System of Systems Readiness Assessment

Jack Ring
OntoPilot LLC
Gilbert, AZ USA
jack@ontopilot.com

Antonio Pizzarello Ph.D.
OntoPilot LLC
Gilbert, AZ USA
tony@ontopilot.com

Oris Friesen Ph.D.
OntoPilot LLC
Gilbert, AZ USA
oris@ontopilot.comline

Byron Davies Ph.D.
OntoPilot LLC
Gilbert, AZ USA
byron@ontopilot.com

Abstract— **Is this system still fit for purpose? This question becomes critical in either a proposed or an active system of systems because many unannounced changes occur throughout the system content and context during its operational life. This challenge arises in multiple usage domains such as defense, national security, industrial, financial, commercial or personal. A system of systems readiness assessment capability must be devised by the systems engineering activity and rigorously vetted during system development, deployment and evolution. The authors describe a set of candidate capabilities and technologies that may enable a comprehensive, fast and trusted assessment capability. Furthermore such assessment capability may be extended to aid in highlighting system resiliency shortfalls, vetting proposed changes to any constituent system and rapidly diagnosing failures.**

Keywords-system of systems, faults, logic simplification, weakest precondition, massively parallel processing

I. PROBLEMATIC SITUATION

Before committing to an operation every user wants to know whether their system is ready for operation. Their alternative is to commit to the operation under the risk of system aborts that can range from wasteful to deadly.

Readiness is determined by estimating the probability that the system will satisfy its measures of effectiveness in the anticipated situation [1] or by detecting whether the system contains faults that will interfere with completion of an intended usage. Estimation involves modeling and simulation and test and evaluation. Fault detection involves inspection of the encoded algorithms. Fault detection becomes increasingly preferable for systems that are higher in extent, variety and ambiguity, where extent signifies the number of cognates involved, variety signifies the number of unique cognates involved, both temporal and semiotic. and ambiguity signifies the uncertainty of SoS usage profiles and SoS intrinsic and extrinsic attributes as well as ramifications of cognitive overload on the part of the SoS designers.

Unfortunately, prevailing recipes for systems engineering of SoS such as the Systems Engineering Guide for Systems of Systems, Version 1.0, Aug. 2008, do not explicitly address designing and vetting a readiness assessment capability. Also, most system engineering recipes ignore the fact that such SoS are constructed in the field, not in system test bays, which means the readiness assessment capability must be finalized in the field. Because the usage situations are somewhat indeterminate then built-in resilience of the readiness assessment is critical. Also, component re-use may be counterproductive because one that worked fine in one configuration may not work at all in another due to changes in usage context as well as in interoperating components. Typical changes are; the next mission engagement profile may be slightly different, one of the constituent systems may have undergone an upgrade or problem fix, reference data values may have been changed, something failed somewhere, or a shared circuit is overloaded, or myriad other causes.

Because operators and their system administration counterparts throughout the various constituent systems are not fully informed [2] many errors are not noticed until attempted mission operations spontaneously abort. This can be precluded by giving users a 'Push to Test' capability at the SoS level just as they have at component levels. 'SoS Push To Test' certifies whether or not a SoS is still fit for purpose even if it was previously. Otherwise, the judgment of operations readiness is made only on the absence of obvious errors.

The rest of this paper describes usage cases, measures of effectiveness, envisioned capabilities and applicable technologies for assessing whether any SoS that is ready for use. The challenge of non-deterministic situations is addressed. Extensions for detecting faults in requirements, concepts, architectures and modularization of an intended SoS from Day 2 of a development project and for automated composition of a SoS appear feasible as summarized at the end of this paper.

II. THE NEW IDEA AND ITS VIABILITY

Readiness assessment is not about proof of correctness. Readiness Assessment determines whether a SoS is Fit For Purpose, where purpose is defined by the intended usage. A SoS may contain faults and still be fit for purpose if the faults are not encountered during intended operations. Readiness Assessment entails asking meaningful questions about the context, content and structure of the SoS, such as 'what assumptions are needed for each desired outcome?' and 'do faults exist that impede the assumptions ?'

A. Errors and Faults.

A system consists of "modules" which change the state of common data or (equivalently) exchange values that are then used for changing private data. For acceptable operation the system data are constrained to some states by *safety/integrity properties, i.e., nothing bad is going to happen.* Also, an acceptable SoS consists of modules that change the data from a legitimate state to a desired (still legitimate) one according to system *progress properties, i.e., something good is happening.*

Modules are composed of algorithms that produce adequate results when executing in an environment that is within the designers' mutual assumptions. This environment must be such that all the safety and progress properties are maintained. In practice systems may be not fully understood, may have been modified, or may have been placed in new contexts/environments that violate the assumptions. Although respective designers may strive to write programs that produce a satisfactory result under their assumptions many if not most of designers' assumptions are unknown to one another and to the users.

Errors and faults are defined by relations over the data values and are indicated by violations of these relations. An error is a mismatch between the actual state and the specified desired state during the execution of a program. A fault is a state violating some safety or progress property of the whole system. The relationship between fault and error is causal: errors are caused by faults. Some faults cause errors that test cases cannot reproduce.

Of the myriad variables in a large system usually very few are involved in an interesting state change at a specific place in the code. It is useful to highlight these and apply the technique of code slicing to filter out all the code that is not relevant thereby rendering the code more comprehendible by responsible personnel.

In the case of autonomous systems the external environment may reach a state that was not considered by the designer thereby causing a fault. Examining not just the SoS but also its interaction with its context is mandatory.

Predicate calculus can describe program states precisely. For example a queue length of -5 is a violation of the informal "queue length cannot be negative," which may be expressed as the predicate "$QL \geq 0$"." Predicates can be manipulated by formally defined rules both for determining fit for purpose and for simplifying the program to make it more understandable to humans. Manipulating predicates formally describing program states is known to range from quite difficult to downright infeasible. This is no longer the case as described in a subsequent section regarding the General Purpose Set Theoretic Processor, GPSTP®.

The weakest precondition, wp, is the least restrictive set of assumptions that must be verified at the start of a computation for it to terminate in some desired state. Given a code segment *S* and a predicate Q defining the desired states at its termination (the postcondition), the wp is a function of *S* and *Q*, wp(*S,Q*), whose value is a predicate describing the minimum set of assumptions for *S* to terminate in a state where *Q* holds. The wp can be used for detecting faults, answering questions about program properties, slicing large code, proving the adequacy of a module in a system and determining the adequacy of a system to external environment actions. The wp calculation is completely formal and practically automatable using proven technologies.

B. *Stakeholders and SoS Usage.*

SoS Readiness Assessment serves the stakeholders involved in such endeavors as net-centric warfare, crisis first responders, industrial supply chains, national transportation modalities, information assurance [12], commercial and personal computing 'in the cloud,' and management of pandemics, whether concerned with human health, piracy, immigration or cybersecurity [12]. In addition to stakeholders concerned with operating SoS's other stakeholders are the in-service engineering practitioners responsible for enhancing and evolving respective constituent systems and for inventing and integrating operations readiness assessment capability into new SoS configurations.

C. *Readiness Assessment Usage.*

At least six usage cases are relevant.

- Users will use the 'push to test' capability prior to commencing actual usage of the SoS. Assessment is done by inspection of the code. No test beds, test cases, original tests or regression tests are required.
- Prior to inclusion of constituent systems users will confirm that those systems are consistent with the readiness assessment coverage envelope by inspecting the actual artifacts that comprise the inclusion candidate.
- During the operational life of a SoS users will want to reconfirm readiness as frequently as indicated by the SoS Mean Time to Change [2].
- Likewise, the capability will be used to confirm that any change, implemented or proposed, is consistent with SoS purpose.
- Earlier, SoS developers may Push to Test an envisioned SoS at its design stage. This becomes more feasible as model-based system engineering produces executable models.
- Finally, users will want confirmation of the effectiveness of the readiness assessment capability in any SoS and across interacting SoS's. Such confirmation may entail assessment of the artifacts that comprise the readiness assessment capability.

D. *Measures of Effectiveness*

Six measures of effectiveness will determine the acceptance of a SoS Readiness Assessment Capability:

- **Acuity:** Incidences of False Positives and False Negatives in the declarations of the existence and locations of faults.
- **Cycle Time:** Readiness Assessment Cycle Time must be less, preferably much less, than the real or anticipated Mean Time to Change, MTTC, of constituent systems comprising the SoS.
- **Trust building:** Revealing the source of confirmed faults not just the existence of the fault. No surprises during SoS operations.
- **Coverage:** Discerning the envelope of Extent, Variety and Ambiguity accommodated by the readiness estimate.
- **Cost:** Development project cost and SoS production and operating costs.
- **Agility:** Span, accuracy and time of responding to changes in a SoS.

III. Enabling Technologies

A. *Non-deterministic systems.*

Faults result not only from mistakes made during system implementation but also from non-deterministic events such as new releases of system software, unpredicted human actions and similar occurrences [2].

Systems are made by the combination of mechanisms, which are or can be faithfully described as algorithms, mostly realized as computer code. Systems also interact with an unpredictable environment made up of hardware, system software (operating system, compilers, loaders etc), databases, sensors, actuators and humans. Even though an individual program in isolation can be formally validated as the computation of a mathematical function, a system interacting with a non-deterministic environment cannot [3]. Furthermore systems that modify themselves according to situational awareness already exist [4]. This reality highlights that the formal methods conceived and occasionally used today to verify programs in isolation are not effective for systems interacting with ambiguous context.

Formal methods by definition require the formal description of the system and of its components. In current practice the only formal description available is the code. Code is a very compact description of the sequence of commands for realizing the separate functions of the system component. The code alone does not contain any possibility of verifying its correctness [5]. Consideration of context is essential.

B. *Sustaining Requisite Variety:*

Recently computer scientists, notably [3] in computability logic and [6] in refinement calculus, have recognized the necessity of considering non-deterministic contexts when reasoning about program correctness, a concept initially introduced by [5]. This has expanded and complicated the field of discourse. A promising approach considers a known program, called the *angel,* to be engaged in a formal game with its context, called the *demon.* The assessment determines whether the *angel* can exhibit the requisite variety [7] to cope with all antics by the *demon*.

C. *Code Analysis.*

Key code analysis technologies are 1) a method of simplifying computer code to reveal the weakest preconditions for proper execution, 2) a unified ontology that spans all kinds of algorithms, notably, software at all stages of realization, but also human interactions with and within a system, and 3) a way of verifying coherence between two or more non-deterministic entities. These enable assessment without execution. No test case development, no test beds, no regression testing. Assessment beats test and evaluation in speed, cost and confidence in results. Integrating these technology advancements into an assessment capability can create the critical mass for a breakthrough in system vulnerability detection and suppression.

The use of formal methods for detecting and correcting faults and for separating the slices of code relevant to faults was demonstrated in 1999 [8], [9]. Although the results were quite encouraging and at times even surprising (detection and correction of non reproducible errors) two important limitations of formal methods were a) the exponential explosion of analyst cycle time in cases that involved indices and pointers and b) the scarcity of people who could be trained to deal with such complexity. Today, both of these limitations can be removed with recent hardware advancements and associated knowledgeware and software.

D. *A General Purpose Set Theoretic Processor.*

The General Purpose Set Theoretic Processor, architecture and process [10] is expected to overcome the combinatorial explosion of possible paths through a set of code that conteins pointers and indices. The GPSTP is a configurable nondeterministic finite state machine. When implemented in modern semiconductor technology a GPSTP® 'chip' will have 2^{16} byte state recognizers that examine input byte streams independently with no parallel processing performance penalty. It will be capable of 2^{65536} states which enables qualification of numerous relational patterns among the bytes. It will have an internal bandwidth of 2^{16} bits per clock cycle (as contrasted to $2^5 = 32$ or $2^6 = 64$ bits per cycle in current commercial computers). When user-configured with a Reference Pattern within these size and complexity limits it processes input bytes as fast as memory technology can serve them up, updates its internal states and reports out any discovered patterns in the input data while wasting no time on instruction cycles. Throughput is constant regardless of the size and complexity of the Reference Pattern or the incidences of relevant bytes in the input data.

IV. Envisioned SoS Readiness Assessment Capabilities

A. *Capability 1.*

Code translation. Most systems are written in a variety of languages. Compiler technology is sufficiently mature to translate all programs into a convenient intermediate language. This allows the creation of tools for one language rather then a plethora of languages for a variety of systems. As an intermediate language the UNITY computational model [11] allows easy inspection and transformation into specifications as well as easy verification of concurrency.

B. *Capability 2.*

Predicate simplifier. Transforming code into UNITY converts long sequences of commands into a more compact set of simultaneous assignments, at the price of constructing rather involved predicates. Predicate calculus is a formal discipline that is amenable to automation, hence the needed predicate simplifier becomes possible.

C. *Capability 3.*

The weakest precondition, wp, [5] allows calculating via pure syntactic manipulation the least restrictive assumptions that are needed for any code to reach a state satisfying a post condition. The wp can be used (and has proven effective) to automatically find faults using a post condition describing the correct output with the erroneous code, to extract slices of interest from a large portion of the system, and to verify the acceptability of code [9]. Although beyond the scope of this paper, the wp calculation may be a way to create unique signatures for code thereby establishing a basis for detecting code that should not be present, i.e., unauthorized code.

The major problem of calculating the wp for indices and pointers is solvable through the use of a new technique associated with the GPSTP hardware described in the preceding section. The calculation of wp for assembly languages can be done using knowledgeware that consists of hundreds of predicates and relational patterns. Further, GPSTP execution speeds may allow these functions to be performed in real time. This is essential for verification of code in the presence of previously undetected malware.

D. Capability 4.

The integrity of data is an essential requirement for any system. Achieving data integrity requires having the knowledge of the constraints on the programs that operate on the data. These constraints are a property of the data which all programs are supposed to, and ensuring that all programs maintain these constraints. This includes requires establishing the conditions for invariants and associated integrity constraints applied to data in addition to computer programs. The integrity constraints define the scope of applicability for systems that access multiple databases and ontologies. These constraints need to be rich enough to take into account how to preserve implicit write sequences when multiple systems can access the database. The possibility of forming unique signatures with invariants found in database management software, schemas and data dictionaries needs to be examined.

E. Capability 5.

Abstract data types are a promising vehicle for establishing a coherent relationship between a) the code/specification world using the entities in the program state space and b) the domain-oriented ontology. Such unified ontology spans from executable code up to the user's view. This may require the application of the rules of the refinement calculus using abstract data types [6].

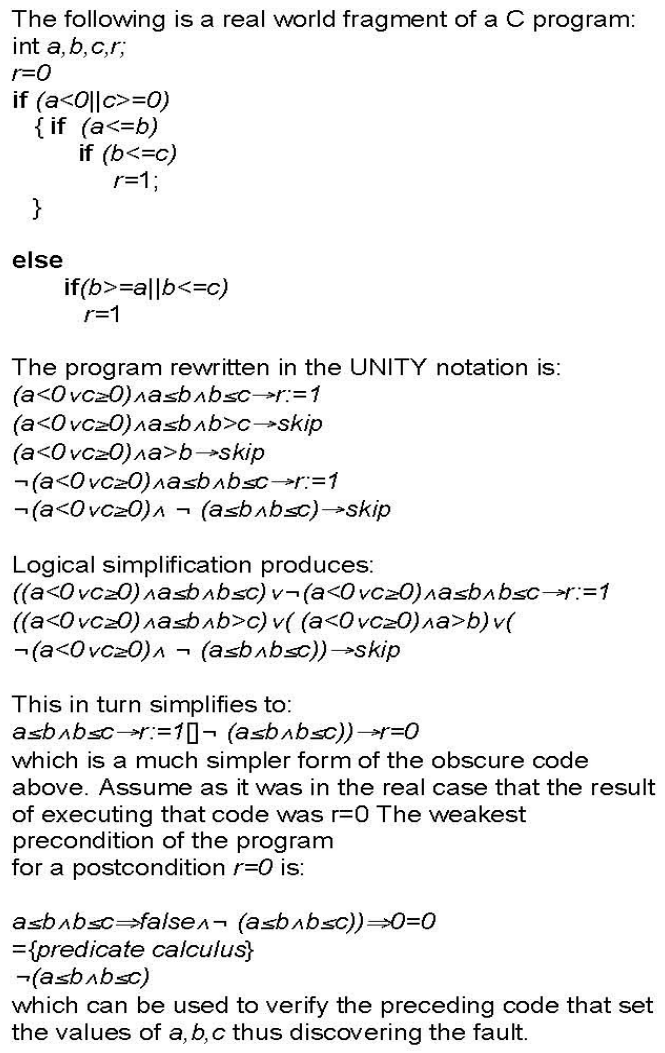
F. Capability 6.

The GPSTP enables efficient recognition of specific patterns in a large set of non-relevant information. This processor can avoid the problem of exponential explosion caused by indices and pointers. In contrast to exhaustive, time-consuming testing this assessment method analyzes computer code as text.

G. Capability 7.

Systems that interact with an environment and that maintain permanent data are not computable functions. The UNITY computation can represent and analyze this kind of system. As long as one deals with a finite environment these can be represented in the UNITY format. This format is essentially a set of (condition)→(action) statements that are assumed as executing in any order and/or simultaneously. An important result in UNITY is the "union theorem" which allows the determination of the properties of two juxtaposed sets of (condition)→(action) statements from the knowledge of the properties of each set. Furthermore it is possible to model the composition of the Hardware/Software system described in the UNITY form with any finite environment that can be represented in UNITY as well. This can be done by simply imagining that the unpredictable actions of the environment are represented by (true)→(action) where true is the constant predicate.

A SIMPLE EXAMPLE



The following is a real world fragment of a C program:

```
int a,b,c,r;
r=0
if (a<0||c>=0)
  { if  (a<=b)
       if (b<=c)
          r=1;
  }

else
      if(b>=a||b<=c)
        r=1
```

The program rewritten in the UNITY notation is:

$(a<0 \vee c\geq 0)\wedge a\leq b\wedge b\leq c \rightarrow r:=1$

$(a<0 \vee c\geq 0)\wedge a\leq b\wedge b>c \rightarrow skip$

$(a<0 \vee c\geq 0)\wedge a>b \rightarrow skip$

$\neg(a<0 \vee c\geq 0)\wedge a\leq b\wedge b\leq c \rightarrow r:=1$

$\neg(a<0 \vee c\geq 0)\wedge \neg\ (a\leq b\wedge b\leq c) \rightarrow skip$

Logical simplification produces:

$((a<0 \vee c\geq 0)\wedge a\leq b\wedge b\leq c) \vee \neg(a<0 \vee c\geq 0)\wedge a\leq b\wedge b\leq c \rightarrow r:=1$

$((a<0 \vee c\geq 0)\wedge a\leq b\wedge b>c) \vee(\ (a<0 \vee c\geq 0)\wedge a>b) \vee($

$\neg(a<0 \vee c\geq 0)\wedge \neg\ (a\leq b\wedge b\leq c)) \rightarrow skip$

This in turn simplifies to:

$a\leq b\wedge b\leq c \rightarrow r:=1 \square \neg\ (a\leq b\wedge b\leq c)) \rightarrow r=0$

which is a much simpler form of the obscure code above. Assume as it was in the real case that the result of executing that code was r=0 The weakest precondition of the program
for a postcondition $r=0$ is:

$a\leq b\wedge b\leq c \Rightarrow false \wedge \neg\ (a\leq b\wedge b\leq c)) \Rightarrow 0=0$

$=\{predicate\ calculus\}$

$\neg(a\leq b\wedge b\leq c)$

which can be used to verify the preceding code that set the values of a,b,c thus discovering the fault.

Figure 1: An example of translation in UNITY and consequent simplification then use of the wp for discovering a fault.

A Note on Simplification: Most of the time spent by any code analysis tool is to find the locations of specific operations such as the state changing operations of assignment. This requires a search that spans over all the instructions in the code. Therefore any tool to determine the position of assignments in a segment of code of length n must perform n operations (comparisons), that is, it has a complexity of O(n). Any of these searches when implemented in the GPSTP requires the equivalent of just one memory fetch, that is, O(1). A similar order of magnitude improvement is expected for logical formula simplifications. The time complexity of this type of simplification in a sequential machine is O(n*s) where n is the number of terms in the formula and s the number of simplifications possible that are available in the library (in all practical situations n<<s). In a GPSTP this complexity reduces to O(n) which in many cases makes possible an otherwise unfeasible task. More importantly the GPSTP can scan for thousands of expressions like this simultaneously at the constant speed of one memory fetch time for each character of program text.

V. Readiness Assessment form factors

Because the readiness assessment capabilities can be accomplished by code inspection only, no test beds, not test cases, no regression tests, an initial readiness assessment system can be a workstation that has access to all the code in the SoS. However, in order to confirm readiness of a dynamically configurable SoS in the field the assessment capabilities must be embedded in the SoS constituent systems. Further, it is likely that a cybersecurity scenario will demand that the embedded assessment be accomplished at the speed of execution of the operational code.

VI. Whole system assessment

The foregoing focused on the assessment of computer code in the operational environment throughout the life of the system because that is the most challenging aspect of SoS readiness assessment. Obviously, however, detecting problems among the SoS constituent systems throughout the SoS life cycle is highly desirable. The capabilities envisioned herein can be extended to detect problems in requirements, design concepts, architectures and modularization of an intended SoS from Day 2 of a development project. For example, if the language processor that produced the executable code is considered as part of the SoS and is confirmed as fit for purpose then the high level language version of the generated code can be assessed as well. Further, if the system design is expressed in a formal system modeling language such as SysML then the SoS model created early in the development stage can be assessed as well. Moreover, modern systems engineering activities prepare executable models of the problematic situation and the underlying problem system for which the SoS is to intervene. This stakeholder conceptualization of the problem system and an intervention strategy can be assessed as fit for purpose thereby avoiding subsequent unintended consequences [13]. Such assessments will not warrant as high a degree of confidence as will the assessment of the executable version but can help preclude subsequent schedule slips and cost overruns by early detection of the faults that are found.

VII. Expedited fault removal

The assessment capabilities envisioned herein make the algorithmic pattern surrounding any detected and located fault much more understandable to those who must maintain the code thereby fostering quicker removal of faults and instant confirmation of success.

Extensions are also feasible for automated composition of a SoS, starting with an answer to the question, will a given SoS be able to participate in an envisioned (larger) SoS?

VIII. Next steps

Prototypes must be prepared for investigations of Proof of Concept, Proof of Feasibilty and Proof of Value. The prototypes must be evaluated with respect to the foregoing Measures of Effectiveness.

Acknowledgements

Dr. Pizzarello thanks Profs. Djikstra, Misra and Mani Chandy for many hours of stimulating dialogue. OntoPilot LLC members acknowledge technical support from Curt Harris, Kennen Technologies LLC, regarding the GPSTP®.

References

[1] J. Ring, H. Eisner, M. Maier, “How can you 'prove' that your systems design will solve the customer's problem *before* you build and prove that design?” Fellows Issue #3, INCOSE INSIGHT, June, 2010.

[2] J. Ring, and A. Madni, Key Challenges and Opportunities in ‘Systems of Systems’ Engineering, IEEE-SMC 2005, Waikoloa, Hawaii, October 21-2005.

[3] G. Japaridze, “Introduction to Computability Logic,” Annals of Pure and Applied Logic, 2003 pp 1-99.

[4] E. W. Dijkstra, “Self stabilizing systems in spite of distributed control,” CACM, Vol17, 11, 643-644 (1974).

[5] E W Dijkstra and C. Scholten, Predicate calculus and program semantics, Springer-Verlag, 1990.

[6] R. Back and J. von Wright, Refinement Calculus, Springer Verlag, 2008.

[7] W. R. Ashby, Introduction to Cybernetics, Chapman & Hall Ltd., 1957.

[8] A. Pizzarello, US Patent 6029002: Method and apparatus for analyzing computer code using weakest precondition Feb 22, 2002.

[9] A. Pizzarello, A new method for location of software defects, AQuis93 Venice, Italy Oct 1993 (143-155).

[10] C. Harris, and J. Ring, U.S. Reg. Patent # 7392229 and Patent # 7487131 assigned to Kennen Technologies LLC.

[11] K. Mani Chandy and J. Misra, Parallel program design: A foundation, Addison-Wesley 1988.

[12] I3P – Information Institute for Infrastructure Protection, Cyber Security Research and Development Agenda, Research Area #3, p. 18, 2003.

[13] J. Warfield, Understanding Complexity: Thought and Behavior, AJAR Publishing Company, 2002, pg. 65.