GAMETE: General Adaptable Metric Execution Tool and Environment

Kurt Rohloff
Raytheon BBN Technologies
Cambridge, MA USA
krohloff@bbn.com

Kyle Usbeck
Raytheon BBN Technologies
Cambridge, MA USA
kusbeck@bbn.com

Joe Loyall
Raytheon BBN Technologies
Cambridge, MA USA
jloyall@bbn.com

Keywords: Complexity metrics; performance metrics; simulation-based verification and validation

Abstract

In this paper we introduce the General Adaptable Metric Execution Tool and Environment (GAMETE) to aid the design, measurement, and analysis of cyber-physical systems (CPSs). GAMETE is a general and extensible environment for evaluating and computing metrics associated with the performance and complexity of CPS designs. GAMETE supports a wide array of metrics that it generates over simulation and experimental output data from CPSs. Key features of GAMETE are its 1) execution environment to host the simulations of CPS models, 2) unified data representation to host simulation and experimental data from CPSs, 3) dynamic metric library that supports the semi-automated evaluation of a wide range of metrics, and 4) standards-based integration with design toolchains. The contribution of this paper is the presentation of a reference architecture for batch metric computation and a case-study where GAMETE helped to quantitatively evaluate the performance of a CPS.

1. Introduction

The cost of modern complex military, aerospace, and other systems and systems of systems has skyrocketed in recent years. These Cyber-Physical Systems (CPSs) comprise major physical components that cannot function properly without integrated cyber components (such as engine control systems and communication buses) [1][7][10][14].

The devices in many, if not all, modern complex CPSs are semi-independent and support platforms with common goals and missions. The model refinement that occurs during multiple design iterations of CPSs creates flexibility that makes systems able to cope with changing environments and extends their useful life. This flexibility can manifest itself in many ways, including being deployed in multiple environments [1] and accommodating reconfiguration through new components [14] or connections between components [12]. The competent design, development, and integration of these systems are critical. However, the cost of modern CPSs is growing exponentially because it is exceedingly difficult to evaluate the integrated design of these complex systems without physical testing [1][7][10][14].

There are extensive models and modeling tools for all of the various CPS components [1][10], but there are no evaluation frameworks (automated or otherwise) that can integrate these models and evaluate these systems during the design phase. Furthermore, as CPSs are designed, developed, and maintained, there is little support to evaluate whether changes to a system are increasing its complexity, and thereby increasing its future maintenance and testing cost.

Sponsored by Defense Advanced Research Projects Agency, Program: META, Issued by DARPA/CMO under Contract No. HR0011-10-C-0108.

We developed the General Adaptable Metric Execution Tool and Environment (GAMETE) to address these concerns and aid in the design and analysis of CPSs. GAMETE is a general and extensible environment for evaluating and computing performance and complexity metrics for CPS design. GAMETE both generates and hosts simulation and experimental output data from CPSs that it uses to compute a wide variety of performance and complexity metrics. Key features of GAMETE are the following:

- An execution environment to host the simulations of CPS models,
- A unified data representation to host myriad data from simulation and experiments of CPSs,
- A metric library that supports the semi-automated evaluation of a wide range of metrics, and
- A standards-based framework to ease the integration of GAMETE with broader design, modeling, and simulation toolchains.

A central part of GAMETE's capabilities and extensibility derives from the only assumption we make on simulation and experimentation engines – that they are processes that can commit data directly to a consumer such as GAMETE.

Although GAMETE is a prototype, we have demonstrated the unified data representation and metrics library capabilities on a variety of data from third parties. We discuss our use of GAMETE to evaluate general classes of metrics as part of the design and V&V of a complex cyber-physical military system. GAMETE identifies system designs that are less complex, more efficient, less likely to fail, less costly, and that have higher performance, and evaluates the behavior of designs during simulation or experimentation for V&V.

This paper is organized as follows. In the next section, we describe the GAMETE execution environment architecture and design. In Section3, we discuss the GAMETE unified data representation. In Section 4, we discuss the GAMETE metrics library. In Section 5, we discuss related work. In Section 6, we conclude with a discussion of ongoing development of GAMETE.

2. Unified Data Representation

A central part of GAMETE capabilities and extensibility derives from the only assumption we make on experimentation/simulation engines – that they are processes that can commit data directly to a consumer such as GAMETE. Examples of experimentation/simulation engines supported by GAMETE include engineered systems that generate data directly from *(i)* sensors such as those that might monitor aspects of system behavior during testing or that gather information during modeling/design, *(ii)* simulations that might generate data over multiple runs, *(iii)* deterministic evaluations of possibly coupled equations, and *(iv)* model representations, such as graph models or equations. GAMETE is designed to semi-automate the collection of data as needed from all of these online or offline data sources.

The GAMETE Unified Data Representation (UDR) is a meta-model which supports the reporting and storage of experimental/simulation/design model data. The UDR enables pluggable metrics and experiments to be developed in the execution framework. Our approach in the design of the UDR is informed by the Semantic Web domain where all information is represented as data graphs with attributes [3]. Our approach is compatible with systems which analyze time-varying resource consumption where resource dependencies are represented as directed graphs.

Recognizing that experimental data is collected from a number of different methods, using different tools with varying frameworks and languages, the UDR provides an easily-supported framework to interface the analysis engine with the variety of experiments and simulations over which metrics are evaluated.

The UDR allows for metrics to be reported and analyzed in any of three formats: resource-consumer relationships, graphs, and sets of key-value pairs (signals). Figure 1 shows the hierarchical nature of the UDR allowing resource-consumer relationships to be viewed as graphs, and simple key/value signals to be attached to graph nodes and edges, thus maximizing the applicability of implemented metrics. The key insight in making the UDR is that the data types in many application domains can be hierarchically ordered. This insight allows metrics to be applied in interesting new ways such as representing resource-consumer relationships as graphs and applying graph-based behavioral metrics to analyze the complexity of resource-consumer relationships.

A reference implementation of the UDR is implemented as a Java library that provides utilities for formatting, persisting, retrieving, and analyzing data. By using the Hibernate framework for data persistence and retrieval, the reference implementation provides an abstraction of the datastore so users are free to use their preferred relational database management system. The current reference implementation is configured to use the open source database, mySQL. Further, the UDR library is designed to be extremely flexible. The same library is used as a component of the Analysis Engine and the Experiment Engine, and is the only component of GAMETE that is necessary for the design and implementation of GAMETE-supported metrics.

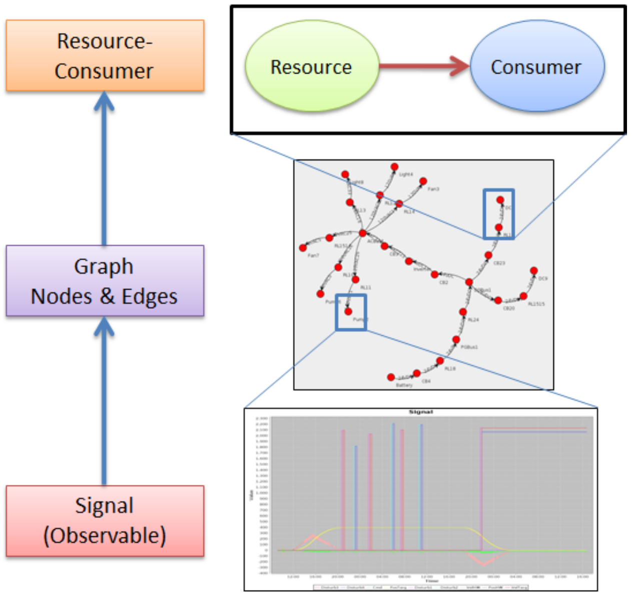

Figure 1: Representing signals, graphs, and resource-consumption in the UDR maximizes the applicability of implemented metrics.

The UDR library also includes several utility functions for common operations, many of which are used in creating the default metric library and include the default metrics.

3. GAMETE Architecture and Design

Figure 2 shows the GAMETE architecture and toolchain interface as well as the dataflow between GAMETE components. GAMETE enables an integrated V&V toolchain by supporting the evaluation of general classes of user-selected metrics on user-selected design variations and user-selected data. Experiments are specified in GAMETE as pluggable components so they can be added and managed separately from metric evaluation and analysis, making GAMETE highly-extensible and adaptable to external toolchains. The output is provided to the user or other consumers in external toolchains as part of design or V&V activities.

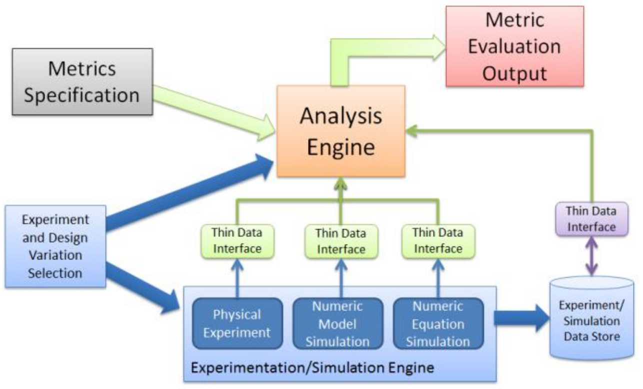

Figure 2: The GAMETE Architecture is designed to support data from "live" experiments, models of systems, and experimental simulations.

The analysis engine is the driver of the execution environment. Its responsibilities include *managing metric specifications*, *controlling metric execution* using the appropriate *managed data source* by acquiring experimental data from the data store, and *presenting the output to the user* through a user interface or other consumers through application programmatic interfaces (APIs). In this section we discuss these responsibilities of the GAMETE components in detail.

Managing Metric Specifications

GAMETE has several natively-supported metrics for the evaluation of model performance which are part of the initial *metric library*. Custom metrics can also be created and stored in the metric library. This allows users to design their own metrics (for use within GAMETE) which may be domain-specific, application-specific, or scenario-specific.

User-created metrics must conform to the Unified Data Representation (UDR) API discussed in Section 2. The UDR provides an API for inspecting the experimental data that passes through the analysis engine. This implies that all metrics must be written in Java and built against the UDR, meaning that compilation must include the UDR library. However, it offers the benefit of compile-time type-checking, which will ultimately help metric designers assure that their metrics will function properly when used in GAMETE. Also, this design decision provides power to the metric designer in that he/she can utilize the full functionality of a Turing-complete language in implementing their metric.

A major design goal of GAMETE is the ability to load custom and third-party metrics either at initialization or when the analysis engine is already running with minimal input from the user. This is enabled by a “`@Metric`” annotation which serves as a *tag* on the methods that the metric designer wishes to expose to GAMETE. Using Java annotations as indications of metrics allows us to reflectively search the classpath periodically to discover functions that are designed to be metrics and promotes loose-coupling of GAMETE architecture components.

Customized metrics can be compiled into Java archive (JAR) files and shared with other GAMETE users. These JARs can then be loaded into GAMETE (even if it is already executing). This feature is vital for any metric execution engine that runs on large datasets so that changes can be made while it is currently processing a large dataset without interrupting the current computation. We use JAR files as the conduit for sharing GAMETE metrics for several reasons. First, JAR files provide a single file that can be conveniently shared between users and loaded into GAMETE. Secondly, using JAR files provides metric designers with all the advantages that come from using Java, including strong-typing, compile-time error checking, efficiency-enhancing integrated development environments, and platform independence. Finally, using JAR files simplifies the implementation of dynamic metric importing. The metric importing utility simply loads the JAR file directly into the classpath and reflectively re-scans the classpath for new metrics (tagged with `@Metric` annotations).

Controlling Metric Collection

The analysis engine controls which metrics are computed and collected during each simulation or experiment execution based on user input. Metric collection occurs on-demand (i.e., when requested by an end-user) to prevent unnecessary data processing. GAMETE provides a user interface for users to select which available metrics they want to collect while a simulation or experiment is executing. We do not automatically collect all metrics at all times because calculation of some metrics is computationally expensive and datasets can be very large.

As new metrics are imported into GAMETE, their names automatically appear in the appropriate categorization (i.e., signal, graph, and resource consumer metrics are grouped separately). Users indicate which metrics should be computed by checking the box adjacent to the metric name, prior to selecting the simulation or experiment.

Managing Experimental and Simulation Data

Another function of the analysis engine is the management of the experimental and simulation data. We designed GAMETE so that it can receive data for calculating metrics from multiple sources, including the *Experimentation-Simulation Engine* or the *Experiment-Simulation Data Store*.

The Experimentation-Simulation Engine ingests data directly from physical experiments through external data ports and from modeling tools that can stream data to other processes during runtime. It then archives the experimental data in the Data Store. The interface from the Analysis Engine to experimental data is read-only – the Analysis Engine cannot modify experimental data as it flows from the source. This provides security for the use of the tool so third-party metrics can only affect their own reports. Other metrics, and more importantly the stored experimental data remains unaffected.

Some modeling tools do not have native support for data streaming to external processes. Similarly, some models and data collection experiments may take too long or are too expensive to run repeatedly. For this reason, we use the Experiment-Simulation Data Store to archive (persist) the experimental data from experimental engine(s).

Presenting Data to the User

The UI displays a listing of all stored experimental runs. When an experiment is selected in the UI, the analysis engine computes the currently-loaded metrics that apply to the highest matching class of experiment type in the UDR (i.e., first resource-consumer, then graph, then signal).

A prototype user interface (UI) for GAMETE performs metric computation on-the-fly. A tabular interface is used as a way to allow system designers to quickly adjust their view of the system’s design or performance and allows multiple views of the system to be displayed simultaneously (for comparison purposes). The ability to present time-varying output provides an intuitive approach for system designers to understand the effects of dynamic interactions between components.

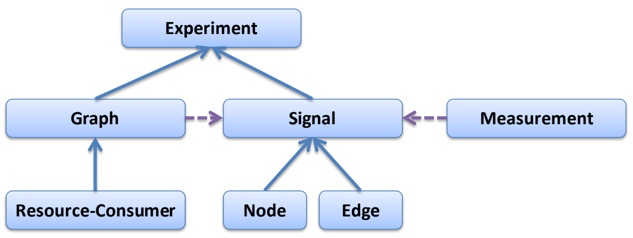

Figure 3: The UDR is represented in this class-level design diagram by inheritance relationships (shown as solid blue lines) and compositional relationships (shown as dotted purple lines).

Experimentation / Simulation Engine

The Experimentation/Simulation engine is a set of libraries for committing experimental results to the data store in the appropriate format. Libraries are currently written for Java and Python. Figure 3 shows the relevant class-level relationships between the UDR components used by the GAMETE client libraries. Using the client libraries to commit experimental data ensures that the data is properly formatted and persisted in the datastore with a user-friendly API, shared by both the python and Java libraries.

As seen in Table 1, the `Engine` class allows the addition of experiments, graphs and resource consumers to an experiment, respectively.

Table 1 **`Engine.addExperiment(name)`**

Parameter	*Type*	*Description*
`Name`	String	An identifying name for the experiment.
RETURN	Experiment	A new experiment with no data.

As seen in Table 2, the `Experiment` class is used for holding and identifying signals, graphs, and resource-consumer data that share the same experimental setup. This is important because it enables the intuitive representation of experiments with multiple components (i.e., one component per graph node), as well as the relationships between those components (i.e., the edges between nodes).

Table 2 **`Engine.addGraph(experiment)`**

Parameter	*Type*	*Description*
`experiment`	Experiment	The experiment to which the graph belongs.
RETURN	Graph	A new graph belonging to experiment with no nodes or edges.

Similarly, one can use resource-consumer terminology when referring to the graph simply by indicating that it is a `Resource-Consumer` graph. In a `Resource-Consumer` graph, resources and consumers are represented by nodes and the consumption relationships are indicated by edges.

As seen in Table 3, `Signal` is not actually a class, but an abstract class or *mix-in* that, in this case, applies the `addSignalData` function to several different classes, including `Experiment`, `Node`, and `Edge`. Thus, all three of these classes can now contain time-series data.

Table 3 **`Signal.addSignalData(time, data)`**

Parameter	*Type*	*Description*
`time`	Timestamp	Time that `data` was collected.
`data`	Float	Quantitative data that was collected at `time`.
RETURN	Boolean	True if the entry was added to the time-series data.

As seen in Table 4 and Table 5, `Graphs` have nodes, which can represent multiple components within a single experiment, and edges, which represent the relationships between those components.

Table 4 **`Graph.addNode(signal)`**

Parameter	*Type*	*Description*
`signal`	Signal	The time-series data collected by the component that is represented by this node.
RETURN	Node	The new node that was created or NULL if the node could not be added to the graph.

Table 5 **`Graph.addEdge(fromNode,toNode,signal)`**

Parameter	*Type*	*Description*
`fromNode`	Node	The node from which this directed edge should start.
`toNode`	Node	The node at which this directed should terminate.
`signal`	Signal	The time-series data associated with the relationship between two components, that represents this edge.
RETURN	Edge	The new directed edge that was created from `fromNode` to `toNode` or NULL if we could not add this edge to our graph.

`Resource-Consumer` has utilities that apply resource-consumer terminology to standard Graph functionality. The API to `Resource-Consumer` is a full wrapper around the `Graph` class. **Table 6** shows the classes and methods from the `Graph` class wrapped by the `Resource-Consumer` classes:

Table 6 Classes and Methods from the `Graph` class wrapped by the `Resource-Consumer` classes

Resource-Consumer Classes/Functions	**Equivalent Graph Classes/Functions**
`Resource`	`Node`
`Consumer`	`Node`
`ResConsRelationship`	`Edge`
`addResource(resource)`	`addNode(resource)`
`addConsumer(consumer)`	`addNode(consumer)`
`addRelationship(resource, consumer)`	`addEdge(resource, consumer)`

4. Example Use of GAMETE

As a demonstration example for using GAMETE to evaluate modeling and simulation, we considered the control signal output by two different control designs for a model of a vehicle ramp, such as the loading ramp on cargo vehicles. These systems use digital controllers to maintain a consistent

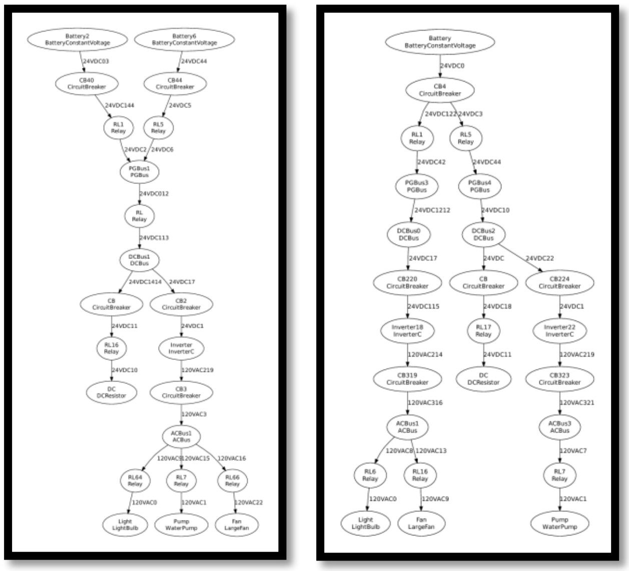

Figure 4: Two design options for the digital controller electronics where design components are represented as graph nodes and their inputs/outputs are edges.

opening and closing speed of the ramp despite changes in ramp torque as the ramp angle changes. The ramp control signal controls the amount of torque output by the ramp motor.

We performed two different digital controller analyses: one analyzing the design, and one analyzing simulation data. First, we analyzed the designs of the controllers by representing the design components as graph nodes and their input/output relationships as edges in a directed graph (as shown in Figure 4). GAMETE showed that their design complexities were similar, so we continued to an analysis looking at simulation data for one control design corresponding to a low-gain controller and another control design corresponding to a high-gain controller which would require different amounts of power and control tuning to operate safely.

We analyzed the input and output control signals of these two digital controllers in Figure 5 as input to GAMETE. One controller, the controller on the right in Figure 5, is a high gain controller with additional actuation circuitry to apply higher magnitude control signals to an electric motor. The figure on the left in Figure 5 represents a lower-gain controller with simpler circuitry. We received this data from a third party and imported it into our GAMETE repository. It took us a matter of minutes to evaluate a standard Shannon-type signal complexity metric [12] for these two signals.

To compute the complexity metric *S(c)* we use the signal frequency spectrum *p(f)* which represents component sinusoid signals which are components of more complex signals. We compute *p(f)* by taking a Fourier transform of the time-varying signal (*c(t)*) and treat that as a probability density function in frequency space (i.e., a normalized power spectrum). The entropy (*S(c)*) of *p(f)* represents the complexity measure.

We found that the control signal from the low-gain controller has a signal complexity 40% lower than the control signal from the high gain controller. This makes intuitive sense because the high gain controller generates a periodic actuator input signal rather than constant steady-state signal which

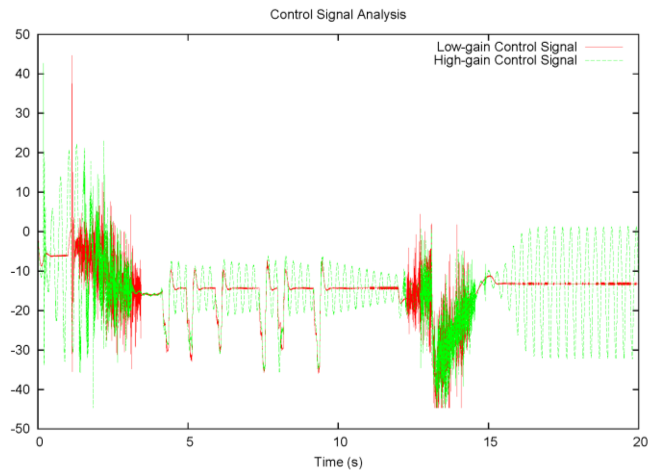

Figure 5: Two control signals, one from a high-gain controller with periodic steady-state behavior (green dotted line) and one from a low-gain controller with DC steady-state behavior (red solid line).

requires lower-cost control circuitry and maintenance. The periodic actuator signal from the high gain controller has higher amplitude sinusoidal components than the low-gain controller, meaning that the frequency specturm of the high gain controller is always greater than the frequency spectrum of the low-gain controller. As a result, the signal complexity of the high gain controller is greater than the complexity of the low gain controller.

The hierarchical relationship we impose through the UDR of resource-consumers, graphs, and signals allows for metrics to be applied in GAMETE in useful and interesting new ways. For example, since resource-consumer relationships are represented in graphs, the graph-based behavioral metrics can be computed to analyze the complexity of resource-consumer relationships. Furthermore, since the graph nodes and edges represent signals, signal metrics can be applied to each edge and node of the corresponding graph. This capability ensures that GAMETE has the effectiveness, general applicability and promise for continued cost-effective improvement that system designers need.

5. Related Work

The contribution of this paper is the presentation of a reference architecture for batch metric computation and a case-study where GAMETE helped to quantitatively evaluate the performance of a CPS. There has been previous research looking at integrated modeling environments [5][6][9][15], but this previous work has focused on the integration of more limited systems. This prior work has focused on software engineering [5][6], general control systems [9] and dependability analysis in cyber systems. Also relevant from a software perspective is the UML approach to integration [4] which provides more of a formalism than a software environment. This UML approach was also used in the OMG Model Driven Architecture standards of XMI and MOF [13] but is separate from the approach we take herein.

There has similarly been prior work on the integration of cyber-physical systems [2][8][11][16]. This prior work has focused on important specific aspects of cyber-physical system composition, including information flow security analysis [2], dependability [8][11], and noninterference [16]. There has been little prior work on model integration environments that can cover the end-to-end breadth of CPS design. and what there is has not provided end-to-end analysis with user interfaces like GAMETE. Our attempt with GAMETE is to provide a comprehensive tool that can be used to analyze all of these important system properties in an integrated environment.

6. Conclusions and Ongoing Development

GAMETE is a research prototype and, while it has shown itself to be an important and useful part of the CPS design process, there is further research to be done, including 1) increasing the scale and breadth of metrics and experimental data sets supported, 2) improving the interactive capabilities of GAMETE with online metric evaluation and 3) aiding the identification of primary and secondary impacts of design alternatives.

Up to now, we have used experiment specifications of relatively-small scales (on the order of millions of datapoints) as produced by other projects. For complex systems with many moving parts and communication pathways, current metric evaluation tools do not scale to larger data sets and system hierarchies because of the interactions between sub-systems that cannot currently be tracked in monolithic metric evaluations. Part of our objectives for future GAMETE development is to increase the size of data sets supported by designing and implementing “divide and conquer” capabilities for incremental metric evaluation in GAMETE.

Current tools only investigate the primary impacts of design decisions on metric evaluation. There have been few practical capabilities to support the identification of design impacts across multiple levels of hierarchies. This limitation is primarily a scaling issue – few tools, if any, can analyze the impacts of design decisions across sufficiently many sub-systems. By instrumenting GAMETE to investigate the propagating impacts of design decisions, we believe we can increase the scale of GAMETE with respect to the levels of hierarchy considered and size of the data logs processed. We have several potential approaches to investigating the propagation of design decisions across hierarchies, including Monte Carlo metric evaluations and cost-benefit metric evaluation which focuses on the important evaluation cases.

Another important area of future investigation is integration of the GAMETE concepts and GAMETE prototype with a CPS design, modeling, and development toolchain. A metric evaluation engine, populated with useful complexity metrics, and utilizing a data representation that can support a variety of inputs (models, signals, etc.), such as we have prototyped in GAMETE, is a key component of a larger design toolchain. Any toolchain that does not include a metrics evaluation engine of the power and flexibility that we provide is missing functionality needed to cost-effectively support the design, development, and maintenance of CPSs. Clearly, a design tool is lacking if it cannot provide the designer meaningful metrics on whether changes to a design are increasing or decreasing complexity, maintainability, and lifecycle cost. Likewise, the toolchain is only providing partial V&V support unless it provides a powerful capability for deriving metrics of the system under test. GAMETE provides this necessary functionality. Furthermore, it provides a powerful set of complexity metrics and the extensibility to add more as they become available. This is important because any metrics framework that does not come with a set of useful metrics is unproven, and any that is limited to a fixed set of metrics will not accommodate the future uses of the cyber-physical design toolchain nor future metrics that are developed.

References

[1] Mark V. Arena, Obaid Younossi, et al., Why Has the Cost of Fixed-Wing Aircraft Risen?, Report No. MG696, RAND Corporation (2008)

[2] Ravi Akella, Han Tang, Bruce M. McMillin, Analysis of information flow security in cyber–physical systems, International Journal of Critical Infrastructure Protection, Volume 3, Issues 3–4, December 2010, Pages 157-173

[3] Antoniou, Grigoris, and Frank Van Harmelen. *A semantic web primer*. MIT press, 2004.

[4] Siobhán Clarke, Extending standard UML with model composition semantics, Science of Computer Programming, Volume 44, Issue 1, July 2002, Pages 71-100

[5] Franck Fleurey, Benoit Baudry, Robert France and Sudipto Ghosh, A Generic Approach for Automatic Model Composition. Models in Software Engineering, Lecture Notes in Computer Science, 2008, Volume 5002/2008, 7-15

[6] France, R.; Fleurey, F.; Reddy, R.; Baudry, B.; Ghosh, S.; , "Providing Support for Model Composition in Metamodels," *Enterprise Distributed Object Computing Conference, 2007. EDOC 2007. 11th IEEE International* , vol., no., pp.253, 15-19 Oct. 2007

[7] Paul G. Kaminski et al., Pre-Milestone A and Early-Phase Systems Engineering, National Research Council (2008)

[8] Lajolo, M.; Rebaudengo, M.; Reorda, M.S.; Violante, M.; Lavagno, L.; , "Evaluating system dependability in a co-design framework," *Design, Automation and Test in Europe Conference and Exhibition 2000. Proceedings* , vol., no., pp.586-590, 2000

[9] Ledeczi, A.; Nordstrom, G.; Karsai, G.; Volgyesi, P.; Maroti, M.; , "On metamodel composition," *Control Applications, 2001. (CCA '01). Proceedings of the 2001 IEEE International Conference on* , vol., no., pp.756-760, 2001

[10] Lee, E.A.; , "Cyber Physical Systems: Design Challenges," Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on , vol., no., pp.363-369, 5-7 May 2008

[11] Jing Lin; Sedigh, S.; Miller, A.; , "A General Framework for Quantitative Modeling of Dependability in Cyber-Physical Systems: A Proposal for Doctoral Research," *Computer Software and Applications Conference, 2009. COMPSAC '09. 33rd Annual IEEE International* , vol.1, no., pp.668-671, 20-24 July 2009

[12] Seth Lloyd, “Measures of Complexity: A Nonexhaustive List,” IEEE Control Systems Magazine, Vol. 24, No. 4 (August 2001)

[13] OMG. “MDA Specifications” Retrieved on 1/25/2013 from http://www.omg.org/mda/specs.htm

[14] Kurt Rohloff, Partha Pal, Michael Atighetchi, Richard Schantz, Kishor Trivedi and Christos Cassandras. “ Approaches to Modeling and Simulation for Dynamic, Distributed Cyber-Physical Systems.” Workshop on Grand Challenges in Modeling, Simulation, and Analysis for Homeland Security (MSAHS-2010), March 2010.

[15] Stott, D.T.; Floering, B.; Burke, D.; Kalbarczpk, Z.; Iyer, R.K.; , "NFTAPE: a framework for assessing dependability in distributed systems with lightweight fault injectors," *Computer Performance and Dependability Symposium, 2000. IPDS 2000. Proceedings. IEEE International* , vol., no., pp.91-100, 2000

[16] Yan Sun; McMillin, B.; Xiaoqing Liu; Cape, D.; , "Verifying Noninterference in a Cyber-Physical System The Advanced Electric Power Grid," *Quality Software, 2007. QSIC '07. Seventh International Conference on* , vol., no., pp.363-369, 11-12 Oct. 2007