MODEL-BASED VERIFICATION OF DIAGNOSTIC SYSTEMS

S. A. Brown[*] and C. Pecheur[**]
[*] Northrop Grumman Integrated Systems, El Segundo, CA
[**] RIACS, NASA Ames Research Center, Moffett Field, CA

ABSTRACT

This paper discusses problems and opportunities inherent in the verification and validation (V&V) of diagnostic (health monitoring) systems. In complex client systems it can be difficult or impossible to even define the system requirements without reference to a functional model of the client system, causing the verification requirements to themselves require verification. This circularity becomes explicit when the diagnostic system to be verified contains, or uses in its development, a functional model of the client system; in some cases identical to or derived from the functional model used to define the verification requirements. In this discussion the system verification problem is decomposed into separate verifications of the algorithms, software engines, and client system models. Formal methods are proposed as being particularly well suited to verification of models representing physical systems.

INTRODUCTION

Modern systems for the diagnostic assessment (health management) of complex systems make extensive use of functional models to encode their knowledge of the client system. In some algorithms (e.g. Model Based Reasoning) this model is explicit, while in others (e.g. neural nets) the model may be implicit through use in the training of the net. Although this paper focuses on diagnostic systems in particular, the discussion presented can be generalized to a large class of reasoning systems based on functional models.

Verification and validation of diagnostic[1] systems is critical to the success of future aerospace systems, because it is expected that these systems will increasingly be used to trigger crew/flight/mission safety critical decisions. In addition, the actual costs associated with informed maintenance will be strongly influenced by how well informed that maintenance is.

The first part of this paper discusses fundamental diagnostic system requirements, which should, in principal, be defined in terms of explicit descriptions of the system nominal and failure modes, in either the infinitely-dimensioned client system space or the finite-dimensioned diagnostic system space. Unfortunately, it is difficult to create an explicit detailed representation of modes in either space, because the parameter values associated with each mode are themselves complex functions of other parameters, and some or all of these parameters may be infinitely valued.

The second part of this paper discusses shifting the focus of verification from verification of the system as a whole to verification of client system specific model(s) used within the system. These models may appear

- explicitly, in run-time algorithms, e.g. model-based reasoning (MBR), or
- implicitly, when used at design time to train or develop algorithms, e.g. neural nets or Kalman filters.

---

[1] Although this paper focuses on diagnostic modeling, the discussion is also relevant to prognostic modeling when the algorithms used are based on explicit or implicit client system models.

This work was performed under NASA contract no. NAS8-01100 with the 2nd Generation Reusable Launch Vehicle Risk Reduction Program, Task Area TA-5, Integrated Vehicle Health Management, Marshal Space Flight Center, Huntsville, AL.

In addition, the diagnostic system requirements may themselves be defined by functional models of system behavior. These definition models will themselves then require validation against the client system design.

VERIFICATION TO DESIGN REQUIREMENTS

The NASA Goddard Independent Verification and Validation Group defines Verification and Validation as:

> *Verification* asks, "Is the product being built right?" It is the process of determining whether or not the products of a given phase of the software development cycle fulfill the established requirements.
>
> *Validation* asks, "Is the right product being built?" It evaluates software at the end of the development lifecycle to ensure that the product not only complies with standard safety requirements and the specific criteria set forth by the customer, but performs exactly as expected[2].

Validation compares the finished product against top-level customer requirements, as discussed in the next section of this paper. Verification does the same, except it compares each phase and/or component in the system development process to a set of derived requirements defined for each phase and/or component. *In verification, the correctness criteria applied at the end of each phase are that the system or component meets the design requirements imposed at the beginning of that phase*. Verification is simpler, in that the requirements for each phase are usually more explicit than the overall validation requirements. On the other hand, this adds the problem of validating the requirements generated for each phase.

PROBLEM STATEMENT IN DIAGNOSTIC SPACE {*D*}

The diagnostic problem consists of examining an engineering system (the client) and determining whether its current state corresponds to one or more distinct failure modes[3]. In particular, a diagnostic system attempts to divide all possible states of the system into:

1. States which represent nominal modes of the system (all functions responding within required parameters).
2. States which represent off-nominal modes of the system, including:
   a) States which represent one or more defined failure modes of the system,
   b) States which represent one or more undefined failure modes of the system[4], and
   c) States which represent some mixture of a) and b).

Ideally a diagnostic system would know everything about the client system; i.e. it would carry exact values for the infinite number of parameters that could be used to describe the client state. In practice, of course, this is impossible. Instead the diagnostic system will define the client system in terms of a finite set of diagnostic parameters $\{D\} = \{c\} + \{s\} + \{u\}$, where:

$\{c\}$ are control parameters with known or imposed values. These values are taken to be known with certainty.

$\{s\}$ are parameters that can be sensed. These values may be taken to contain some element of uncertainty (e.g. due to sensor noise, or possible sensor failure). If $\{s\}$ is assumed known with certainty then the distinction between $\{s\}$ and $\{c\}$ becomes merely semantic.

$\{u\}$ are parameters that must be inferred from $\{c\} + \{s\}$, and/or evolved through the prior history of the system. These are essentially the unknowns for which the diagnostic system is solving.

---

[2] http://www.ivv.nasa.gov/faq/index.shtml

[3] Two modes are distinct if they are distinguishable, given all possible information about the client system. We will assume that, by definition, all failure modes are distinct from all nominal (non-failure) modes.

[4] The client is considered to be in an undefined failure mode when its current state is determined to be off-nominal, but does not correspond to any known failure mode either.

In general a diagnostic parameter

- may be enumerated, integral, or continuous
- need not be finite in range
- need not be independent of the other diagnostic parameters.

In principal every valid[5] point in $\{D\}$ can be classified as belonging to

a) one or more nominal modes of operation, and/or
b) one or more failed modes of operation, or
c) an undefined mode of operation; i.e. neither a) nor b).

A given point in space may belong to multiple modes, because the diagnostic space is limited in representation, mode definitions may contain some uncertainty, and our presumed point in the space may contain some uncertainty.

Defining the diagnostic problem requirements then hinges on our ability to define nominal and failure modes, each of which corresponds to some region in $\{D\}$. As one simple example, a given mode definition might be valid over a fixed range of some subset of these parameters $\{d\} \subseteq \{D\}$. The mode definition itself would then typically define nominal ranges for the complimentary set $\{n\} = \{D\} - \{d\}$.

Example: Assume the diagnostic space $\{D\}$ = {*launch_phase*, *temperature_1*, *temperature_2*}, where *launch_phase* can take on the enumerated values [*on_pad, lift_off, cleared_tower*].

- If $\{d\}$ = {*launch_phase*}, then a particular nominal mode could be defined as limits on the complementary set {*temperature_1*, *temperature_2*} for *launch_phase* = *lift_off* and *launch_phase* = *cleared_tower*.
- If ((*launch_phase* = *lift_off*) or (*launch_phase* = *cleared_tower*)) and either *temperature_1* or *temperature_2* were outside of the nominal range, the state would be considered anomalous; that is, we know what nominal should look like for these launch phases, and the current state does not match.
- In the subspace corresponding to *launch_phase* = *on_pad* the mode would be undefined.

In this simple example the hyper-rectangle in $\{D\}$ defined by *launch_phase* ∈ [*on_pad, lift_off*] is divided into two regions, nominal and anomalous. Nothing is said about the space outside of this hyper-rectangle (it is neither nominal nor anomalous to this mode).

Unfortunately, real modes are typically much more complex in topology, making them very difficult to define directly as regions in $\{D\}$. A more common practice is to provide a functional model $F(\,)$, or the information required to build such a model, including the ability to turn failure modes on or off. The model itself is defined over $\{D\}$, and typically allows the computation of $\{s\}$ and part or all of $\{u\}$, given $\{c\}$ and the complementary part of $\{u\}$:

$$F(\{c\}, \{u\} \subseteq \{u\}, \{\textit{failure modes}\}) \rightarrow \{s\}, \{u\} - \{u\}$$

- Nominal modes are defined by the system response when {*failure modes*} = $\{\varnothing\}$.
- Anomalous and undefined modes are both characterized by points that cannot be resolved by (are not valid in) the functional model. Most functional models are not robust enough to distinguish between these two cases.

---

[5] Some points in $\{D\}$ may not be physically valid for the client system represented. For example, if two parameters defined for the system corresponded to mean flow rates at the inlet and outlet of a small volume pump, then any region of $\{D\}$ where the flow parameter values were not equal would represent physically invalid regions of $\{D\}$. Alternatively these could be modeled as valid points corresponding to a failure in either or both of the flow rate sensors.

VERIFICATION OF THE DIAGNOSTIC SYSTEM REQUIREMENTS

The true client system is defined by an infinite number of possible engineering parameters, covering both its internal state and its environment (external state). The diagnostic system approximates this infinite parameter space with the finite dimensional diagnostic space {*D*}, which implies that the diagnostic system can never exactly express the full range of the client system. General verification of the diagnostic system thus consists of two problems:

1. *Verification of the Diagnostic System Requirements*: Verifying that the diagnostic space, and the problem description defined in terms of that space, are adequate to express the requirements demanded of the actual client system. This requirement can, in turn, be broken down into two sub-requirements:
   a) Verifying that the diagnostic space, and the problem description defined in terms of that space, are adequate to express the requirements demanded of the actual client system *as those requirements are understood at design time*.
   b) Verifying that the actual deployed client system (including its environment) match the design time understanding of that system to an acceptable degree.

2. *Verification of the Implemented Diagnostic System*: Verifying that the implemented diagnostic system fulfills the requirements given *as defined within that diagnostic space*.

In general, verification of the implemented system against the diagnostic system design requirements (2) is a more tractable problem than verifying the design requirements against the actual client system requirements (1).

FUNDAMENTAL (BLACK BOX) REQUIREMENTS

The fundamental requirements of a diagnostic system can, in principal, be defined without regard to the manner in which the requirements are implemented; i.e. treating the diagnostic system as a black box that infers (dynamically over time) current modes based on values of the known parameters {*c*} + {s}.

The first set of requirements define correct operation of the diagnostic algorithm(s):

1. *Recognition of Nominal Mode(s)*
   For all possible states of the system *for which nominal mode descriptions have been provided*, the system shall correctly identify all states that correspond to defined nominal modes.

2. *Recognition of Anomalous Modes*
   For all possible states of the system *for which anomalous mode descriptions have been provided*, the system shall correctly identify all states that correspond to defined anomalous modes. Anomalous modes may be defined explicitly, or implicitly by defining a certain definition space for a nominal mode, and then defining a subspace of that to be nominal.

3. *Nominal Robustness*
   For all possible states of the system, the system shall correctly identify all states that do not correspond to any defined nominal or anomalous mode (i.e. ones for which no nominal mode descriptions have been provided). Nominal robustness prevents the system from reporting an anomalous mode simply because it does not know what nominal would look like in this state. Instead the system should report undefined state . In the example given above, if this were the only nominal mode defined then states with the parameter *launch_phase* = *on_pad* would be considered neither nominal nor anomalous; they are simply undefined.

4. *Fault Recognition*
   For all possible states of the system *for which description(s) of a given failure mode have been provided*, the system shall correctly identify all states that correspond to the defined failure mode.

5. *False Alarm Avoidance*
   For all possible states of the system *for which description(s) of a given failure mode have been provided*, the system shall correctly identify all states that do not correspond to the defined failure mode.

6. *Fault Robustness*
   For all possible states of the system, the system shall correctly identify all states for which a given failure mode has not been defined. In these cases the system should not report the failure mode as either active or not; it should properly report that it has no knowledge of the failure mode in this state.

The robustness requirements (3 & 6) simply formalize the understanding that for any given point in {*D*} we may say one of three mutually exclusive things about a particular defined mode:

- The point *does* correspond to the defined mode (e.g. the failure *has* occurred)
- The point *does not* correspond to the defined mode (e.g. the failure has *not* occurred).
- The presence or absence of the mode is *not defined* (visible) at that point (nothing can be said about the failure at this point).

Assuming that requirements 1-6 are all met, we can then define an additional *desirable* condition for the system:

Ideal Fault Isolation
For all possible states of the system, there should

- never be a state which corresponds to one or more defined nominal modes *and* one or more defined failure modes (nominal/failure ambiguity)
- never be a state which corresponds to two or more distinct failure modes (failure ambiguity) unless the ambiguities in question have been specifically allowed

It is important to recognize that, while requirements 1-6 define correctness of the algorithm, the maximum possible extent to which fault isolation can be achieved is a property of the problem definition itself. The diagnostic system space {*D*} spans a finite number of parameters from the infinite number possible, and continuous parameters will be represented by a finite set of values in {*D*}. It is quite common to have cases in which, for some region(s) of the diagnostic space, a theoretically distinct failure mode overlaps one or more different nominal and/or failure modes due to this condensation of the state space.

- Given this diagnostic space and problem definition, *no possible algorithm can isolate beyond this ambiguity group*.
- Failure of an algorithm to meet requirements 1-6 can, however, increase the level of ambiguity beyond the minimum possible.

Thus a feasible requirement on fault isolation must be worded something like:

7. *Fault Isolation*
   For all possible states of the system, to the limit possible within the given diagnostic state representation {*D*}, there shall
   - never be a state which corresponds to one or more defined nominal modes and one or more defined failure modes (nominal/failure ambiguity)
   - never be a state which corresponds to two or more distinct failure modes (failure ambiguity) unless the ambiguities in question have been specifically allowed.

In principal it is always possible to disambiguate two modes, given additional system information necessary and sufficient to the disambiguation in question. Put another way, if the diagnostic space spanned all possible system parameters, all distinct modes could, by definition, be disambiguated. In practice what this means is that even if a given diagnostic space cannot disambiguate a set of modes, sometimes a second system can by employing a different diagnostic space.

Diagnosing Under Uncertainty: For the purposes of this discussion we are treating modes and mode sensing as deterministic. In practice, however, real diagnostic systems typically include uncertainty in various ways; e.g.

- Sensor readings of physical quantities, such as strain or temperature, introduce noise errors.
- Mode signatures may themselves contain some degree of uncertainty in some or all of the measured parameters.
- Some diagnosis algorithms, in addition to simply reporting all modes consistent with the current belief state, also track the relative likelihood of these modes based on an evolution from a priori estimates and/or the probabilistic matching between modes and sensed parameters.

In practice these issues tend to have two effects on the results of diagnoses:

- Fault isolation groups get larger, and
- Faults within a group can be assigned relative probabilities (confidence).

Fortunately, these kinds of uncertainty do not significantly alter either the arguments or the conclusions of this discussion, and so this aspect has been ignored in the rest of this discussion.

DERIVED (IMPLEMENTATION SPECIFIC) REQUIREMENTS

Unfortunately, the only direct way to prove the fundamental requirements 1-7 would be to examine every possible point in the system space (exhaustive search). Even if we restrict ourselves to verifying the diagnostic system against its design requirements, the number of test points in $\{D\}$ typically ranges from large to infinite.

In addition, as discussed above, real systems are generally too complex to admit explicit definitions of the nominal and failure modes. Mode definitions themselves are given in the form of a client system functional model, typically of the form:

$$F(\{c\}, \{u\} \subseteq \{u\}, \{\textit{failure modes}\}) \rightarrow \{s\}, \{u\} — \{u\}$$

In order to determine what failure modes are associated with a given point in $\{D\}$ the functional model must somehow be back-solved for {*failure modes*}; but this brings us right back to the original diagnostic problem. Since the definition of modes is central to stating the requirements for a diagnostic system, we are now faced with this conundrum:

> *When mode definitions are given implicitly by a client system functional model, extraction of these modes for requirements validation becomes a problem equivalent to the diagnostic problem originally posed.*

In other words, the requirements model is itself equivalent to the diagnostic model, and so must itself be verified. But, by definition, the requirements *are* the verification standard!

An alternative approach that indirectly verifies the fundamental requirements can be applied when three conditions are met:

1. *Algorithm(s)*: The diagnostic system employs a specific set of algorithms, methods and procedures that are known to meet requirements 1-7, so long as the problem specific inputs to the system (the model) are correct.
2. *Engine(s)*: The specific implementation of these algorithms, methods and procedures, is shown to be correct.
3. *Model(s)*: The problem specific model is shown to be correct. In particular, the model is both
   a) semantically correct, i.e. correctly represents the client system to an acceptable level of fidelity, and
   b) syntactically correct, i.e. correctly represents the client system in terms of the specific language and assumptions of the selected algorithms and engine.

Validation of any specific algorithm set is beyond the scope of our current TA-5 program. In practice certain algorithms are usually trusted , and accepted without further validation.

The level of validation typically applied to a specific implementation (diagnostic engine) depends on both its trust level and the tools available for validation. Because the diagnostic engine is typically written in a general purpose programming language, it can be very difficult to prove its correctness. Engines are normally validated through general SEI-type practices (i.e. written under a formal Software Development Plan, validated through some kind of test suite). In addition, one of the authors has been developing an automated testing/simulation tool for Livingstone (Livingstone PathFinder), which could be adapted to test one implementation of a specific diagnostic engine (Livingstone) against another trusted implementation.

Under the assumptions listed above, it should be clear that it is the client system functional model which contains all of the diagnostic system s knowledge about client system specific nominal and failure modes. The way in which that knowledge is represented, and what constitutes verification of that knowledge, depends on the specific form of the model (which is, in turn, a function of the specific algorithm and engine to be applied). In addition to the diagnostic functional model, which is used explicitly or implicitly in the diagnostic system, there may be a separate functional model used to define the requirements for the diagnostic system[6]. In this case both models need to be verified; the definitional model to the true client system, and the diagnostic model to the definitional model.

Verification of the model to the engine is primarily a matter of syntax; i.e. validating that the specific representation of the model provided to the engine will be interpreted as intended with reference to the underlying algorithm. This leaves only verification of the model with respect to the original design problem (client system), where verification is understood to be within the context of the underlying algorithms to be applied.

DEFICIENT ALGORITHMS/MODEL FORMS

Above, it was stated that the algorithm and implementation applied are required to meet fundamental requirements˚1-7, so long as the model itself is correct. In practice we frequently use diagnostic algorithms, engines, and/or models that are known to fail these requirements in some way. Two common examples of deficiencies are:

- Model forms/algorithms that are unable to distinguish between anomalous and undefined mode spaces.
- Completeness of diagnosis (the ability to find all possible modes) may be compromised for performance purposes. These compromises are necessary, as the fundamental diagnosis problem is commonly of very high computational complexity.

In each of these cases, verification of the system has to identify those deficiencies, which then must be either corrected (if correctable) or accepted (if inherent to the algorithm).

MODEL VERIFICATION AND VALIDATION: FUNCTIONAL (CLIENT SYSTEM) MODELS

Above it was claimed that, by using trusted algorithms and engines, the diagnostic system verification problem could be reduced to verification of its client system specific model. In particular, algorithms such as Model-Based Reasoning (MBR) represent the nominal and failure modes of a system by constructing a functional model of the physical system itself.

- The parameters of this model constitute the diagnostic space $\{D\}$.
- The absence or presence, and perhaps severity of each defined root failure mode is typically a parameter of the model, with the observable effects of the failure on the system (fault signature) generated through causality relations built into the model.

---

[6] In principal there are always two models, one defining the diagnostic system requirements and one implementing those requirements. Because they are similar in function, however, it is not uncommon to build a single model to serve both ends. In this case the diagnostic model verifies to the requirements model by definition, and all of the focus is on verification of the requirements model to the actual client system.

- Nominal modes are defined as the system response in the absence of any defined failures.
- The diagnostic model may be validated only over some subspace $\{d\} \subseteq \{D\}$, typically defined as some expected range of external environment and commanded states. One of the characteristics of a well-behaved model is the ability to recognize when the values of $\{d\}$ are out of a range for which the functional model is known to be valid (robustness requirement).

Because functional models directly model the physics of the client system, they have some interesting verification characteristics. In particular:

1. The diagnostic model itself may be the only extant detailed definition of the system s normal and failure modes. If there *is* an external definition of the client system modes, it will typically be in the form of an even more elaborate functional model (which would itself require verification).
2. Validation of the diagnostic system requirements against the client system requirements, at design time or in operational use, consists of validating the functional model against the client system.

Based on the above arguments, we will now claim the following:

Verification of a diagnostic system based on a functional model can be reduced to:

1. *Universal verification (or trust) of the diagnostic algorithms to be employed.*
2. *Universal verification (or trust) of the specific (software) implementation of these algorithms.*
3. *Specific verification of the client system functional model(s) against the physical characteristics of the client system, including:*
    a) *verification that the representation of the client system is correct (within the limitations of the given model form), and*
    b) *verification that the representation of the client system is adequate to express the behaviors of interest, including specifically the response of sensed observable parameters to internal and external system states*.

Verifying correctness of the functional model can include the use of formal methods to prove certain characteristics and/or invariants that are known to be true of the client system, either because they are true of the client system specifically, or because they are true for *all* physical systems. The physical world, including the client system, is always self-consistent, and there are a number of physical laws (e.g. conservation of mass) which must always hold. This means that large regions of the parameter space $\{D\}$ can be declared invalid, and any legal transition within the model from a valid state to an invalid one represents an error in the model.

Design verification of the diagnostic system against its client system is more difficult, because it requires comparison to an independent and more trusted representation of that system. Ultimately verifying the design understanding of the system against the physical reality can occur only after a sufficient number of deployed systems have seen service, to establish both baseline behavior and variance over the life of the system.

Finally, although the arguments of this section were focused on diagnostic models that are directly based on functional models, they apply to some extent to any model that uses a functional model in its construction. For example, a neural net trained using a simulation (functional model) of the client system will, in general, not be valid unless the functional model used for training is also valid.

CONCLUSIONS

In the development of diagnostic systems both the overall validation requirements and the specific verification requirements are expected to be of the form:

> *Given a set of sensible parameters {s} $\subseteq$ {D}, detect failure mode x (if present), isolated to within the fault isolation group { y }*

Unfortunately, the normal and failure modes of the system will typically be described by reference to a functional model of the client system. Diagnostic systems which themselves operate on a functional model of the client system are, therefore, well suited to solving the stated problem; but are difficult to verify, because they create the detailed requirements specification within their own implementation. They are, in as sense, self-verifying, but only if the detailed requirements can be validated against the actual client system. Two general approaches to this problem have been identified as applicable to the design phase:

1. Quality software processes, augmented by formal methods applied to the functional model. Formal methods can be used to verify that certain universal and problem specific constraints and invariants are met for all applicable states of the model.
2. When available, simulation-based testing. This involves presenting the diagnostic model with a sequence (one or more) of inputs (control and/or sensible parameters), and then comparing its resulting diagnoses against results which are either known to be correct, or are at least trusted to a higher degree than the diagnostic model. Selection of test points for maximal utility or coverage can be developed automatically using formal methods based procedures.

Final validation of the system requires operational deployment of the client system, in order to generate test sets which are certain to be correct. Even in this case, however, the client system may never explore all possible failure states, and discovering incorrectly represented failure states may have catastrophic costs. For these reasons our primary focus will remain on up front design verification of the system, despite its inherent difficulty.