Verifying Consistency Between Models

August Schwerdfeger Hazel Shackleton Steve Vestal
Adventium Labs, USA,
{august.schwerdfeger,hazel.shackleton,steve.vestal@adventiumlabs.com}

Abstract

Numerous aircraft development programs have suffered cost and schedule delays due in part to unplanned rework that occurred during integration and acceptance testing. Many of the errors that required rework can be traced back to inconsistencies between different specifications and models developed by or for different disciplines and suppliers early in the development process. We describe a novel method for specifying and verifying complex consistency properties between different kinds of models. This method makes use of a gray-box model integration framework and an SMT verification tool. We report on the application of this method to one specific challenge problem, verifying that a logical computer system architecture specified in AADL and a solid model specified in Creo together satisfy a particular consistency property.

Keywords model consistency, virtual integration, model integration, SMT, verification, defect detection

1. Introduction

Several aircraft development programs have suffered from cost and schedule overruns due to unplanned rework late in the project. Among the reasons cited for the A380 were problems with design configuration management between the different suppliers [1]. For the B787, delays during software and system integration were cited [3]. Delays in software and system integration were also cited as a problem for the F-35 [13]. Studies of software failures have reported that a significant proportion of defects are associated with interfaces between modules or between requirements and implementation rather than a design or coding error within a single module [11, 18, 19]. The System Architecture Virtual Integration (SAVI) project being conducted by a consortium of civil aircraft manufacturers and suppliers has identified verification of model consistency as a priority need [12]. A key goal of the SAVI program is analysis of models in early program phases to detect defects that currently remain latent until integration or acceptance testing, which they have estimated could save $400M in an aircraft development program [24].

The 4th Analytic Virtual Integration of Cyber-Physical Systems Workshop
December 3, 2013, Vancouver, Canada.
Copyright 2013 Adventium Enterprises. The US Government has unlimited rights in this work in accordance with BPA W31P4Q-05-A-0031, Task Order 32. This material may be reproduced by or for the U.S. Government pursuant to the copyright license under the clause at DFARS 252.227-7018. The proceedings are published by Linköping University Electronic Press. Proceedings are available at:
http://dx.doi.org/10.3384/ecp13090
AVICPS website:
http://www.analyticintegration.org/

One approach to assure consistency between different models is to define a Domain-Specific Language (DSL) and then generate different kinds of models for different kinds of analyses from a common specification. An example of this is the SAE standard Architecture Analysis and Design Language (AADL) for embedded computer system architectures. A variety of safety, security, performance, and behavioral models (and system integration and configuration data) can be generated from a common AADL specification. These models can then be analyzed by appropriate back-end tools. This avoids the need to manually generate these various models, which is what is done in the absence of an appropriate DSL and tool set. More importantly, consistency between these various models (and the integration code and data) is built-in to the generators. All are generated from a common input AADL specification. A correctness property of the tools is that the generated models (and integration code and data) are consistent with the input AADL specification and with each other.

A DSL approach only works if a domain is sufficiently bounded and understood to define such a DSL and widely enough used to merit the investment. In contrast, vehicle development is a multi-domain problem that involves computer architecture, solid, control, fluid dynamics, electrical, hydraulic, and many other kinds of models. In a study of ground vehicle development, BAE identified dozens of different modeling languages and tools and a hundred different developer roles [4]. The investment in legacy training, tools, and models is enormous. It has been estimated that the DoD has been spending billions of dollars annually on modeling and simulation [15]. Autodesk alone reports investing about $250M annually in R&D in their tool domain [2]. This paper deals with verifying consistency between different models from different domains, and our approach is based on the use of a model integration framework rather than a new DSL.

One technology that can detect a class of inconsistencies between models is type checking. Vehicles have been lost due to a simple units mismatch [6]. We earlier reported on a multi-view gray-box model integration framework called FUSED that includes a powerful, extensible, abstract type system [7]. In our own experience with a set of UAV models, we found one case of units mismatch and one case of frame-of-reference mismatch between different models.

However, type checking only detects one class of defects. In this paper we report on an approach that allows us to specify and verify consistency properties over the structures of different models. By model structure, we mean the objects and relationships between objects declared by the user in a model. Using the capabilities of our multi-view

gray-box model integration framework, we provide the system engineer with an abstract model structure type. This is a graph structure in which nodes and edges are typed. A structure for almost every kind of model can be represented in this format at some level of abstraction. The mapping from a specific model in a specific language to a graph that abstractly represents the structure of that model is defined when the modeling environment is initially integrated into the framework.

The overall goal is illustrated in Figure 1. Given two models that provide distinct views of a cyber-physical system that are used for different engineering purposes, the model integration framework can provide the system engineer with a gray-box view of each as an abstract structure graph. These abstract structure graphs are then imported into a Satisfiability Modulo Theories (SMT) environment. The SMT specification has "subscribe" statements that direct the framework to import the two model structure abstractions as a set of SMT declarations. The desired consistency property is manually specified in the standard SMT-LIB language. The class of consistency properties that we explore in this paper has the form that there exists a mapping from one model structure to another model structure that satisfies a set of typing and connectivity assertions. Intuitively, the consistency property we verified is that the processors, buses, devices, etc. specified in the avionics model map to corresponding solid objects with a corresponding connection topology. However, our long-term goal is general methods for specifying and verifying a broad class of complex consistency properties.

Examples of previous work to define and verify consistency between different models are synchronization between UML models [9], treating multiple models as extended projections of a central common model [22], consistency between different views and a base architecture [5], and traceability between models[16]. All of these share certain similarities. The mappings between models are partial, each may contain information that does not appear in and cannot be generated from any of the other models. Most use annotated graph representations of the static structure of models and define mappings between such graphs. Our approach does not require an expert in meta-modeling or formal languages or the integration framework to explicitly specify particular transformations or mappings or trace links between models. It uses concise specifications of desired properties and applies verification technology that automatically checks for the existence of a satisfying mapping between models. Our approach does not require an explicit single base architecture or common central view. It assumes models are written in existing languages (nine were used in an earlier series of UAV demonstrations[7]).

In the remainder of this paper, we will present the two models in our challenge problem: an AADL model of the logical software and hardware architecture for an aircraft mission system; and a solid model of the boards, enclosures, and cables for that system. We then present the SMT-LIB specification for a desired consistency property and a FUSED workflow specification that orchestrates the overall verification task. We conclude with lessons learned from this exercise and the results of some synthetic benchmarking experiments to assess scalability.

2. Models and Workflow

The two models that are to be verified consistent with each other are a 3D solid model specified using the commercial Creo (formerly ProE) tool and a logical avionics architecture model specified in AADL. Consistency means the two models satisfy a consistency property that we specify in the SMT-LIB language, which is a third model. Finally, there is a FUSED workflow specification that composes these three models to automate the sequence of actions necessary to perform the actual verification. FUSED workflows are themselves models, and this workflow specification is a fourth model. All of these models have an associated modeling environment, which is a particular modeling language supported by a particular toolset. These four environments are integrated into the FUSED multi-view gray-box model integration framework, which is necessary to execute a workflow automatically.

2.1 Equipment Solid Model

Solid models are three dimensional geometric models created by specifying 2D and 3D shapes and combining them to form more complex shapes, parts, and assemblies. Shapes (closed regions in 3D space) can be created by applying operations such as extrusion and rotation to 2D shapes. More complex shapes and parts can be specified by applying constructive solid geometry operations such as union, intersection and difference to simpler shapes. Geometric constraints can be specified to combine parts into assemblies and to combine parts and assemblies into increasingly more complex assemblies. A variety of properties can be specified for objects in the models, such as material and surface appearance properties. Most tools allow user-defined properties to be associated with solid objects.

Almost all the tools of which we are aware impose a hierarchical containment structure on the solid objects in a model. Every shape, part, and assembly (other than the root) is uniquely contained in one other shape, part or assembly. A containment tree view is provided to users to facilitate model navigation and object selection. Geometric constraints define relationships that may cut across this tree structure to form general graphs of relationships between solid objects. For example, parts in different subtrees can be aligned or mated using geometric constraints. Geometric constraints can create relationships between almost any pair of nodes in the containment tree structure.

There are standard exchange formats for solid models such as STEP and IGES, but model creation is done using a GUI unique to each tool vendor. Most commercial vendors provide a variety of solid model analysis capabilities, for example mass properties and structural analyses. Most tools provide an API for third-party tool developers.

Figure 2 is a rendering of the solid model for the simple avionics system used in this study. We developed our own solid model based on data sheets that could be downloaded from vendors of ruggedized embedded electronics modules and associated standards. Actual production models would contain more detail than we were able to include in our model, a subject to which we will return in the remarks section. (At least one vendor we talked to could supply detailed solid models, but only for purchased equipment under an NDA.)

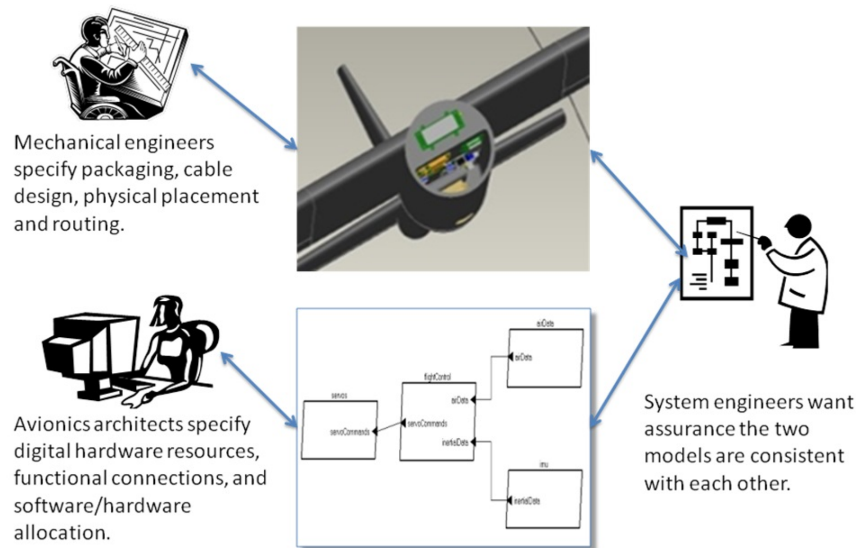

Figure 1. The goal is to verify consistency properties between different kinds of models.

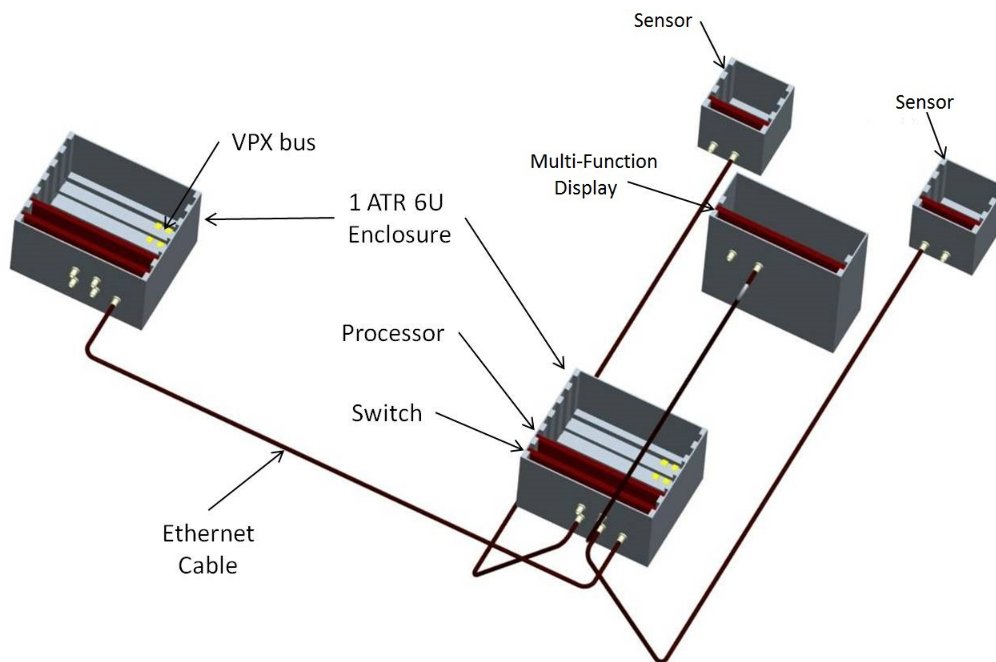

Figure 2. The 3D solid model shows enclosures, boards, and cables. Boards are inserted and cables connected by declaring geometric constraints. Special tool plug-ins collect electrical and type data.

These assemblies would normally be located (using additional geometric constraints) within an airframe solid model. Given the effort involved, placing these assemblies within an outer airframe model was not deemed significant enough for this exercise because the consistency property we used only applies to solid objects of particular types. We specified types for the objects in our model using a user-specified property, and only objects typed as devices, processors, switches, enclosures, cables, etc. were referenced by our consistency property.

Wiring harness design is a complex problem, e.g. routing, electromagnetics, reliability, installation, maintainability. Special add-in tools are available for most commercial solid modeling tools for the sub-domain of wiring harness design and placement. These specialized wiring harness add-ins capture electrical connectivity data and also add geometric constraints to the model. There are thus relationships between solid objects in the solid model that are typed as electrical connections.

We did not have a wiring harness add-in tool. We manually drew cabling using our own geometric constraint pattern to specify connections, shown in Figure 3. We added user-defined properties to explicitly identify electrical connections between solid objects. In our experiment we used the latter to identify electrical connection relations between solid objects (i.e. to strongly type certain relations between solid objects as electrical connections). In practice this could and should be done using only data that is already available from a wiring harness add-in without any additional user input.

2.2 Avionics Architecture Model

AADL is an SAE standard language used to model software and hardware architectures for embedded computer systems [20]. Software objects such as subprograms, objects, threads, processes, partitions, etc. and hardware objects such as memories, devices, processors, buses, etc. can be connected and bound to each other. There is a sophisticated typing system with extension, generic, and refinement capabilities. The language has a well-defined semantics and a large set of standard properties to enable a variety of analyses for performance, safety, security, behavior, etc. Tools also exist to generate system integration code and data (detailed specifications for individual components that are being integrated are usually written in some other suitable language, e.g. C, Ada, VHDL, SimuLink). The language includes packaging and hierarchical structuring features for components, data types, and connections. The standard defines textual, graphical, and tool interchange formats for AADL specifications.

We wrote our specification using the Open Source AADL Tool Environment (OSATE) [21]. Figure 4 shows three of the many diagrams from the graphical representation of the model. Hierarchical relationships are indicated in this figure by dotted lines that show which of the lower diagrams provides internal design detail for which boxes in the upper diagram. (In the actual tool, buttons and clicks are used to navigate up and down the design hierarchy between diagrams.) AADL specifications, in the graphical view, have the boxes-and-arrows look-and-feel of many modeling languages used in the computer field (but with well-defined semantics that enable a variety of analysis and generation tasks to be automated).

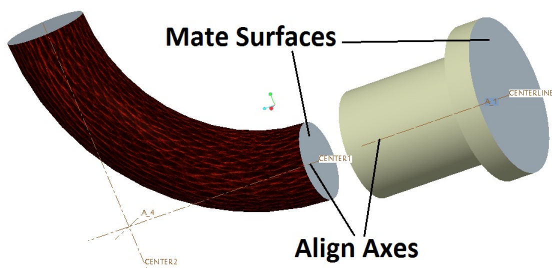

Figure 3. Connections are declared in the solid model using a geometric constraint pattern and properties on solids.

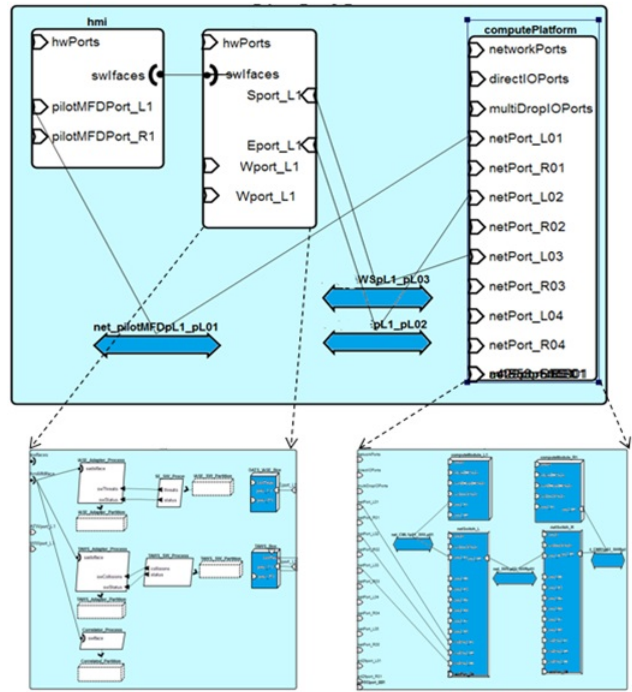

Figure 4. Three of the several diagrams in the graphical view of the AADL model. The dotted lines indicate where the lower diagrams show internal detail for boxes in the upper diagram.

The boxes that represent hardware objects have been shaded blue in the figure. AADL is a system language, and software and hardware objects are mixed in various places in a complex model. For example, in our model there is a common hardware computing platform shared by multiple hosted functions. Each hosted function consists of software subsystems together with special hardware equipment for that particular function, such as sensors. AADL is used to specify an integrated system in which multiple hosted functions are integrated onto the common computing platform both by binding software elements and by connecting specialized hardware. The overall hardware architecture is typically not contained in a single diagram. The hardware components are scattered among multiple specifications (typically developed by multiple suppliers) plus integration specifications developed by the system integrator.

An AADL specification is analogous to a set of class declarations. For example, a declaration of a processor called PowerPC is actually a specification for a class of processors. Multiple instances can be created from this declaration, for example by declaring multiple components of this class inside another declaration. A model for a specific system instance is obtained by "instantiating" a designated root system declaration in the AADL specification. This produces a data structure that has a specific set of instance objects that represent a specific system.

2.3 FUSED Models and Workflows

One way to better coordinate multidisciplinary modeling is to use an environment that supports capture of traceability links between different models in an environment [17]. Some engineering workflows that involve different kinds of

models can be automated using a black-box model integration framework [25, 10]. In a black-box model integration framework, models are abstracted as functions from design parameters to quality metrics that can be evaluated at design time. Model integration frameworks allow the specification of “workflows” that automate engineering tasks that involve multiple different kinds of models. For example, a solid model can be “evaluated” to map choices of wingspan to vehicle weight and moment of inertia, which can in turn be used as input parameters of a dynamical systems model, which can then be solved or simulated to obtain performance metrics of interest to the engineer.

For our challenge problem, which requires a higher degree of visibility into the models being integrated, we used a prototype gray-box model integration framework called FUSED [7]. Among the goals for FUSED are improved specification and management of system (multimodel) configurations and design trade spaces, automated inference of traceability data from higher-level declarations of dependencies between models, interoperation with multiple model repositories and servers across multiple organizations, and more complex workflows such as combining multiple design automation environments (e.g. trade space visualization, multidisciplinary design optimization) with system models that are themselves compositions of many design models. Another example of a complex workflow is the subject of this paper, the use of a gray-box structural abstraction viewpoint into models to support automated verification of complex consistency properties.

The type system we are developing in FUSED combines concepts from programming language type systems (e.g. type constructors, multiple inheritance, interfaces, generics) and ontologies (e.g. description logics). All of this is done with the perspective that any single FUSED type is just one possible abstract viewpoint of some aspect or element of a model that has a more detailed and precise meaning and structure in the semantics and type system of its own language and modeling environment.

In the exercise reported here, several types play several roles. Both the solid modeling and the logical architecture modeling languages have their own type systems (e.g. primitive types of surfaces and shapes, types of relations such as part-of or electrical connection). At the FUSED level in this exercise, these domain-specific type systems are abstracted down to sets of enumeration literals (think of abstracting all types in a software program down to a list of class names). At the FUSED level in this exercise, we also use a graph type whose nodes and elements are labeled with these domain-specific type names. For example, the solid model is abstractly represented as a graph whose node labels name types of solid shapes and whose edge labels name types of solid relationships. (Inheritance in the domain-specific type systems is currently abstractly represented by allowing nodes and edges to have multiple type labels.)

The gray-box viewpoint used in this exercise could be graphically examined and navigated using a FUSED GUI by the engineer (who is developing the verification workflow specification). The nodes of the abstract model graph (a FUSED type of thing) represent objects statically declared in the model specification (e.g. component declarations but not objects that would be dynamically created during an execution or simulation of that model – those might appear in a different viewpoint). The edges of the graph represent relationships between objects that exist in a model. The nodes and edges are labeled with the names of domain-specific types declared in the detailed models. Figure 5 shows a graphical view of the abstract structure graph for the AADL model used in this exercise in which hardware (versus software and system) nodes have been shaded blue and connection (versus containment) edges have been shaded green (shading is used here in liu of displaying lists of domain-specific type labels).

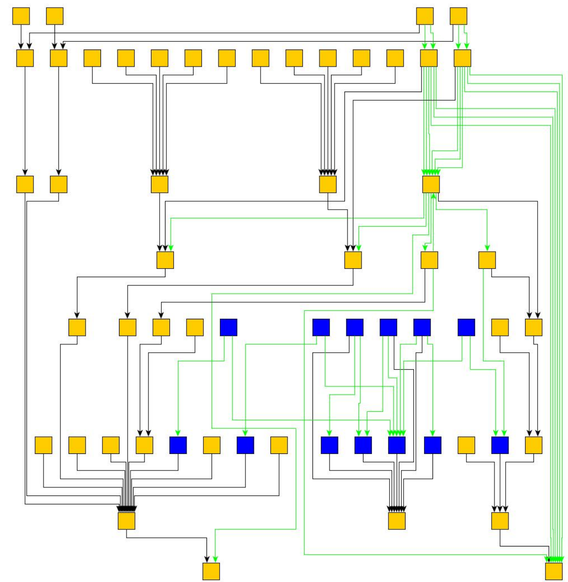

Figure 5. One layout for the abstract architecture view. Hardware objects are shaded blue, connection relations are shaded green.

The FUSED workflow used in this exercise is shown in Figure 6. This specification identifies the desired solid and architecture models and their configurations, specifies that abstract structural views are published for these two models and used to satisfy subscriptions (dependencies on external data) within a specified SMT model, and specifies that the SMT model is verified within the SMT modeling environment. The SMT model contains the specification of the consistency property to be verified.

2.4 SMT Consistency Property

The overall “proof structure” is to verify that a particular consistency relationship exists between two abstractions of the two models. The abstraction steps were done conventionally and without any formal definition or verification beyond strong typing of the data structures. The consistency relation is typed subgraph isomorphism. This is captured as an SMT-LIB specification and verified using an SMT checker.

Satisfiability Modulo Theories (SMT) tools can determine whether a model, as the term “model” is used by formal logicians in mathematics, exists for a specified set of many-sorted first-order logic predicates over an available set of theories [8]. The set of theories over which predicates may range can vary from tool to tool, but almost all support theories of integers, reals and arrays. SMT has been widely applied to formally verify properties for software. In this exercise SMT technology is applied for a novel purpose, to

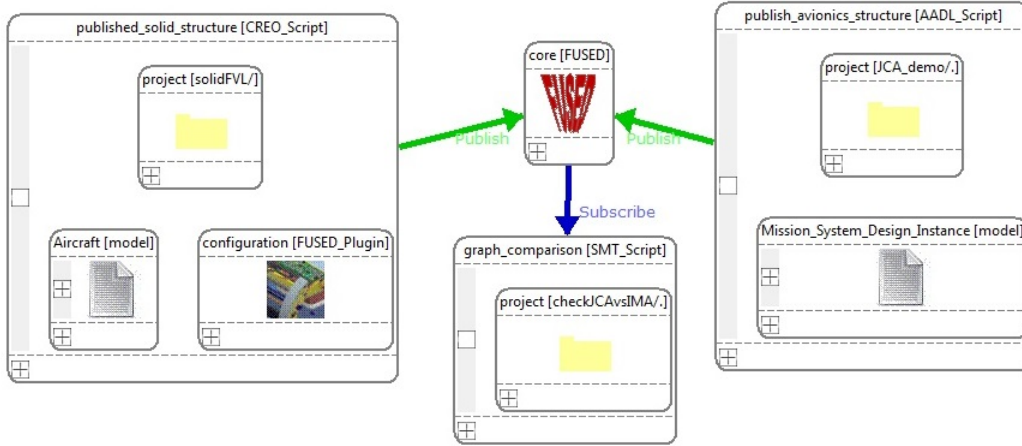

Figure 6. The FUSED workflow specification directs that abstract structural views be published by the solid and architecture model and provided to the SMT model, which is then executed to verify the consistency property.

formally verify a consistency property between the abstract structures of two different kinds of models. The consistency property desired is written in the SMT-LIB language by a subject matter expert familiar with this modeling and verification technology.

Figure 7 shows a redacted version of the consistency property specification used in this exercise. This can be broadly divided into two sections, the declarations for the abstract model structures to be checked and the declarations for the consistency property itself.

The declarations for the two abstract model structures are trivial for the SMT user. FUSED was built using language extension technology from the University of Minnesota [23, 14]. This makes it easy to integrate new modeling environments at a FUSED site. It also makes it easy to layer carefully defined new features on top of any modeling language at the time that modeling environment is integrated into FUSED. For example, this can allow type specification and checking not available in the original language (e.g. units, frames of reference). In this exercise, we layered a "subscribe" declaration on top of the SMT-LIB language. This new declaration allows the SMT specifier to identify data that is to be provided from an external source as specified by the system engineer in a FUSED workflow specification.

The FUSED framework will automatically perform type checking and representation conversion as required and expand the subscribe declarations into standard SMT-LIB declarations before invoking SMT tools. In the SMT-LIB language, nodes and types are declared as enumeration literals that are satisfied by an assignment of unique integers to them. Nodes are typed by a predicate over nodes and types. Edges are a predicate over pairs of nodes and an edge type. Nodes and edges may have multiple types, e.g. inheritance or casting in the original modeling language.

The second part of the specification must be manually written. This can be divided into two subparts. The first subpart consists of predicates that identify the nodes and edges (by type) for which a mapping must exist and specify type compatibility rules between nodes and edges that are mapped from one model to the other. The second subpart is the declaration of an array that maps nodes from one model to another plus a declaration that says this map must be a subgraph isomorphism where the nodes and edges satisfy the type compatibility rules.

The predicates that identify the types of nodes and edges that must be mapped and the type compatibility rules need to be modified for each new pair of models, but their structure is simple. The subgraph isomorphism declarations can probably be reused for a different pair of models with minor modifications.

The consistency relation between the two abstractions is typed subgraph isomorphism. Although theoretically a computationally challenging problem, graph isomorphism has been well-studied, and reasonably scalable algorithms exist. Adding the type compatibility constraints, pre-filtering the abstract graphs to nodes and edges having types of interest, and seeding with known mappings (e.g. "assert avionics leftCabinet maps to solid leftEnclosure") should further simplify the problem.

Our choice of SMT for this specific consistency property could reasonably be challenged as overkill. However, our hypothesis is that a variety of complex properties may emerge, and the number and nature of such properties is still a research issue. We selected SMT because it is a powerful general-purpose verification technology. A research question is whether a many-sorted first-order logic over a few fixed theories can easily express a wide class of useful properties.

3. Results

SMT performance was surprisingly good. Our challenge problem required about three dozen nodes and edges to be mapped. Verifying satisfiability or unsatisfiability required only a few seconds.

Most SMT tools provide their own languages and support unique features. There are multiple ways to encode a problem, especially when the set of alternatives considered include tool-specific language features. For example, some languages support special forms of declaration for enumerations. We conducted some preliminary synthetic benchmarking exercises, summarized in Figure 8, that both scaled the size of the problem and experimented with different formulations. There were some surprises, e.g. the

```
;; Subscribe to SMT declarations for elements of the abstract model graph type.
;; FUSED expands these to enumerations and functions for nodes, edges, and types.
(FUSED-subscribe Solid)
(FUSED-subscribe AADL)

(define-fun mapAADL ((ao AADL.Object)) Bool
    ;; return true if AADL.Object must have a mapping to a solid model object
))

(define-fun typeCompatible ((ao AADL.Object) (se Solid.Object)) Bool
    ;; return true if the avionics and solid object types are compatible
))

;; The model consistency property is true if a satisfying value for "map" exists.
(declare-fun map ()(Array AADL.Object Solid.Object))

;; The typed graph isomorphism property the map must satisfy.
;; For all avionics objects that must map to solid objects...
(assert (forall ((ao AADL.Object)) (=>  (mapAADL ao)
     (and
         ;; the map from avionics to solid model is 1-to-1
         (forall ((aoB AADL.Object))
              (=> (and (mapAADL aoB) (= (select map ao) (select map aoB))) (= ao aoB)))
         ;; if <ao,aoB> is an avionics edge that must map to a solid edge
         ;; then solid edge <map(ao), map(aoB)> with compatible type must exist
         (let ((so (select map ao)))
             (forall ((aoB AADL.Object))
                 (=> (and (AADL.attached ao aoB AADL.connection) (mapAADL aoB))
                     (Solid.attached so (select map aoB) Solid.Mate))))
         ;; avionics type of ao is compatible with solid type of map(ao)
         (typeCompatible ao (select map ao))))))
```

Figure 7. A redacted SMT consistency property specification used in this exercise.

use of special enumeration declaration forms could be less tractable than a more primitive declaration of literals plus a series of uniqueness assertions. We hypothesize that further experimentation in collaboration with practitioners skilled in the art could significantly expand the size of problem solvable. It is still an open question whether this can scale to problems of real-world size, but even if not, SMT may be useful to explore for consistency properties for which efficient tailored verifiers should be developed.

For practical use, it is essential that the method provide clear and useful indicators of why two models are inconsistent when the consistency property is unsatisfiable. The feed-back provided by the SMT tool in our demo was not directly useful when debugging model inconsistencies. SMT tools will identify unsatisfiable cores in such cases, but the output may in essence be the internal encoding, and the core may be an arbitrarily large subset of the entire specification. In our case, the tool would report that the entire set of assertions was unsatisfiable. We know that our SMT declarations of the abstract structure graphs are satisfiable in isolation. The unsatisfiabilities of interest are due to the nonexistence of a satisfying solution for the map array alone. It should be possible to use these properties to obtain useful error reporting.

Fully detailed solid designs are much larger and more complex than our model. Special-purpose add-in tools for wiring harness design are typically used. It would be an interesting future exercise to use a solid model with increased fidelity and publish/subscribe the wiring harness electrical analysis results view for use in consistency verification.

There are broader research questions that remain. How can we better formalize and verify the abstraction relations between the typed graphs and the detailed models? Can this approach specify and verify a usefully broad range of consistency properties that are effective in detecting defects that occur in practice? (We hypothesize this exercise could be adapted to any consistency property of the form "there exists a mapping between typed graph abstractions of model static structures that satisfies a set of properties.") Would applying these methods early in the process (for example to preliminary design models) significantly reduce defects that otherwise would remain latent until integration or acceptance testing? To address these questions, we are working to establish collaborations with developing organizations to perform studies using real-world models and defect data.

Acknowledgments

This work was supported by US Army AMCOM, Bruce Lewis, Army POC, bruce.a.lewis.civ@mail.mil.
DISCLAIMER: Reference herein to any specific commercial, private or public products, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement, recommendation, or favoring by the United States Government. The views and opinions expressed herein are strictly those of the authors and do not represent or reflect those of the United States Government. The viewing of the presentation by the Government shall not be used as a basis of advertising.

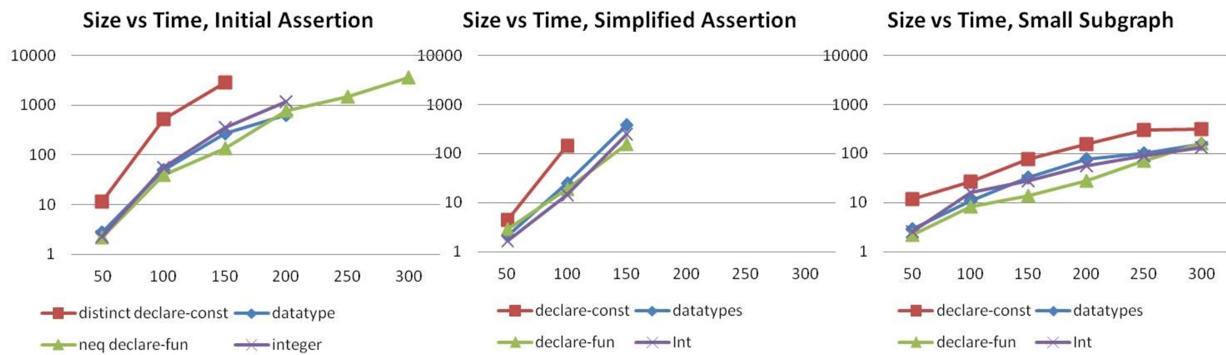

Figure 8. Initial synthetic benchmark results for various problem sizes and formulations were encouraging.

References

[1] Airbus A380. online, September 2013. http://en.wikipedia.org/wiki/A380.

[2] Autodesk Annual Report. online, April 2012. http://investors.autodesk.com/phoenix.zhtml?c=117861&p=irol-reportsAnnual.

[3] Boeing Reschedules Initial 787 Deliveries and First Flight. online, September 2013. http://www.boeing.com/news/releases/2007/q4/071010d_nr.html.

[4] Steven Bankes, Daniel Challou, David Cooper, Todd Haynes, Hillary Holloway, Paul Pukite, Jorge Tierno, and Christopher Wetland. META Adaptive, Reflective, Robust Workflow (ARRoW) Phase 1b Final Report. Technical Report TR-2742, BAE Systems, October 2011.

[5] Ajinkya Bhave, Bruce H. Krough, David Garlan, and Bradley Schmerl. View Consistency in Architectures for Cyber-Physical Systems. *International Conference on Cyber-Physical Systems*, 2011.

[6] Mars Climate Orbiter Mishap Investigation Board. Phase I Report, 1999. ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf.

[7] Mark Boddy, Martin Michalowski, August Schwerdfeger, Hazel Shackleton, and Steve Vestal. FUSED: A Tool Integration Framework for Collaborative System Engineering. *Analytic Virtual Integration of Cyber-Physical Systems Workshop*, 2011.

[8] David R. Cok. The SMT-LIBv2 Language and Tools: A Tutorial, March 2013. http://www.grammatech.com/resource/smt/SMTLIBTutorial.pdf.

[9] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM SYstems Journal*, 2006.

[10] iSight and the SIMULEA Simulation Engine. online, September 2013. http://www.3ds.com/products-services/simulia/portfolio/isight-simulia-execution-engine/latest-release/.

[11] David N. Card. Learning From Our Mistakes with Defect Causal Analysis. *IEEE Software*, January 1998.

[12] Peter H. Feiler, Jorgen Hansson, Dionisio de Niz, and Lutz Wrange. System Architecture Virtual Integration: An Industrial Case Study. Technical Report CMU/SEI-2009-TR-017, Software Engineering Institute, November 2009.

[13] GAO. Joint Strike Fighter Restructuring Places Program on Firmer Footing, but Progress Still Lags. Technical Report GAO-11-325, General Accounting Office, April 2011.

[14] Jimin Gao, Mats Heimdahl, and Eric Van Wyk. Flexible and Extensible Notations for Modeling Languages. *Proceedings of Conference on Fundamental Approaches to Software Engineering*, 2007.

[15] Paul Gustavson, Ali Nikolai, Roy Scrudder, Curtis Blaise, and Richard Daehler-Wilking. Discovery and Reuse of Modeling and Simulation Assets. online, September 2013. The M&S Journal, http://www.msco.mil/.

[16] Simon Frederick Königs, Grischa Beier, Asmus Figge, and Rainer Stark. Traceability in Systems Engineering – Review of industrial practices, state-of-the-art technologies and new research solutions. *Advanced Engineering Informatics*, 2012.

[17] Simon Frederick Königs, Grischa Beier, Asmus Figge, and Rainer Stark. Traceability in Systems Engineering – Review of industrial practices, state-of-the-art technologies and new research solutions. *Advanced Engineering Informatics*, 2012.

[18] Maggie Hamill and Katerina Goseva-Popstojanova. Common Trends in Software Fault and Failure Data. *Transactions on Software Engineering*, 1978.

[19] Robin Lutz. Analyzing Software Requirements Errors in Safety-Critical, Embedded Systems. *IEEE Requirements Engineering*, 1993.

[20] Architecture Analysis and Design Language. Technical Report AS5506, SAE, September 2012.

[21] OSATE 2. online, September 2013. https://wiki.sei.cmu.edu/aadl/index.php/Osate_2.

[22] Charles Simonya, Magnus Christerson, and Shane Clifford. Intentional Software. *OOPSLA*, 2006.

[23] Minnesota Extensible Language Tools, September 2013. http://melt.cs.umn.edu/index.html.

[24] Don Ward, Steve Helton, and Greg Polari. RoI Estimates from SAVI's Feasibility Demonstration, 2011. Systems Engineering Conference.

[25] Scott Woyak. Simulation Driven Design: Creating an Environment for Managing Simulation Tools, Processes, and Data. Technical report, Phoenix Integration, March 2010. http://www.phoenix-int.com/documents/pdf/white_papers/simulation-driven-design.pdf.