Reflective Systems Need Models at Run Time

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

Kirstie L. Bellman
Topcy House Consulting, Thousand Oaks, California
Email: bellmanhome@yahoo.com

***Abstract*—We think that run-time models are a minimum necessity for significant self-awareness in Computationally Reflective systems. For us, a Computationally Reflective system is not only "self-aware" (it can collect information about its own behavior and that of its environment), but also "self-adaptive" (it can reason about that behavior and make adjustments that change it). Moreover, when the system is also "self-modeling", that is, defined by the models it creates and maintains, then it can modify those models to better align its behavior with changes in its operating environment.**

In this paper, we show that building and managing these models presents some difficulties that are largely independent of the modeling mechanisms used. We also provide some mechanisms for coping with these difficulties, from a kind of "Behavior Mining" and "Model Deficiency Analysis" for current models to "Dynamic Knowledge Management", of which major components are "Knowledge Refactoring" and "Constructive Forgetting". None of these methods eliminate the problem, but each alleviates some of the difficult aspects of the problem, which allows bigger and more complex systems to exist in more complex and diverse environment for longer periods of time.

Keywords: Self-Aware Systems, Self-Adaptive Systems, Self-Modeling Systems, Computational Reflection, Computational Semiotics, Run-Time Model-Building, Wrapping Integration Infrastructure, Problem Posing Interpretation

I. Introduction

We argue that run-time models are a minimum necessity for significant and effective self-awareness in Computationally Reflective systems. A Computationally Reflective system [28] [29] [4] must not only be"self-aware" (it can collect and organize information about its own behavior and that of its environment), but also "self-adaptive" (it can reason about that behavior and make adjustments that change it). We are primarily interested in systems that are also "self-modeling", which extends the notion to systems that construct and maintain their own models, and use those to generate their behavior. We think that such systems must be able to change not only the models but also the modeling mechanisms they use at run-time, not just adapt parameters or switch between operating modes.

A Computationally Reflective system can **collect** information about its own structure and behavior, and it can also **examine** that information to **decide** how to **modify** the behavior, **retain** and **analyze** it over extended time periods, **adjust** its decision processes to "optimize" it towards selected success criteria, and even **change the success criteria** to respond to changes detected in the environment (which may also include newly imposed purposes or goals).

In this paper, we explore some seemingly far-fetched theoretical notions, show that they are eminently likely in self-modeling systems, and provide some suggestions for alleviating their effects.

A. Self-Modeling Systems

Self-modeling systems [24] are self-aware systems that have the capability to build and analyze models of their own behavior, and use those models to produce as much of that behavior as the designers intend. We have already shown [20] [22] that we can build systems for which all of the system's components are models, including the model interpreter. While these Computationally Reflective systems are usually originally deployed with an initial set of models and model construction guidelines, their main self-modeling work is to adjust those models according to their observations of their operational external environment and their own internal environment, and produce their behavior by interpreting the models that they create. An engineering decision to be made by the developers is to decide how much of these behavior generating models are fixed and which ones can be variable (and the boundary can often usefully be changed at run time, depending on the situation).

Because these models are intended to be interpreted by the system, the common "4+1" view [16] is not entirely appropriate: we must mainly be concerned with process and interconnection models (behavior and interaction), and not much with developmental or use case views. Moreover, the system needs to build and maintain models of its operational environment also, not just its own structure.

The questions then become: how does a system

- learn about its own structure and behavior?
- make decisions about its own structure and behavior?
- decide on changes to its own structure and behavior?
- record and use the decisions and their consequences over time?

For us, all of these are about models or involve or require models. The corresponding engineering design problems are:

- how extensive these reflective mechanisms need to be for a given application,
- how much we want the system itself to construct and modify the models, and
- how much of the system behavior is to be driven by the models.

Models cost time for development and execution, which makes this set of engineering problems central to the success of self-adaptive systems in complex environments. If we make

this extensive use of models, we are brought to the notion of using the models to define the behavior.

These are self-modeling systems.

B. Representational Issues

In our prior work on self-modeling systems, we have identified two fundamental representational issues that must be addressed by any self-aware system of this kind:

1) the need for identifying repeated patterns and inventing new symbols for them, to reduce the computational load;
2) the need for occasionally re-inventing the representational mechanism itself, to avoid the inevitable increase in rigidity in a complex knowledge-intensive system.

These are both issues of Computational Semiotics [18] [21], since they involve the use and meaning of symbols in computing systems. The first issue is a generalization of some well-known optimization techniques, analogous to training, that replace the computation of appropriate responses to recognition of the appropriate situations (which is usually much faster). The second issue follows from our "Get Stuck" Theorems [18] [21], which essentially show that any increasing knowledge structure will eventually become unwieldy and extremely difficult to extend (even if humans are putting in the new knowledge, so it is not a computability problem). The theorems are formalizations of the long experience of failure in building extensible languages, systems, and programs.

These theorems are not insurmountable, but they do pose a serious design consideration. In addition, it can be seen that they are inevitable in any systems that build increasingly fine-grained models. That means that they are not unlikely; they must be addressed.

Helpfully, there are several processes that can be used to alleviate their effects to some extent. We think that these helpful processes are necessary in any case for a self-modeling system to operate in a complex environment for any non-trivial amount of time.

"Behavior Mining" is the process of recording and analyzing the system's own behavior to make descriptive or prescriptive models.

"Model Deficiency Analysis" is the process of evaluating the effectiveness of a model (not just success and failure, but also performance, resource implications, and interference with other models), and analyzing where improvements can be made.

"Knowledge Refactoring" is the process of rearranging the hierarchical knowledge structures (e.g., ontologies) to reduce dependencies, duplication and unused elements.

"Constructive Forgetting" is the process of simplifying knowledge structures by conflating and even removing some terms and relationships.

Each of these will be discussed in the rest of the paper.

None of these methods entirely avoids the problems presented by the theorems, of course; the theorems are still true. They do, however, push the bounds farther out, which allows bigger and more complex systems to exist in more complex and diverse environments for longer periods of time.

C. Structure of Paper

The rest of this paper is organized as follows. In the next Section II, we describe more of the structure of self-modeling systems.

In Section III, we describe our own approach to constructing self-modeling systems, using activity loops driven by knowledge bases.

In Section IV, we describe some fundamental results from Computational Semiotics that have relevance to model building at run time.

In Section V, we show how some partial approaches can alleviate the problem of getting stuck, and finally, in Section VI, we describe some conclusions and prospects for further development.

II. Self-Modeling Systems

Our goal is to design and build systems that are able to operate in hazardous or remote environments that cannot be completely known. The assumption is that they are too far away or too dynamically variable for a human controller, so they will need to operate largely autonomously.

To that end, we want these systems to build models of that environment, so they can reason about it and their potential activities. We expect the models to help them interact with their environment more effectively. Moreover, if we also make the systems build models of their own behavior, external and internal, then they can reason about that behavior and its connection to the environment. This approach to reflective systems therefore expects the system to use processes that build and use models of the system's external environment and processes, and also its internal environment and processes (see Figure 1). We have argued that this modeling is an

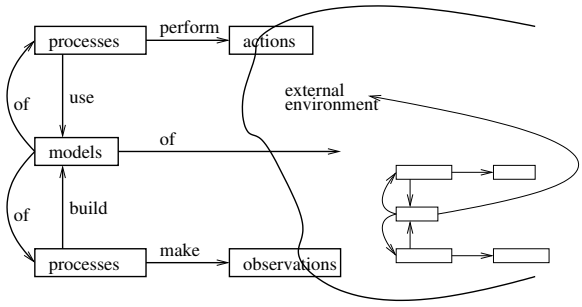

Fig. 1. Reflective System in Environment

important enabler for the flexibility required to adjust behavior to external conditions [20] [22].

During the course of their activities, these systems will build models of common situations and appropriate responses, so they can subsequently recognize a situation to help decide on a response more quickly. Then they can separate effective and ineffective responses, identify environmental or internal characteristics that distinguish these two classes, and extend the relevant situation descriptions to include the newly identified property. Therefore, these systems are in a continual state

of making new distinctions, thereby increasing the knowledge base size and interconnectivity.

However, this process cannot continue indefinitely, as we shall show in Section IV.

We have shown [24] how to build systems that consist entirely of models, including the model interpreter. These systems build models of system behavior (or are provided them by the designers), and use the model interpreter to execute them. The "Moby Hack" [33] [20] shows how to make this circular process well-defined. Because it seems unusual, we have a short explanation here.

We start with the interpreter written in the modeling language (which does put constraints on how simple the modeling language can be), and an external program (that will be used exactly once), written in any convenient language, that compiles the modeling language into an executable process. Then any model subsequently produced can be interpreted, including changes to the interpreter (see Figure 2). Changes

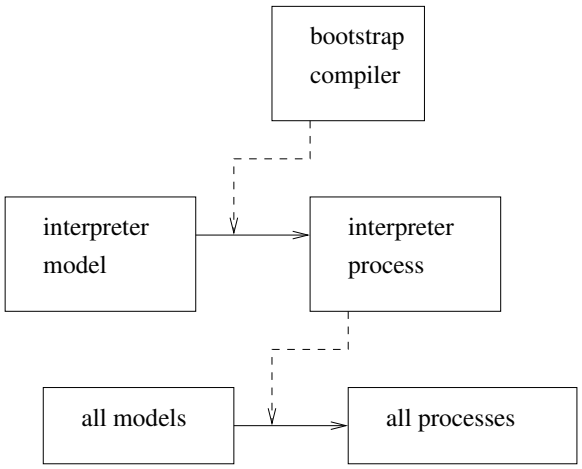

Fig. 2. Interpreter Bootstrap

to the modeling language are more problematic, and are left for further studies. In many applications, the interpreter will not change at run time, simplifying this process somewhat.

III. Our Approach

To manage the large collection of necessary models, we use the Wrapping integration infrastructure [17] [23], which allows us enormous flexibility in selecting and employing models. It was first designed to help study infrastructure decisions. What follows in this section is a short description of Wrappings, with its focus on resource management. More details can be found in the cited references.

A. Problem Posing Interpretation

The Problem Posing Interpretation is a different way to interpret programs that provides an enormous amount of flexibility consider in the assignment of semantics to syntax.

The basic idea is to separate information service *requests*, such as function calls, message sends, etc. from information service *providers*, such as function and method definitions. Compilers and interpreters can always tell the difference, so there is no reason to require them to use the same name (which is what we usually do).

That separation immediately allows (and requires) a mechanism to reconnect requests to providers. This change of interpretation allows many interesting properties:

- Code reuse without modification;
- Delaying language semantics to run-time,
- Migration to standards;
- Run-Time flexibility.

This last property is our interest here.

B. Wrapping Properties

Our choice for mapping problems to resources in context is to use a Knowledge Base that defines the various uses of each resource in different contexts; it allows different resources to be used for the same problem in different contexts.

This is a kind of implicit invocation.

There are five essential properties of Wrappings:

- ALL parts of the system are resources;
- ALL activities in the system are resource applications;
- Relevant system state is given by context;
- *Wrapping Knowledge Bases (WKBs)* contain the mappings;
- *Problem Managers (PMs)* interpret the Wrappings to perform all of the infrastructure and application functions.

C. Wrapping Processes

The processes in the Wrapping approach are as important as the data in the WKBs (it must be remembered that declarative programs do not DO anything without an interpreter). The PMs organize the computations and interpret the Wrappings. Two basic classes of PMs are the *Coordination Manager (CM)* and the *Study Manager (SM)*. These two PMs separate the control loop at the center of our approach into the looping "heartbeat" that keeps the system moving forward (the Coordination Manager or CM) from the basic planning steps within the loop (the Study Manager or SM).

1) Coordination Manager: The CM is a class of resources that are analogs of a system's main program. The simplest one consists of the following steps:

Find Context	Collect initial context.
loop :	
Pose Problem	Select one problem to study.
Study Problem	Find and use one resource for this problem.
Present Results	Make adjustments to context (and possibly Wrappings).

This is the analog of an extremely simple sequential main program (and also clearly of the classic "read-eval-print" style of basic LISP interpreter). There is a CM corresponding to any "main program" style (parallel, cooperative, etc.). Remember that each of these steps is also a posed problem, using Wrappings to map them to appropriate resources in context.

2) Study Manager: The Study Manager is a class of resources that are designed to address the problem "Study Problem" (from the CM). It reads and interprets the WKB (which is described in the next subsection). The simplest one consists of the following steps:

Match Resources	Match resources to problem in context (simple WKB filter).
Resolve Resources	Resolve resources via negotiation between problem and Wrappings to produce candidate resources.
Select Resource	Choose one of the remaining candidate resources.
Adapt Resource	Adapt resource to problem and context (finish instantiating all necessary invocation parameters).
Advise Poser	Record (announce) resource application (what is about to be done), together with relevant context (that was used for its construction).
Apply Resource	Use the resource for the problem (invoke the resource application).
Assess Results	Evaluate success or failure.

As always, each of these steps is a posed problem, using Wrappings to map them to resources. This is an extremely simple-minded planner. The main power of Wrappings is that any improvement in planning processes can be incorporated as other resources that can address these posed problems. Other SMs apply all relevant resources in turn until one succeeds, or accumulate the results of all successful applications, etc..

The interaction of the two main PMs is shown schematically in Figure 3, which we find to be a useful mnemonic for which steps occur when. The interaction of these two processes is the

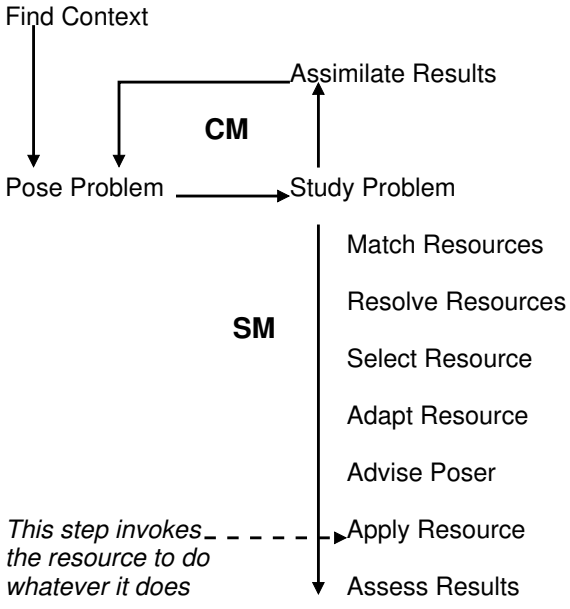

Fig. 3. CM and SM Steps

foundation of our self-modeling systems [22] [17].

D. Wrapping Knowledge Bases

The WKB contains entries called *Wrappings*. An individual Wrapping is a map from a problem specification to a resource application in context. Its main information components are:

PROB	problem name
COND	context condition (may be repeated)
RES	resource name
SYM	symbol name
FILE	file name

The *problem specification* consists of a problem name and a list of actual parameters. This is the form in which problems are posed. When we write about a *problem*, we are usually referring to a class of problems identified by the problem name (e.g., integrate, optimize, infer, etc.). The *context condition* is a sequence of boolean conditions (assumed conjunctive) on the problem parameters and the context variables (this allows the same problem to map to different resources if there are different problem parameters). The *resource application* is the fully bound invocation of the resource (all required parameters have values assigned). The *symbol and file names* are used to find the executable code for the resource. Most of our implementations use dynamic loading and unloading to reduce the size of the executable and retain the flexibility to add new or different resources for the same or different problems at run time.

The actual format of a WKB can vary (even heterogeneously); it is constrained only by the fact that it is read only by a few of the SM steps (minimally Match, Resolve, Apply), and as the resources for those steps can be selected differently at run time, so can the WKB have different syntax at run time.

E. Wrapping Summary

The power of Wrappings comes from the notion that each of the steps is a posed problem [19], for which the entire Wrapping infrastructure is available to identify an appropriate resource to apply.

It should also be clear that any system with such an explicit central activity loop (whether or not it is organized in the way we described) has a ready-made identifier for the events that it performs. The event labels can be considered as the basic representational language for modeling behavior.

F. Use of Wrappings

The use of Wrappings provides several conceptual unifications in system design:

- ALL activity is requested as posed problems.
- ALL activity is performed by resources, ALL processing is resource application.
- ALL activity can be recorded (Advise Poser in SM).

Wrappings help with some of the organization processes in self-modeling systems. They do not solve all of the hard problems, but they do help organize the solutions.

They allow a system

- To manage its collection of models;
- To coordinate its activities with other systems.

IV. Getting Stuck

As our self-modeling systems, or indeed any systems that constructs situational models at run time, behave in their operational environment, the natural tendency for us as designers is to assume that they will build more detailed models to make more detailed distinctions among the phenomena they encounter. This clearly could be a problem, and the following results show that it is definitely a problem.

The "Get Stuck" Theorems [18] [21] show some fundamental limits on representing knowledge in symbolic structures in computing systems, in particular, that systems operating in complex environments will eventually need to change their own internal representations. These theorems formalize the decades of observations that have the common theme that extending existing systems becomes more difficult and error-prone over time (e.g., Lehman's laws of software evolution [26] [27] [12] for programs that operate in the real world, among other informal notes), and in many cases has simply precluded anything but minor maintenance.

The first theorem is based on the fact that we can count how many structures of a given size can exist, so representing behavior in a sufficiently complex environment will eventually need to use so many expressions that they will become too large for the system to use either effectively or soon enough. The second theorem is based on the fact that each new element in a complex knowledge structure will need to attach to more of the other elements, and that therefore the practical constraints on adding new knowledge become more and more difficult to satisfy.

The first theorem is relatively straightforward to prove. If the computing system has a fixed finite symbol system (finitely many basic symbols, finitely many fixed arity combination rules), then we can count the number of expressions of any given size, using the well-known mathematical technique of generating functions [36]. The point is not that the values are large or small, it is that there are only finitely many expressions of any given size. That means, as the system tries to represent more and more complex phenomena or relationships, or make more detailed distinctions, it will need more and more expressions, so they will get larger and larger. The system will eventually run out of expressions small enough to process effectively.

Similarly, if we are building and extending a knowledge base, with references from entities to other entities, then the interconnection density eventually becomes large enough that the implications of potential changes take too much time to analyze. The important point here is that it will happen even if humans are doing the extensions.

These two theorems have implications for our computing systems: they will have to change their representational mechanisms. In the next Section, we describe some processes that can at least delay the problem.

V. Getting Unstuck

We have shown that if our system is effective at discovering new distinctions and making more refined models, then it will eventually "Get Stuck". In this Section, we describe some processes that will help alleviate the problem, but not, of course, eliminate it. The processes are:

- Behavior Mining,
- Model Deficiency Analysis,
- Dynamic Knowledge Management, and its main component processes
  - Knowledge Refactoring and
  - Constructive Forgetting.

These processes help delay the problem, and we believe that more such alleviations are necessary.

A. Behavior Mining

We use the term "Behavior Mining" for any analysis of the sequence of events performed by a system (there may of course be multiple parallel analyses at different resolutions and / or scopes). The intention is to identify and encapsulate common sequences, so that they may not need to be computed separately repeatedly. This is akin to encapsulating sequences of operations into "macros" to reduce computation (the target computation is the same, but the decision processes that construct the sequence are eliminated). The system will invent new symbols as event labels for these common event sequences, as well as other recurring structures (repetitions, alternations, etc.), and use them in a continual analysis.

The technical area of interest for this process is grammatical inference [7] [9] [3] [5] [13], which is about discovering higher-level structure among sequences of relatively low-level events. For different sets of assumptions about the high-level structure of these common sequences, different algorithms can be used to infer the higher-level organization of those structures. This is an enormous field, and we have included some references to illustrate some of the different approaches and theoretical results. One of the theorems is that most interesting classes of structures cannot be effectively inferred without having both positive and negative examples [10] [1] [2] (see [25] for a summary of these results).

Our Wrapping infrastructure allows for both positive and negative examples [22], since it makes available not only the context and reasons for choosing a particular resource to perform a computing step, but also the conditions that eliminated other possible resources. It therefore seems possible that these systems can infer more complex behavioral structures than other mechanisms can provide.

Once the system has new labels for some collection of common event structures, it becomes interesting to consider the possibility that all events can be placed into one of the identified commonalities ("macros"). When this happens, there is the possibility of defining the events solely in terms of the macros. That is, the event succession can be re-expressed in terms of the new symbols. If the system can actually express its behavior using these new symbols only, then we write that the new symbol set is "semiotically closed" [18] [21], which we believe is an essential component of emergence. This line of reasoning is important, but beyond the scope of this paper.

B. Model Deficiency Analysis

The processes we collectively call *Model Deficiency Analysis* are intended to detect problems with the models in use and provide suggestions for reducing the problems and improving the models. The system needs to be able to determine that a model has a problem, isolate the problem to particular sections of the model, and determine potential changes that will improve the model.

There are three key issues:

- How to monitor and validate the models,
- How to decide which model (if any) has failed or (preferably) is about to fail,
- How to use the failure to improve the model.

We can view these collected process as one step in a learning mechanism that is intended to improve the effectiveness of the model behavior. However, most learning applications assume a fairly class of parameterized models, and their process is to adjust the parameters to improve the behavior (which can be quite effective in some circumstances). Very few treat the models as explicit objects of study, and attempt to diagnose what is wrong when the models fail (as opposed to performing less well than they could).

A key property to this kind of analysis is that in Wrapping-based systems, there is a ready-made class of symbols that can be used for event names (the problem names), and a ready-made class of events that can be labeled (the resource applications announced in the SM). This property allows the system to construct models of its own actual behavior, which can be compared to the behavior intended by the designers (as specified, inevitably, by another model). Precise point of failure can be identified by incorrect or missing resource applications, and the event context (also reported by the SM in the Advise Poser step) and the WKB can be studied directly for errors and omissions in the context that was used to create the incorrect resource application (or not select the correct one).

Further discussion from a different context can be found in [23].

This kind of analysis is helpful for the infrastructure models, but not quite sufficient for creating and improving the domain models. For that part of the analysis process, we rely on some combination of simulation and evolutionary programming techniques [6] [11] This area is under active investigation.

C. Dynamic Knowledge Management

Wrappings are one kind of knowledge base in these systems, but there are certain to be many others to represent domain knowledge and other information resources.

We have not specified what form the knowledge bases are to take, since these methods are largely independent of that, and can be applied or adapted to most kinds of declarative knowledge. One of the most powerful time savers in accessing knowledge is context: the use of situational awareness to greatly reduce the search space for any posed problem. We describe two forms of context-based Dynamic Knowledge Management, time-based forgetting and context-based search reduction.

We envision this knowledge base with meta-knowledge on each element that contains a measure of its "popularity", that is, how often it has been used in recent reasoning processes. These values are adjusted as the system reasons, and the data structures used for the knowledge base should reflect that ordering. This kind of "dynamic rule promotion" keeps commonly used rules near the front of the search system, and is known to reduce search times. Then the system can occasionally discard some of the oldest and least used elements, as another way to do Constructive Forgetting.

Context describes the current and intended state of affairs of the system (taking goals and capabilities into account), and its best guess for the current and expected state of affairs of the environment (taking its predictive models of the environment into account). Then the available knowledge can be searched in an order determined by expected relevance (this ordering involves the indexing mechanisms, and does not need to move the knowledge around). In some cases, this re-ordering will delay the discovery of unexpected paths to a goal, or even prevent the system from finding a good course of action, but it is helpful on the whole. We clearly need better indexing and ordering mechanisms, relevance computations, and dynamically steerable search algorithms.

D. Knowledge Refactoring

Even if the system cannot define away its problems, it might be able to re-organize its knowledge structures to require much shorter expressions. Because this is a more complicated operation than the previous two, we spend more time discussing it. There are two parts to this kind of re-organization: the first is a simple change of emphasis for all of the models and rules, and the second is an actual re-arrangement. This re-arrangement has come to be known as refactoring, and is very popular in object-oriented coding circles [8] [15] [30]. There are many different kinds of refactoring operations, and which one to use first depends on some characteristics of the interconnection graph that have come to be called "smells" in the object-oriented coding world. For our application, we need "smell" analogs for knowledge bases, which are relatively easily adapted from [8], in addition to those that can be derived from first principles.

As an example of the latter, we note the property of "tangleness", that is, many elements all referring to each other. One simple measures of tangleness is the size of maximal cliques in the directed reference graph. Each maximal clique is a set of elements that all refer to each other, so it seems that there should be some partition of the set into subclasses, or even some new element that they are all connected to. For an n-clique, this changes the n * (n -1) / 2 edges between every pair of elements to n edges, from each of the elements to the new one. We have a speculative notion that small world graphs may be useful for this kind of structure [34] [35].

The notion of graph cliques brings up an entirely new speculation that is interesting and may become important. It

is well-known that the collection of cliques in an undirected graph is a simplicial complex, and therefore many topological methods, and in particular persistent homology, can be applied to understand the structure of the knowledge base at all scales [14] [32]. Our interest is in the recent improvements in computational speed [37] [38], which may allow these result to be computed on the fly. The question is which of these methods provides useful information.

Another intriguing possibility is about reconstructing the *symbolic dynamics* of a nonlinear system (a computable definition of what the system does and will do). Mischaikow et al. [31] show that, given a sequence of measurements of the system, a symbolic model with a finite number of states can be computed, and that model used for predictions and other analyses. Its utility is in proving that a certain dynamical system is chaotic (which is largely of theoretical interest), in separating noise from essentials (which is of great practical interest), and in determining invariant sets (sets that the dynamics preserves, which include all *attractors*, and gives hints about long-term behavior).

Each of these possibilities warrants much more study.

We close with some of the refactoring "smells" and corrective actions that may apply to knowledge based systems, whether they are embodied as rules or some other declarative mechanism. We describe such a knowledge base as having several sets of element definitions that define what objects are being considered, how they relate to each other and how they change. The entities and attributes define the objects, and there are relationships among the objects, and activities that change those objects. The key interconnection is the reference from one of these entities to another.

An entity with too many attributes should be separated into a hierarchy of multiple entities that group the attributes in a sensible way. An entity with too few attributes could be too general, needing some new attributes to make appropriate distinctions (of course, the system needs a reason to make those distinctions). Two entities with too many relationships should be better modeled as subordinate to a more general combined entity. Many different entities with the same set of attributes should be better modeled using a common subordinate containing those attributes. These and other changes are relatively easy. The old element references can be adjusted to these new ones (though the system will have to make up some names).

Note that this kind of refactoring is almost entirely experience driven, and has essentially nothing to do with simplifying different aspects of original models, since they usually do not exist.

The more radical restructurings of the knowledge base depend on the semantics of the elements and their connections, to which the system has little or no access, so these are generally left for the designers, for now (though there are certainly claims in the code refactoring world that many of these small changes can add up to a large one). Eventually, we expect to go farther along the path of having the system identify new core concepts, adjust the knowledge base, and translate old expressions into new ones.

There are many other warning signs that the knowledge base structure might be unwieldy, but none as strong as search delays or mistakes, which the system may not be able to detect when they happen, but might be able to diagnose after the fact.

This kind of refactoring is time-consuming, and very likely too static for the "on-line" system, so we expect our runtime systems to rely on the continual rearrangement of rule priorities according to context and situation, and to do more serious refactoring during down times (if there are any).

E. Constructive Forgetting

Since we know that the system will eventually run out of space, the system needs ways to reclaim that space. Our notion of "Constructive Forgetting" is about how to reduce the knowledge base without sacrificing performance. For this purpose, we think that the system can make simplifications by grouping similar clusters or sequences, and simply treating them all the same. If differences are eventually identified, then either the system can recover some of the distinctions, or it can recognize that it does not have the resources to cope.

This is an application of grammatical inference, and identifying classes of structures that have the same or a similar effect, or to which the system responds nearly identically. In those cases, the distinction is irrelevant to the response and it can be discarded. Of course, if subsequent experience shows that the distinction matters, the normal elaboration process will make that refinement.

It is extremely important for managing the balance among these processes that the system record all of the elaborations and collapsings, so they and their preconditions can be studied. For example, when the system operation seems to be getting slow (or the stored knowledge large), the threshold for approximate matching can be lowered, so that more responses are collapsed by this process. When system operation is relatively fast, the elaboration mechanism can be turned up a little bit, so that finer distinctions can be considered.

VI. Conclusions and Future Prospects

In this paper, we have described an application of self-modeling systems to autonomous systems that we expect to operate in hazardous or remote environments, and to adapt their behavior to the external (and internal, e.g., low battery) conditions they encounter. These systems refine designer-provided knowledge bases that describes what they should expect to encounter from the environment, from themselves, and from the complexities of their interactions. We have also described some methods for reducing the size and complexity of the knowledge bases in an attempt to stave off the "Get Stuck" Theorems, which will allow these self-modeling systems to operate autonomously for longer periods of time or in more dynamic environments.

In general, the question of how to balance expansion and contraction, and tune them to the complexity and dynamics of the environment, is the main sustainable system question, but it is our recommendation to keep the system near the higher

boundary, to maximize the available size of the semantic space. A more general answer requires experimentation with some engineering applications, and we expect the result to be quite application specific.

The important design question is how much of this variation machinery we actually need for a given application, and how much of the system structure can remain fixed (and unmodeled). This engineering question can only be answered in the context of a particular application.

The important implementation question is how to make the analysis fast enough to be useful, or how much of the model update process the system can afford to do. This question also is dependent on the particular application, because that will determine how much computing power is available for these sustainability functions.

We know that we can build these self-modeling systems. We think that we can learn to tune the balance rules effectively. We expect that they will be useful for difficult applications.

REFERENCES

[1] Dana Angluin, "Finding Patterns Common to a Set of Strings", J. Computer and System Sciences, vol.21, pp.46-62 (1980)

[2] Dana Angluin, "Inductive Inference of Formal Languages from Positive Data", Information and Control, vol.45, pp.117-135 (1980)

[3] Dana Angluin, Carl H. Smith, "Inductive Inference: Theory and Methods", *Computing Surveys*, vol.15, no.3, pp.237-269 (Sep 1983)

[4] 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)

[5] J. Berstel, L. Boasson, "Context-Free Languages", Chapter 2, pp.59-102 in Jan van Leeuwen (Managing Editor), *Handbook of Theoretical Computer Science vol. B: Formal Methods and Semantics*, MIT (1990)

[6] C. A. Coello Coello, G. B. Lamont, and D. A. Van Veldhuizen, *Evolutionary Algorithms for Solving Multi-Objective Problems*, Springer Science+Business Media (2007)

[7] J. M. Foster, *Automatic Syntactic Analysis*, American Elsevier (1970)

[8] Martin Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley (1999)

[9] K. S. Fu, T. L. Booth, "Grammatical Inference: introduction and survey", *IEEE Trans. Systems, Man, and Cybernetics*, vol.SMC-5, pp.95-111, pp.409-423 (1975)

[10] E. M. Gold, "Language Identification in the Limit", *Information and Control*, vol.10, pp.447-474 (1967)

[11] D. Hadka, P. Reed, "Borg: An Auto-Adaptive Many-Objective Evolutionary Computing Framework", *Evolutionary Computation*, vol.21, no.2, pp.231-259 (2013)

[12] Israel Herraiz, Daniel Rodriguez, Gregorio Robles and Jesus M. Gonzalez-Barahona, "The evolution of the laws of software evolution: A discussion based on a systematic literature review", *ACM Computing Surveys*, Vol.1, No.1 (June 2013)

[13] Colin de la Higuera, *Grammatical Inference: Learning Automata and Grammars*, Cambridge U. Press (2010)

[14] Danijela Horak, Slobodan Maletić and Milan Rajković, "Persistent homology of complex networks", *J. Stat. Mech.: Theory and Experiment* arXiv 0811.2203 (2008) https://arxiv.org/pdf/0811.2203 (availability last checked 24 Jul 2016)

[15] Joshua Kerievsky, *Refactoring to Patterns*, Pearson (2004)

[16] Philippe Kruchten, "Architectural Blueprints The 4+1 View Model of Software Architecture", *IEEE Software*, vol.12, no.6, pp.42-50 (November 1995)

[17] Christopher Landauer, "Infrastructure for Studying Infrastructure", *Proc. 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)

[18] Christopher Landauer, Kirstie L. Bellman, "Situation Assessment via Computational Semiotics", pp.712-717 in *Proc. ISAS'98: The 1998 International MultiDisciplinary Conference on Intelligent Systems and Semiotics*, 14-17 Sep 1998, NIST, Gaithersburg, Maryland (1998)

[19] 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)

[20] 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)

[21] Christopher Landauer, Kirstie L. Bellman, "Semiotic Processing in Constructed Complex Systems", *Proc. CSIS2002: The 4th International Workshop on Computational Semiotics for Intelligent Systems*, *JCIS2002: The 6th Joint Conference on Information Sciences*, 08-13 Mar 2002, Research Triangle Park, NC (2002)

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

[23] Christopher Landauer, Kirstie L. Bellman, "Model-Based Cooperative System Engineering and Integration", *Proc. SiSSy 2016: The Third Workshop on Self-Improving Self-Integrating Systems*, 19 Jul 2016, Würzburg, Germany (2016)

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

[25] Lillian Lee, "Learning of Context-Free Languages: A Survey of the Literature", Technical Report TR-12-96, Harvard U. (1996)

[26] Meir M. Lehman, "Programs, Life Cycles, and Laws of Software Evolution", *Proc. IEEE*, vol.68, No.9, pp.1060-1076 (1980)

[27] M. M. Lehman and L. A. Belady (eds.), *Program evolution. Processes of software change*, Academic Press (1985)Professional, Inc., San Diego, CA, USA

[28] Pattie Maes, "Computational Reflection", technical report 87_2, Vrije Universiteit Brussel, Artificial Intelligence Laboratory (1987)

[29] Pattie Maes, "Concepts and Experiments in Computational Reflection", pp. 147-155 in *Proceedings OOPSLA '87* (1987)

[30] Q. Ethan McCallum, *Bad Data Handbook*, O'Reilly (2012)

[31] Konstantin Mischaikow, Marian Mrozek, J. D. Reiss, Andrzej Szymczak, "Construction of Symbolic Dynamics from Experimental Time Series", Phys. Rev. Lett. vol.82, pp.1144 (8 Feb 1999)

[32] Giovanni Petri, Martina Scolamiero, Irene Donato, Francesco Vaccarino. "Topological Strata of Weighted Complex Networks", PLOS One, vol.8, no.6, e66506 (2013)

[33] Ken Thompson, "Reflections on Trusting Trust", *Comm. of the ACM*, vol.27, no.8, pp.761-763 (Aug 1984), also at URL http://www.acm.org/classics/sep95/ (availability last checked 24 Jul 2016); see also the commentary in the "back door" entry of "The Jargon Lexicon", which is widely available on the Web, and in particular, is findable by searching with Google for "back door moby hack" (availability last checked 24 Jul 2016)

[34] Duncan J. Watts, Steven H. Strogatz, "Collective dynamics of 'small-world' networks". *Nature*, vol.393, pp.440-442 (1998)

[35] Duncan J. Watts, *Small Worlds: The Dynamics of Networks between Order and Randomness*, Princeton U. Press (2003)

[36] Herbert S. Wilf, *generatingfunctionology*, Academic Press (1990)

[37] Afra Zomorodian, "The Tidy Set: A Minimal Simplicial Set for Computing Homology of Clique Complexes", pp.257-266 in Afra Zomorodian (ed.), *Proc. SoCG10: 26th Symposium on Computational Geometry*, 13-16 Jun 2010, Snowbird, Utah (2010)

[38] Afra Zomorodian, *Topological Data Analysis: Proc. 2011 AMS Short Course on Computational Topology*, 04-05 Jan 2011, New Orleans, Louisiana, Symposia in Applied Mathematics, v.70, AMS (2012)