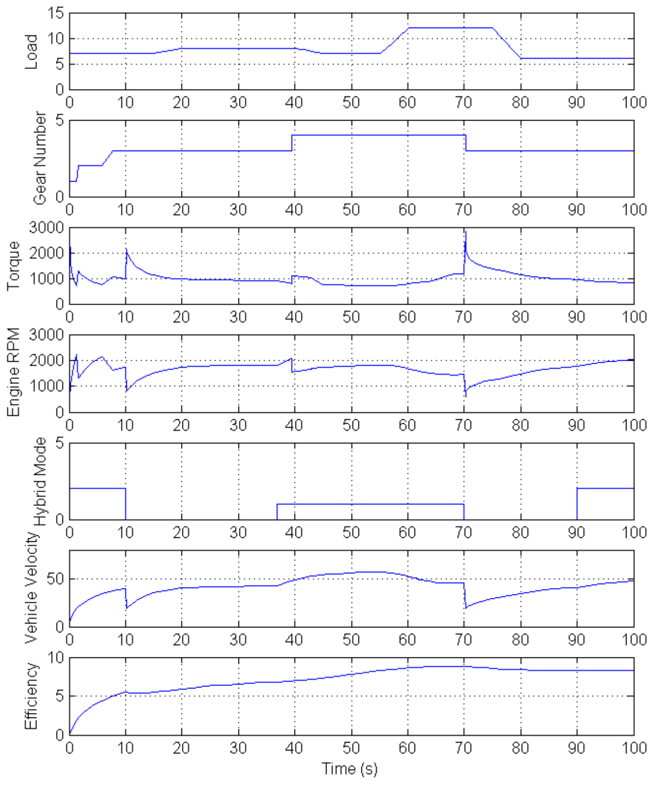


FIGURE 10. TEST BENCH RESULTS

Metrics and parameters define the configuration space for a test bench. Parameters may be given by the user or propagated automatically by the CyPhyML formula evaluator. For example, the formula evaluator calculates the total weight of a given design configuration from its constituent parts, and then propagates that value to a parameter block specified in the test bench that uses the vehicle model as the SUT. Metrics are the key performance parameters of the system. *Metric* objects specify the output of a test bench model, and define dependencies on state variables of the SUT. The final values of the metrics represent the 'quality' of the tested subsystem. For instance, the example shown in Fig. 9 includes metrics objects for average speed, maximum speed on the test track, and efficiency. In the model metrics are derived from the collected output signal values.

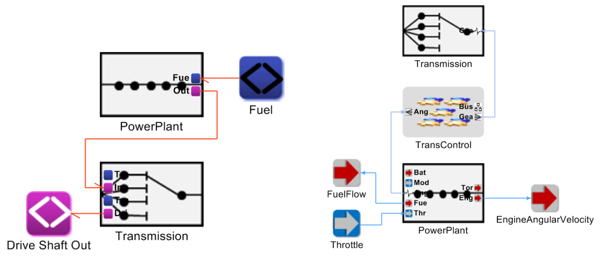


FIGURE 11. DESIGN SPACE INTEGRATION OF TRANSMISSION COMPONENTS

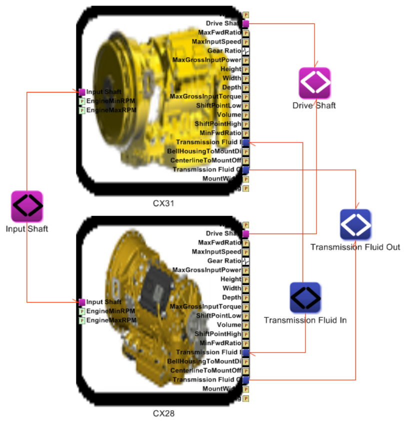


FIGURE 12. DESIGN ALTERNATIVES FOR TRANSMISSION

The example in Fig. 9 examines the performance of the vehicle power plant (hybrid drive) and transmission subject to a rotational load. A test driver (*TypicalTerrainCourse_Driver*) creates an input signal which varies the test load according to a specified terrain profile. Evaluation blocks collect data required to compute the metric values. Three of them are shown here, while others were removed for clarity. A speed controller test component was included to simulate the behavior of a cruise control. Figure 10 displays various quantities from the test run.

Figure 11 shows the details from two aspects within the *PowerPlant_and_Transmission* block. On the left is the *Dynamics* aspect, which displays the physical ports (rotational and fluid). The *PowerPlant* and *Transmission* share mechanical rotational power, consistent with their actual configuration in a vehicle. In the figure on the right the *Cyber* aspect shows the input throttle signal, and the angular velocity and fuel flow signals which are exported to the test bench evaluators. The transmission controller appears in this aspect. It contains the logic to change the gear ratio based on angular velocity, and is specified by the alternative container shown above, in Fig. 7. During simulation model synthesis either an ideal Simulink controller can be generated, or a Simulink model of the deployed control software implementation using the TrueTime toolkit. The physical *Transmission* model is also a design container as shown in Fig. 12. The automated design space exploration process selects one or the other transmission for the final design. A model interpreter re-creates the test bench using parts selected from the available alternatives.

Realization of Simulation Models

To illustrate the generation process once a design candidate has been selected, the following steps produce a Simulink simulation from a test bench model containing bond graph elements. A script is generated, which when run uses the Simulink API to create the model. The steps describe the order in which elements are added to the generation script.

1. In the target Simulink model, create the hierarchical structure and elements (except Bond graph and ports) for the model based on the hierarchical structure of the CyPhyML design model.
2. In CyPhyML, collect all Bond Graph elements for components in the design and create a flattened out bond graph in memory. Bonds are resolved across the hierarchy.
3. Create Bond Graph elements in Simulink for the model.
4. Create power port elements in Simulink.
5. Create signal ports in Simulink.
6. Update the Simulink layout.
7. Create connections between elements in Simulink.
8. Add commands to open Simulink scopes automatically.
9. Copy all referenced Matlab models into the output directory.

These steps have been implemented in automated CyPhyML model interpreters. A similar procedure creates composed Modelica models from a CyPhyML test bench. A few additions are required to create TrueTime models to evaluate the deployed controller implementations:

1. The CyPhyML to ESMoL translator must be invoked to create the ESMoL deployment model.
2. The ESMoL tools are run to create the controller source code, schedule the tasks, and generate scripts to create the TrueTime models as described in [11].
3. The TrueTime generation scripts are invoked by the bond graph dynamics generation scripts, integrating the TrueTime controller models with the generated dynamics model in Simulink.

Mobility Simulation

A design candidate can also be evaluated in the context of its expected operating environment using a mobility simulation. The mobility simulation exercises the vehicle in a 3D virtual world and runs in real-time, giving the user the interactive experience of driving this particular vehicle configuration over a specific, detailed terrain. The simulation provides feedback on the maximum obstacle heights the vehicle can traverse, suspension performance over a given terrain, and power train performance over a specific terrain. Delta3D is used as the 3D rendering engine and for management of 3D solid objects. Open Dynamics Engine (ODE) is a rigid-body dynamics simulator used to simulate the physical interactions of the terrain/world objects and the vehicle elements (wheels/tires, suspension). A co-simulation approach is used to combine the benefits of the rigid-body simulation with the powertrain simulation implemented in Simulink. The co-simulation is implemented with UDP socket communications that provide inputs from the 3D simulation including user throttle/brake and loading on power train due to terrain as monitored from vehicle kinematic feedback to the dynamics simulation. The Simulink simulation of the powertrain returns the torque produced from the power train that is fed back to the 3D simulation and translated as a torque on the wheels in the rigid-body physics simulation.

Evaluation of Candidate Designs

Table 1 displays values from the dashboard that can be determined by component properties, the assembly rules, and the selection of a particular design candidate. For space we only include two design candidates. Note that these numbers are only representative to illustrate the kinds of values that can be measured in the design environment.

TABLE 1. DASHBOARD RESULTS FOR DYNAMIC METRICS

Design ID	C025	C016
Max off-road speed (m/s)	32.0	29.8
Max on-road speed (m/s)	32.3	36.7
Acceleration (0-13.4 m/s in s)	7.1	7.7
Max obstacle height (m)	0.612	0.757
Min turn radius (m)	0.713	0.663
Range (km)	612.4	345.8
Ground Pressure (kPa)	30.3	34.5
Fuel Efficiency (kpl)	1.87	1.66

RELATED WORK

State of Current MBSE Tools

Reichwein and Paredis offer a canonical overview of concepts, history, and directions for the field of Model-Based Systems Engineering (MBSE) [13] - their work surveys some of the basic terminology and fundamental issues. Our work has much in common with current research efforts in SysML and related tools.

In contrast to current MBSE approaches, our integration methods aim towards a more strict 'glue' role, which does not prescribe particular technologies/standards, but selects and integrates from best of breed alternatives for attacking particular aspects of the problem. We use abstractions for each of the integrated capabilities (component definition, DSE, and generation) so that users can work in different modeling tools and integrate their existing component libraries.

The Core Product Model (CPM) from NIST is a standard which covers many of the same design issues covered by CyPhyML and its associated tools [14] [15]. CPM defines a metamodel to capture components and their features, as well as the design of assemblies using those components. CPM is multi-domain, and seeks to separate the specification of function (i.e., intended behavior), form (i.e., physical realization), and behavior (i.e., implemented behavior) within design models. From the point of view of their ability to design and synthesize simulations of model dynamics, both CPM and CyPhyML include the notion of component port objects, which allow the designer to specify a design as a hierarchy of components with interconnected ports representing joint behavior of multiple components. CyPhyML relies on a static, canonical set of port types for physical and software systems. This allows analysis tools to work with all CyPhyML models which conform to the standard. CyPhyML also has acausal power ports, which represent direct variable sharing betwen intertonnected components. Where CPM semantics are generic and adaptable, CyPhyML strives for a precise behavioral semantics in each design domain, by which well-formed models can be formally verified with respect to requirements. The static port type system also increases the compatibilitiy of components submitted to the component library from separate sources.

Model-Based Testing

Hause et al discuss the use of UML and SysML to support testing efforts [16]. They describe using test bench structures to map test sequences onto implementation models and interpret the results. This shields the testing concern from the implementation concern, and also allows tests to be re-generated if the design or testing procedures change.

SysML Extensions for Modelica

Paredis et al present a transformation specification to integrate Modelica models with SysML models. SysML4Modelica [17] contains the Modelica blocks and its equations as well. Our tools abstract away from the specifics of Modelica and focus only on the details required to perform the integration of physical components.

ACKNOWLEDGMENT

This work was supported by DARPA under contracts FA8650-10-C-7082 and FA8650-10-C-7075.

REFERENCES

[1] Sangiovanni-Vincentelli, A. L., 2007. "Quo Vadis SLD: Reasoning about Trends and Challenges of System-Level Design". *Proc. of the IEEE,* ***95***(3), March, pp. 467–506.

[2] Sztipanovits, J., Koutsoukos, X., Karsai, G., Kottenstette, N., Antsaklis, P., Gupta, V., Goodwine, B., Baras, J., and Wang, S., 2012. "Toward a Science of Cyber-Physical System Integration". *Proceedings of the IEEE,* ***100***(1), Jan, pp. 29–44.

[3] Karsai, G., Sztipanovits, J., Ledeczi, A., and Bapty, T., 2003. "Model-integrated development of embedded software". *Proc. of the IEEE,* ***91***(1), Jan, pp. 145–164.

[4] Wrenn, R., Nagel, A., Owens, R., Yao, D., Neema, H., Shi, F., Smyth, K., Ceisel, J., vanBuskirk, C., Porter, J., Bapty, T., Neema, S., Mavris, D., and Szipanovits, J., 2012. "Towards Automated Exploration and Assembly of Vehicle Design Models". In Proc. ASME International Design Engineering Technical Conf. & Computers and Information in Engineering Conf. (IDETC/CIE 2012).

[5] Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., IV, C. T., Nordstrom, G., Sprinkle, J., and Volgyesi, P., 2001. "The generic modeling environment". *Workshop on Intelligent Signal Processing*, May.

[6] Willems, J., 2007. "The behavioral approach to open and interconnected systems". *Control Systems, IEEE,* ***27***(6), Dec, pp. 46 –99.

[7] Lattmann, Z., 2010. "A multi-domain functional dependency modeling tool based on extended hybrid bond graphs". Master's thesis, Vanderbilt University.

[8] Otter, M., Elmqvist, H., and Mattsson, S. E., 2007. "Multidomain Modeling with Modelica". In *Handbook of Dynamic System Modelling*, P. A. Fishwick, ed. Chapman & Hall/CRC, ch. 36, pp. 36.1 – 36.27.

[9] Cervin, A., Arzen, K.-E., Henriksson, D., Lluesma Camps, M., Balbastre, P., Ripoll, I., and Crespo, A., 2006. "Control loop timing analysis using TrueTime and Jitterbug". In Proceedings of the 2006 IEEE Computer Aided Control Systems Design Symposium.

[10] Hemingway, G., Porter, J., Kottenstette, N., Nine, H., vanBuskirk, C., Karsai, G., and Sztipanovits, J., 2010. "Automated Synthesis of Time-Triggered Architecture-based

TrueTime Models for Platform Effects Simulation and Analysis”. In RSP ’10: 21st IEEE Intl. Symp. on Rapid Systems Prototyping.

[11] Porter, J., Hemingway, G., Nine, H., vanBuskirk, C., Kottenstette, N., Karsai, G., and Sztipanovits, J., 2010. “The ESMoL Language and Tools for High-Confidence Distributed Control Systems Design. Part 1: Language, Framework, and Analysis”.

[12] Neema, S., Sztipanovits, J., Karsai, G., and Butts, K., 2003. “Constraint-based design-space exploration and model synthesis”. In *Embedded Software*, R. Alur and I. Lee, eds., Vol. 2855 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, pp. 290–305.

[13] Reichwein, A., and Paredis, C. J., 2011. “Overview of Architecture Frameworks and Modeling Languages for Model-Based Systems Engineering”. In Proc. ASME 2011 International Design Engineering Technical Conf. & Computers and Information in Engineering Conf.

[14] Fenves, S., 2001. A Core Product Model for Representing Design Information. Tech. Rep. NISTIR 6736, NIST.

[15] Fenves, S., Foufou, S., Bock, C., and Sriram, R., 2008. “CPM2: A Core Model for Product Data”. *Journal of Computing and Information Science in Engineering,* ***8***(1).

[16] Hause, M., Stuart, A., Richards, D., and Holt, J., 2010. “Testing Safety Critical Systems with SysML/UML”. In Engineering of Complex Computer Systems (ICECCS), 2010 15th IEEE International Conference on, pp. 325 –330.

[17] Paredis, C. J., Bernard, Y., Burkhart, R. M., de Koning, H.-P., Friedenthal, S., Fritzson, P., Rouquette, N. F., and Schamai, W., 2010. “An Overview of the SysML-Modelica Transformation Specification”. In 2010 INCOSE International Symposium.