Active System Integrity Fences

Christopher Landauer
Topcy House Consulting, Thousand Oaks, California
Email: topcycal@gmail.com

***Abstract*—This paper describes the architecture of a Wrapping-Based Self-Modeling System that includes all of the models and processes we have mentioned in our previous papers on the subject of methods for mitigating the effects of the Get Stuck theorems. To make the description more concrete, we have selected a particular application domain example: "active system integrity fences", that protect and defend a complex computing system or network (called the host system) from all enemies, foreign and domestic.**

An "active integrity fence" sits on an interface between two system components, with an explicit notion of the characteristics expected in the traffic (in both directions), derived from expectations provided at design time and observations collected at run time. We do not describe a particular host system, since our primary interest is in the protector system, and we expect that it applies to any host system for which we can specify the expected internal interactions.

We define the relevant models and processes, and how they interact with each other and with the host system. We also provide a short description of the Wrapping infrastructure that holds the system together and organizes and executes all of its behavior, both protective and mission.

Keywords: Self-Aware Systems, Self-Adaptive Systems, Self-Modeling Systems, Get Stuck Theorems, Behavior Mining, Dynamic Knowledge Management, Wrapping Integration Infrastructure

1. Introduction to the Problem

This paper is a further continuation of and companion to previous papers on Self-Modeling Systems [31] [26], concerning new and extended methods for mitigating the effects of the "Get Stuck" Theorems (see [31] for a more recent explanation. and [26] for a discussion of issues). These theorems basically say that the system needs to be able to rearrange its own knowledge structures for compactness and efficiency, if it expects to survive for long periods of time in a demanding environment [30] [31]. The mechanism for doing that must include comparison of design-time expectation models with run-time observations of behavior. Self-Modeling Systems will do these comparisons themselves, as much as feasible. Though, strictly speaking, this is a way to decide on adaptations, not an adaptation itself; the adaptation processes we use fall out of the Wrapping infrastructure.

1.1. Example: Active System Integrity Fences

In this Subsection, we describe our application example: "active system integrity fences", which are modules charged with a single simple (if difficult) goal: explore system behavior and look for anomalies.

We use the term "fences" intentionally, to emphasize a distinction in purpose: guards keep things out, wardens keep things in, watchers and monitors do neither, and fences do both.

We consider complex engineered systems that are managed or controlled by software, but that have essential hardware components driven by the software. These systems may be distributed, and operate in environments that are too large, too remote, too hazardous, or too rapid for direct human operation. They therefore need a great deal of autonomy, and even local adaptivity.

These systems are not entirely software. Hardware in systems is usually accompanied by very specific operating conditions, provided by the manufacturer. Complex hardware often has a large and diverse set of commands that it interprets, and part of the role of the active fences here is to guarantee that the command streams do not drive the hardware outside its operating envelope without a certain level of authorized over-ride (there are frequently multiple levels of envelopes for any hardware component ranging from "not recommended" to "this will break it".

The purpose of an active system integrity fence is to protect and defend a complex computing system or network (called the host system) from all enemies, foreign and domestic.

An active integrity fence sits on an interface between two system components (or at the common entry interface of one component), with an explicit notion of the characteristics expected in the traffic (in both directions, both temporal behavior and semantics), derived from expectations provided at design time and observations collected at run time.

In the simplest case, it can throttle the data volume to an acceptable level (by silently ignoring or pointedly rejecting input), but it can also use constraints on the contents of data to make similar decisions (e.g., for routing, acceptance, and even explicit rejection or tacit dropping of data).

The criteria are a combination of "type" constraints on the content characteristics of the data and associated "allowed data volumes" (more complex examples have explicit state-machine protocol identifiers and constraints on their allowed transition volume).

The main purpose is to identify unexpected consequences and prevent them from damaging system performance: for example, even after a system password compromise, the notion that certain external entities can access the system is a failure that should be rejected.

What are the likely dangerous unexpected consequences?

- system leaks (provides data content it should not)
- system overloads
- system thrashes
- system forgets (loses data)
- system breaks (still runs, but wrong answers)
- system stops

Available computational resources dictate whether the fence is continuous (checking whenever anything in the interface changes) or sporadic (occasional, periodic, or other event based activity rule).

Our focus here is on the computational mechanisms that enable this kind of protection.

We can also consider mobile fences, moving around in the system or network, either transferring from one machine to another, or just changing focus on different interfaces (which clearly needs a map of the interfaces to define the space of movement.

1.2. Structure of Rest of Paper

The structure of the rest of the paper is as follows. In Section2, we provide some background on Wrappings, to make the paper more self-contained.

In Section3, we provide some comments on why we consider self-modeling systems, and how they relate to the self- world. We also briefly introduce the "Get Stuck" theorems, which motivate this mitigation approach.

In Section 4, we provide a notional system architecture, based on Wrappings, that is sufficiently flexible to allow the "behind-the-scenes" knowledge management that is performed by our mitigation processes, and within which the mitigation models interact.

In Section 5, we provide a description of the models used in our application and in the mitigation processes, and describe the mitigation processes in more detail,

Finally, in Section 6, we present our conclusions and prospects for further advances.

2. Background on Wrapping

We provide a short description of Wrappings in this Section, since there are many other more detailed descriptions elsewhere [27] [25] [6], and especially the tutorials [33] [34]. The Wrapping integration infrastructure is our approach to run-time flexibility, with its run-time context-aware decision processes and computational resources. The basic idea is that Wrappings are Knowledge-Based interfaces to the uses of computational resources in context, and they are interpreted by processes that are themselves resources.

The basic idea starts with the "Problem Posing" interpretation of programs [27], which replaces explicit invocation of computationsl resources with an implicit request to address a problem.

Thus, programs interpreted in this style do not "call functions", "issue commands", or "send messages"; they "pose problems" (these are *information service requests*). Program fragments are not written as "functions", "modules", or "methods" that do things; they are written as "resources" that can be "applied" to problems (these are *information service providers*).

Because we separate the problems from the applicable resources, we can use more flexible mechanisms for connecting them than simply using the same name.

We have shown that this approach leads to some interesting flexibilities, when combined with the "meta-reasoning" approach of Wrappings [5], including such properties as software reuse without source code modification, delaying language semantics to run-time, and system upgrades by incremental migration instead of version based replacement.

We specifically want to make the mapping from problems to resources explicit, because implicit mechanisms are hard to study, so for Wrappings we use a Knowledge Base.

The Wrapping integration infrastructure is defined by its two complementary aspects, the Wrapping Knowledge Bases and the Problem Managers.

The *Wrapping Knowledge Bases* (WKBs) contain the Wrappings that map problems to resources in context. They define the entire set of problems that the system knows how to treat (there are usually also default problems that catch the ones otherwise not recognized). The mappings are problem-, problem parameter-, and context-dependent.

The *Problem Managers* (PMs) are the programs that read WKBs and select and apply resources to problems. The meta-recursion follows because the PMs are also resources, and are Wrapped in exactly the same way as other resources, and are therefore available for the same flexible integration as any resources. These systems therefore have no privileged resource; anything can be replaced. Default PMs are provided with any Wrapping implementation, but the defaults can be superseded in the same way as any other resource. These are the processes that replace the implicit invocation process, allowing arbitrary processes to be inserted in the middle of the resource invocation process. This choice leads to very flexible systems [33] [34].

The basic notion is the interaction of one very simple loop, called the "Coordination Manager", and a very simple planner, called the "Study Manager".

The default Coordination Manager (CM) is responsible for keeping the system going. It has only three repeated steps, after an initial FC = Find Context step as shown in Figure 1.

To "Find Context" means to establish a context for problem study, possibly by requesting a selection from a user, but more often getting it explicitly or implicitly from the system invocation. It is our placeholder for conversions from that part of the system's invocation environment that

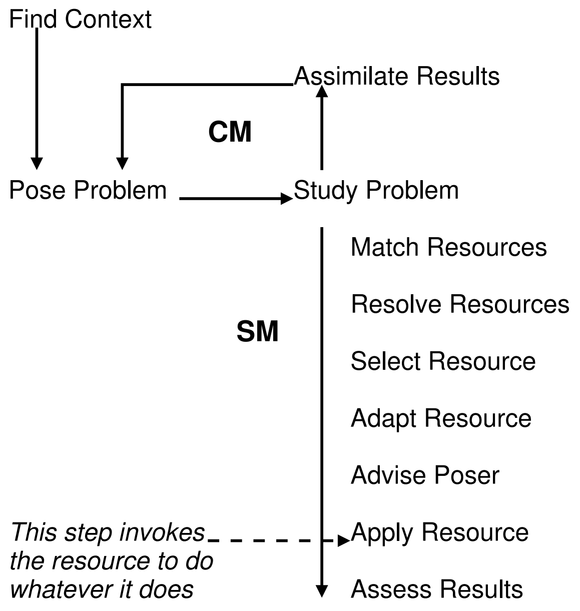

Figure 1. CM and SM Steps

is necessary for the system to represent to whatever internal context structures are used by the system.

To “Pose Problem” means to get a problem to study from the problem poser (a user or the system), which includes a problem name and some problem data, and to convert it into whatever kind of problem structure is used by the system (we expect this is mainly by parsing of some kind).

To “Study Problem” means to use an SM and the Wrappings to study the given problem in the given context, and to “Assimilate Results” means to use the result to affect the current context, which may mean to tell the poser what happened. Each step is a problem posed to the system by the CM, which then uses the default SM to manage the system’s response to the problem. The first problem, “Find Context”, is posed by the CM in the initial context of “no context yet”, or in some default context determined by the invocation style of the program.

The main purpose of the default CM is cycling through the other three problems, which are posed by the CM in the context found by the first step. This way of providing context and tasking for the SM is familiar from many interactive programming environments: the “Find context” part is usually left implicit, and the rest is exactly analogous to LISP’s “read-eval-print” loop, though with very different processing at each step, mediated by one of the SMs. In this sense, this CM is a kind of “heartbeat” that keeps the system moving.

If the Coordination Manager is the basic cyclic program heartbeat, then the Study Manager is a planner that organizes the resource applications. The CM and SM interact as shown schematically in Figure 1.

We have divided the “Study Problem” process into three main steps: “Interpret Problem”, which means to find a resource to apply to the problem; “Apply Resource”, which means to apply the resource to the problem in the current context; and “Assess Results”, which means to evaluate the result of applying the resource, and possibly posing new problems. We further subdivide problem interpretation into five steps, which organize it into a sequence of basic steps that we believe represent a fundamental part of problem study and solution. These are implemented in the default Study Manager (SM).

To “Match Resources” is to find a set of resources that might apply to the current problem in the current context. It is intended to allow a superficial first pass through a possibly large collection of Wrapping Knowledge Bases.

To “Resolve Resources” is to eliminate those that do not apply. It is intended to allow negotiations between the posed problem and each Wrapping of the resource to determine whether or not it can be applied, and make some initial bindings of formal parameters of resources that still apply.

To “Select Resource” is simply to make a choice of which of the remaining candidate resources (if any) to use.

To “Adapt Resource” is to set it up for the current problem and problem context, including finishing all required bindings.

To “Advise Poser” is to tell the problem poser (who could be a user or another part of the system) what is about to happen, i.e., what resource was chosen and how it was set up to be applied.

To “Apply Resource” is to use the resource for its information service, which either does something, presents something, or makes some information or service available.

To “Assess Results” is to determine whether the application succeeded or failed, and to help decide what to do next.

Finally, we insist that every step in the above sequences is actually a posed problem, and is treated in exactly the same way as any other, which makes these sequences “meta”-recursive [1]. That means that if we have any knowledge at all that a different planner may be more appropriate for the context and application at hand, we can use it (after defining the appropriate context conditions), either to replace the default SM when it is applicable, or to replace individual steps of the SM, according to that context (which can be selected at run time).

Of course, we also have to have something to replace or supersede. We have therefore provided default resources for each of the CM and SM steps, to be used when no other is selected to supersede it (as the above SM is the default resource for the problem “Study Problem”). A simple complication occurs with the default among many possible resources for the “Select Resource” problem: we want to allow other resources to be used, so we insist that the default resource (which otherwise might just pick the first resource on the list) not pick itself if there is another choice when it is addressing the “Select Resource” problem.

In addition, since the resources that read the WKB are selected in context as is any other, the WKB can be heterogeneous, with context determining which reader is used for which format of Knowledge Base. This helps greatly for implementing improvements to programs, since the new and old formats can exist simultaneously, unti the old format is no longer needed.

We have used these algorithms many times to explain and implement autonomous and reflective agents and systems [28] [29], and shown that they provide the appropriate level of manageable flexibility and auditable integration. The advantage in flexibility this approach provides over other activity loops that have been proposed is that the SM and CM steps are "meta"-steps, with posed problems for the activities, allowing one further level of abstraction and indirection when it is useful. There are a number of other activity loops that we have seen described in various places [5], especially the popular MAPE-K loop of autonomic computing [21] [24] [3] [40], and we have shown that our CM / SM meta-recursive interaction subsumes all of them. The meta-interpretation style [1] of Wrappings [27] can of course be applied to any of them to make them much more flexible.

We have implemented several different kinds of CMs in addition to the simple default CM defined above. There are CMs that short cut the reflection by calling the default step resources directly, and fully recursive versions that have extra levels of problem posing. Some of them are described in other papers in the references.

We have also used different SMs, beyond the default one that tries only one resource: one SM tries all applicable resources and returns with the first success, another tries them all and evaluates them to return the best success, and one collects all successes and summarizes. There are also different kinds of SM steps. The Match and Resolve resources that read XML WKBs are different from the ones that read text only WKBs. A different Match or Resolve might invoke a more sophisticated planner if there are no matches. A different Select might choose all compatible resources, then negotiate among them. Different versions of apply, beyond the default function call, might send a request message, or invoke an interpreter or other process. Another one might simply add the resource to a configuration, instead of invoking it.

Wrapping-based systems support run-time decisions about which resources to apply in the current context, both at the application level (the resources that perform the task at hand) and at the meta-level (the resources that are used to select and organize the application level resources). This flexibility does come with a cost, but there are also mechanisms based on partial evaluation [13] [20] [27] [41] [15] for removing any decisions that will be made the same way every time, thus leaving the costs where the variabilities need to be.

3. Self-Modeling Systems

Our approach is related to the architectures of robots and autonomous vehicles [2] [37], but we add some features not computationally feasible at the beginning. Our systems are Self-Adaptive [6] [9] [23], which means that they can observe their own behavior, reason about it, and use the results to adjust the behavior.

We are especially enamored of Self-Modeling Systems [28] [29], which have models of their own behavior, derived from original specifications of intent (from the designers), as modified by observation of actual behavior, because (in this author's paraphrase):

> No plan ... extends with any certainty beyond the first contact with ... [reality], reference [39], p.92

These models are interpreted to produce the system's behavior. That is to say, the behavior, sometimes including the interpreter itself, is the interpretation of the models (this is not as hard as it seems [42] [28] [31]).

The reason for self-modeling systems is to retain as much flexibility in the operational system as possible, and to allow the system to depart wildly from its original design specifications (under carefully controlled or appropriately identified situations, of course). Additionally, it allows the system to examine itself, looking for anomalies:

> O wad some Power the giftie gie us To see oursels as ithers see us! [10]

This architectural approach also supports the processes that mitigate the effects of the "Get Stuck" Theorems, which essentially say that any software-intensive system that makes models of its environment and behavior will eventually need to reorganize its knowledge structures. To that end, we defined several mitigation processes in [26] [32], and this paper is a description of a notional architecture in which to implement the processes.

4. Architecture

In this Section, we describe an architectural context for the mitigation processes, using a Wrapping infrastructure to provide the flexibility of operations that we want. In the next Section, we describe the processes in a little more detail and show how they might interact.

A notional picture of the architecture under consideration is in Figure 2.

The top part of the picture is the usual Wrapping CM / SM loop, accessing the WKB in context to apply resources to posed problems, possibly also adjusting the context (and allowing the context to be changed by the system environment). This is the standard behavior of a Wrapping-based system.

The other processes in the main system, that is, the ones in the application domain, are invoked by the SM, and build and maintain the domain knowledge bases that are the focus of the mitigation effort (though, of course, the same mitigation processes apply equally well to the Wrapping Knowledge Base. due to reflection).

The mitigation processes BM, KnRef, and DKM collect information from the choices made in the SM and adjust the WKB, and they collect information from the domain knowledge bases and adjust them also.

5. Description of Models Used

There are three classes of models (processes): the infrastructure models, such as the CM, SM, and other PMs;

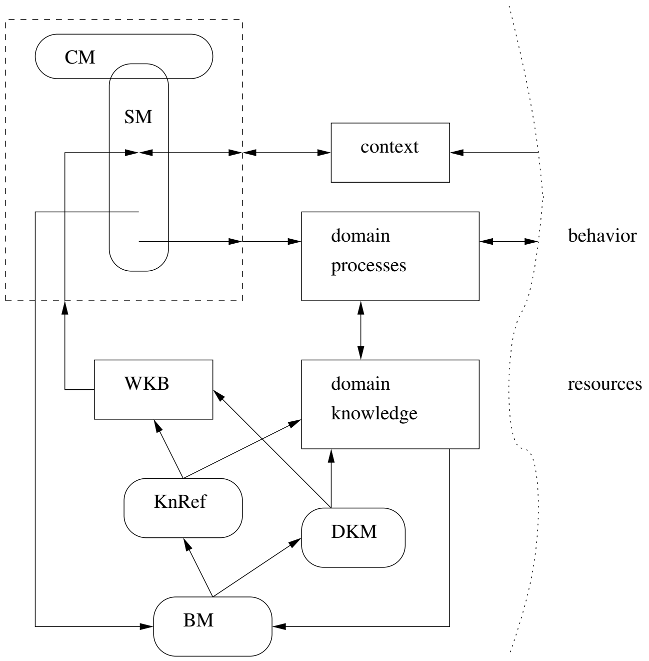

Figure 2. Notional Architecture

the application models that perform the system's objective activity, and the mitigation models that sit behind that activity and keep it running.

5.1. Infrastructure Models

The infrastructure has been described before, in Section 2. It contains the default PMs, which are the various flavors of SM (default is straight line plan, others are try all until success, try all and combine, and others. These can be fully recursive or not.

It contains the various flavors of CM used in the system, such as the default simple loop and the Mud CM for distributed components, and these can be recursive or not. It contains the WKB and the context knowledge base, and the reflective nature of Wrappings allows both of them to be heterogeneous, since the knowledge base readers are also resources, selected according to posed problems in the usual way.

5.2. Fences Application Models

These are the models that do the fences application work. Each fence is responsible for one class of interfaces (often just one interface).

- instrumentation: record summaries of data types and volume for all (or just representative) crossing the interface in either direction; in certain cases, the content matters also;
- movement: for mobile fences, decisions about when and where to move (perhaps to track down a problem or just randomly spot check); this process clearly needs a map of system interfaces;
- examination: build models of the data crossing the interface (in both directions), infer actual protocol with performance measurements; in the simplest case, just measuring close approaches to acceptability boundaries is enough model;
- retention: store these models in a system domain knowledge base;
- communication: announce current status and observed behavior, trends and variability distributions; interact with other fences to discover and isolate problems; announce to the system that there is an issue in certain places (so that the context can be chand to avoid them);
- cooperation: interact with other fences to discover and isolate problems; apply a requested data throttle at an interface;
- escalation: when a problem gets dangerous or large or widespread enough (criteria supplied by the designers). ask for help.

Each of these processes has relatively simple inputs, and only the examination process has any difficult algorithms, Most of the difficulty lies in the criteria for data extraction, modeling , and escalation.

5.3. Mitigation Models

These are the ones that do the mitigations [31] [26]:

- Behavior Mining
- Model Deficiency Analysis
- Knowledge Refactoring
- Dynamic Knowledge Management
- Constructive Forgetting
- Continual Contemplation

We describe the intent of each of these processes and how they might interact.

5.3.1. Behavior Mining.

This process examines every decision in context, recording the following data:

- problem with parameters,
- relevant context,
- resource application selected and constructed.

The relevant context is the set of context conditions that were examined and succeeded for the selection. In fancier cases, this will also include resources not selected for this problem, and why (according to the context conditions that eliminated them).

In general, problems are stratified into layers, based on their resolution (in time, space, or concept). Strictly speaking, this should be called semiotic content, but that discussion leads beyond the scope of this paper.

The BM process also examines changes to the domain knowledge base, looking for infelicities, which can be statistical or even esthetic (unbalanced trees, different amounts of detail in different areas, unique or very low frequency references, which often results from specification errors). In this case, the models are "plausibility" models, since there is no available basis to declare them correct or incorrect.

The BM process then feeds its results to the MDA process for resolution.

5.3.2. Model Deficiency Analysis. This process attempts to determine where models have gone wrong (this is a retrospective analysis, not predictive). The simplest form of this begins with a behavioral assertion about model effects that has been violated.

We expect the assertions to be provided initially by the designers, since models are expected to provide some information service, and the model creators decide what that is. After all, there was a purpose for creating the model in the first place, and these assertions define the designers' expectations for it.

Every behavioral assertion involves some of the variables within the model (or some performance parameters). In the most intrusive case, every change to any of those variables causes the assertion to be (it is possible to reduce this a little bit if if can be proven that the change cannot make an assertion go fro true to false).

When an assertion fails, the hard part begins: the assignment of blame, that is, how can the system decide where the failure is and who did it. This is especially difficult for assertions in the usual kinds of first-order logic, since mathematically there is no such culprit.

However, the assertions are not the only information available to the MDA process. It also has access to the component behavior models constructed by the BM process, and sometimes it can use those to decide which part of an assertion is less likely to be wrong. This assessment can often be done by maintaining a reliability "reputation" for each of the assertions, each of its components and each of the processes that produce the variables tha occur in those components. The reputation of an assertion component is enhanced by its success frequency (tempered by a measure of how well the input data to each variable producer fits the input assumptions).

In addition, the MDA process gets warnings from the BM process about models that may be incorrect due to statistical or esthetic considerations. It tries to decide when a strange structure is an error and when it is just strange. Of course, it can't really do that, mainly for undecideability reasons, but it can discover certain kinds of problems and announce the others to the system monitors to try to get help. If no help is forthcoming, then the issue is simply recorded as a problem that the MDA process cannot solve, and if enough instances of those occur, it escalates the issue to a deficiency warning.

5.3.3. Knowledge Refactoring. This process is partly housekeeping: re-arranging knowledge for efficiencies: it can be applied to the context conditions for the same problem (e.g., to get the shortest average time for decision, given the time distribution of conditions). It is also related to the esthetic criteria in the BM process: if we consider any knowledge representation mechanism as a graph with labeled nodes and directed labeled edges, then nodes with an excessive number of edges may be too general, and nodes with too few edges may be too specific.

This process is also sensitive to the access patterns of the knowledge elements. We want elements that are accessed very often to have shorter access paths, which can distort any nice *a priori* ontology (the goal in this case is not readability; it is efficiency; other semantics-preserving refactorings may be needed for readability).

This process is intended to be transparent to the users of the knowledge bases, in the sense that their access mechanisms remain exactly the same; only the resulting search performance is affected.

5.3.4. Dynamic Knowledge Management. This process organizes the knowledge for quick reasoning; it subsumes Constructive Forgetting (a mitigation from [31] [26]). We called that one out explicitly because it is not normally acceptable to throw knowledge away, but we have shown that it is inevitable.

The idea here is that there are different frequencies of access for different parts of the knowledge base, and rearranging the knowledge base can take that into account.

The simplest version of this is to pushd the Least Recently Used (LRU) knowledge elements off to longer access paths (this is much the same process as defining Huffman codes [19] [36] [14]). This measure of recency can be weighted by importance of consequences and gathered by relevant context (big context change implies much knowledge re-ordering). There are also other relevant factors that will be different for each different application.

5.3.5. Continual Contemplation. This process is the general term for all the background concurrent examination processes that examine everything: some examine as-is models (from Behavior Mining), and compare with as-designed models (from system definition). These functions are described earlier in this Section.

6. Conclusions and Prospects

We have argued that the flexibility of self-modeling systems make them well-suited to handle remote, complex, and / or hostile environments, but that flexibility comes with the obvious cost in performance, and a less obvious cost in organization. How much of this infrastructure machinery is used in any given application is an application-specific engineering decision that depends on the expected level of hazard and the required level of performance.

We have shown that the Wrapping infrastructure supports a very flexible interweaving of domain resources and infrastructure resources, and can now include processes that mitigate the effects of the "Get Stuck" Theorems, which limit the lifetime of any autonomous system that builds and maintains models.

One of the most exciting prospects is that the system has enough knowledge about its own behavior that it can explain what it is doing, or what it is about to do, and why, and most particularly, what it is not doing and why (the SM manages the selection, so it has the data to explain what it does not select). This requires the system to reason about the situation it is in [4], so they can describe it.

A key advance in the state of the reasoning processes would be to provide tools for them to reason about incomplete and inconsistent information [7] [8] [12], since that describes essentially all of the system's knowledge. Similarly, various kinds of advanced learning methods [11] [38] [18] [17] could be applied in the model building processes, to avoid needing to specify a model type or structure in advance. These could be of great use to the model building processes in these systems.

However, learning methods such as XCS [43] [44] have not much place here, since the behaviors in these systems are not well modeled by MDP (Markov Decision Processes) or even POMDP (Partially Observable MDP) in most cases, and these methods are not model-free; they assume a state transition model that can be described as a POMDP.

We also described mobile fences, moving around in the system or network, with the responsibility to explore, detect, decide and act or escalate. If these fences are also knowledgeable about more of the system behavioral expectations, then they should be able to detect certain software errors, in addition to external anomalies.

We can also imagine some physical existence for "touch points" (like the little blue police / watchman boxes for periodic checking in), so the fence can monitor and examine its own progress.

We think that these systems can be made much safer than they are now, but that requires an engineering judgment based choice of how much protective infrastructure to include.

References

[1] Harold Abelson, Gerald Sussman, with Julie Sussman, *The Structure and Interpretation of Computer Programs*, Bradford Books, now MIT (1985)

[2] James S. Albus, Alexander M. Meystel, *Engineering of Mind: An Introduction to the Science of Intelligent Systems*, Wiley (2001)

[3] Paolo Arcaini, Elvinia Riccobene, Patrizia Scandurra, "Modeling and Analyzing MAPE-K Feedback Loops for Self-Adaptation", *Proceedings SEAMS 2015: The 2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 18-19 May 2015, Florence, Italy (2015)

[4] Jon Barwise, *The Situation in Logic*, CSLI Lecture Notes No. 17, Center for the Study of Language and Information, Stanford U. (1989)

[5] Dr. Kirstie L. Bellman, Dr. Christopher Landauer, Dr. Phyllis R. Nelson, "Managing Variable and Cooperative Time Behavior", *Proceedings SORT 2010: The First IEEE Workshop on Self-Organizing Real-Time Systems*, 05 May, part of *ISORC 2010: The 13th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing*, 05-06 May 2010, Carmona, Spain (2010)

[6] Kirstie L. Bellman, Christopher Landauer, Phyllis Nelson, Nelly Bencomo, Sebastian Götz, Peter Lewis and Lukas Esterle, "Self-modeling and Self-awareness", Chapter 9, pp. 279-304 in [22]

[7] Leopoldo E. Bertossi, Anthony Hunter, Torsten Schaub (eds.), *Inconsistency Tolerance*, Springer Lecture Notes in Computer Science, Volume 3300, Springer Verlag (2004)

[8] Jean-Yves Beziau, Walter Carnielli and Dov Gabbay (eds.), *Handbook of Paraconsistency*, King's College (2007)

[9] Robert Birke, Javier Cámara, Lydia Y. Chen, Lukas Esterle, Kurt Geihs, Erol Gelenbe, Holger Giese, Anders Robertsson and Xiaoyun Zhu, "Self-aware Computing Systems: Open Challenges and Future Research Directions", Chapter 26, pp. 709-722 in [22]

[10] Robert Burns, "To a Louse" (1768); *Robert Burns in Your Pocket*, Waverley Press (2009); along with many other collections and web sites

[11] Jaime G. Carbonell, "Learning by Analogy: Formulating and Generating Plans from Past Experience", pp. 137-161 in [38]

[12] Walter A. Carnielli, M.E. Coniglio and J. Marcos, "Logics of Formal Inconsistency", pp. 15-107 in [16]

[13] C. Consel, O. Danvy, "Tutorial Notes on Partial Evaluation", p.493-501 in *Proceedings PoPL 1993: The 20th ACM Symposium on Principles of Programming Languages*, 10-13 January 1993, Charleston, SC (January 1993)

[14] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*, MIT Press (1990) Second Edition, McGraw-Hill (2001), Section 16.3, p.385392

[15] Marcus Denker, Orla Greevy, Michele Lanza, "Higher Abstractions for Dynamic Analysis", pp.32-38 in *Proceedings PCODA'2006: the 2nd International Workshop on Program Comprehension through Dynamic Analysis*, Technical report 2006-11 (2006)

[16] D. Gabbay, F. Guenthner (eds.), *Handbook of Philosophical Logic*, vol. 14, Reidel (2007)

[17] Ian Goodfellow, Yoshua Bengio, Aaron Courville, *Deep Learning*, MIT (2016)

[18] Trevor Hastie, Robert Tibshirani, Jerome Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, Springer (2009), 2nd ed. Springer (2016)

[19] David A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes", *Proceedings of the IRE*, v.40, no.9, p.1098-1101 (1952)

[20] N. D. Jones, "Partial Evaluation", *Computing Surveys*, Volume 28, No. 3 (September 1996)

[21] J. Kephart, "Feedback on feedback in autonomic computing systems", in *Proceedings FC 2012: the 7th International Workshop on Feedback Computing*, San Jose, California (2012)

[22] Samuel Kounev, Jeffrey O. Kephart, Aleksandar Milenkoski, Xiaoyun Zhu (eds.), *Self-Aware Computing Systems*, Springer (2017)

[23] Samuel Kounev, Peter Lewis, Kirstie L. Bellman, Nelly Bencomo, Javier Cámara, Ada Diaconescu, Lukas Esterle, Kurt Geihs, Holger Giese, Sebastian Götz, Paola Inverardi, Jeffrey O. Kephart and Andrea Zisman, "The Notion of Self-aware Computing", Chapter 1, pp. 3-16 in [22]

[24] Philippe Lalanda, Julie A. McCann, and Ada Diaconescu, *Autonomic Computing: Principles, Design and Implementation*, Undergraduate Topics in Computer Science Series, Springer (2013)

[25] Christopher Landauer, "Infrastructure for Studying Infrastructure", *Proceedings ESOS 2013: Workshop on Embedded Self-Organizing Systems*, 25 June 2013, San Jose, California; part of *2013 USENIX Federated Conference Week*, 24-28 June 2013, San Jose, California (2013)

[26] Christopher Landauer, "Mitigating the Inevitable Failure of Knowledge Representation", *Proceedings M@RT@ICAC 2017: The 2nd International Workshop on Models@run.time for Self-aware Computing Systems*, Part of *ICAC2017: The 14th International Conference on Autonomic Computing*, 17-21 July 2017, Columbus, Ohio (2017)

[27] Christopher Landauer, Kirstie L. Bellman, "Generic Programming, Partial Evaluation, and a New Programming Paradigm", Chapter 8, pp.108-154 in Gene McGuire (ed.), *Software Process Improvement*, Idea Group Publishing (1999)

[28] Christopher Landauer, Kirstie L. Bellman, "Self-Modeling Systems", pp.238-256 in R. Laddaga, H. Shrobe (eds.), "Self-Adaptive Software", Springer Lecture Notes in Computer Science, vol.2614 (2002)

[29] Christopher Landauer, Kirstie L. Bellman, "Managing Self-Modeling Systems", in R. Laddaga, H. Shrobe (eds.), *Proceedings Third International Workshop on Self-Adaptive Software*, 09-11 Jun 2003, Arlington, VA (2003)

[30] Christopher Landauer, Kirstie L. Bellman, "Model-Based Cooperative System Engineering and Integration", *Proceedings SiSSy 2016: 3rd Workshop on Self-Improving System Integration*, 19 July 2016, part of *ICAC2016*: *13th IEEE International Conference on Autonomic Computing*, 19-22 July 2016, Wuerzburg, Germany (2016)

[31] Christopher Landauer, Kirstie L. Bellman, "Self-Modeling Systems Need Models at Run Time", *Proceedings M@RT 2016: the 11th International Workshop on Models@run.time*, 04 October 2016, Part of *ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*. 02-07 October 2016, Palais du Grand Large, Saint Malo, Brittany, France (2016)

[32] Christopher Landauer, Kirstie L. Bellman, "An Architecture for Self-Awareness Experiments", *Proceedings SeAC 2017: 2nd Workshop on Self-Aware Computing*, Part of *ICAC2017: The 14th International Conference on Autonomic Computing*, 17-21 July 2017, Columbus, Ohio (2017)

[33] Dr. Christopher Landauer, Dr. Kirstie L. Bellman, Dr. Phyllis R. Nelson, "Wrapping Tutorial: How to Build Self-Modeling Systems", *Proceedings SASO 2012: The 6th IEEE Intern. Conf. on Self-Adaptive and Self-Organizing Systems*, 10-14 Sep 2012, Lyon, France (2012)

[34] Dr. Christopher Landauer, Dr. Kirstie L. Bellman, Dr. Phyllis R. Nelson, "Wrapping Tutorial: How to Build Self-Modeling Systems", *Proceedings CogSIMA 2013: 2013 IEEE Intern. Inter-Disciplinary Conf. Cognitive Methods for Situation Awareness and Decision Support*, 25-28 February 2013, San Diego, California (2013)

[35] Dr. Christopher Landauer, Dr. Kirstie L. Bellman, Dr. Phyllis R. Nelson, "Modeling Spaces for Real-Time Embedded Systems", *Proceedings SORT 2013: The Fourth IEEE Workshop on Self-Organizing Real-Time Systems*, 20 June 2013, part of *ISORC 2013: The 16th IEEE International Symposium on Object / component / service-oriented Real-time distributed Computing*, 19-21 Jun 2013, Paderborn, Germany (2013)

[36] Jan Van Leeuwen, "On the construction of Huffman trees", p.382-410 in *Proceedings ICALP 1976: the Third International Colloquium on Automata, Languages and Programming*, 20-23 July 1976, Edinburgh (1976)

[37] Alexander M. Meystel, James S. Albus, *Intelligent Systems: Architecture, Design, and Control*, Wiley (2002)

[38] Ryszard S. Michalski, Jaime G. Carbonell, Tom M. Mitchell (eds.), *Machine Learning: An Artificial Intelligence Approach*, Tioga Press, Palo Alto, CA (1983)

[39] Helmuth Karl Bernhard Graf von Moltke, *On Strategy* (in German), translated in Daniel J. Hughes and Harry Bell, *Moltke on the Art of War: Selected Writings*, Presidio Press (1993); paperback Presidio Press (1995)

[40] E. Rutten, "Feedback Control as MAPE-K loop in Autonomic Computing", Research Report RR-8827, INRIA Sophia Antipolis - Méditerranée, INRIA Grenoble - Rhône-Alpes (10 Dec 2015)

[41] Gregory T. Sullivan, "Dynamic Partial Evaluation", *Proceedings PADO II: Second Symposium on Programs as Data Objects*, 21-23 May 2001, Aarhus, Denmark (2001)

[42] Ken Thompson, "Reflections on Trusting Trust", *Comm. of the ACM*, vol.27, no.8, pp.761-763 (Aug 1984), http://dl.acm.org/citation.cfm?id=358210 (availability last checked 03 Apr 2017); see also the "back door" entry of "The Jargon File", widely available on the Web, and other comments findable by searching for "back door Ken Thompson moby hack" (availability last checked 03 Apr 2017)

[43] Stewart W. Wilson, "Classifier Fitness Based on Accuracy", *Evolutionary Computation*, v.3, no.2, p.149175 (1995)

[44] Stewart W. Wilson, "Generalization in the XCS Classifier System", p.665674 in John R. Koza et al. (eds.), *Proceedings GP 1998: the Third Annual Conference on Genetic Programming*, 22-25 July 1998, University of Wisconsin, Madison, Wisconsin (1998)