Formal United System Engineering Development (FUSED)
Scientific and Technical Reports:
Final Report

July 29, 2011

CONTRACTOR:
Name of Contractor: Adventium Enterprises, LLC
Principal Investigator: Dr. Mark Boddy
Business Address: 111 3rd Ave. S, Suite 100, Minneapolis, MN, 55401
Phone Number: 651.295.7126

TABLE OF CONTENTS

LIST OF FIGURES

1.0 SUMMARY

The goal of this program was to develop a new extensible system representation metal-language, which we named the Formal United System Engineering Development (FUSED) language, to specify complex relationships between models written in multiple languages at multiple levels of abstraction and used by developers from many disciplines for many purposes during a concurrent development process.

Our approach was to leverage the enormous investment in the great variety of existing domain-specific modeling environments by creating a language that would complement these existing ones. FUSED is a language to compose collections of models written in these various languages in a way that they can be used in a synergistic, verifiable, multi-disciplinary, model-based development activity. Existing languages, tools, and model libraries are used within the various domain-specific modeling environments to perform the specialized engineering activities for which those environments were created. FUSED operates at the requirements and system engineering levels, where complex relationships between models of varying kinds and at varying levels of abstraction must all be used collaboratively to perform overall system development.

We developed a preliminary FUSED language specification and supporting toolset. These were successfully applied to a number of demonstration development tasks based on a small UAV development scenario (a UAV that was based on an actual Lockheed-Martin product line). Collectively these tasks demonstrated the composition of 10 different types of models (requirements, abstract and mixed-fidelity equational, solid/geometric, aerodynamic, dynamical, avionics, trade space, verification, design optimization) in 9 different modeling languages (SysML, Excel, Creo/ProE, AVL, Modelica, AADL, ATSV, SMTLib, MiniZinc). The compositions, when executed, performed activities such as moving analysis and simulation data between models with strong type checking, verifying consistency conditions across multiple models, and performing collaborative multi-disciplinary design optimization over models.

2.0 INTRODUCTION

FUSED is a language used to specify compositions of design engineering models written in a variety of other modeling languages (e.g. SysML for requirements, Creo/ProE for solid/geometric models, Modelica for dynamical systems models, Excel for simple one-off engineering models). A FUSED composition identifies a set of models and specifies relationships between those models (e.g. parametric dependencies, part/whole assembly, inter-model consistency conditions). A FUSED composition specification is a model just like any other and can be mixed-and-matched to form more complex FUSED compositions.

FUSED allows engineers to use proven and accepted modeling languages and environments within established engineering domains (e.g. Cre/ProEo users can use Creo/ProE and their legacy models, Modelica users can use Modelica and their legacy models). We use an extensible language approach to add those few features needed to cleanly interface with the system engineering capabilities provided by the FUSED infrastructure (e.g. publish or subscribe model elements to be obtained from or provided to the system engineer or other models, specify type qualifier information for which no capability exists in the language currently). This is done in a way that is a natural fit with the look-and-feel of the existing modeling environment (e.g. new clause to declare uncertainty in standard Modelica parameter declarations, SysML extensions appear as new profiles). The primary users of the FUSED language proper are requirements and systems engineers, for whom it provides novel capabilities.

In addition to the obligatory editor of graphical FUSED specifications, FUSED tooling provides the following capabilities:

- Automatically transfer elements between models with strong type checking (support a single source of truth), including complex types (e.g. constraints, function signatures, abstractions of the model structure itself) and type qualifiers (e.g. units, frame of reference, uncertainties). The ability to deal with complex model types enables many kinds of compositions not possible with current simulation workflow engines, e.g. compose models with specialized consistency verification modeling environments.

- Automatically invoke analyses or simulations specified for a composition of models. The possible combinations reflect the capabilities of the selected modeling languages and tools. For example, a Creo/ProE model could be analyzed to obtain mass properties while a Modelica model could be analyzed to obtain a simulation trajectory. Engineers typically think of these as simulations, but we have paid particular attention to the ability to exchange analysis and formal verification results as well (e.g. don't simulate a real-time schedule and deal with a set of test traces, do a schedulability analysis and use guaranteed bounds).
- Automatically invoke verifications and design tools on a composition of models. Models in a composition are often component models (e.g. a wheel), but they are also often verification or design aid models (e.g. a model to obtain a Pareto frontier for another model, a model to automatically optimize certain design configuration parameters of another model).
- Track dependencies and change propagation between model elements and models. This currently includes conventional make/build change tracking, with a capability to manage and cache results across multiple configurations of a model. We are working towards smarter and finer-grained ripple effects analysis (e.g. determine that a change is not significant enough to trigger re-analysis based on type qualifier information such as uncertainty or sensitivity) within the context of a highly concurrent and collaborative development process.
- Support for configurable models and requirements and design evolution. Features are included to explicitly support definition and use of configurable models, integrate trade space visualization and exploration environments, and integrate specialized design optimizers at various points in the process.
- Support for multiple kinds of model composition. We currently support compositions that allow models to use as inputs values that are obtained from other models as determined by a FUSED model composition specification (what we call publish/subscribe). We plan to add some support for part/whole compositions, both within matching modeling environments (e.g. compose multi-model components having matching domain-specific tools, such as composing solid component models within

Creo/ProE while simultaneously composing dynamical component models within Modelica); and composing models using diverse but synergistic modeling environments (e.g. co-simulation, such as composing a Modelica simulation with a FlightGear environment simulation).

FUSED is extensible. It is designed so that support for selected modeling languages and tools, and selected types of data and meta-data to be made visible and manageable at the multi-model composition level, can be added. This is somewhat analogous to creating a plug-in to Eclipse, except that FUSED is build using advanced language extension technologies (from the University of Minnesota extensible languages group) so that much of this can be done at a more concise and abstract level than writing Java code. However, it is still a programming-like exercise, and extending FUSED to support a new modeling language or a new set of types would be done by engineers trained for those particular tasks (e.g. a support group within an organization, a vendor of a tool, a small company in that business). Eventually, we would hope a set of plug-ins for common languages and tools would be available and could be installed by users, analogous to the way it can be done with Eclipse today.

3.0 METHODS, ASSUMPTIONS, AND PROCEDURES

3.1 Assumptions and Approach

We assumed that it is not currently possible to develop a unified theory and semantics for everything. We assumed it was not practical to layer-over existing languages with some new meta-language and still replicate all the power and detail of the myriad of domain-specific languages that exist. This is the basis of our approach to allow domain-specific users to keep using modeling environments and methods they are already familiar with and focus on a meta-language that adds new capabilities to compose multiple models from different domains.

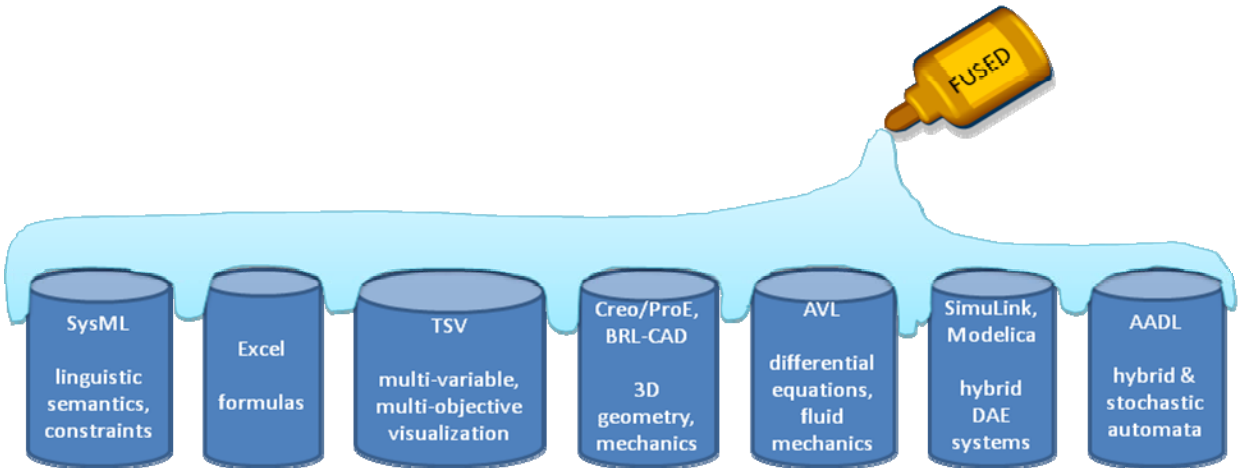

Figure 1: FUSED Glues Together Models in Existing Languages

One of our conjectures was that advanced formal language notations and tools and extensible language technologies would provide a powerful way to specify and implement our new FUSED language and its supporting tools. We selected the Silver higher-order attribute grammar system from the University of Minnesota to carry out our work. We believed then (and still do) that this basis in formal programming languages and type and logic theory provides greater power and assurance than alternative approaches based on, say, UML or semantic web concepts.
We also felt that concurrent, collaborative engineering raised challenges that had to be supported. FUSED is more than a language for specifying static compositions of models. It includes semantics, and the toolset includes support for, projects in which hundreds of engineers may be concurrently modifying thousands of models.

3.2 FUSED Language and Capabilities

FUSED Language and Capabilities were summarized in the **INTRODUCTION** section.

3.3 FUSED User Experience

We currently do our demos within the Eclipse environment. This offers the usual conveniences, but also brings with it some inconveniences. The only real dependency we currently have on

Eclipse is that our graphical FUSED editor is built using Eclipse GEMS, otherwise vehicle engineers could do their work using another IDE or none at all.

Engineers within a specific domain create an Eclipse project of the desired type, e.g. create a Modelica project. An inconvenience is that most of the tools used in vehicle engineering lack Eclipse plug-ins, so the special features Eclipse provides when plug-ins are available is lost (unless someone develops these plug-ins, something that could be done as part of creating a FUSED extension for a particular language -- but something we haven't been doing for our demos). The users see what they expect from their chosen flavor of project, but with a few extensions. They see a few extensions available for the modeling language (e.g. ways to subscribe to model elements they want to be provided to them, ways to declare things that cannot be declared in the standard language such as uncertainties or frames-of-reference). There is also added build functionality to support the FUSED capabilities listed above. If the engineer is developing a configurable model, then they may need to add a few special specifications or follow a few special guidelines if different model configurations make use of different model files in a way the build cannot automatically deduce.

FUSED composition specifications are created in a new project that allows FUSED graphical specifications to be developed using our FUSED graphical editing tool. These can be compiled and executed to carry out some specified activity (e.g. perform consistency checks across the set of models, do some sort of overall analysis of the composed set of models, run a design optimization over the composition, perform sampling to estimate the Pareto frontier and explore that).

An overall project isn't just one engineer carrying out one sequential set of engineering activities. We are concerned about hundreds of engineers concurrently making changes to thousands of models across multiple engineering disciplines at multiple levels of abstraction. We are trying to put in capabilities and features to help people stand on each other's shoulders instead of each other's toes, but successful collaborative engineering is a matter of cultural and management norms and processes that effectively use the collaboration features. These FUSED

features are obviously just a subset of the collaboration support provided by an overall common repository such as VehicleForge, and they need to be thoughtfully integrated into that.

3.4 FUSED Structure and Operation

This section gives a high-level overview of how FUSED is built and operates. We overview how languages and types (ontologies) are specified and how tools that operate on them are generated. We overview how compositions of models are specified and how the tooling that automatically does things with compositions works.

3.5 Specifying Model Languages and Types

We use the Silver higher-order attribute grammar language and tools to specify and implement tools that understand and extend the various modeling languages of interest. At a very high level, Silver can be thought of as a scanner/parser generator on steroids (it has the power of a full functional programming language). The library of Silver specifications we have written so far could be divided, at a high level, into two groups: concrete grammars and abstract types.

3.6 Concrete Grammars

There are several concrete grammar specifications for various modeling languages. For example, we took the standard Modelica grammar, tweaked it into the exact syntax used by Silver, and added attributes so that we can generate a tool that can parse and perform some semantic analysis on Modelica files. There are a couple of additional things we do beyond traditional scanning/parsing.

Silver has special support for defining and implementing sets of extensions to existing languages. It is easy to write add-ons that extend existing rules of the grammar, and it is easy to say how these can be implemented by expanding (“forwarding” is the buzzword in the language extension community) these into the host language (e.g. in our interval uncertainty demo, a clause

specifying an interval uncertainty for a parameter causes the parameter declaration to be split into two declarations of a min and max parameter, and all statements that use that parameter get split into two versions). The tool that is generated from these specifications will read a string in the extended language and then write a string in the standard language. Note that Silver specifications can thus include languages whose strings are to be written as well as languages whose strings are to be parsed.

We often write specifications that just extract data. A modeling toolset uses many more file formats than just the language written by users; there are typically a variety of intermediate and analysis result file formats. We also generate parsers for some of these so that we can obtain the results of various operations performed on models. For example, we generated tools that can parse Creo/ProE mass properties analysis files and AVL aerodynamic stability derivatives files.

3.7 Abstract Types

Our library of Silver specifications also includes definitions of the types of model elements the FUSED infrastructure knows how to deal with. These are types of data of interest to systems engineers, or types of data that need to be exchanged or compared across different kinds of modeling environments. This is our ontology, specified in a format that allows us to use Silver to generate various tools that can operate on instances of these types.

Semantically and structurally, we are coming at this from a formal languages perspective. A type of model element can have a structure as complex as Silver extending scanning and parsing technology can handle (easily as complex as any current programming language), and its semantics can be similarly as complex. For example, we have a constraint type (inequalities), we have a typed object graph type that captures an abstraction of the overall static structure of a model, we have bits of a function signature type (so one model can call another as a function during some analysis or evaluation activity).

Of course, that belies all the hard work of actually formally defining semantics for each particular type. Pragmatically, we do what language people do today -- it is a matter of degree. For a chosen modeling language, we have the semantics to work with that are provided by that community. The modeling language communities have to bear the brunt of clearly defining and formalizing their semantics as much as they can be convinced to do so. For our ontology types, which cross domains, we have to come up with a semantics common across all the languages that might potentially use that type/concept – again, we can do no better than those language communities have done. The semantics of any particular ontology type comes from an abstract semantics space that is common across all the modeling languages in which that type has any meaning, and arguments need to be made that the appropriate abstraction relations hold within each modeling environment.

We currently do not impose much of any structure on our collection of ontology types. We have multiple roots, and multiple inheritance is supported. We don't pick any particular model for parametric typing, that falls within the scope of what our approach has the power to specify – different modeling languages have different parametric typing systems, and we can support that. There is a lot of potential power and flexibility in our approach, but of course with that comes potential complexity. Effective ways to structure an overall ontology for this purpose is an area of research.

One thing we do have is a concept we call type qualifiers. System engineers can add-on typing information that cannot be specified in this or that particular modeling language. We currently have type qualifiers for units, frame-of-reference, and interval and stochastic uncertainties.

Getting back to nuts and bolts, we use Silver to generate tools that can extract model elements of the various types from files written in supported modeling languages, convert them to a canonical internal representation (which inside the tool takes the form of a higher-order attributed abstract parse tree), and convert from a canonical representation to any other language representation in which elements of that type make any sense.

One concrete representation that we have for every ontology type is an XML representation that we have defined. Any collection of model elements in any hierarchical namespace structure can be written to a file in this format.

We also build a set of basic operations on elements of our ontology types. These are things like extracting a subset of elements from a collection, composing elements to form a new collection, adding type qualifiers, checking simple properties for an element, renaming, etc. These are the sorts of mundane things a system engineer may need to specify in a FUSED composition in order to get a particular set of models pasted together.

3.8 Doing Things with Compositions of Models

The FUSED composition language currently exists as a graphical language. There are an editor and a compiler that are implemented using Eclipse GEMS, which makes it much easier to develop and change than if it were implemented in Java (say). From the beginning we've taken a need-driven approach to language design – we identified the capabilities we felt were needed, then identified how to go about providing them, and finally identified a high-level syntax in which to specify them. The language continues to evolve.

The compiler generates ant build scripts. Executing a composition means executing a target in one of these ant build scripts. The overall execution is actually a hierarchy of build scripts that call each other. These scripts can be roughly divided into two kinds, those that are generated entirely by the compiler for a specified composition, and builds for a particular model developed in a particular modeling environment (which are the leaf builds in a tree of builds invoked for a particular purpose by the system engineer).

A part of the process for developing FUSED extensions for a particular modeling environment is the development of a template build script for that kind of modeling environment. The build templates have different targets (operations) for the different kinds of things an engineer might want to do with a model and the kinds of things the associated toolset is capable of. For

example, the Creo/ProE build template has targets to publish a mass properties analysis and publish a typed object graph abstraction of a model, while the Modelica build template has a target to produce a simulation trajectory.

Every build template accepts as an input parameter a path to an XML file containing a collection of model elements in the FUSED common format. A build template may return a path to an XML file containing a collection of model elements. The build template invokes the FUSED operations and the local modeling tool invocations necessary to do all the model element extractions and conversions, and all the extension-expanding pre-processing and output-generating post-processing, that are needed to accomplish the desired task for the desired configuration of that model.

We definitely encourage component developers to produce parametric or configurable models. Sometimes the build scripts need to automate some of this, e.g. know which subsets of files need to be used in which configurations. There may also be a need to restructure the inputs. For these reasons, the build templates themselves need to subscribe or publish certain values. This is still a little awkward now, model developers sometimes need to follow certain guidelines or hand-edit pieces of the build template. We're working on a concept of "FUSED wrappers" that makes this more concisely specified and automated and allows model/component developers to control the interfaces presented to the system engineer.

The FUSED compiler proper generates ant build scripts that invoke other ant build scripts. It passes paths to collections of FUSED elements in canonical representation between model builds, performing FUSED operations on these collections as specified to get the various models to talk to each other.

The build scripts currently do the usual dependency tracking and change propagation, using the typical coarse-grained, semantics-free notions of file time-stamping or differencing. One of the things we're working towards is leveraging semantic and meta-data available to the tools to do this in a smarter way. For example, if a solid model changes but a new mass properties analysis

says that various values have changed relatively little compared with the uncertainty meta-data associated with those values, then don't bother updating all the analysis results of all the other models that depend on that value. This is a research area.

Build scripts can receive as parameters a path to another build script. This is done, for example, in our trade space exploration demos, where the trade space builder repeatedly invokes the build of a parameter model to sample the design space.

A model target may be invoked for a particular configuration of a model and with a particular set of parameters for that analysis. For example, when requesting an aerodynamic analysis of an AVL model, the user needs to specify vehicle configuration parameters (e.g. choice of wing) and needs to specify a set of constraints that determine the "trim point" at which analysis is to be performed. Some of our build scripts can automatically cache analysis results, using the configuration and analysis parameters as a cache tag. This is a demo capability right now. To make this useful in practice, we'd have to add typical cache management capabilities. We'd also have to add somewhat novel ways to specify "close enough" semantics when determining a cache hit, e.g. "a request for analysis with a mass of 3.17 is close enough to an earlier analysis that used a mass of 3.12."

Building on CCM and dependency tracking and change management technologies, adding smart ripple effects analysis and change propagation, and providing a capability to do results caching, are features intended to help support concurrent, collaborative engineering. But much more thought needs to go into exactly how these would be used in conjunction with other VehicleForge collaboration features in the context of a crowd-source acquisition process.

4. RESULTS AND DISCUSSIONS

Our demonstrations were based around a development scenario for a small UAV. This scenario was based on real-world experience gathered from the Lockheed-Martin Desert Hawk, illustrated

in the figure. This is a small, reusable UAV whose parts can be carried in a backpack. It can be quickly assembled and hand-launched to perform local tactical surveillance missions.

Figure 2: Lockheed-Martin Desert Hawk UAV is a Real-World Basis for Demos

In our overall demonstration scenario we developed 10 different models, written in 9 different languages, at different levels of abstractions. These were used in various compositions as we demonstrated how FUSED and its tools could be used to perform selected system engineering tasks with greater automation and assurance. The following figure illustrates the complete set of models.

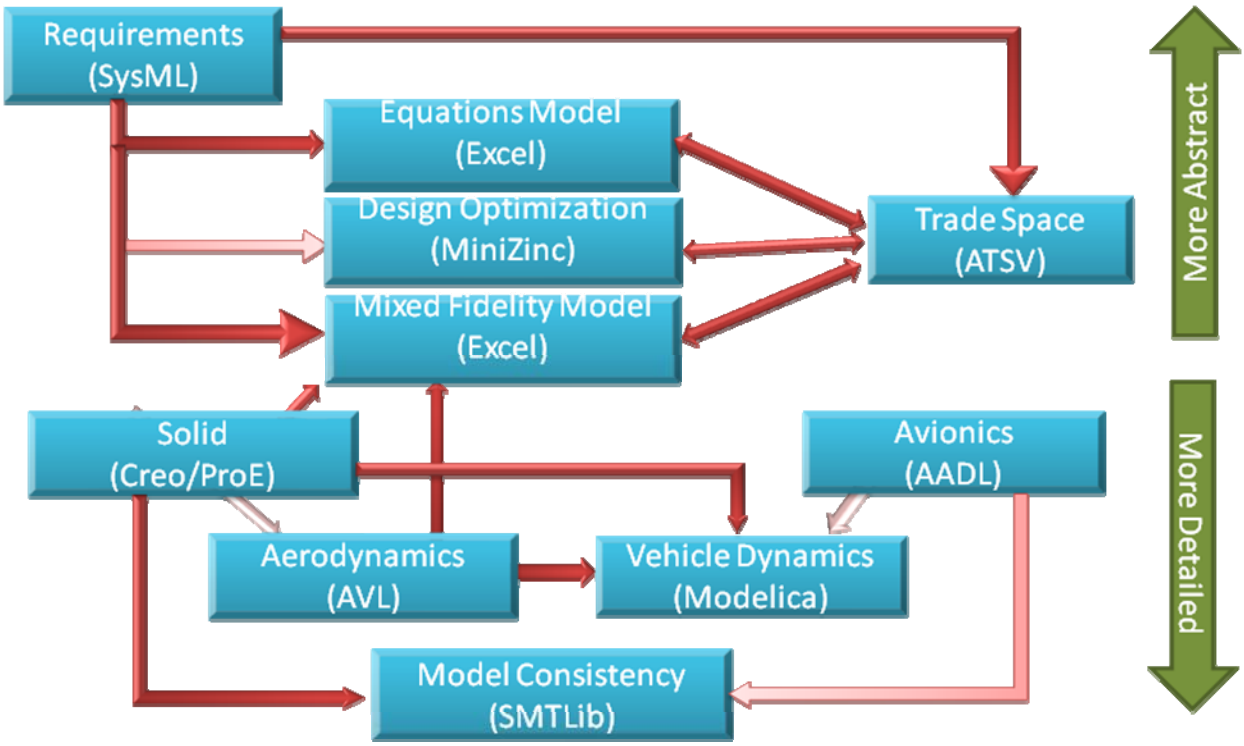

Figure 3: Demo Includes 10 Engineering Models in 9 Different Languages

The following sections describe the FUSED compositions we developed for the various demonstration tasks.

4.1 Trade Space Exploration of Abstract Equational Model

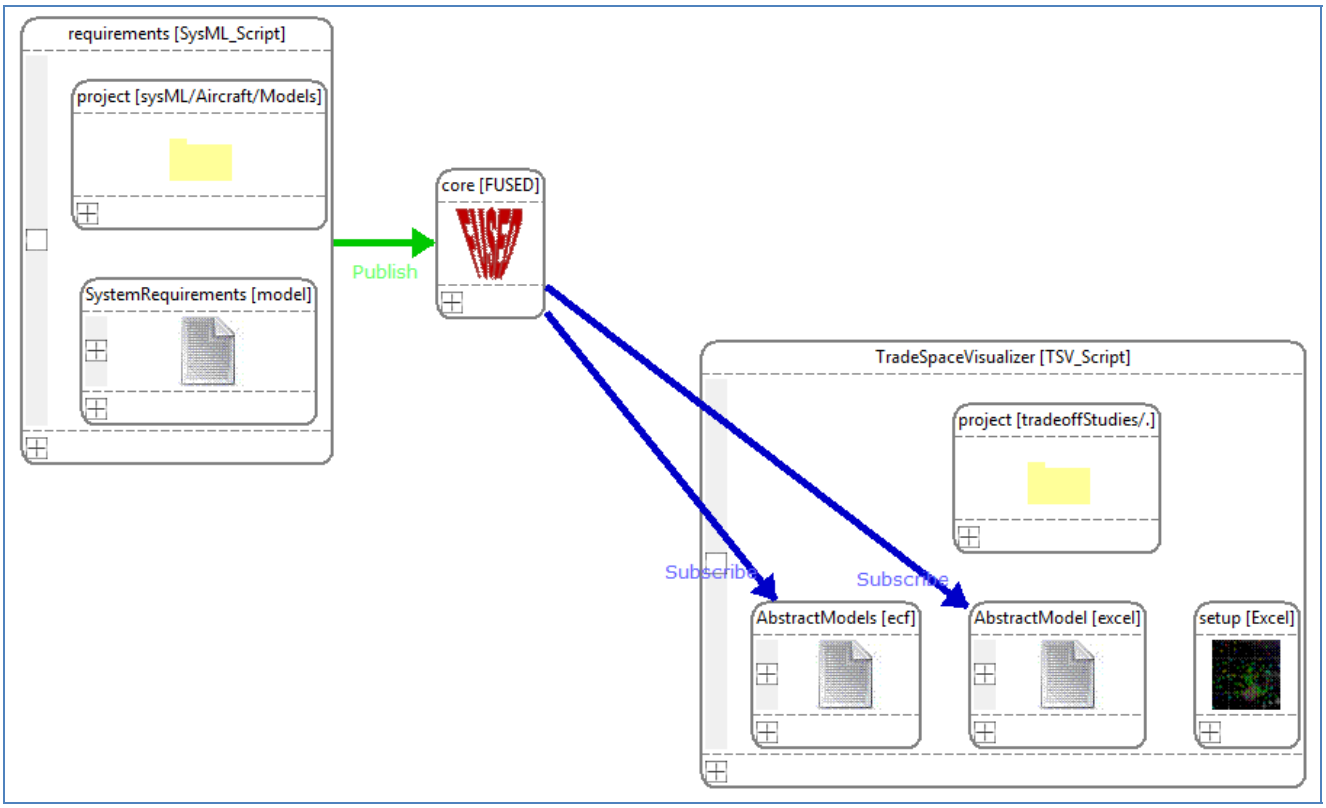

Figure 4: FUSED Composition of Trade Space, Requirements, Equational Models

In this scenario, very early in the development process the system engineer has been given a preliminary set of requirements and has developed an abstract equational model (in Excel) of the UAV from basic aeronautical principles and formulas. This model includes design configuration choices, such as wing parameters and choices of batteries and propellers and motors, that he is uncertain about. In the above FUSED composition specification, requirements such as take-off and cruise velocities, are published to the abstract equational model. This is then composed with ATSV, a trade-space exploration and visualization tool that allows the requirements and systems engineers to explore the trade-offs between requirements and design choices using a variety of model sampling and visualization methods.

4.2 Trade Space Exploration of Mixed Fidelity Model

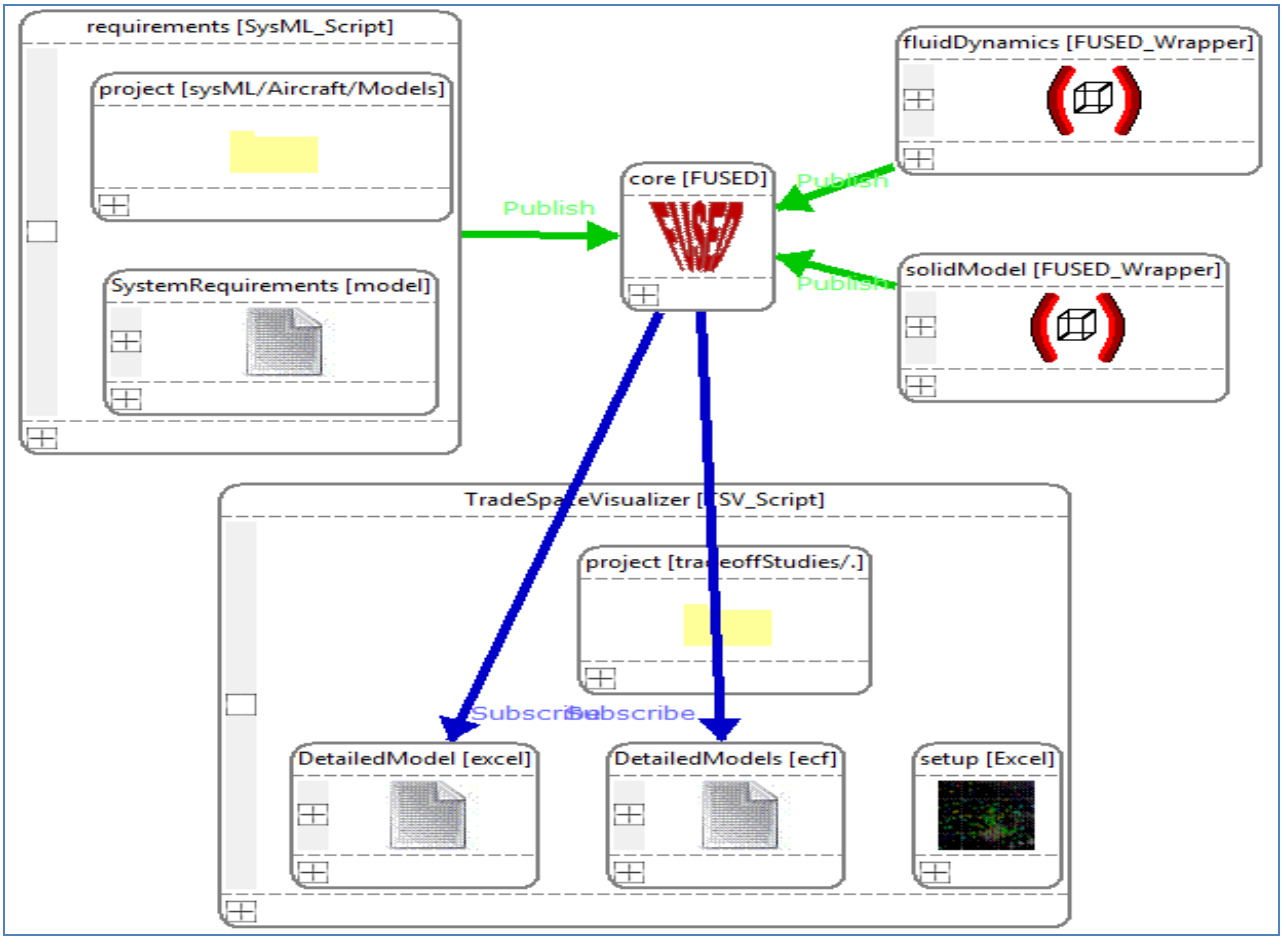

Figure 5: FUSED Composition of Requirements, Trade Space, Mixed Fidelity Models

In the demonstration scenario, we imagine that the system engineer has determined that the choices seem very sensitive to properties such as the exact surface areas of control surfaces and vehicle mass properties, and to the coefficient of drag. Preliminary solid and aerondynamics models are developed (in Creo/ProE and AVL, respectively) and analyzed to obtain more precise estimates of the parameters. The composition of all these models is further combined with ATSV so that trade space explorations can be performed over the hybrid model that has reduced uncertainties.

4.3 Vehicle Dynamics Simulation Using Solid and Aerodynamics Data

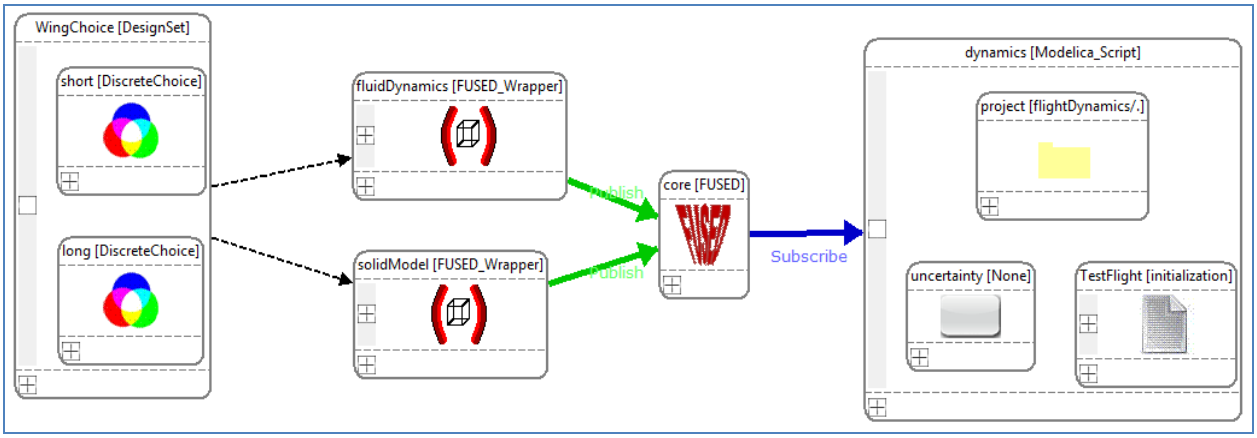

Figure 6: FUSED Composition of Solid, Aerodynamics CFD, and Vehicle Dynamics Models

The above specification shows a composition of a solid/geometric model of a UAV, an aerodynamics/fluid dynamics model of that same UAV, and a Modelica model for the flight dynamics. The right-most box indicates that the system engineer is interested in two specific configurations, either a short or a long wing configuration (the solid and aero models are both configurable for either of these choices). When this specification is compiled and executed, it invokes operations on the solid model to publish the results of a mass properties analysis (e.g. surface areas of control surfaces, total mass, moments of inertia); invokes operations on the aerodynamics model at a set of trim points to obtain stability derivatives and other aerodynamic coefficients; and then passes this data into the Modelica model. The Modelica model includes extended subscription declarations to extract what it needs from this data to simulate a trajectory for the selected vehicle configuration.

4.4 Verification of Multi-Model Consistency

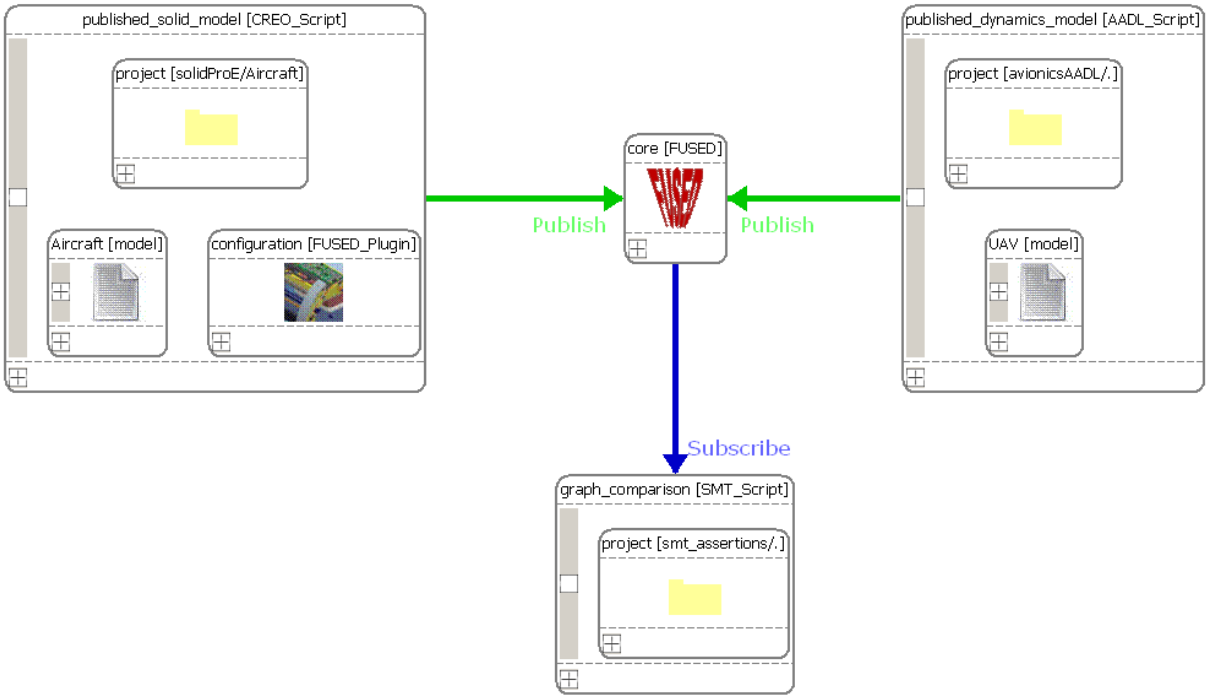

Figure 7: FUSED Composition of Solid, Avionics, and Inter-Model Consistency Verifier Models

In the demo story, the above specification was created by a system engineer who was concerned that the solid modeling team (a team of mechanical engineers) and the avionics team (a team of computer system engineers) might produce specifications that are not fully consistent with each other. In particular, he wants ongoing checking that the resources the avionics people say they need do in fact have space and cabling allocated in the geometry. When executed, the above specification causes an abstraction of model structure (an ontology type called a typed object graph) to be published for both the solid model (in Creo/Proe) and the avionics model (in AADL). A specification that there exists a suitable mapping from logical processors and busses and devices in the avionics model onto physical elements of the solid model is written in the SMTLib language (a semi-standard language that can be used to make logical assertions about

models). This SMTLib model subscribes to the two abstract structural representations and uses one of the many SMT tools to model-check that such a mapping exists. Note that the SMT model is not a model for a component, it is a design aid model used to verify a complex property about the design itself.

4.5 Collaborative Multi-Disciplinary Design Optimization

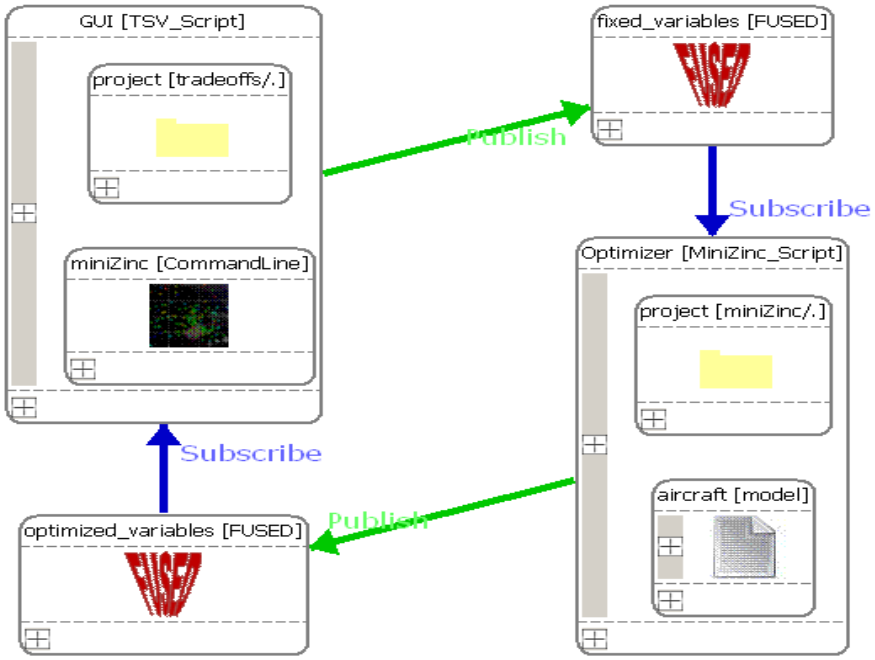

Figure 8: FUSED Composition of Trade Space and Design Optimization Models

In the demo story, customers working with requirements engineers are somewhat uncertain about requirements. They would like to visualize and explore alternatives, and in particular they would like to see the Pareto frontier of what existing technologies could provide them. For a UAV, customers are concerned with things like range, endurance, and cruise velocity. A Trade Space Visualizer (TSV) model is used to define the trade space, explore it using a number of sampling methods, and visualize it using a number of common display formats.

TSV could be used to explore the overall design space including all the detailed engineering design alternatives like choice of propeller, wing design parameters, etc. However, this demo adds a collaborative multi-disciplinary optimization (MDO) twist to things. Rather than confuse the customer with detailed engineering concerns, the TSV model is not composed with a raw model of the vehicle. Instead, it is composed with a design optimization tool that uses a model written in the standard MiniZinc language. When executed, the TSV tool will open and can be commanded to collect samples of the trade space. Each sampling invokes the design optimization model, which will automatically find a good set of lower-level design configuration parameter values for that particular requirements sampling point.

5. CONCLUSIONS

We believe our choice of higher-order attribute grammar and extensible language technology as an enabler to do our work (to build the FUSED infrastructure with) is better than the other alternatives we've seen – basing it on UML technologies or on web technologies. In our experience, it has much more semantic power than the others. For example, it has been fairly easy for us to deal with XML and XMI formats from the UML world, moreover with much more semantics than provided by, say, XML schemas. The natural tie-in between our use of formal languages and language semantics (e.g. type theories) provides a natural way to deal with complex semantics in modeling languages.

Based on our experience, we also believe our tendency to allow subject matter experts in a particular domain to use their favored languages and tools, rather than have them go through some sort of layering on top, is the best choice. It seems a waste of effort to try and include features in a META-Language that are already present in this or that domain-specific language; in fact, it seems impractical to add the full power and convenience and conciseness. Our focus has been in a META-Language that deals with relationships between these various models, rather than one that tries to provide an integrated duplication of some portion of their capabilities and semantics.

Having said that, it is important to deal with inter-model consistency. Our approach to this has been two-fold. First, provide an extensible and rich type and meta-type system that allows strong type-checking of elements that are moved between models or are of significance at the system engineering level. Second, provide export of abstractions of models that can be used with powerful verification technologies to show that complex consistency conditions hold between two models. We performed demonstrations of both of these. However, more experience is needed with both of these in order to mature these technologies.

Our approach to providing a sounder semantic basis for all this is to view the model elements and associated types as abstractions that are published by or subscribed to from different models. The semantics of these common types must fall within the intersection semantics of all the kinds of models that may produce or consume them. More research is needed to extend typing theory to deal with this situation. We need more powerful ways to structure complex ontologies of types and meta-types. We also need a richer set of ways to deal with abstractions, in particular ways to more formally define and verify what it means for an element that has been exported from a model to be an abstraction for something within that model that is suited for the purposes for which it is being used.

Additional research is also needed in support of more highly collaborative and concurrent engineering processes. We have demonstrated some features to support this. More work is needed in the area of ripple effects analysis (smart change propagation) so that designers are not constantly dealing with trivial changes made in other areas while at the same time being certain to stay consistent with changes that do impact their work. We have been working with teams developing various Monte Carlo methods for uncertainty and global sensitivity analysis, and we have done some work ourselves with interval solutions to DAE models, but much more work is needed in the area of support for a rich set of uncertainty models.

6. REFERENCES

Barrett, C., Sebastiani, R., Seshia, S.A., Tinelli, C., "Satisfiability Modulo Theories," in **Handbook of Satisfiability**, IOS Press, Amsterdam, 2008, pp.737-797

Hoyle, C., Tumer, I.Y., Kurtoglu, T., Chen, W., "Multi-State Uncertainty Quantification for Verifying the Correctness of Complex System Designs," *Proceedings of the ASME 2001 International Design Engineering Technical Conferences*, August 2001.

Marler, R.T., Arora, J.S., "Survey of Multi-Objective Optimization Methods for Engineering," *Structural and Multidisciplinary Optimization*., **26**, 6, April 2004 pp. 369-395.

Minnesota Extensible Language Tools, University of Minnesota, http://melt.cs.umn.edu/index.html

Rihm, R. "Interval Methods for Initial Value Problems in ODEs," in **Topics in Validated Computations: Proceedings of Imacs-Gamm International Workshop on Validated Computations, Oldenburg, Germany, 30 August - 3 Sept**, Elsevier Publishing Company, Amsterdam, 1994

Trade Space Exploration, Penn State University, http://www.atsv.psu.edu/

Van Wyk, E., "Semantics of Attribute Grammars and their Roll in Language Development," University of Minnesota, November 2010