Modeling Diagnostic Constraints with AI-ESTATE

John W. Sheppard
ARINC Incorporated
2551 Riva Road
Annapolis, Maryland 21401 USA
sheppard@arinc.com

Jonas Åstrand
Chalmers University of Technology
Kronotorpsgaten 5
S-418 77 Goteborg, Sweden
d0jon@dtek.chalmers.se

Abstract: **The Artificial Intelligence and Expert System Tie to Automatic Test Equipment (AI-ESTATE) subcommittee of the IEEE Standards Coordinating Committee 20 (SCC20) has been developing a set of standards for exchanging diagnostic knowledge in intelligent test systems. To date, AI-ESTATE has developed models for fault trees and enhanced diagnostic inference models (EDIMs). Since the start of committee work, it was believed that AI-ESTATE needed to address the issue of defining constraint knowledge, which could be used to guide and refine diagnostics. In this paper, we discuss early efforts by AI-ESTATE to define such a constraint model.**

I. INTRODUCTION

The AI-ESTATE standard P1232 is being developed to standardize the interfaces between test systems and artificial intelligence based systems [1]. In addition, AI-ESTATE is including standard representations for several types of knowledge bases and databases. So far, the standard has focused on representing static diagnostic strategies, in the form of fault trees, and a basic inference mechanism, the EDIM, usable by several types of reasoning systems [2].

AI-ESTATE is being developed using a cooperative processing model. Under this model, all processes communicate across a communications pathway or bus and access other parts of the system through a set of services. The objects communicating in an AI-ESTATE system include a test system, a reasoner, a human presentation system, a maintenance data collection system, a unit under test (UUT), and an operating system. Any data used by the objects on the pathway will be specified in some standard representation or accessed through a standard set of services.

The current model representations defined by the AI-ESTATE P1232.1 Data and Knowledge Specification are specified using the Express modeling language. Currently, AI-ESTATE has specified three models including a common element model (a collection of entities and attributes used by all models), a fault tree model (a standard representation for a static fault tree), and the Enhanced Diagnostic Inference Model (a standard representation that extends the concepts of the dependency model and the information flow model [3]). This paper introduces work being done by the AI-ESTATE committee to produce a new model representation—the diagnostic constraint model.

II. MODELING DIAGNOSTIC CONSTRAINTS

The model described in this paper is derived from an approach for expressing diagnostic knowledge in terms of constraints. The model is of sufficient power to express all important diagnostic constraints to be imposed on a test system, and the model is consistent with current approaches to modeling constraint satisfaction problems (CSPs) in the literature [4,5]. The specification was also designed to facilitate ease of use and natural expression of diagnostic knowledge.

Constraint-based reasoning is a paradigm for formulating and reasoning about knowledge as a set of constraints without specifying the method by which these constraints are to be satisfied [4]. Formally a constraint satisfaction problem is composed of:

- A set of variables $\mathbf{X} = \{x_1, x_2, \ldots, x_n\}$ and their related domains $\mathbf{D} = \{d_1, d_2, \ldots, d_n\}$. Where each x_i takes values from its associated domain d_i (i.e., $x_i = d_i^j$, $d_i^j \in d_i$).

- A set of constraints $\mathbf{C} = \{c_1, c_2, \ldots, c_m\}$. Each of the constraints is expressed as a relation defined on some subset of variables.

The goal of reasoning by constraint satisfaction is to assign a unique domain value to each variable without violating any of the given constraints [6]. In system test and diagnosis, we have two sets of *variables*:

- **DIAG** = $\{diag_1, diag_2, \ldots, diag_n\}$ a set of *diagnosis variables.*

- **TEST** = $\{test_1, test_2, \ldots, test_m\}$ a set of *test variables.*

DIAG contains every diagnosis that can be made on the system. Each diagnosis is tied to a specific system aspect or

0-7803-2621-0/95 $3.00

group of system aspects in the UUT. The value of each diagnosis can be *good*, *bad*, or *unknown*. This means that all $diag_i$ have the same domain d^{diag} = {good, bad, unknown}. The goal of diagnosis is to assign each $diag_i$ a value $d_i \in d^{diag}$, where d_i = good identifies a unit in proper working order, d_i = bad identifies a malfunctioning unit, and d_i = unknown identifies a unit whose failure state is indeterminate.

TEST contains all the tests that can be performed on the UUT. Each $test_i$ can be assigned a value from its own domain (the different outcomes of the test) d_i, i.e., $test_i = d_i$ where $d_i \in d_i^{test}$. Test values are assigned to $test_i$ based on testing and inference which in turn ultimately assign values to each $diag_i$. In the current model, these two sets of variables are constrained by three types of constraints–logical, temporal, and global. Each of these types of constraints are discussed in the following sections.

III. SOLVING CONSTRAINT SATISFACTION PROBLEMS

Several procedures exist for solving CSPs. Usually these procedures are categorized as exhaustive, consistency-based, or structure-driven. The most common exhaustive search procedure is backtracking, in which labels are assigned following depth-first search. If a value assignment violates one or more constraints, the search backtracks to the point causing the inconsistency and continues with a new label. This procedure is the easiest to implement, but is computationally the most expensive.

The most common consistency-based procedure, called arc-consistency, is based on the concept of i-consistency. The definition of i-consistency is recursive. For example, 2-consistency verifies that for any value assigned to a single variable within that variable's domain, the value assignment of any other single variable in the network is consistent with the constraint (if any) between those two variables. This is the base case of the definition. In general, i-consistency states that a locally consistent assignment of any $i - 1$ values to the variables in a constraint network (CN) is consistent with all constraints between those $i - 1$ variables and any i^{th} variable. Arc-consistency is simply 2-consistency. Path-consistency is 3-consistency.

Unfortunately, i-consistency has computational complexity that is exponential in i, so arc-consistency or path-consistency is generally all that is implemented. Further, consistency-based methods are usually combined with backtracking and heuristics to transform the problem and minimize backtracks. For many problems, the transformation to eliminate backtracks (or dead ends) is just as expensive as general i-consistency. However, it has been demonstrated empirically that arc-consistency generally eliminates a large number of dead ends, and path-consistency generally eliminates almost all dead ends [4]. Backtracking is then applied to the result.

Structure-driven approaches provide a framework for implementing either exhaustive or consistency-based algorithms. Their advantage comes in applying graph-theoretic techniques to perform the transformations mentioned.

Graph theory suggests that any network can be transformed into an equivalent binary network [4]. A binary constraint network is defined to be a CN in which every constraint subset, S_i, involves at most two variables (i.e., $|S_i| \leq 2$). A primal constraint graph is a binary CN in which variables are nodes, and edges exist between any two nodes with a constraint between them. General CNs where $|S_i| > 2$ require representation as a hypergraph unless the transformation to a binary CN takes place. A dual-constraint graph represents each constraint subset, S_i, as a node and associates an edge between any two nodes for which the constraint subsets share variables. This representation provides an easy transformation of any CN to an equivalent binary CN.

Next, a binary CN that is a tree (i.e., no cycles exist in the graph) can be solved in linear time. Several approaches exist for converting a CN to binary form, applying arc- or path-consistency, and solving the resulting network by applying backtracking.

Because of the complexity of the diagnosis problem, anything that reduces the number of calculations will improve diagnostic performance. The constraint satisfaction model provides this capability, but CSP algorithms also have computational limitations. In particular, labeling a constraint graph with n nodes by using backtracking can require up to $O(2^n)$ time to solve. If we apply i-consistency to a binary constraint network of n nodes, the time complexity is $O(n2^i)$ given no backtracking. For $i << n$, this provides a tremendous savings over pure backtracking if no dead ends are encountered, but reduced complexity can be guaranteed (without transformation of the CN) only if $i = n$. Otherwise, complexity associated with backtracking from a dead end must be traded off with level of consistency. As mentioned above, typically $i = 2$ is sufficient to cover most dead ends, and $i = 3$ covers almost all dead ends.

Another area that affects complexity is the definition of the constraints. Well-defined constraints simplify computations by pruning the search space in the most optimum way. Heuristics exist for processing constraints (i.e., expanding nodes in the CN) that provide insight into the definition of constraints.

These heuristics include the following:

- Modify constraints in the network by using current value assignments. This approach is called constraint propagation.
- Instantiate the variable (expand the node) that participates in the most constraints. This approach has the effect of selecting a variable that constrains the search space the most.
- Assign a value to the selected variable that results in the greatest number of options for instantiation of remaining variables. Otherwise, the probability of finding a dead end increases.
- As dead ends are encountered, add new constraints that characterize the encountered conflict. As a result, future encounters of this conflict will be avoided.

These heuristics suggest that constraints having as many variables as possible should be defined, resulting in maximum reductions of the search space. On the other hand, variable assignments should be made that provide minimal constraints to maximize options in labeling the constraint network. These two objectives appear to be inconsistent, but they are not. Nodes (variables) are labeled with values from the allowable domain, and these labels restrict other labels that can be assigned downstream. Applying maximally restrictive labels may result in over-constraint of the search space. This could result in missing feasible solutions. As shown in Fig. 1, labeling the variables in constraint C_1 more restrictively eliminates the portion of the search space containing the solution.

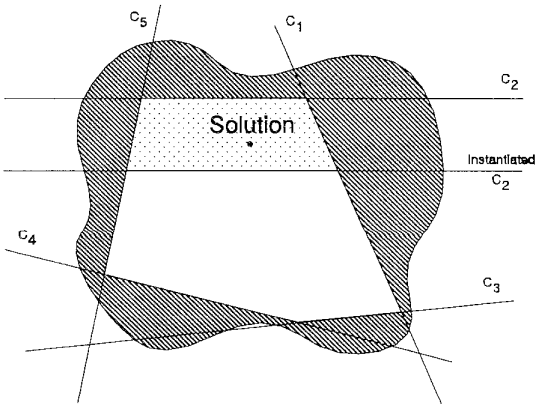

Fig. 1 Over-Restrictive Variable Labeling

In defining the constraints for a problem, we actually define the search space for the problem. Therefore, the constraints must be general enough to include all of the solutions. Constraints that eliminate solutions do not represent the true search space and indicate errors in modeling. On the other hand, constraints that are too general unduly increase the size of the search space, thus increasing the complexity of the algorithm. In Fig. 2, the appropriate constraints for the problem are represented by lines labeled C_1 to C_5. The lines labeled S_1 to S_5 represent constraints that are too general and include an inordinately large search space.

Determining proper constraints is a modeling problem. The easiest constraints to specify are numeric constraints, and if all constraints are numeric and linear, linear programming can be applied. For nonlinear numeric constraints, other "programming" algorithms may be applicable, including dynamic programming, integer programming, gradient descent, and polynomial programming. Diagnosis frequently limits test outcomes to discrete values (e.g., pass or fail) and fault candidate to either failed or not-failed. This would tend to indicate that integer programming would be applicable.

IV. LOGICAL CONSTRAINTS

We call the first type of constraint the "logical constraint." In the current draft of the P1232.1 standard [2], the model of the enhanced diagnostic inference model (EDIM) is intended to address most logical inference problems. However, it is possible that some diagnostic reasoners will operate from an equivalent constraint model. The logical constraints provide an equivalent representation for many of the constructs in the EDIM using the CSP formalism.

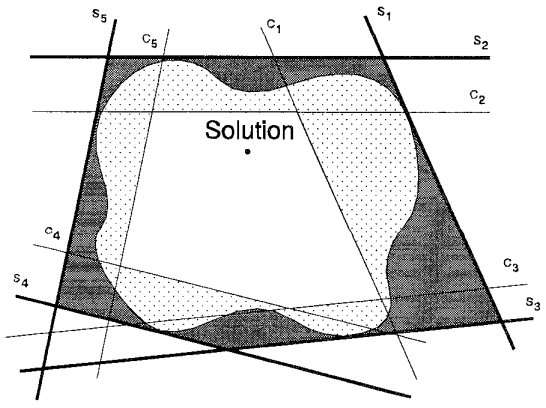

Fig. 2 Under-Restrictive Constraint Specification

Let **CL** = $\{cl_1, cl_2, \ldots, cl_p\}$ be a set of *logical constraints* where a logical constraint is a constraint that has to be satisfied to be able to assign a value to a variable. Each cl_i = $[(v,d), \{(v_1,d_1), (v_2,d_2), \ldots, (v_n,d_n)\}]$ consists of (a) a single attribute-value pair where the attribute corresponds to the variable v, and the value of the attribute is d (this value can be assigned only if the logical constraint is true) and (b) a set of pairs (v_i,d_i) of variables and their associated values. All pairs in the set (b) must be valid to make the assignments in set (a).

A. Example 1

Suppose we have the constraint cl_i = [($test_5$, fail), {($test_2$, fail)}]. This can be interpreted as the following: If $test_2$ has outcome fail, constraint cl_i is true, resulting in assigning $test_5$ the outcome fail. Of course, a value can always be assigned to a test variable simply by performing the associated test. But the constraints make it possible to express inference between tests or diagnoses.

B. Example 2

Suppose $diag_i$ and $diag_j$ are diagnosis variables, $test_p$ and $test_q$ are test variables, and we define the logical constraint cl_i = [($diag_i$, good), {($test_p$, pass), ($test_q$, 7.8), ($diag_j$, good)}]. This constraint can be translated to: $diag_i$ is good if $test_p$ has outcome pass, $test_q$ has outcome 7.8, and $diag_q$ is good.

C. Example 3

Assume we know the following:

If $test_5$ passes then $diag_2$ is good and $test_6$ will have outcome 10.3.
If $test_5$ fails then $diag_2$ is bad and $test_7$ will pass.

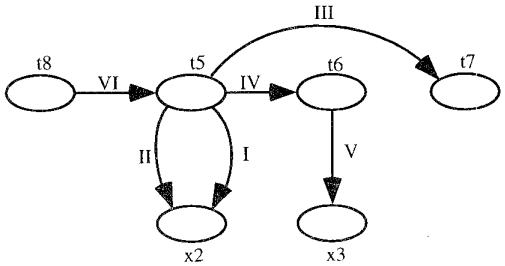

Fig. 3 Diagnostic Constraint Graph

These rules can be expressed with the following constraints:

I. [($diag_2$, good), {($test_5$, pass)}]
II. [($test_6$, 10.3), {($test_5$, pass)}]
III. [($diag_2$, bad), {($test_5$, fail)}]
IV. [($test_7$, pass), {($test_5$, fail)}]

Suppose we also have the following constraints:

V. [($diag_3$, good), {($test_6$, 10.3)}]
VI. [($test_5$, pass), {($test_8$, pass)}]

This can be shown graphically in the form of a constraint network (Fig. 3) where the nodes represent the variables and the edges represent the constraints. This graph shows that there exist inference paths from $test_8$ to both $diag_2$ and $diag_3$. This means that if the constraints associated with the edges are true, we would be able to assign values to both $diag_2$ and $diag_3$ just by performing the test associated with $test_8$. If the reasoner decides to perform $test_8$ and $test_8$ passes, the graph would be changed to reflect the propagation of the value assignments through the constraints (Fig. 4). Observe that all the constraints that can be no longer satisfied (i.e., constraints II and III) have been removed.

D. Example 4

Suppose we wish to represent a "conditional" constraint (i.e., a constraint for which the value assignment is not completely specified). For example, suppose we have

If $test_5$ fails then $diag_1$ is bad or $diag_2$ is bad.

This constraint is expressed in the following way:

[($diag_1$, bad), {($test_5$, fail), ($diag_2$, good)}]
[($diag_2$, bad), {($test_5$, fail), ($diag_1$, good)}]

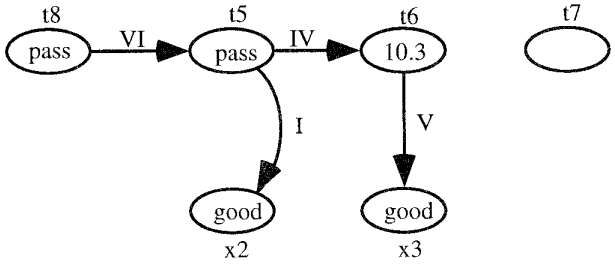

Fig. 4 Revised Diagnostic Constraint Graph

V. TEMPORAL CONSTRAINTS

When testing, time relations between tests (or other events) are often taken as exceptions to a process. For example, $test_a$ must precede $test_b$, or several tests must be performed in a specific order. These requirements would be treated as exceptions and overrides in a traditional test process. While exception handling has worked in many cases in the past, the increasing complexity of systems suggests that a more robust method for reasoning about time is needed.

Reasoning about time can be modeled as a constraint satisfaction problem. By defining the events that may occur in a test system and the temporal constraints between these events, a diagnostic planner (i.e., reasoner) can use approaches to solve CSPs that would assist in reasoning about these events. The temporal constraint model is intended to address issues of representing these temporal relations in a diagnostic model.

Let $\mathbf{CT} = \{ct_1, ct_2, \ldots, ct_q\}$ be a set of *temporal constraints* that indicate constraints on when certain events can occur. A *time event* is an event that has a known duration in time or whose duration can be determined. Examples of time events could be performing a test, requiring a resource, or some other action. The time constraints make it possible to define an ordering between these time events. The ordering is expressed in terms of the time events themselves, so time events cannot be ordered in absolute time, just in relation to each other.

Time constraints can be specified with a single construct,

$$TRelate(T_a, T_b, \delta_1, \delta_2, \delta_3, \delta_4)$$

where T_a and T_b are time events and $\delta_i \in \mathbb{R}$ is duration between endpoints of the intervals. T_a is the time event that is being *constrained*, and T_b is the time event that *constrains* T_a. The δ_i define the following durations in time and have some agreed upon and consistent units (Fig. 5):

$$\delta_1 = Start(T_b) - Start(T_a) \qquad \delta_2 = Stop(T_b) - Start(T_a)$$
$$\delta_3 = Start(T_b) - Stop(T_a) \qquad \delta_4 = Stop(T_b) - Stop(T_a)$$

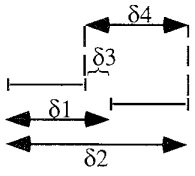

Fig. 5 Interval Endpoint Relations

The *TRelate* construct specifies that the event associated with T_a is constrained to start δ_1 seconds before T_b starts, start δ_2 seconds before T_b stops, stop δ_3 seconds before T_b starts, and stop δ_4 seconds before T_b stops. Note that, in general, each of the δ_i can be either positive or negative, but all of the δ_i must be consistent (i.e., $\delta_2 = \delta_1 + \delta_4 - \delta_3$).

A. Example 1

Suppose that for event T_a to occur, T_b must start at the same time as T_a. Note, however, that for T_b to occur, there is no constraint on T_a. This is represented as $TRelate(T_a,T_b,0,x,x,x)$. Here, x is treated as a "don't care."

B. Example 2

For event T_a to occur, T_b must start at the same time as T_a; for T_b to occur, T_a must start at the same time as T_b. In addition, T_b must stop 2.5 seconds before T_a stops. This is represented using two constraints, $TRelate(T_a,T_b,0,x,x,2.5)$ and $TRelate(T_a,T_b,0,x,x,-2.5)$.

C. Example 3

For event T_a to occur, T_a must be started and the T_b must be initiated 4 seconds later. In addition, T_a must stop 0.5 seconds before T_b stops. This is represented as $TRelate(T_a,T_b,4,x,x,0.5)$.

D. Example 4

For event T_a to occur, T_b must start at the same time as T_a starts and T_a must stop at the same time as T_b stops. This is represented as $TRelate(T_a,T_b,0,x,x,0)$. This is the same as defining the *equal* temporal relation.

E. Example 5

If we wish to combine two constraints, such as $TRelate(T_a,T_b,1,x,x,x)$ and $TRelate(T_a,T_b,x,x,x,-1)$, we can consolidate these constraints into $TRelate(T_a,T_b,1,x,x,-1)$ (i.e., we take the equivalent of a logical OR of the two constraints).

F. Example 6

Suppose we wish to limit the duration of T_a to 10 seconds. Then we use $TRelate(T_a,T_a,0,10,x,x)$ which states that to do T_a, T_a must be started at the same time as T_a (i.e., a tautology on the start of T_a is given) and stopped 10 seconds after T_a starts. This actually limits the duration of T_a to 10 seconds. Fully specifying this constraint yields $TRelate(T_a,T_a,0,10,-10,0)$.

G. Example 7

Suppose we specify the constraint, $TRelate(T_a,T_b,11,x,1,x)$. This constraint states that to do T_a, T_a must start 11 seconds before T_b and stop 1 second before T_b starts. This is depicted graphically in Fig. 6.

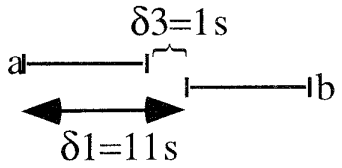

Fig. 6 Interval Relations with Specific Times

VI. GLOBAL CONSTRAINTS

It may be necessary to specify constraints on a global basis. In other words, some constraints may apply to the complete test process and have no specific bearing on any single, identifiable event in the test process. Currently, we have identified three global constraints; however, many more can be specified. The global constraints **CG** = $\{cg_1, cg_2, cg_3\}$ defined in the model so far include the following:

- cg_1 = *analysis_limit*. This is a time limit for the overall diagnosis of the system.
- cg_2 = *test_limit*. This is a time limit for the overall testing of the system. This limit is specified from the start of the first test to the end of the last test.
- $cg_3 = \{cg_3^1, cg_3^2, .., cg_3^T\}$. This is a set of constraints that limits the use of resources. A resource limit constraint is a pair $cg_3^i = (r,t)$ where r is the resource that is limited, and t is the time in some agreed-upon units (e.g., seconds). If t is not specified, then it is assumed that $t = 0$ (i.e. the resource r is not available at all). For example, if time event T_a uses resource r for 25 seconds, T_b uses r for 0.5 seconds, and the resource r is available for only 25 seconds, then the resource limit constraint is $cg_3^i = (r,25)$. Note that it is not possible to perform both T_a and T_b, so a resource manager must select the appropriate event to occur.

VII. CONCLUSION

In this paper, we outlined the efforts of the AI-ESTATE subcommittee of SCC20 to standardize interchange formats for diagnostic constraint models in intelligent test systems. We also provided a theoretical basis for the model and a description of the model as it is currently specified in the AI-ESTATE draft standard. It is the hope of the AI-ESTATE subcommittee and its working groups that people interested in the standard become directly involved in its development. We welcome all interested parties to attend the meetings of the committee.

REFERENCES

[1] IEEE Std 1232-1995. *Trial Use Standard for Artificial Intelligence and Expert System Tie to Automatic Test Equipment (AI-ESTATE): Overview and Architecture*, New York, IEEE Press. 1995.

[2] IEEE P1232.1. *Trial Use Standard for Artificial Intelligence and Expert System Tie to Automatic Test Equipment (AI-ESTATE): Data and Knowledge Specification*, Draft 4.3, (in ballot), New York, IEEE Press. 1995.

[3] William R. Simpson and John W. Sheppard. *System Test and Diagnosis*, Norwell, Massachusetts: Kluwer Academic Publishers. 1994.

[4] Rina Dechter. "Constraint Networks: A Survey," *The Encyclopedia of Artificial Intelligence*, Stuart C. Shapiro (ed.), New York: Wiley. 1992.

[5] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems*, San Mateo, California: Morgan Kaufmann Publishers, Inc. 1988.

[6] Nils Nilsson. *Principles of Artificial Intelligence*, San Mateo, California: Morgan Kaufmann Publishers, Inc. 1980.