A domain-specific language for model composition and verification of multidisciplinary models

Conference Paper · March 2016

CITATIONS
4

READS
296

5 authors, including:

Amogh Kulkarni
Vanderbilt University
5 PUBLICATIONS **7** CITATIONS
SEE PROFILE

Daniel Balasubramanian
Vanderbilt University
44 PUBLICATIONS **380** CITATIONS
SEE PROFILE

Gabor Karsai
Vanderbilt University
389 PUBLICATIONS **8,616** CITATIONS
SEE PROFILE

Anantha Narayanan
University of Maryland, College Park
50 PUBLICATIONS **607** CITATIONS
SEE PROFILE

2016 Conference on Systems Engineering Research

A domain-specific language for model composition and verification of multidisciplinary models

Amogh Kulkarni[a], Daniel Balasubramanian[a,*], Gabor Karsai[a], Anantha Narayanan[b], Peter Denno[c]

[a]*Institute for Software Integrated Systems, Vanderbilt University, 1025 16th Ave South, Nashville, 37212, USA*
[b]*University of Maryland, College Park, MD 20742*
[c]*National Institute of Standards and Technology, 100 Bureau Dr., Gaithersburg, 20899, USA*

Abstract

Complex, engineered products and manufacturing processes often necessitate integrated analysis that cuts across physical domains and engineering disciplines. When the domain-specific models that contribute to the overall analysis process are available then the problem can be addressed by composing them into an analysis workflow which then can be executed using some execution platform. Such a composition and integrated analysis is essentially a systems engineering approach applied to an engineering process. In this paper we describe a model integration language that allows the rapid composition of models, the verification of the composition and the generation of executable code and other engineering artifacts that are needed for model execution on a software platform. The language is based on OpenMDAO, a widely-used model execution framework, and it improves the engineering process by checking composition constraints that must be satisfied by the integrated model and by automatically generating executable code that facilitates the run-time integration of the models. This paper describes the design of the language, illustrates its use through a running example and outlines future work.

Keywords: MDAO; OpenMDAO; DSML; Modeling; Model composition and verification

1. Introduction

Multidisciplinary design and optimization (MDAO) is an engineering methodology concerned with design problems that span across multiple domains and engineering disciplines. The goal is to integrate models and tools from multiple disciplines to provide high fidelity analysis and optimization for multidisciplinary systems. MDAO is

* Corresponding author. Tel.: +1-615-343-7472; fax: +1-615-343-7440.
E-mail address: daniel@isis.vanderbilt.edu

gaining use in the aerospace industry[2] and has also been applied in the manufacturing domain[3].

Nomenclature

CONMIN	Constraint Minimizer
DOE	Design Of Experiment
DSML	Domain-specific modeling language
MDAO	Multidisciplinary design and optimization
SLSQP	Sequential Least Squares Programming

MDAO allows problems that would normally be described in many, individual domain tools to be analyzed in a single tool. For example, consider selective laser melting, an additive manufacturing process, consisting of multiple stages such as heat-transfer from the heat source to the powder bed, enthalpy change of the powder bed and change in temperature and phase. Phase changes include multiple physical phenomena like melting, pooling and shrinking[1]. Each sub-process and/or phenomenon falls under a distinctly different discipline such as optics, thermodynamics and fluid mechanics. MDAO allows this overall selective laser melting process to be modeled with the help of single platform instead of using different ones for each one of the distinct sub-processes.

However, one of the biggest difficulties when using many approaches to MDAO is that the underlying integration code that connects the individual tools and models must be created manually. Creating this integration code manually is problematic for two reasons. The first reason is that the code at the integration level does not understand the semantics of the models being integrated. For example, a developer may mistakenly write code which connects a variable representing one physical quantity in a constituent model to another variable that represents a different physical quantity; and as this kind of mismatch is syntactically valid, the semantics must be understood to detect the true problem. The second reason that creating the integration code manually is problematic is that it makes debugging very difficult. If analysis and optimization yield unexpected results, it could be due to errors in the individual domain models, or due to the errors in the integration code or even due to non-compliance with the underlying integration framework. Not knowing whether to investigate the domain models or the integration code as the potential cause of errors is both time-consuming and tedious.

To address these challenges, we have developed a domain-specific modeling language (DSML) for representing MDAO problems. This DSML provides an intuitive, high-level abstraction around the integration language of MDAO that shields users from the low-level details of manually creating low-level integration code. Instead of creating low-level integration code by hand, users can create graphical models of the integration. These models can then be checked for conformance to predefined constraints and used to automatically generate the low-level integration code. This approach minimizes the chance for errors in the low-level integration code, thus allowing experiments to be prototyped more quickly by letting MDAO users concentrate on their domain models rather than the integration code.

This paper describes our experiences designing such a DSML for OpenMDAO[4], a popular open-source MDAO framework. We include a description of our constraint framework and our prototype code generation capability. While our initial framework is domain-independent, meaning that it can be used to model the integration for models coming from any underlying domains, we also describe future extensions that are aimed at additive manufacturing.

The rest of this paper is organized as follows. Section 2 gives background information on OpenMDAO, DSMLs and model composition. Section 3 provides a running example that we use with our DSML. Section 4 describes the actual design and implementation of our DSML, constraint framework and prototype code generator. We survey related work in Section 5, provide future directions in Section 6 and conclude in Section 7.

2. Background

The ability to compose models is a desirable trait of any component framework. Model composition allows users to build large, complex system models by connecting smaller, individual models, often from disparate domains. The “component” models may express disparate viewpoints. The principal challenge in composing models lies in knowing what reasonably can be inferred from the models in composition. For example, a production scheduling model can be expressed as the composition of models of product demand, process plans, production equipment

types, and production equipment instances. However, the process plans might refer to equipment used in production in a way that makes it difficult to know which of the actual equipment types might be applied apply. Further, the models in composition might refer to what might appear to be the same thing, but in actuality may reflect different measurement contexts, or may be expressed in different units of measure.

However, composition of the smaller parts of the models can be a non-trivial task in MDAO problems. As the systems being modeling tend to be complex and contain models from various disciplines, the interfaces between them can be hard to define. Additionally, non-causal relationships can exist between models from different disciplines, which may not be supported by the underlying integration framework. The component scheduling strategy used when multiple components simultaneously have all inputs required to execute can lead to non-deterministic results. Due to these issues, manually creating an MDAO integration model without higher-level tool support can be error prone. Graphical modeling tools, like the one proposed in this paper, can help developers by providing an intuitive interface and automatically generating the lower-level implementation artifacts.

OpenMDAO[4] is an open-source platform for model composition, analysis and optimization of multidisciplinary systems. At its core, OpenMDAO is implemented as a Python framework that provides the integration layer connecting the different parts of a model of a multidisciplinary system. An OpenMDAO system model consists of mainly two logical parts: one part representing the actual system in terms of the relation between inputs and outputs, and another part representing how the analysis and/or optimization is going to be carried out.

There are four major concepts in an OpenMDAO model: *Component*, *Assembly*, *Driver* and *Workflow*. Component and Assembly are the entities which define the behavior of the system in terms of its input/output relationship. Driver and Workflow are the entities which together define the execution of the optimization or analysis problem. In concrete terms, each of these is implemented as a Python class that can be extended for customization by the user.

A component is the atomic building block of an OpenMDAO model. It contains inputs, outputs and the relationship between the two. OpenMDAO provides the component concept through a Python class (named "Component") that the user extends to implement this input/output relation. Specifically, the user implements a method named "execute" with the functionality of their custom component.

A driver provides a set of inputs to a component (or group of components), fetches their outputs, draws results and optionally repeats the process. Every driver is associated with a workflow, which is a set of components representing the components which should be executed by the associated driver. An assembly is a container for components, drivers and other assemblies. If an assembly contains more than one driver, one of them is designated as the top-level driver responsible for ordering the execution of the others.

The behavior of an assembly is similar to that of a component: an assembly has inputs and outputs, and can be contained in a driver's workflow. However, the execution of a component means executing its "execute" method, whereas the execution of an assembly executes every OpenMDAO entity (component, assembly and/or driver) in the workflow of its top-level driver. Thus, instead of an "execute" method, each assembly has a "configure" method in which the user selects the top-level driver of that assembly. Users can also instantiate other OpenMDAO objects and either associate them with the driver's workflow, connect their data ports or configure driver(s).

As with any component framework, OpenMDAO users must have a strong understanding of its concepts and their relationships in order to create, compose and analyze models. In addition to grasping the main concepts in the preceding paragraphs, users must also understand many lower-level details in order to create an integration model. However, many potential users are experts in their individual domains but lack either the technical background or time to learn the low-level details of the OpenMDAO component framework. Further, even with an understanding of these details, creating relatively low-level integration code is a tedious process. A simulation or analysis with unexpected results could be the result of either errors in the integration code or errors in the individual domain models; reducing the possibilities for integration errors is essential.

One way to reduce the possibilities for errors in the integration code is to raise the level of abstraction. This can be accomplished by *modeling the integration* itself and giving users an intuitive way of representing and interacting with such integration models. One way to do this is using domain-specific modeling languages (DSMLs).

DSMLs are becoming increasingly important as the tools and frameworks developers are working with are growing in complexity. DSMLs provide a layer of abstraction above the underlying implementation of the domain on a computational platform, which allows users to think about the problem they are trying to solve in terms of the

concepts from that domain. Thus, DSMLs can help reduce development time and cost[5] by providing higher-level concepts which can be automatically mapped to lower-level details. Combining a DSML with automated tools such as constraint checkers and code generators can produce a powerful tool-suite that allows domain-experts to increase their productivity and efficiency with related tools.

Having DSMLs coupled with code generators improves developer productivity because automatically generated code tends to have fewer errors as compared to hand-written code; this is especially true in rapid prototyping environments where the low-level code must be synchronized to changes in the higher-level models. Additionally, validators can be used before the code generation to ensure code is being generated from a valid model. Validators can be used to provide both syntactic and semantic correctness checks.

3. Running example

Our running example, which we model using our DSML in Section 4, is a based on an example included in the official OpenMDAO tutorials[6]. It consists of a simplified model of a car containing of three elements: a transmission, engine and chassis. The transmission element is responsible for transforming the engine output torque to torque at the wheels and for calculating the engine RPM. The engine element calculates (1) the torque value passed to the transmission, and (2) the current fuel burn rate. The chassis element provides the vehicle acceleration for the current torque value provided by the transmission.

Fig. 1 shows how this model can be represented with OpenMDAO. There are three atomic component elements: one for each of the transmission, engine and chassis. Each component reads its inputs and calculates its outputs independently. The three components are coupled together in an assembly named “Vehicle”. In order to use this assembly for analysis, its parent assembly also needs a driver. The driver provides the necessary values to the input ports of the assembly, which in turn are connected to the inputs of the internal components, as shown in the Fig. 1, via a concept known as a *Passthrough*. Passthroughs allow promoting the ports of internal objects as the ports of the object containing them. These inputs from the Driver are called *Parameters* and they act as design variables.

We wish to execute this model with one of two main goals: a *design of experiment* (DOE) or optimization. A DOE is used to observe the response of the system to a set of values given to inputs, while an optimization is used to find the optimal input parameters that minimize the amount of time for the car to accelerate from 0 mph to 60 mph. For a DOE type of problem, the outputs of the Vehicle assembly are captured in the driver via an interface called *Responses*, whereas for an optimization problem, an interface called *Objectives* might be used for a similar purpose. OpenMDAO provides several of the pre-implemented drivers to help designing both types of problems, and these drivers can be classified into two categories according to the kind of interface they provide – DOE or optimization.

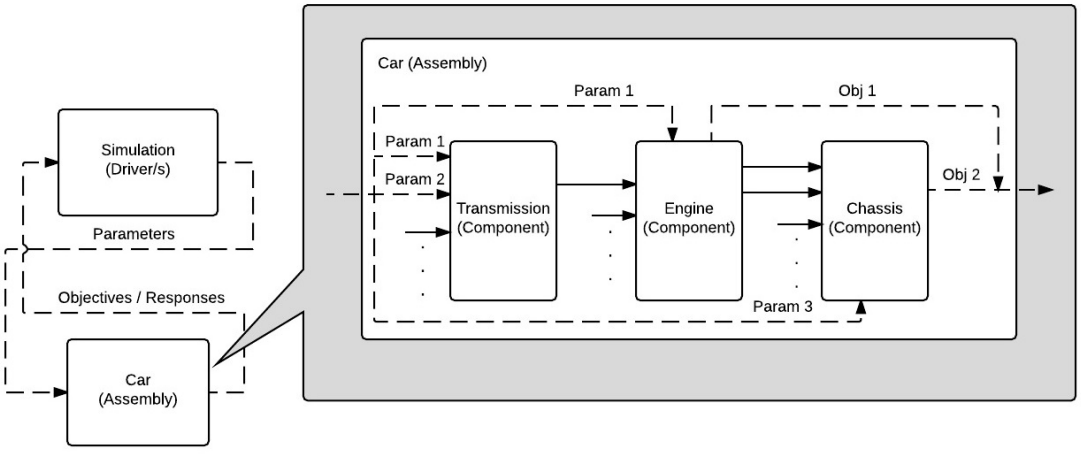

Fig. 1. Simplified model of a car in OpenMDAO.

A model (i.e. an assembly or a single component) can also be driven by an application-specific driver. Such custom drivers contain a customized implementation of the logic that decides what values should be given to the inputs of the model and what decisions should be taken according to the values of its outputs. In our example, a custom driver drives the Vehicle assembly to simulate its run from 0 mph to 60 mph. It calculates the time taken by the car to go from 0 mph to 60 mph by iteratively setting the values to some of the input variables of Vehicle assembly and fetching the value of output variables.

The custom driver sets values of current gear, velocity and throttle as the inputs of the Vehicle assembly. The assembly, for a given set of values of these three quantities produces two outputs, namely overspeed and acceleration. Depending upon the value of overspeed, the driver decides whether it should change the gear ratio provided to the Vehicle assembly for the next iteration, signifying the shift of the gear. Similarly, the driver updates the value of velocity depending upon the value of acceleration produced by the Vehicle assembly. This process is repeated in every single iteration of the simulation until the value of velocity being calculated reaches 60 mph.

The example presented here can also be seen as a DOE where the time taken by the car to accelerate from 0 mph to 60 mph is calculated for various values of spark angle. Thus, the example in addition to an application-specific driver also contains a DOE driver. The DOE driver can set the value of one or more inputs to the Vehicle assembly and can save the time taken to accelerate from 0 mph to 60 mph, calculated by the application-specific driver, for that particular set of values. In this example, we decided to change the value of spark angle in a given range. The values set to spark angle and corresponding values of time taken to accelerate from 0 mph to 60 mph are stored in the DOE driver and are accessible after the execution of the DOE driver's iterations complete.

4. Implementation overview

This section describes the implementation of our model composition DSML, which we have named MOCA (MOdel Composition and Analysis). The abstract syntax is specified using a metamodel, which defines the domain concepts, along with their attributes and relationships. The metamodel defines the structure of instance models that are said to conform to that metamodel. Instance models use a concrete syntax of graphical icons, which provide an intuitive syntax for users.

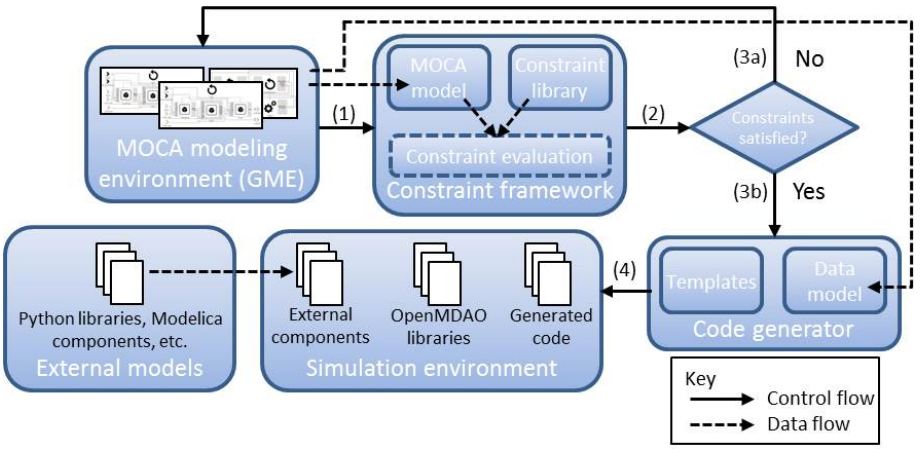

Fig. 2. Overview of our tools and workflow.

The modeling tool that we used is the Generic Modeling Language (GME[5]). GME provides a meta-programmable environment that allows a metamodel to be created and then used to configure GME for creating instance models of that metamodel. Additionally, we used the extension mechanisms of GME to implement (1) a constraint framework that ensures models satisfy constraints that are not enforceable through syntax alone (for instance, all elements have unique names), and (2) a code generator that can generate the underlying OpenMDAO integration code from a MOCA instance model.

4.1 MOCA Metamodel

A metamodel is a formal representation of a given domain's entities and the relationships between those entities. In GME, metamodels are described using a meta-language that is very similar to UML class diagrams[7]. Class diagrams use *classes* to represent the entities and *associations* to represent the relationships between those entities. A class can contain *attributes* that hold information describing the corresponding domain entity.

Fig. 3 shows a simplified version of the MOCA metamodel using the UML Class Diagram notation. Our metamodel is closely coupled to the OpenMDAO component model, and so many of the OpenMDAO concepts are mapped directly to corresponding classes in the metamodel. Fig. 3 shows a top-level class named Assembly, which can contain Assemblies, Components and Drivers (also called 'Drivables') as signified by the solid lines ending in a diamond shape. Drivables are associated (indicated by a dashed line ending in an arrow) with a Driver, representing the fact that a Driver drives them. Every Drivable can contain DataPorts that allow them to communicate with other DataPorts. In addition to these DataPorts, Drivers can contain DriverInterfaces. DataPorts can either be connected to other DataPorts or DriverInterfaces, whereas DriverInterfaces can only be connected to DataPorts.

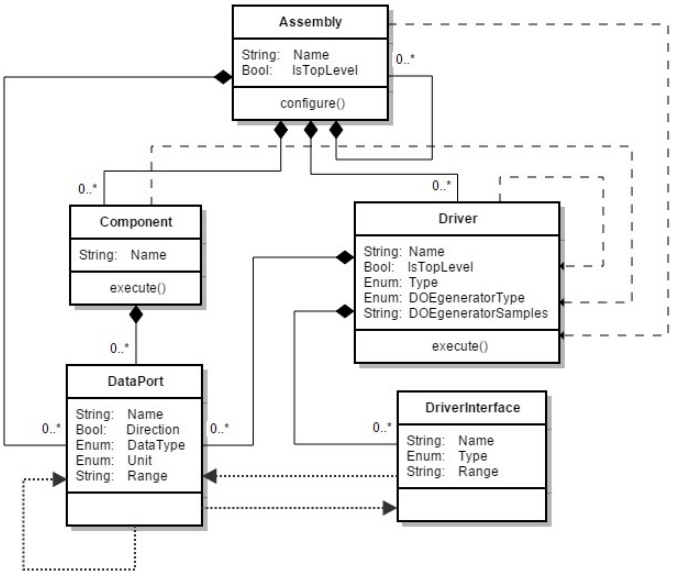

Fig. 3. Simplified metamodel of MOCA.

In addition to the containment relationships, some of the classes have typed attributes and methods. For instance, the DataPort class has attributes for its Name, Direction, DataType, Unit and Range. Instances of the DataPort class can then have different values for these attributes. Attributes which can have only Boolean values (e.g. the direction of a DataPort) are of Boolean type; attributes which can be assigned one value from a finite set of values (e.g. the DataType of a DataPort) are of Enum type, and attributes which can be assigned with a value expressed in text format (e.g. Unit of a DataPort) are of Field type.

4.2 Constraint Framework

The simplified metamodel described above allows the context-free rules describing the domain elements and their relationships to be expressed. However, our MOCA language also contains constraints that are not context-free. Such constraints include, for instance, that all model elements must have unique names. Because MOCA instance models are eventually translated into lower-level (Python) code that provides the integration layer for simulations/optimizations, detecting errors as early as possible before they can propagate into the simulation/optimization is crucial.

In order to check these additional types of constraints, our MOCA language uses an additional *constraint framework* layer. Having a constraint framework to provide additional model validation augments the notion of having a DSML as a development tool tailored for a particular domain. With large OpenMDAO models, validating models can greatly reduce the number of runtime errors. One reason for this is that in an interpreted language like Python, it is beneficial to have early detection of the potential causes of run-time errors, because runtime errors can remain dormant until even later stages of execution.

Concretely, we implemented our constraint framework using GME's extension framework[5]. The overall process works in the following way. First, each constraint is encoded as a C# method. Next, these methods are all compiled into a dynamically linked library, which is a binary executable code that can be loaded into the modeling tool on demand. When the constraint framework is invoked, it loads both the binary representation of the MOCA model (made available through the GME programmatic API) and the dynamically linked library containing the constraint rules. The framework then iterates over the elements in the MOCA model. For each element on which a constraint rule is defined, the corresponding method from the constraint library is invoked and the constraint is checked programmatically. This is labeled as step (1) in Fig. 2.

Some of the constraints that are implemented in the constraint checker include –

- Names of the entities in a model must be unique.
- An assembly should contain one and only one top-level driver.
- Ranges associated with connected ports must be compatible.
  (I.e. an output port cannot be connected to an input port that has a narrower range.)
- A non-numerical valued port should not have a range associated with it.
- Data ports that connected to each other should be of compatible data type.

4.3 Code Generator

MOCA uses a code generator to translate instance models into Python code targeting the OpenMDAO integration framework. By first ensuring that the instance models conform to the domain-rules described by the MOCA metamodel, and then using the constraint framework described above to check additional constraints on the instance models, our framework eliminates many potential sources for error. Automated code generation provides another way to minimize errors by automatically generating the corresponding Python code.

Besides removing potential errors, automated code generation offers at least two additional benefits. First, it decreases the amount of time required to change the corresponding Python integration code if changes are made in the model. Second, it provides the ability to target additional integration frameworks. The integration code for additional frameworks can be supported by writing an additional code generator. This allows the benefits provided by the DSML and constraint framework to be leveraged.

Similarly to the constraint framework, our code generator is also implemented using GME's extension framework and works as follows. First, the binary representation of a MOCA instance model is obtained using the GME's programmatic API. The model is then traversed, and as it is traversed, a *data-model* is populated. This data model can be thought of as an intermediate representation of the instance model that contains only the information necessary to generate code. Once this data model is fully populated, it is used by a *template processor*[8] (we use the Microsoft T4 Text Templates) to populate a set of templates. These templates are a mixture of verbatim text and variables whose values are obtained from the elements of the data model. For example, an assembly element in a MOCA instance model would be used to populate a template representing a Python class, and the name of the assembly element in the model would be inserted as the value of a variable in this template. In this way, the Python classes and infrastructure code corresponding to an instance of a MOCA model are generated. This part of the workflow is labeled as steps (2) through (4) in Fig. 2.

The generated code contains the definitions of OpenMDAO entities, their instances and connections between them. In most cases, the code for an assembly does not need manual modification because it does not contain a user-defined "execute" method like components do. For custom drivers and components, the "execute" methods that define the relationship between the inputs and outputs need to be added manually to the generated code. If the model uses pre-implemented OpenMDAO components and drivers (such as the CONMIN and SLSQP optimization drivers

shipped with the OpenMDAO stack)[6], the MOCA code generator can generate the code that does not require manual modification in order to execute the OpenMDAO model.

4.4 Evaluation

Fig. 4 shows two separate portions of the GME model for the running example (Section 3). The right hand side of Fig. 4 is analogous to Fig. 1 and is the 'simulation' part of Fig. 1. It is modeled using two drivers named 'DOEDriver' and 'SimAcceleration'. 'Vehicle' is an assembly containing three component blocks for the transmission, engine and chassis, which are shown in the callout box on the left. A driver interface of 'DOEDriver' is connected to an input of 'Vehicle' assembly which sets different values for the spark angle. The 'SimAcceleration' driver simulates the run of 'Vehicle' assembly from 0 mph to 60 mph as discussed in Section 3. This driver assigns values to the inputs and reads outputs from 'Vehicle' assembly via its interfaces. The 'SimAcceleration' driver produces the time taken for one run as its own output, and this output is read by 'DOEDriver'.

The development of a model like this starts with the creation of the components and assemblies. In this particular example, the 'Vehicle' assembly and the three components it contains were created first. In order to connect the components and assemblies, inputs, outputs and passthroughs were created. In addition to a name attribute, these ports can have other attributes associated with them, such as a default value and data type. Moreover, each assembly needs a top-level driver, as discussed in Section 2, and so a driver for 'Vehicle' assembly (with its 'IsToplevel' attribute set to true) was created. A driver also needs to be associated with the objects it is required to drive, i.e. the three components in 'Vehicle'. Finally, this assembly can be a part of an outer assembly having two more drivers, one for simulating the car and another for setting the value of its spark angle. The creation of the model is completed after connecting these two drivers to the 'Vehicle' assembly via their interfaces. This model creation step is shown in the upper-left hand portion of Fig. 2 labeled, "MOCA modeling environment (GME)."

Before generating the code for the model, the constraint framework is invoked (steps (2) and (3) in Fig. 2) to further validate various aspects of the model, such as the type-correctness of data connections, the absence of algebraic loops and compatibility between connected ports on the basis of range and units of the physical quantity they represent. For example, in the model in Fig. 3, if the 'overspeed' port of the Engine component (which is a 'Boolean' type) gets connected to the 'engine_mass' port of the Chassis Component (which is a 'Float' type), the constraint framework would raise an error indicating the two ports are incompatible with each other.

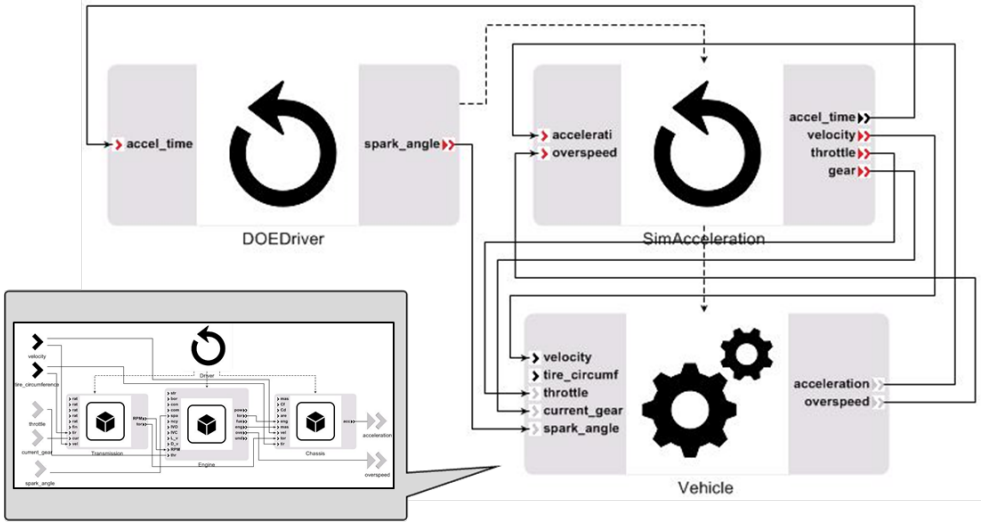

Fig. 4. Screenshots of the GME model for the running example.

Running the code generator (step (4) in Fig. 2) on the model for our running example produces the Python integration code, which contains approximately 140 lines of code. This generated code uses the OpenMDAO libraries and can be thought of as the "skeleton" code for an OpenMDAO integration model. It contains Python class definitions for the interfaces between various components such as 'DOEDriver', 'SimAcceleration' driver (i.e. application-specific driver), Vehicle assembly and each of the internal components. The implementation for the 'execute' methods (which include model-specific relationships between inputs and outputs) of the components and drivers is then inserted manually by the user.

5. Related work

In paper[9], the authors present a method to create executable system specifications and analysis models to simulate discrete and continuous system behavior. The authors use ModelicaML, which is a profile extension of the UML meta-model, to integrate continuous-time behavior descriptions in Modelica with discrete-time system descriptions in SysML. These models are checked by a custom validator for constraints and possible inconsistencies, and Modelica code is generated from them. The generated Modelica model can then be simulated or analyzed using various tools. The validation of the ModelicaML model itself is not standards based, and the authors point a number of discrepancies between Modelica and ModelicaML that arise due to the nature of these modelling languages. These arise primarily from the event-based UML activity diagrams used in ModelicaML. For instance, transition between states is instantaneous in ModelicaML. We believe that our approach is more standards-based. We do not impose discrete and event based behavior on OpenMDAO model, allowing for a truer representation of system behavior. There are similar efforts in using SysML and Modelica.

Optimica[10] is an extension of Modelica supporting dynamic optimization. The extension allows users to specify cost functions, constraints, optimization interval etc. A custom compiler is developed that transcribes the optimization problem and generates AMPL[11] code, which can be sent to an AMPL solver. AMPL is a proprietary language, and does not support object-oriented modelling or engineering units. Open MDAO is free and open source, and provides a library of solvers and optimizers.

Papers[12,13] are other notable works in model composition (not specifically directed towards optimization). Paper[12] provides a classification of model composition approaches. The author discusses different model composability approaches in terms of modelling formalisms. The paper discusses ways of composing optimization models as a control layer over other system models. Paper[13] provides a structural model composition approach for computer aided V&V of automation systems. The method is implemented in the form of graph templates supported by standardized interfaces and protocols and model transformation techniques.

6. Future work

Even with our framework, there are still challenges with integrating analyses across multiple disciplines. One of these is dealing with analytic dependency loops i.e. when design decisions taken after carrying out an analysis conflict with that of another analysis[14]. Identifying and suggesting algorithmic approaches to solve such dependency loops on a higher level of abstraction than our MOCA language provides to developers would be beneficial.

Causal relationships between inputs and outputs of components in a component-based framework can sometimes limit the ways to approach to a problem. For example, a complex multidisciplinary problem having several inter-related processes with multiple interacting variables might be difficult to fit into a framework where relationships between the variables have to be strictly causal. In such cases, a higher-level modeling tool can help developers to describe non-causal connections between variables, and then such non-causal models of the system might be used to determine the causal model for a specific analysis with some variables marked as design parameters and some of them as optimization objectives. Providing this feature also applies the "don't repeat yourself" principle to the development process, as the same non-causal model can also be used for other purposes (like analysis or optimizations) with a completely different set of parameters and objectives.

A low-barrier to entry is one of the advantages that DSMLs provide over generic modeling languages. This aspect of DSMLs can be more useful for developers who are not very accustomed to programming environments or

frameworks. Solutions to optimization problems in areas like additive manufacturing have to consider many interacting disciplines, such as thermodynamics, optics and fluid mechanics[1], and modeling the problems in these specific areas is difficult using generic modeling languages. What would be useful is a library of built-in components targeted towards special fields, like additive manufacturing, which developers could "drag-and-drop" into models and configure by setting their attributes.

7. Conclusions

Complex engineering design problems necessitate the execution of complex analysis processes, where the analysis cuts across physical domains and engineering disciplines. The problem is especially acute in the case of advanced manufacturing, where the desired product emerges from the interactions of multiple physical processes. To analyze and control such interacting, heterogeneous processes a complex model integration task needs to be solved. In other words, systems engineering techniques should be applied to the problem.

In this paper we have shown how a domain-specific modeling language can be used to address this problem. The language is based on the OpenMDAO framework that provides the underlying execution platform for integrating (executable) models. The language offers higher level abstractions for model composition and the accurate control of the execution of analysis tasks using those models. From these models code that configures the OpenMDAO framework is generated. Additionally, the explicit, domain-specific models support the early detection of potential model composition problems; well before the actual computational analysis is executed. The language can also be extended with a sophisticated type system, to support the verification of the model composition. The usability of the tool has been illustrated using an example.

In conclusion, the efficiency of the engineering design and analysis processes can be clearly improved by tools that allow engineers to compose their analysis workflows. Currently, a big problem is that engineers often have to fight with the tools, translate data from one tool to another tool, and create complex analysis scripts – activities that can clearly be done with 'meta-level' tools that orchestrate the execution of domain-specific models and apply systems engineering techniques to the process itself. However, the technology to support such engineering processes is the subject of active research.

References

1. Verhaeghe F, Craeghs T, Heulens J, Pandelaers L. A pragmatic model for selective laser melting with evaporation, In: Acta Materialia 2009; **57**:6006-6012.
2. Del Rosario R. An MDAO Perspective. National Science Foundation Workshop on The Future of Multidisciplinary Design and Optimization: Advancing the Design of Complex Systems 2010.
3. Sztipanovits S, Bapty T, Neema S, Howard L, Jackson E. OpenMETA: A Model- and Component-Based Design Tool Chain for Cyber Physical Systems. In: Lecture Notes in Computer Science 2014;**8415**:235-248.
4. Moore K. Calculation of Sensitivity Derivatives in an MDAO Framework. 14th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference 2012.
5. Ledeczi A, Bakay A, Maroti M, Volgyesi P, Nordstrom G, Sprinkle J, Karsai G. Composing Domain-Specific Design Environments. IEEE Computer 2001;34:44-51.
6. OpenMDAO official website. http://openmdao.org/.
7. Rumbaugh J, Jacobson I, Booch G. The Unified Modeling Language Reference Manual, 2nd ed. Addison-Wesley Professional; 2004.
8. Niemeyer P, Leuck D. Learning Java, 4th ed. O'Reilly Media; 2013.
9. Schamai W, Fritzson P, Paredis C, Pop A. Towards unified system modeling and simulation with Modelicaml: Modeling of executable behaviour using graphical notations. In: Proceedings of the 7th Modelica Conference 2009;612-621.
10. Akesson J. Optimica – an extension of modelica supporting dynamic optimization. In: Proceedings of the 6th International Modelica Conference 2008.
11. Fourer R, Gay D, Kernighan B. AMPL: A Modeling Language for Mathematical Programming. 2nd ed. Cengage Learning; 2002.
12. Sarjoughian H. Model Composability. Proceedings of the Winter Simulation Conference (WSC) 2006;149-158.
13. Vyatkin V, Hanisch H, Pang C, Yang C. Closed-loop modeling in future automation system engineering and validation. IEEE Transactions on Systems, Man, and Cybernetics Part C 2009;17-28.
14. Ruchkin I, Schmerl B, Garlan D. Analytic Dependency Loops in Architectural Models of Cyber-Physical Systems. 8th International Workshop on Model-based Architecturing of Cyber-Physical and Embedded Systems 2015.