Design Space Exploration and Manipulation for Cyber Physical Systems

Himanshu Neema, Zsolt Lattmann, Patrik Meijer, James Klingler, Sandeep Neema, Ted Bapty, Janos Sztipanovits, Gabor Karsai

Institute for Software Integrated Systems, Vanderbilt University
1025 16th Ave S, Suite 102, Nashville, TN 37212

{himanshu, lattmann, patrik85, jklingler, sandeep, bapty, sztipaj, gabor}@isis.vanderbilt.edu

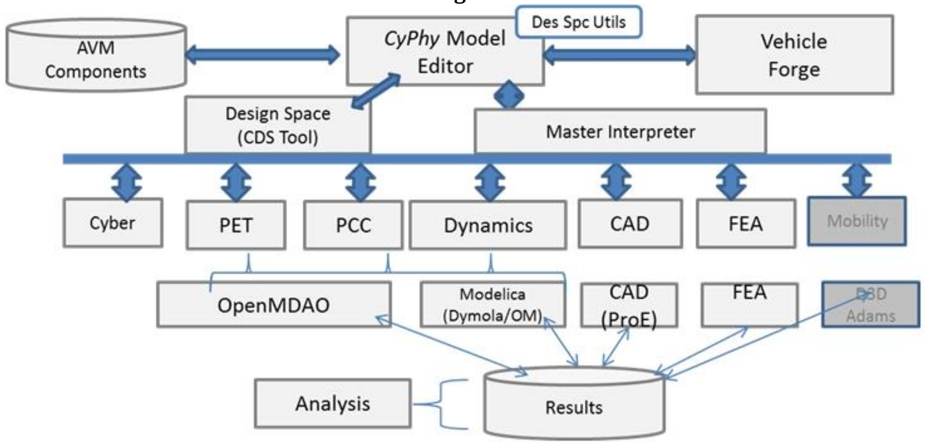
Abstract. Cyber-Physical Systems (CPS) [1] are engineered systems that require tight interaction between physical and computational components. Designing a CPS is highly challenging [2] because these systems are inherently complex, need significant effort to describe and evaluate a vast set of cross-disciplinary interactions, and require seamless meshing of physical elements with corresponding software artifacts. Moreover, a large set of architectural and composable alternatives must be systematically explored and evaluated in the context of a highly constrained design space. The constraints imposed on the selection of alternatives are derived from the system's functional, performance, dimensional, physical, and economical objectives. Furthermore, the design process of these systems is highly iterative and requires continuous integration of design generation with design selection and manipulation supported by design analyses. Existing computer-aided design tools are not well-suited for this method of design. To facilitate the iterative design process for CPS-s, we have developed a design toolchain, OpenMETA [4] [9], built around a Domain-Specific Modeling Language (DSML) [3], called the Cyber-Physical Modeling Language (CyPhyML). In this paper, we present parts OpenMETA that address the requirements of Design Space Exploration and Manipulation (DSEM) for CPS-s.

Keywords: Design Space Exploration, Design Space Manipulation, Cyber Physical Systems, Ordered Binary Decision Diagrams, Symbolic Search, Model-Integrated Computing, Domain-Specific Modeling Language

1 Introduction

Cyber-Physical Systems (CPS) [1] are systems that require tight interaction between physical and computational components. These systems span several engineering domains such as mechanical, electrical, thermal, and cyber (i.e., digital control). CPS design is highly challenging [2] because these systems are inherently complex, need significant effort to describe and evaluate a vast set of cross-domain interactions, and require seamless meshing of physical elements with their corresponding software artifacts. CPS design involves a set of architectural and composable alternatives (a ‘design space’) that must be systematically explored and evaluated. Design spaces formulated with these alternatives can grow exponentially large due to combinatorial explosion of design choices. A way to limit discrete design selections is to utilize various known *hard* design constraints. These constraints can be derived from the system’s functional (e.g., gas, electric, or hybrid drivetrain), performance (e.g., min. torque of engine), dimensional (e.g., max. height or capacity), physical (e.g., weight, join structures), and economical (e.g. vendor acquisition cost, system’s operating cost) objectives. In this way, only those design configurations are generated that satisfy these design constraints. Using existing Computer-Aided Engineering (CAE) tools, the process of fully specifying all of the possible alternatives is cumbersome and has the potential of being extremely time-consuming, especially if a user were to apply all these design constraints manually.

Fig. 1.



OpenMETA Architecture

Good system design must further consider several factors such as manufacturability, stability, complexity, reliability, risks, time-to-market, etc. However, we consider these factors as secondary, coming after the basic set of constraints has been satisfied. The discrete selection of composable and architectural alternatives based on these primary constraints form the initial Design Space Exploration (DSE). The resulting full-system alternative designs (i.e., configurations or design points) can then be analyzed for evaluation against the secondary requirements. Existing CAE tools do not quite help in making this task easy, as few support any kind of parallel multi-model simulation execution and comparison of results. The result of these detailed system-level analyses in terms of valid design selections and reformulations should then be incorporated into the original design space, which must be re-explored to generate a new set of valid design configurations. This iterative nature of the design process with strong bidirectional coupling between design activities and system analysis and verification is a key to efficient and effective CPS design.

To streamline the iterative design process for CPS-s, we have developed a design toolchain called *OpenMETA* [4] [9] (see **Fig. 1**), built around a Domain-Specific Modeling Language (DSML) [3], called the Cyber-Physical Modeling Language (CyPhyML). The CyPhyML captures integration interfaces of system components across multiple design domains (e.g., *Cyber*, *CAD*, *FEA*) as well as generic assembly rules given in terms of composable and architectural alternatives and *hard* design constraints for the final assembly. OpenMETA supports multi-level and multi-fidelity exploration of system-level architectural and parametric tradeoffs. These tools facilitate the iterative design process by integrating formal qualitative reasoning methods, Design Space Exploration and Manipulation (DSEM), and analysis of design alternatives. In this paper, we focus on tools that support DSEM for CPS design.

The rest of the paper is organized as follows: Section 2 provides an overview of the design space portion of the CyPhyML. Section 3 describes our DSEM tools. Section 4 puts our DSEM tools in the context of the iterative design process for CPS-s. Section 5 presents a case study that shows DSEM tools usage and presents analyses results. Related work is given in Section 6, and Section 7 concludes the paper.

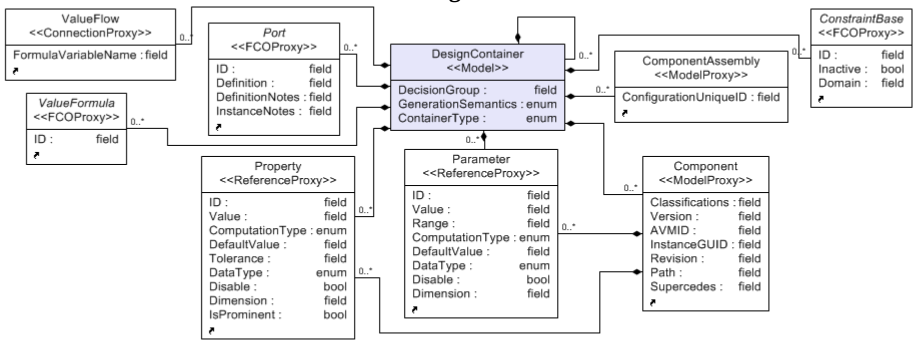
2 Modeling Language

The CyPyML uses Model-Integrated Computing (MIC) [3] techniques to support design-time integration of vast number of system-level design aspects and methods, and the automated exploration and manipulation of design spaces. Model-Integrated Computing (MIC) is the core technology on which CyPhyML and its tools are built. MIC focuses on the formal representation, composition, analysis, and manipulation of models during the design process. It utilizes models as the common concept throughout the entire life-cycle of systems, including specification, design, development, verification, integration, and maintenance. The Generic Modeling Environment (GME) is a metaprogrammable toolkit that enables definition and use of Domain-Specific Modeling Languages (DSMLs) [3] such as CyPhyML. In MIC, DSMLs are configured through metamodels, expressed as UML class diagrams, specifying the modeling paradigm of the application domain. Metamodels characterize the abstract syntax of the DSML, defining which objects (i.e. boxes, connections, and attributes) are permissible in the language. Simplistically, DSML is a schema or data model for all possible models that can be expressed by a language. A DSML for finite state machines would consist of states, and transitions, from which any valid state machine can be realized. The inherent flexibility and extensibility of GME via metamodels make it an ideal platform for CPS design and analysis using CyPhyML.

CyPhyML captures the concepts of design models in various CPS domains, specifies how these concepts are organized and related, and specifies the rules governing their composition (i.e., the relationships, inter-compatibility, and connectibility of discrete component models and subsystems). OpenMETA consists of a number of model interpreters and analysis tools, which can be used to generate system and analysis artifacts from system designs, and perform various structural and dynamic analyses.

Any DSML requires precise specification of the language's syntax and semantics. **Fig. 2** provides a simplified view of the design space part of the CyPhyML metamodel. As shown, the central modeling element in the language is called a *DesignContainer*. The key attribute of a design container is *ContainerType*, which can have one of the following three values: *Compound*, *Alternative*, or *Optional*. All elements of a compound design container must be part of the final system design. Alternative design containers are used to capture design choices/trade-offs. The final assembly will include only one of the choices from alternative design containers. Optional design containers are similar to alternative, but also allow 'none' as an option.

Fig. 2.



Simplified metamodel of the design space

A key element of this language is that design containers can contain child design containers. This enables construction of a hierarchical AND/OR design space. The concrete elements of the design are *Component* and *ComponentAssembly* (CA). A CA is a system that can only contain components and child CAs (i.e. subsystems), and represents a system that has been fully explored, analyzed, and finalized.

As can be seen in **Fig. 2** components, component assemblies, and design containers have special elements called *Property* and *Parameter*. A property represents a static property of a component or a component assembly. Properties cannot directly be changed at design time during design space model construction. Examples of properties are engine's power rating, or a driveshaft's mass. A property of a design container may correspond to a basic property at that level in the design space or it may be a representative property that is calculated based on the chosen sub-elements of the design container. Alternatively, CyPhyML parameters can be used to specify a range of acceptable values. A key element of our tools is that parameters are automatically translated into design constraints to ensure that the values of the parameters generated for selected configurations lie within the ranges specified.

CyPhyML also supports combining the properties and parameters using *ValueFormula*. The *ValueFlow* connections are used to connect properties and parameters to value formulae. A simple example could be to calculate mass of the system by adding the masses of its sub-components. CyPhyML supports two different kinds of value formulae: *SimpleFormula* and *CustomFormula* (not shown in the figure). A *SimpleFormula* is used for basic arithmetic operations on incoming *ValueFlow* properties, while a *CustomFormula* is used in situations when a derived property needs to be calculated using complex operations, e.g., *Cosine()* and *Sqrt()*.

For the specification of high-level as well as fine-tuned system requirements, CyPhyML also provides a large number of constraint types to support design constraints arising from component interactions and due to functional and practical system requirements. We provide details of the supported constraint types in the next section.

3 Design Space Tools

CyPyML provides a collection of modeling methods and tools for exploration and visualization of designs and design spaces, solving complex design constraints, and effective management of the design spaces and designs. A brief overview of these tools is given below. Examples of tool usage and analyses are given in Section 5.

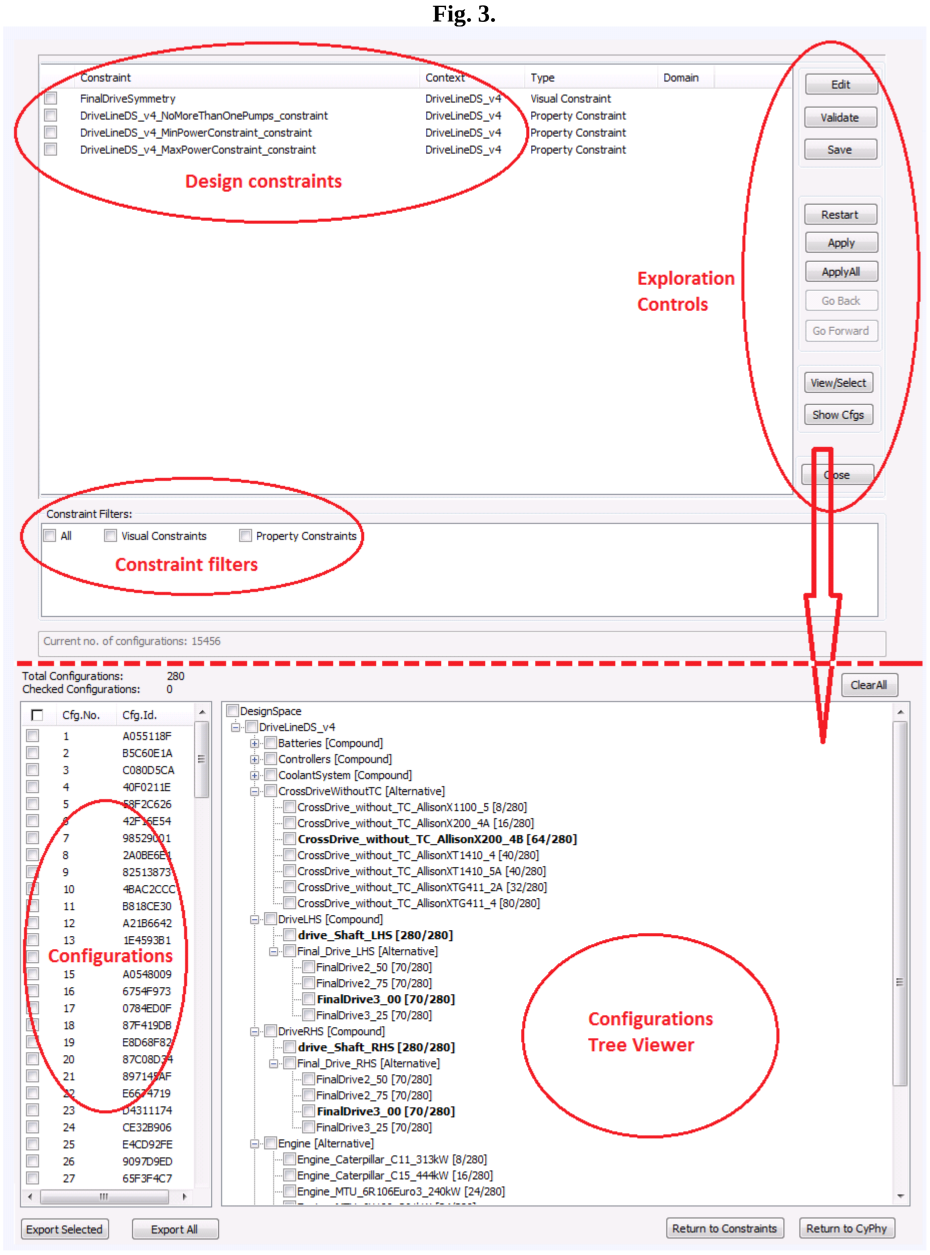
3.1 Design Space Exploration Tool (DESERT)

This is our key tool for design space exploration. It uses symbolic constraint satisfaction for design space exploration using Ordered Binary Decision Diagrams (OBDD-s) [5]. The choices in the design space come from the AND-OR-LEAF tree structure as well as from the variability of values that can be bound to properties and parameters. The design space is a cross product of all possible choice outcomes. The process begins with a binary encoding of the design space, including the AND-OR-LEAF tree and the design constraints. Each node in the design space tree is assigned a unique integer identifier (ID). These IDs are then translated into BDD variables such that the encoding reflects the design container containment semantics. The properties and constraints are also handled symbolically as BDDs [6]. For handling variable properties, we extended BDDs to Multi-Terminal BDDs (MTBDDs), which enables values other than 0 and 1 as terminals of BDDs [6]. With this encoding, the constraint satisfaction amounts to the creation and composition of design constraints and the symbolic design space representation. The resultant BDD represents the pruned design space. The symbolic representation has been proven to handle very large design spaces consisting of up to 10^{80} design configurations [6]. The interested reader is referred to [5] for an overview of BDDs and to [6] for detailed constraint-driven design space exploration algorithm used in OpenMETA.

The key elements of DESERT are shown in **Fig. 3**. The exploration controls allow the users to manage constraints and enable them to selectively apply them to explore design spaces. The number of viable constraints may reduce or increase as new constraints are applied or reverted respectively. It also permits grouping constraints using their types and domains for selective constraint application. Further, design elements can also be selected to be included in all configurations.

The right-hand side of **Fig. 3** shows the configurations in a tree view. The left panel lists the configurations and the right panel shows the corresponding design space tree with selection frequency for each design element. Users can select configurations and corresponding design elements are highlighted. Users can also select a particular alternative element and corresponding configurations on the left become checked. The selected configurations can be exported back to design space.

Fig. 3.



Constraint	Context	Type	Domain
FinalDriveSymmetry	DriveLineDS_v4	Visual Constraint	
DriveLineDS_v4_NoMoreThanOnePumps_constraint	DriveLineDS_v4	Property Constraint	
DriveLineDS_v4_MinPowerConstraint_constraint	DriveLineDS_v4	Property Constraint	
DriveLineDS_v4_MaxPowerConstraint_constraint	DriveLineDS_v4	Property Constraint	

Constraint Filters:
All   Visual Constraints   Property Constraints

Current no. of configurations: 15456

Total Configurations: 280
Checked Configurations: 0

Cfg.No.	Cfg.Id.
1	A055118F
2	B5C60E1A
3	C080D5CA
4	40F0211E
5	58F2C626
6	42F16E54
7	98529001
8	2A0BE6E1
9	82513873
10	4BAC2CCC
11	B818CE30
12	A21B6642
13	1E4593B1
[illegible]	[illegible]
15	A0548009
16	6754F973
17	0784ED0F
18	87F419DB
19	E8D68F82
20	87C08D34
21	897145AF
22	E6674719
23	D4311174
24	CE32B906
25	E4CD92FE
26	9097D9ED
27	65F3F4C7

```
DesignSpace
  DriveLineDS_v4
    Batteries [Compound]
    Controllers [Compound]
    CoolantSystem [Compound]
    CrossDriveWithoutTC [Alternative]
      CrossDrive_without_TC_AllisonX1100_5 [8/280]
      CrossDrive_without_TC_AllisonX200_4A [16/280]
      CrossDrive_without_TC_AllisonX200_4B [64/280]
      CrossDrive_without_TC_AllisonXT1410_4 [40/280]
      CrossDrive_without_TC_AllisonXT1410_5A [40/280]
      CrossDrive_without_TC_AllisonXTG411_2A [32/280]
      CrossDrive_without_TC_AllisonXTG411_4 [80/280]
    DriveLHS [Compound]
      drive_Shaft_LHS [280/280]
      Final_Drive_LHS [Alternative]
        FinalDrive2_50 [70/280]
        FinalDrive2_75 [70/280]
        FinalDrive3_00 [70/280]
        FinalDrive3_25 [70/280]
    DriveRHS [Compound]
      drive_Shaft_RHS [280/280]
      Final_Drive_RHS [Alternative]
        FinalDrive2_50 [70/280]
        FinalDrive2_75 [70/280]
        FinalDrive3_00 [70/280]
        FinalDrive3_25 [70/280]
    Engine [Alternative]
      Engine_Caterpillar_C11_313kW [8/280]
      Engine_Caterpillar_C15_444kW [16/280]
      Engine_MTU_6R106Euro3_240kW [24/280]
```

Design Space Exploration Tool

DESERT supports a number of different types of constraints, most of which can be specified graphically. However, for constraints that involve several complex mathematical functions, an extended Object Constraint Language (OCL) [3] for their specification is also supported. Functions supported include all *trigonometric* and *logarithm* functions as well as *sign*, *rint*, *sqrt*, and *abs*. The example constraint given below ensures that the vehicle can accelerate at 2 m/s^2 on a 20-degree uphill.

$$(\text{Powerplant\_maxTorque}() * 1.3558) >= ((\text{Tire\_radius}()/1000) * (\text{Vehicle\_weight}() + 13000) * \sin(0.35) + ((\text{Vehicle\_weight}() + 13000)/9.8) * 2) \quad (1)$$

Fig. 4.

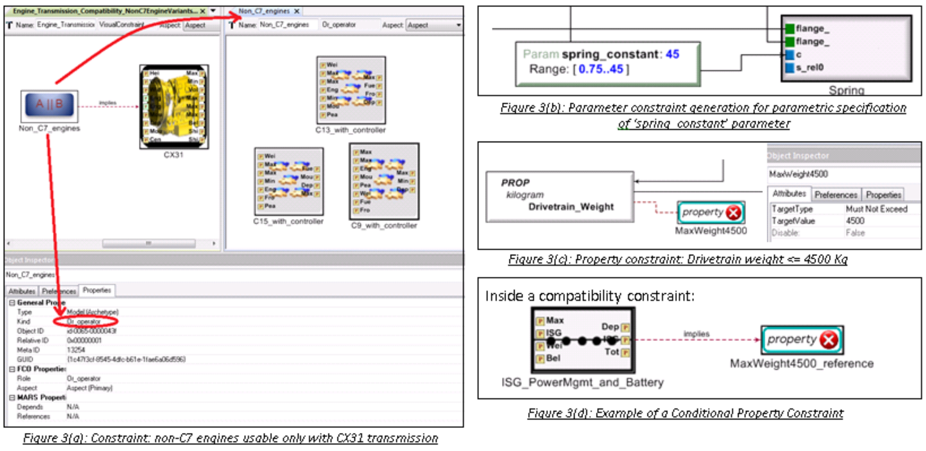


Figure 3(a): Constraint: non-C7 engines usable only with CX31 transmission

Figure 3(b): Parameter constraint generation for parametric specification of 'spring_constant' parameter

Figure 3(c): Property constraint: Drivetrain weight <= 4500 Kg

Figure 3(d): Example of a Conditional Property Constraint

Constraints type examples in DESERT

DESERT allows *compatibility constraints* represented graphically as *Implies* connections between design elements. Using AND/OR groups of design elements, these can be nested hierarchically to create complex Boolean expressions for compatibility constraints. **Fig. 4(a)** shows constraint that non-C7 engines are only compatible with CX31 transmission. The parameter specifications are also translated into constraints. In **Fig. 4(b)**, the range given is translated into a *parameter constraint* to ensure that the value of *spring_constant* parameter lies between 0.75–45.0 N/m. Many requirements can be captured using simple limit constraints on properties. **Fig. 4(c)** shows a *property constraint* example. For property constraints applicable only for certain selection of alternatives, a *conditional property constraint* can be used. In **Fig. 4(d)**, *MaxWeight4500* is applied only if *ISG_PowerMgmt_and_Battery* is in a configuration. We also support succinct specification of constraints for some special cases. For example, all 4 tires of a vehicle might have same alternatives, but all must be same in any full-system configuration. Instead of specifying cross-product of compatibility constraints, *DecisionGroup constraints* can be used, where the user simply collects the alternative design containers, the equivalent compatibility constraints are autogenerated.

3.2 Component Assembly Export Tool (CAExporter)

The configurations exported by DESERT contain only a flat set of references to the design elements that were selected during the DESERT process. The CAExporter tool is then used to convert these configurations into fully-specified component assemblies that include the full hierarchical and structural composition of all the components and sub-assemblies, along with all the ports and connections.

3.3 Design Space Refinement Tool (DSRefiner)

DSRefiner allows the user to select a subset of configurations of a design space and generate a *refined* design space using these configurations. The refined design space has the exact same hierarchical structure, ports, and connections as the original design space, but omits design elements from the original design space that are not part of the selected configurations. Further, original design constraints are removed, but a new compatibility constraint is added that ensures that when DESERT is run on the refined design space, the exact same set of configurations is generated. This avoids repetition of the analyses that were done in the original design space. DSRefiner is highly useful for gradually building design spaces, performing coarser-grained analyses, and incorporating the results for refining and manipulating the design space.

3.4 Design Space Manipulation Tool (DSRefactorer)

DSRefactorer is a key design space tool that directly supports iterative design process for CPS-s. Design spaces evolve continually during the design process as more knowledge is gathered as well as when more refinement is done or parts of the design space are finalized. We developed a dedicated tool that can manipulate design spaces according to various domain-specific use-cases, while maintaining the hierarchical placement, port restrictions, and valid connections. DSRefactorer is context sensitive such that the refactoring choices presented to the user and the actions taken depend on where and on what design element the tool is invoked. Following are some of the key refactoring use-cases: (1) Component Assembly (CA) to a Design Container (DC) for design space extension, (2) Component or CA to an Alternative DC with original component or CA as a choice, (3) Extract design elements of a CA or a Compound DC in the parent DC, (4) One or more components and/or CA-s to inside a new CA or a new compound DC, (5) DC to a new Alternative DC with original DC as one choice, (6) Compound DC to a CA, and (7) Optional DC to a Compound DC.

3.5 Design Space Criticality Meter (DSCriticalityMeter)

The DSCriticalityMeter generates the frequency of design elements in the set of configurations generated from a design space. Using this metric, designers can make informed decisions regarding resource allocation: if an element is ubiquitous (or absent) in the generated configurations, it might merit increased scrutiny.

3.6 Supporting Tools

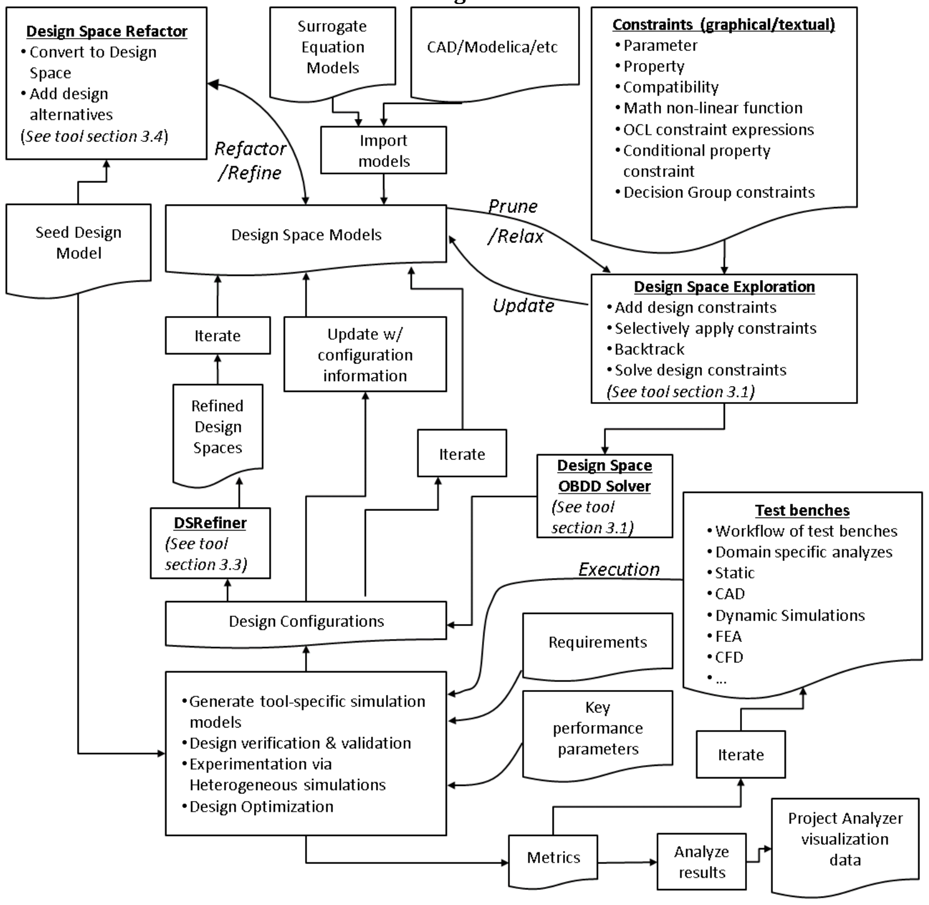
OpenMETA has other interpreter tools that are associated with DSE. The *Component Authoring Tool* provides importing capability from various domains (e.g. CAD, Modelica) into CyPhyML, essentially wrapping domain models with CyPhyML interfaces and making them readily usable in the OpenMETA toolchain. After the component library is populated the *Component Library Manager* helps to discover and insert different instances of the same component types into an alternative design container. Once these containers (i.e., component- and subsystem-placeholders) are composed into a design space and design configurations are exported, the *Master Interpreter* automates the translation of each unique design configuration into an executable domain-specific model, essentially parsing the CyPhyML design configuration and generating a valid, simulation-ready compound model which incorporates preexisting component models. After these model transformations, the *Master Interpreter* transfers the generated executable models (along with the appropriate domain-tool analysis packages) to the *Job Manager*, which executes the analyses using domain specific tools (e.g., Dymola, Creo, etc.) via the *Job Manager*. OpenMETA supports parallel execution of analyses, either locally on user's computer or remotely on a cloud of Jenkins-managed 'slave' job executors, which have the appropriate domain-specific tools installed.

4 Iterative Design Process and Design Flows

CPS design is a major integration problem because of their inherently complexity and unexpected component interactions. As shown in [7], the design process must allow for the continuous existence of an executable system, with a concrete architecture, well-defined interfaces, and an executable form. This allows designers to analyze their designs earlier during the design process and obtain useful feedback. This facilitates less error-prone designs, saved manpower, and manageable design spaces. To enable the iterative design process for CPS-s, OpenMETA supports three key design flows.

Fig. 5 shows the iterative design process in OpenMETA pictorially. As can be seen a classic design flow is to begin with creating a design space model with top-level design container and then adding further design elements in it along with constraints on those elements according to design requirements. A second design flow show in the figure, begins from a single seed design. This design flow is highly applicable for the real-world design use-cases, where there are existing designs and design processes. Starting from the seed design, the user extends the design space (using the tools mentioned above) to add alternatives in place of concrete design elements such as components or component assemblies. In this fashion, the user grows a larger design space from that seed design. Another supported design flow is when the user does not have concrete design elements or assemblies to work with. In this case, the user can use *surrogate equations* in place of design space elements. The design space can still be explored and analyzed. These surrogates can then be replaced with more accurate models as they become available. Surrogates are also helpful for performing coarse-grained analyses, the results of which can be used to refine the design space.

Fig. 5.


Design Space Refactor
• Convert to Design Space
• Add design alternatives
(See tool section 3.4)
Surrogate Equation Models
CAD/Modelica/etc
Constraints (graphical/textual)
• Parameter
• Property
• Compatibility
• Math non-linear function
• OCL constraint expressions
• Conditional property constraint
• Decision Group constraints
Import models
Refactor /Refine
Seed Design Model
Design Space Models
Prune /Relax
Update
Design Space Exploration
• Add design constraints
• Selectively apply constraints
• Backtrack
• Solve design constraints
(See tool section 3.1)
Iterate
Update w/ configuration information
Refined Design Spaces
Iterate
Design Space OBDD Solver
(See tool section 3.1)
DSRefiner
(See tool section 3.3)
Test benches
• Workflow of test benches
• Domain specific analyzes
• Static
• CAD
• Dynamic Simulations
• FEA
• CFD
• ...
Execution
Design Configurations
Requirements
Key performance parameters
• Generate tool-specific simulation models
• Design verification & validation
• Experimentation via Heterogeneous simulations
• Design Optimization
Iterate
Project Analyzer visualization data
Metrics
Analyze results

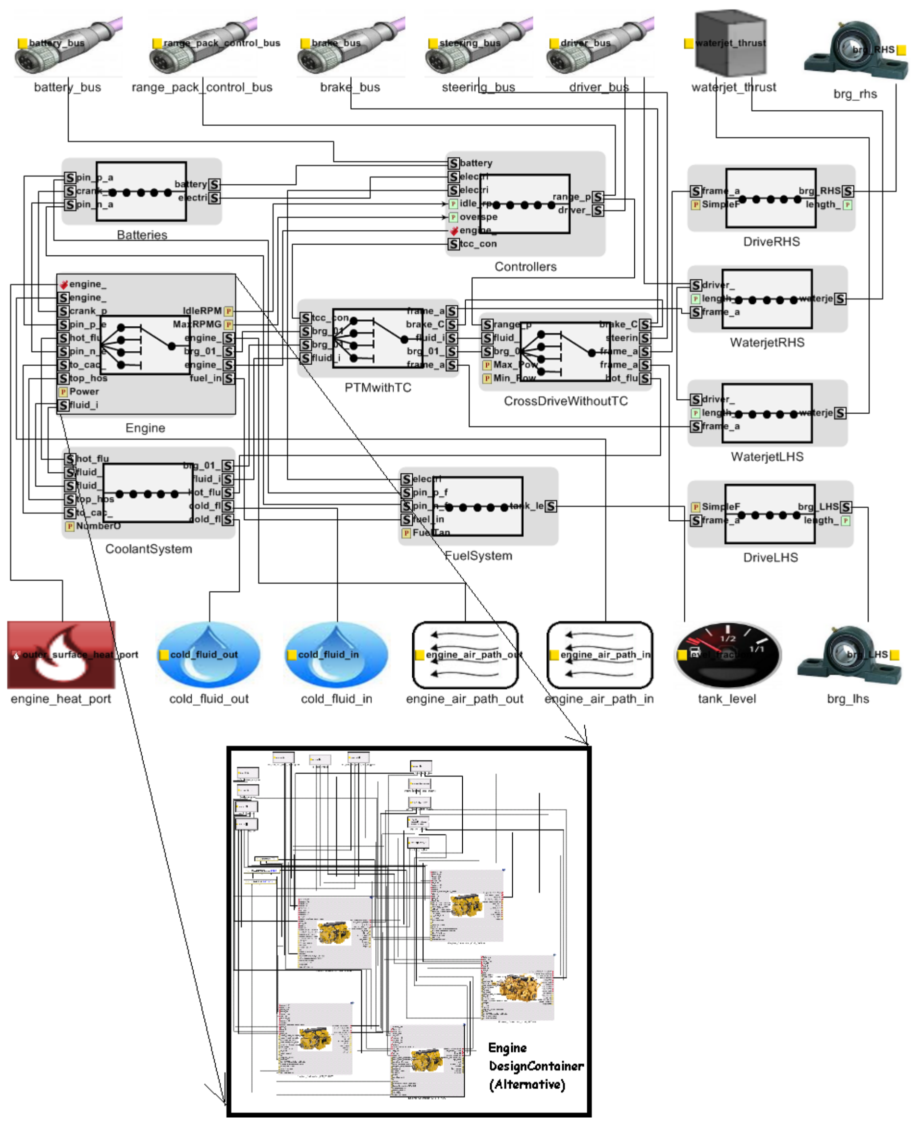
Iterative Design Process in OpenMETA

It is important to note that while there are several design flows that users can exercise in OpenMETA, all of the design space tools, such as DSRefactorer and DSRefiner, are equally applicable. Different design flows do not eliminate the need to continually evolve design spaces using a closed-loop integration of design and analysis activities. As shown in **Fig. 5**, and described previously, OpenMETA provides several supporting tools to build, test, and analyze design configurations.

5 Case Study

In this section we present a simplified drive line model as a case study for design space exploration. Many CAE tools (e.g., CAD, FEA, CFD, Modelica) have great capabilities to analyze a design, but creating and extending a design in CAE tools often takes significant time, and invariably requires subject matter expertise. To improve the process, OpenMETA allows users to alter a seed design with alternative component instances, and then to evaluate which combination performs best. We use DSEM tools in OpenMETA to capture architectural and composable alternatives and specify design constraints, generate configurations that satisfy these constraints, and then apply a set of model translations to generate fully-specified executable models, ready for simulation by the CAE tools.

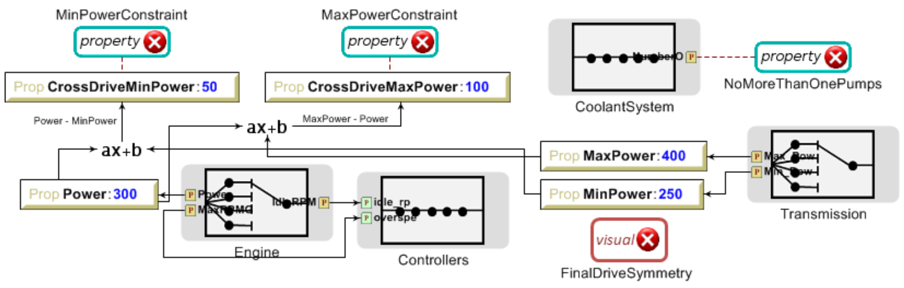
Fig. 6.



Driveline Design Space Model

Fig. 6 shows the design space of a simplified drive line that contains several subsystems. In the figure, each dotted icon in the center represents a DesignContainer and system interfaces are at the periphery of the image. The architecture and composition of the design space was derived from a single design point built in Modelica. The design space is extended with additional engine, power take-off module, transmission, final drive, hydraulic fan, and hydraulic pump alternatives. This leads to a large design space – 15456 combinatorial configurations – many of which are not even viable due to design constraints. **Fig. 7** shows 4 constraints (see Section 3.1 for constraint types). *FinalDriveSymmetry* constraint ensures that the left and right drives have the same gear ratios. *MinPower* and *MaxPower* constraints assert that the engine's nominal power lies between the transmission's min. and max. power ratings. *NoMoreThanOnePumps* ensures the design uses no more than one hydraulic pump. Application of these constraints reduces the viable number of designs to 47 – a manageable set that users can analyze.

Fig. 7.

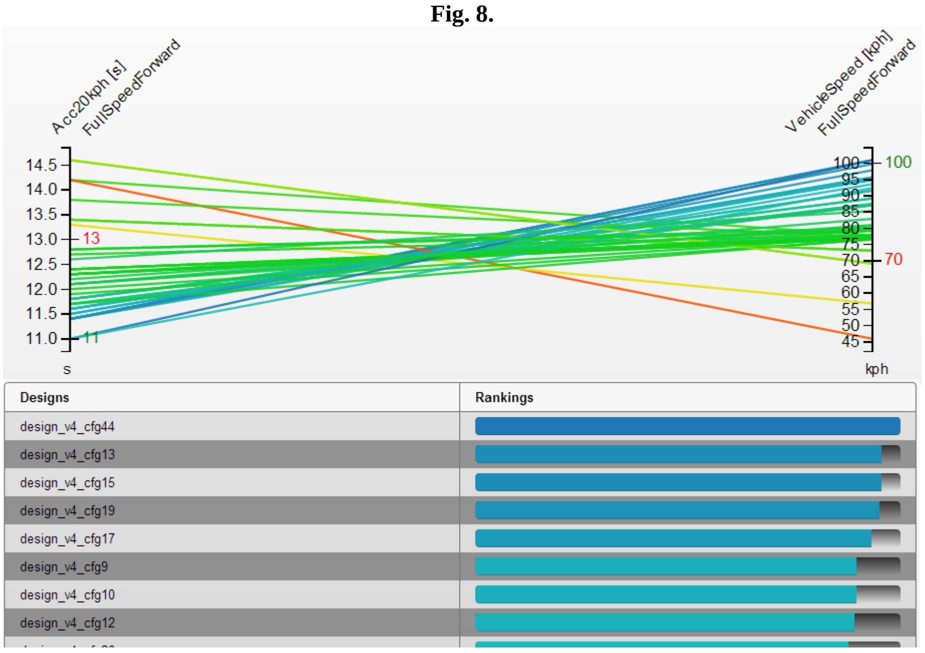


Constraints used in the drive line design space

The generated configurations capture the relevant information across multiple domains, but in this case study, we focus on the dynamic behavior of the systems: the generated Modelica system models, simulation execution, and visualization of results. Results are collected locally and presented using the *Project Analyzer* tool. Visualization capabilities include requirement analysis, design point grouping, design point comparison, parallel axis plots, multivariate plots, surrogate model views, multi-attribute decision analysis, and physical limit violations across the entire design space.

We used OpenMETA to analyze the behavior of the selected 47 configurations using the Modelica simulation tools, and the collected results are shown in **Fig. 8**. It shows two visualization capabilities of the Project Analyzer (a) parallel axis plot and (b) multi-attribute decision analysis. The parallel axis plot has vertical axis for each variable of interest from the analysis and each colored plot represents a design configuration. The requirement objective and threshold values are shown with green and red colors respectively. The multi-attribute decision analysis widget shows an ordered list of configurations based on the user's specified weighting of each variable of interest. This is an interactive widget that helps to quickly identify differences between designs and choose the best design based on user preferences.

Fig. 8.



Designs	Rankings
design_v4_cfg44	
design_v4_cfg13	
design_v4_cfg15	
design_v4_cfg19	
design_v4_cfg17	
design_v4_cfg9	
design_v4_cfg10	
design_v4_cfg12	

Project Analyzer: Parallel axis plot and multi-attribute decision analysis

6 Related Work

Design Space Exploration for complex CPS cannot be realized as a closed form analytical search procedure, and requires multiple techniques, at multiple abstractions and fidelity, and involves complex iterations. Our toolchain is uniquely placed in that respect incorporating a comprehensive suite of methods for design space exploration (DESERT – discrete combinatorial design space, PET – parametric design space, Simulation based design metric evaluation, Dashboard for design space metric visualization). However, there are several comparable efforts implementing constraint-based search procedures using various methods. The use of OBDDs for design space exploration is well known [6]. In the work presented, we use these concepts and techniques, but extend the framework with a library of tools (e.g., translation, parallel execution of simulations, and simplified result comparison) that are necessary for the iterative design process for cyber-physical systems. There are also satisfiability solvers such as CoBaSA [10], but these do not provide means for exploration and manipulation of design spaces, an issue that is addressed by OpenMETA. Other efforts involve addressing semantics of integration [12] [14] and model-based rapid synthesis of heterogeneous simulations [13] [15]. Many efforts exist that focus on finding globally optimal solutions [11], but often the system architecture is not fully defined in the initial design stages. Hence, it becomes difficult to define an objective function (which is paramount to this approach), and designers must iteratively build the design based on experimentation with a large, and often cumbersome, set of alternatives.

7 Conclusions and Future Work

In this paper, we have identified some of the challenges of Cyber-Physical System design and highlighted the merits of using an iterative design process. We presented a model-based design environment, called OpenMETA, which provides a number of tools for design space exploration and manipulation.

OpenMETA is in use by 'beta' testers from several industry-leading vehicle design companies (Caterpillar, Oshkosh, Ricardo, among others) in conjunction with DARPA's Adaptive Vehicle Make program, and we are incorporating their feedback into improving both the robustness and the ease-of-use of the toolchain. Our current work also specifically focuses on extending the toolchain for (i) additional design space manipulation tools, (ii) increasing the library of supported design constraints including those that get automatically generated from specifications and ones that facilitate succinct representations of existing methods, (iii) increase feedback messaging from analysis tools into design space exploration and manipulation.

8 Acknowledgements

This research is supported by the AVM Program [8] of the Defense Advanced Research Project Agency (DARPA) under award # HR0011-12-C-0008 and the National Science Foundation under award # CNS-1035655.

References

1. Sztipanovits J.: Composition of cyber-physical systems. In: 14th Annual IEEE Int'l. Conference and Workshops on the Engineering of Computer-Based Systems (ECBS '07), Washington, DC, USA. IEEE Computer Society, 2007, pp. 3–6.
2. Lee E.: Cyber physical systems: Design challenges. In: Proc. of the 11th IEEE Int'l. Symposium on Object Oriented Real-Time Distributed Computing (ISORC '08), May 2008, pp. 363–369.
3. Sztipanovits J., Karsai G.: Model-Integrated Computing. In: IEEE Computer 30, 1997, pp. 110-112.
4. Wrenn R., Nagel A., Owens R., Yao D., Neema H., Shi F., Smyth K., van Buskirk C., Porter J., Bapty T., Neema S., Sztipanovits J., Ceisel J., Mavris D.: Towards Automated Exploration and Assembly of Vehicle Design Models. In: ASME 2012 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference (IDETC '2012), Chicago, Illinois, USA, August 12-15, 2012. Volume 2: 32nd Computers and Information in Engineering Conference, Parts A and B, pp. 1143-1152. doi:10.1115/DETC2012-71464.
5. Bryant R.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers, vol. C-35, pp. 677-691, 1986.
6. Neema S., Sztipanovits J., Karsai K.: Constraint-based design-space exploration and model synthesis. In EMSOFT, 2003, pp. 290–305.
7. Karsai, G., Sztipanovits, J.: Model-Integrated Development of Cyber-Physical Systems. In: Proceedings of the 6th IFIP WG 10.2 international workshop on Software Technologies

for Embedded and Ubiquitous Systems, October 01-03, 2008, Anacarpi, Capri Island, Italy. doi: 10.1007/978-3-540-87785-1_5.
8. DARPA Adaptive Vehicle Make Program. www.darpa.mil/Our_Work/TTO/Programs/Adaptive_Vehicle_Make__(AVM).aspx.
9. Lattmann Z., Nagel A., Scott J., Smith K., van Buskirk C., Porter J., Neema S., Bapty T., Sztipanovits J., Ceisel J., Mavris D.: Towards Automated Evaluation of Vehicle Dynamics in System-Level Designs. In: ASME 2012 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference. Volume 2: 32nd Computers and Information in Engineering Conference, Parts A and B, Chicago, Illinois, USA, August 12–15, 2012, pp. 1131-1141. doi:10.1115/DETC2012-71378.
10. Manolios P., Subramanian, G., Vroon D.: Automating component-based system assembly. In: ISSTA 2007, pp. 61-72.
11. Gries M.: Methods for evaluating and covering the design space during early design development. In: Integration, 38(2):131{183, 2004.
12. Porter, J., Lattmann, Z., Hemingway, G., Mahadevan, N., Neema, S., Nine, H., Kottenstette, N., Volgyesi, P., Karsai, G., Sztipanovits, J.: The ESMoL Modeling Language and Tools for Synthesizing and Simulating Real-Time Embedded Systems. In: 15th IEEE Real-Time and Embedded Technology and Applications Symposium, San Francisco, CA, April 2009.
13. Neema, H., Gohl, J., Lattmann, Z., Sztipanovits, J., Karsai, G., Neema, S., Bapty, T., Batteh, J., Tummescheit, H., Sureshkumar, C.: Model-Based Integration Platform for FMI Co-Simulation and Heterogeneous Simulations of Cyber-Physical Systems. In: Proceedings of the 10th International Modelica Conference, pp. 235-245, March 2014, Lund University, Solvegatan 20A, SE-223 62, Lund, Sweden.
14. Simko, G., Levendovszky, T., Neema, S., Jackson, E., Bapty, T., Porter, J., Sztipanovits, J.: Foundation for Model Integration: Semantic Backplane. In: Proceedings of the ASME 2012 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference, IDETC/CIE, 2012.
15. Hemingway, G., Neema, H., Nine, H., Sztipanovits, J., Karsai, G.: Rapid Synthesis of High-Level Architecture-Based Heterogeneous Simulation: A Model-Based Integration Approach. In: SIMULATION, vol. March 17, 2011 0037549711401950, no. March 17, 2011, Online, Simulation: Transactions of the Society for Modeling and Simulation International, pp. 16, March 2011.