Mitigating the Inevitable Failure of Knowledge Representation

DRAFT last changed 23:00 PDT, 05 April 2017 at nguyen

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

***Abstract*—This paper is a continuation of a previous paper on self-modeling systems, concerning mitigation methods for the Get Stuck Theorems, which are powerful theorems about the limits of knowledge representation. The First Get Stuck Theorem says that since there are only finitely many data structures of any given size, it follows that as a system tries to save more and more data / information / knowledge, the structures necessarily get larger, and eventually they are too large for effective computation.**

The mitigations we described are Behavior Mining, which is about building models of the system and environment behavior, Model Deficiency Analysis, which is about assessing the efficacy of those models and determining how to improve them, Knowledge Refactoring, which is about restructuring the saved data for more efficient access and smaller storage, and Constructive Forgetting, which is about explicitly discarding some data that is deemed to be less critical.

We argue that these classes of mitigations, and a couple of new ones, can help a system retain effectively computable knowledge structures in a dynamic environment at higher levels of difficulty (of course, as the environment gets more dynamic, all systems, including individual biological organisms and even species, eventually fail).

Keywords: Self-Aware Systems, Self-Adaptive Systems, Self-Modeling Systems, Get Stuck Theorems, Behavior Mining, Model Deficiency Analysis, Knowledge Refactoring

I. Introduction

This paper is a continuation of a previous paper on Self-Modeling Systems [55], concerning mitigation methods for mitigating the effects of the Get Stuck Theorems [48] [51] (see [55] for a more recent explanation). These theorems are powerful theorems about the limits of knowledge representation, and are easily proved using the classic mathematical technique of generating functions [77] [28]. The First Get Stuck Theorem says that since there are only finitely many data structures of any given size, it follows that as a system tries to save more and more data / information / knowledge, the structures necessarily get larger, and eventually they are too large for effective computation.

The Second Get Stuck Theorem says that as knowledge structures become richer, they become more interconnected, and it becomes harder for new knowledge to be added consistently, even when humans are doing the adding. This problem has been noticed in every extensible programming language, knowledge base, and ontology development project since the Rapidly Extensible Language (REL) Project at Caltech in the 1970's. It is wonderful at the beginning, gradually gets more difficult, and unless there is a dedicated coordinating team to keep things organized, it gets too hard to use and fades out. There are some approaches to alleviating this problem directly, with certain kind of knowledge refinement used to incorporate more distinctions by expanding the reliance on context, but discussing this further would go beyond the scope of this paper. It will be treated in future papers.

A. Why Self-Modeling?

The question then naturally arises: why use Self-Modeling Systems if they have these problems? Firstly, these theorems apply to all systems that build models; it is only that Self-Modeling Systems build a lot of them. We remember that Self-Modeling, like Self-Awareness, is an approach to reach a goal, not an end in itself. The goal is survivability in these difficult environments, and as much operational competence as possible, and we think that Self-Awareness is essentially the only effective way to do that [9] [44] (and we particularly want to achieve Self-Awareness via Self-Modeling [50]). The construction methods we use are fairly complicated [49] [53] [54], but we have repeatedly shown that this level of difficulty is warranted in sufficiently hazardous environments.

We want to build systems that are capable of operating in enormously difficult environments, such as search and rescue robots for collapsed buildings, deep ocean exploratory vehicles, remote astronomical probes, etc.. These environments are dynamic, largely unpredictable, and almost completely uncontrollable. Some of them are inherently harsh (e.g., due to pressure, corrosive chemicals, radio interference, radiation, vacuum and many other features), and some are so remote that human operators have no effective way to handle emergencies. The systems must do for themselves.

For these systems to be effective, they must have a large degree of autonomy, and have excellent adaptation capabilities to enable robust and resilient behavior. We think that run-time models are a minimum necessity for significant self-awareness, and our architecture makes significant use of models throughout [50]. In fact, all knowledge in the system is contained in models, some dynamic, some descriptive of dynamics, some just descriptive of relationships.

Of course, we live and die by our models, and the extensive use of models brings a few problems of its own [55], which we have begun to address.

B. Mitigations

The mitigations we described before are Behavior Mining, which is about building models of the system and environment behavior, Model Deficiency Analysis, which is about assessing the efficacy of those models and determining how to improve them, Knowledge Refactoring, which is about restructuring the saved data for more efficient access and smaller storage, and Constructive Forgetting, which is about explicitly discarding some data that is deemed to be less critical.

The new mitigation processes are Dynamic Knowledge Management (which was mentioned in passing before), which is about how to organize the saved data for more efficient access, and Continual Contemplation, which is a process that runs in the background all the time (except during emergencies), that continually examines system and environment behavior, looking for converging evidence of interesting phenomena (of course, the definition of "interesting" depends on the available data, goals, and resources).

We showed how these mitigations can help a system retain effectively computable knowledge structures in a dynamic environment at higher levels of difficulty (of course, as the environment gets more dynamic, all systems, including individual biological organisms and even species, eventually fail).

In this paper, we explore these topics more deeply. We cannot prove that these methods are sufficient, so there may be other mitigating methods that are useful for a system to retain effectively computable knowledge structures in a dynamic environment, but we believe that these ones go a long way towards that goal.

C. Structure of Rest of Paper

The structure of the rest of this paper is as follows: in the next Section II, we describe the behavioral architecture, which defines what the system has to do. In the following Section III, we introduce the various mitigations, expanded from the previous paper [55], explain why we think that they are important, and provide some indication of how they might be implemented.

We do not discuss the system architecture we use, or the Wrapping infrastructure that gives it its power. We have been developing these for some time [6] [49] [52] [56] [47] [55], but most of the details have to be left to these other references.

Suffice it to say that in contrast to most reference architectures [4] [12] [23] [35], a Wrapping-based system has the meta-level processes in the same conceptual level as the object-level processes (we have shown that this can work [52] [8] [57], despite its seemingly intentional confusion, and there is a famous example that show it is not even very hard [73]), so any part of the system can be the subject of a reflection process, including the parts performing the reflections.

The infrastructure also avoids the undecideability issues in a typically pragmatic engineering way: if the system can't figure something out in a small enough amount of time, it moves on to another, presumably easier, version of the problem, perhaps giving up some enhanced performance or capability for lack of ways to show it will work in the current context and under the current time constraints (both for showing and for doing).

And finally, in Section IV, we present our conclusions and prospects for further advances.

II. Behavioral Architecture

The Behavioral Architecture of a Self-Modeling System has several fundamental activities that we believe must be present at some level of detail. These functions generally relate to a practical compromise among goals requested by the operators or imposed by the designers, current capabilities of the system, and exigent conditions in the system environment. Where the designers place that compromise is an engineering decision. We strongly support the notion that design time models are (the initial) run time models [74], since design time does not end with deployment.

These systems should recognize their own failures (to a certain extent), change their own behavior via model adjustment (when possible), use Knowledge Refactoring and Constructive Forgetting for efficiency, and report on their operational and maintenance activities (this is system telemetry, generally lower priority than mission data, but essential nonetheless for longer term system improvement).

In this Section, we introduce the functions, compare them to some recent reference architectures [4] [23] [33] [34], and then in the next Section III, we explain how the mitigation methods address those functions.

A. Situation Awareness

The system has to be able to

"know" what is going on (situation awareness).

That knowledge includes possible system actions and environmental effects, which requires predictive models of behaviors and simulations to examine them. This is the Self-Awareness ("introspection" in [17], which also includes reasoning) of Self-Modeling Systems. We believe that it is important to have these models exist at multiple levels of resolution in time and scope [1] [63], so that reasoning can take place at the lowest adequate level of detail.

These models will also include historical knowledge at multiple granularities. The relevant reasoning processes will need to smoothly move from one level of detail to another, and to relate them by generalization, simplification, analogy and metaphor, abstraction and reification. These reasoning processes and knowledge representations, and how they interact with uncertainties and inconsistencies, are beyond the scope of this paper (they are an entire research area by themselves).

This is where the almost all of the model creation occurs.

There are many ways to create models of behavior, but all of them rely on being able to observer behavior "from the inside", so to speak. That is, it is far simpler to have internal instrumentation that shows what the behavior is intended to be than to try to infer it from external observations of effects. On the other hand, the difference between the intentions and the actions is an important reality check (see Section II-C below).

These systems need methods for generating some kind of execution traces, whether by infrastructure adjustment [3] or inherently as in Wrappings [49].

The context models of [4] fulfill some of this function, but we did not see very much about the creation of new models (see [5] for one example approach). The internal and external sensors of [23], together with learning processes that are not described, fulfill this function. There is essentially no model-building in [33] [35], only modules that have certain control responsibilities. The description of Descartes in [34] explicitly mentions the adjustment of models, and addresses the initial creation via extraction.

B. System Expectations

The system has to be able to

> "know" what should or should not be going on (behavior expectations).

These are expressed as specification and/or expectation models, which are mapped to the current symbol system or they cannot be interpreted, They are generally computed from the purposes and goals of the system, and mapped into these models at design or construction time. In the most interesting cases, the provided goals are sufficiently high level that the system must do some of these mappings at run time, since the goal interpretation cannot always be completed outside the current situation (this author's paraphrase):

> No plan ... extends with any certainty beyond the first contact with ... [reality], reference [64], p.92

This topic represents a difficult issue for designers, since in some cases, direct contravention of one goal is the only viable solution for another (especially in emergencies, the goals will be different, and this contingency should be part of the design). It means that effective goal specifications are more difficult than they seem, and some kind of preference mechanism needs to be included.

We have described a system engineering process that takes into account stakeholder expectations, defined via scenarios and corresponding system behavior expectations, and produces the components used in a Wrapping-based system [46], though clearly other processes could be used [7] [53] [54]. It should now be combined with the appropriate assurance methods [25] or other behavior controlling methods [65] to allow designers to show that the system's run time changes are not going to cause trouble.

The Goal Management Layer of [4] addresses this function. There is very little about the source of goals in [23]. There is no discussion of the source of the goals in [33] The description of Descartes in [34] does not mention the source of the goals.

C. Error and Anomaly Detection

The system has to be able to

> figure out whether, what and why anything went wrong (anomaly detection and diagnosis).

This process uses self-monitoring to gather the original data, and compare it to expectations. Problems that occur in the execution of the system can be due to incorrect assumptions about environment behavior or incorrect implementation or interpretation of the system specification. Failures of environmental expectation models can only be corrected at run time. Incorrect implementations can be examined by serious system verification at system construction time, and a continuing examination of specifications and operations in any newly modified processes. Incorrect interpretation of the system specification can be examined by serious system validation at design time, incorporating an effective number of scenarios defining expected and expectable situations and the stakeholder expectations for corresponding system behavior. These are much harder to correct at run time because they are errors of expectation, not errors of behavior.

Even if we now assume that the right system expectations were provided (and validated), and that the system was implemented flawlessly (and verified), and that the hardware has not failed or degraded, there are still anomalies and other potential problems to be considered. For our purposes, the term *anomaly* means some peculiarity of behavior that might be an error, but might not, and usually only domain expertise can determine which. Our systems may have a lot of domain knowledge, as provided by domain experts, but they will generally not have domain expertise.

A different kind of problem can occur when there is not enough knowledge to reason about system behavior. These gaps in the knowledge are difficult to detect; we are only beginning to address methods for the system to find out that something is missing.

Certain kinds of performance issues can be considered to be failures also, and the system itself can only go so far to help (optimizing expressions, remembering common combinations of actions, etc.). These issues are more properly considered at design time, because they are difficult, and our recommendation is to have multiple approaches to some of the more time-consuming analyses, with varying levels of accuracy and resolution, so that at run time, the system can decide how much time it has and act accordingly.

For our Wrapping-based systems, the context specification in a Wrapping may be incorrect, which can cause resources to be used inappropriately (conditions too weak), or the right ones to be missed (conditions too strong). Here we can only be very careful about the conditions under which each resource is applied. In addition, the behavior descriptions can be incorrect, for which we can only be very careful about what the resources do. This care can seem to be a burden at design time, but it is known to be extremely valuable at run time.

The analyzer of the Configuration Management Layer of [4] addresses this function. There is little mention of this function in [23]. There are analyze and reason modules in [33], but no discussion of error detection and diagnosis. The description of Descartes in [34] explicitly mentions detecting problems.

D. Model Change Design and Test

The system has to be able to

> design and test changes to its models accordingly.

This is the Self-Adaptation ("intercession" in [17]) of Self-Modeling Systems. This is one of the hardest areas, because it seems to be the most "creative", finding solutions to problems by designing models.

The reasoner of the Configuration Management Layer of [4] partially addresses this function, and the Capability and Plan Models of the Base Layer are also involved in the modification of models, but we did not see much about creation of models. There is little mention of this function in [23], except to refer to learning methods. There is essentially no mention of this function in [33]. The description of Descartes in [34] explicitly mentions the construction of adaptation plans, but does not explain how the models have to be changed.

This function is an essential part of self-adaptation, but none of the mitigations is about model change design and test, since we do not have very good approaches yet. For example, this is where we might use evolutionary programming or some other search technique to find a suitable model [18] [40], but we have seen no truly successful examples.

Other methods for detecting the need for changes include the short term model deficiencies described in Section III-B below, and the longer term system evolution. If the self-aware system retains models as databases, then there are methods for detecting and making changes [36] (this is also related to Knowledge Refactoring), though there seems to be an expectation that the impetus for changes is external to the system.

Changing the nature of a system at run time is not restricted to self-aware systems [21], and the more general versions can also be applied. In fact, it can be argued that having an explicit design model can help a reflective system decide on some of its own changes by reflecting on the design [20]. This process includes comparing behavior to intentions specified with the design, and deciding what needs to change and how to change it. The latter steps are both fairly difficult.

Of course, some changes are predictable; we know that the system will want to re-organize its priorities in the light of environmental behavior, and we know that some of the clever code we have inserted will never be used (we typically just don't know which parts). We can assist the systems we build with a set of adaptation plans that it can use when the occasion warrants, and make them also adaptable [61].

E. Get Stuck Theorem Mitigation

The system has to be able to

mitigate the Get Stuck theorems.

This is one of the newest functions, not mentioned in the other Reference Architectures. We believe that it must be part of managing all of these many models, generally using some process for Constructive Forgetting (along with others). We call the other processes described in this Section mitigations also because they set up the knowledge structures in a way that facilitates Constructive Forgetting.

III. Mitigations

There are six basic mitigation strategies:

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

In this Section, we describe each in turn, and provide some indication of how they might be implemented. It will be clear that each of these has many different kinds of implementation, at differing levels of effort and capability.

In the simplest terms, Behavior Mining makes models (models are represented knowledge), Model Deficiency Analysis detects deficiencies and changes models, Dynamic Knowledge Management organizes the knowledge for efficient access, Knowledge Refactoring detects infelicities and changes the represented knowledge, Constructive Forgetting simplifies the represented knowledge, and Continual Contemplation watches all the processes looking for improvements or anomalies.

A. Behavior Mining

Behavior Mining is the process of creating models of the behavior of the system (and its environment), based on observations from external sensors and internal instrumentation. These models can be used for process improvement and error correction, and for prediction of likely environment behavior and likely effects of system decisions. These models are the knowledge that the system has about itself, its environment, the interaction between them, and the goals / constraints it has been given. In a strong sense, the desire for these capabilities underlies all Models@Run.Time approaches.

Behavior Mining is where the various methods and algorithms for model construction are used, to build prediction and expectation models at multiple time and subject scale and scope. These methods include grammatical inference [39] and much more sophisticated mathematical methods.

We expect that the exigencies of system development mean that we cannot expect the system to be capable of building all of its own models, so whatever mechanisms we choose must allow some models to be provided by the designers: typically goals, anti-goals (things not to do), and starter models for the system, the expected environment and their interaction. We also must allow some models to be fixed by the designers (not spontaneously changeable), but then mounting evidence of problems can be used to request changes (fundamental system goals are likely to be in this category).

These models can come from many sources: the simplest are the performance models used to find bottlenecks [71], which have been widely used for a long time. Models can be elicited from users or extracted from running software [78] [5].

The general process of building and refining knowledge is akin to concept formation: the system distinguishes contexts that have different implications (via environment behavior changes or differing responses), splits concepts according to differing context and usage (classic knowledge elaboration methods), and uses whatever learning processes are available

to refine them [40] (and many other learning methods, which are beyond the scope of this paper).

There is a strong sense in which Behavior Mining is not new; every self-aware system has to be doing some of these processes already, and we are just emphasizing it as a major function, according to our emphasis on the models.

For the simpler arena of sequence identification, methods of grammatical inference [39] have been known to be useful, in combination with parsing methods for testing inferred models in a simulated environment during a training phase at construction time [29]. We can also use formal languages and power series as a reality check on generated grammars [26] [13] [70] (there is a simple eigenvalue test for finite expected length of strings generated by a context-free grammar; it can be used to bound the expected length of strings generated by a context-sensitive grammar).

A powerful way of creating models of observations is inductive inference [2], which goes beyond sequence pattern identification to look at other commonalities, such as sequence identification and next term prediction, function definition from examples, etc..

Other dynamic models are also interesting [78]. Temporal logic allows us to build models that have notions of "eventually" and "always" [69] [79] [80], which are useful to discover or verify system invariants (properties that are always true or always false, such as the low-level "this variable is an integer between 0 and 63", or the higher-level "these two processes are never operating concurrently"), and semi-invariants (properties that can sometimes be allowed to be true, but the system always eventually makes them false again, such as the low-level "the task queue is always in a consistent state" or the higher-level "the knowledge base has no unreachable elements").

When the sequences are intended to have embedded timing information, the process is a little harder (because the time intervals and the observations thereof may not be precisely the same for every observation).

And, as always, whatever logics and other reasoning processes are used need to deal with inconsistency [14] [15] [19] and uncertainty [45] [32] in a coherent way. This is especially important in our case, when the system is creating and adjusting many models at the same time.

The simplest argument for models of internal behavior is performance modeling to find bottlenecks [71].

A permanent problem in all of these approaches is bad data [62]. We need processes to recognize it (not just outliers), ignore it when it should be ignored, and process it when we are trying to understand it.

B. Model Deficiency Analysis

Model Deficiency Analysis is the process of analyzing model behavior, detecting anomalies, mistakes and other deficiencies, ascribing them to particular model structures or interactions when possible, inventing adjustments to structures or interaction protocols that fix the problem, or at least reduce it, and changing the models accordingly.

It includes comparison of behavior models with intention models, deciding what component or interaction failed and why, creating potential changes. and internal simulations to assess effects of potential changes.

We think that parts of this process are very hard in general, but nonetheless essential. For example, in many complex systems, the components are distributed, heterogeneous, not particularly well-specified, and none of them was designed to work with others. A common important problem is what we call *emergent failure*: no one component or interaction is to blame; something else happens. Correcting emergent failure is extremely difficult, since we need to examine a much larger space of timed interactions than is necessary for any one of the component systems or for any of the component pair interactions.

In CARS (Computational Architecture for Reflective Systems), [8] [57], we have a set of reflective processes comparing sensor readings with the effector commands issued (more precisely, the expected effect on the sensor readings due to the command), to detect inconsistencies and initiate a fault-management process.

This fault-detection via sensor prediction is also nicely treated in [41]. Another method is presented in [82]. While model mismatch is a source of problems [32] it is not exactly a source of uncertainty (uncertainty is more like measurement precision; model mismatch is usually an error).

In addition to the model change need detection mechanisms described above in Section II-D, model deficiencies are an important reason to want to change them.

For example, we can use run-time probabilistic model checking [66] to get a notion of the likelihood of a model failure, or we can perform a "root cause analysis" [72] to determine what caused a failure and sometimes how to fix it (since most errors seem to be simple ones). Even when the model under consideration is a reasoner [24] [67], we can examine reasoning failures to improve it [22], by separating reasoning problems from lack of information.

C. Dynamic Knowledge Management

Dynamic Knowledge Management is the process of organizing the data structures that represent knowledge in the system for efficient access. The intention is to make frequent (or important) accesses quick, and allowing rare and less important accesses to take longer. The notion of which accesses are more important is also dynamic, as it depends on the operational context at the time. That means that the knowledge is undergoing continual assessment and rearrangement, tempered by notions of "it is good enough for now"; there is no optimization in this process.

One approach to organizing this knowledge is to use a multi-level archive, separated by immediacy, much like all modern computing storage, within the CPU (caches) and not (memory, disks, etc.). These levels might be selected as

- direct and indexed (cache);
- direct and less indexed (main storage);
- indirect and less indexed (hard drives);

- easy compressed (less compression, quicker access);
- hard compressed (more compression, slower access);
- gone;

Of course, experiments are needed to determine how much of this organization is warranted for any particular application.

This is not the same as organizing by generality, with more specificity intended for more efficiency? then choices and distinctions are retained, and only the most specific things are archived / discarded, but that can be used as part of the criteria. Other re-ordering criteria can include popularity or assessed importance (use in many reasoning arguments), the utility of information, or the utility of distinctions (e.g., is it important that these two labels remain distinct?). It is also essential to note the trade-off between compressing the data and the length and kind of indexes [27]; the indexing should move up in abstraction levels as the data gets more remote, (which can mean old, unreferenced, superseded, or any other component of the re-ordering criteria.

Another intriguing idea is to consider small world graphs [75] [76] as an organizing principle, because ordinary association graphs are not like this. The main question to be considered here is how to make these graphs, and what criteria to use. Then the system can use difficulty to infer the existence of new nodes. This is another little-studied research topic.

D. Knowledge Refactoring

This approach is part of Dynamic Knowledge Management, but we separated it because it is more specific, and because there is other work in a related area, refactoring of program source code [30] [42].

Knowledge Refactoring is the process of re-arranging knowledge structures and their interconnections for better access properties (better can mean faster, but also flexibility and abstraction are important). This process includes determining that a refactoring is necessary (or just useful; a relevant criterion is how busy the system is at the moment, and how much disruption it can tolerate just now).

This is the point of Knowledge Refactoring: how to decide what to fix and how. The first question, how and why to dislike a knowledge structure, is about deciding what features are so bad about a structure that it warrants refactoring. Since our purpose for the knowledge structure is to be quickly informative, our criteria are mainly about ease and speed of access.

The simplest way to treat this problem is as a graph re-arrangement problem, where we imagine each relationship among knowledge elements as an edge (of course, what the actual knowledge elements are, especially their granularity, remains to be decided). Then we can weight edges by how often they were traversed during normal (and anomalous) operation, and try to make heavily weighted paths shorter (by collecting the knowledge elements near each other). Similarly, lightly weighted edges can be allowed to be longer, so some can be saved farther away. Here, near and far refer both to direct pointers and to different levels of immediacy in the Knowledge Archive Structure described above. Clearly, there is much experimentation to be done here.

This function includes testing the effects of proposed model changes to validate the changes in multiple situations. For example, there will be internal simulation of hypotheticals with observation of results, to evaluate the effectiveness of a proposed change. The system will maintain a small collection of canonical problems that it can use for any such speculative reconfiguration.

E. Constructive Forgetting

Constructive Forgetting is the process of throwing out knowledge that has become obsolete, overly burdensome, or just not important enough to keep right now. The importance measures are a balance between frequency of use (frequently used reasoning processes get higher importance) and danger of damage (extremely damaging possibilities get higher importance, even if unlikely).

This process competes / cooperates with Behavior Mining and Knowledge Refactoring for access to the knowledge structures. It shares the notion of importance assessment assigned by Knowledge Management, and moves elements from nearer to farther in the access hierarchy.

In fact, we can treat the various Knowledge management functions as separate agents, and use distributed system models to help them coordinate [68].

F. Continual Contemplation

Continual Contemplation is the background process of examining all available knowledge or previously unknown relationships, with the intent of discovering properties that are present in the system, but not yet properly correlated with others. Such a process should run at a low priority, so it does not interfere with the operational behavior of the system, especially in emergencies.

This is an inherently n^2 problem if n is the size of the knowledge base, so there must be some content-based partitioning to make it feasible. It will create hypotheses, run simulations to evaluate them, possibly using some version of evolutionary programming or simulated annealing to improve them, definitely including certain kinds of time series analysis and may other advanced mathematical methods. This is a classic set of very hard problems, not considered further for this paper.

An even harder problem is to decide that something cannot be concluded from the available data. This problem may arise in coordinating multiple actions and planning complex actions, when a certain goal is actually not reachable using the available methods and data. As before, we will avoid any decideability issues using resource and time bounds (i.e., if the system cannot draw the conclusion in a useful amount of time, it abandons it without assuming that it is false; it is recorded as merely being too hard to decide).

IV. Conclusions and Prospects

,

We have shown earlier that any self-aware system in a complex environment will run afoul of the Get Stuck Theorems. In this paper, we have described several processes and algorithms that mitigate the effects of the theorems, though, of course, they do not prevent the effects, only delay them.

We have argued that many of these processes are necessary for self-aware systems to survive for long periods of time, and acknowledged that each application will have a different trade-off for how much of this machinery is needed.

Many mathematical methods are useful in these modeling analyses, but many of them are likely to be too slow. We want to examine them to determine which ones would be more useful if they could be made faster (even if only approximate). Our recommendation for multiple levels of resolution helps in this regard, since the system can compute only to its available time, and then incrementally increase the resolution as needed when it is not so busy.

These issues and others (e.g., collective awareness) will be studied in the context of our CARS application.

For the future, we have only a little to add to the outlook in [16] [60], mainly in the area of much more complicated information and knowledge processing, and not so much in infrastructure organization, since we already start with an infrastructure based on Wrappings that is capable of managing all this flexibility.

REFERENCES

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

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

[3] Lorena Arcega, Jaime Font, Øystein Haugen and Carlos Cetina, "An Infrastructure for Generating Run-time Model Traces for Maintenance Tasks", in [38]

[4] Uwe Aßmann, Sebastian Götz, Jean-Marc Jézéquel, Brice Morin and Mario Trapp, "A Reference Architecture and Roadmap for Models@run.time Systems", p.1-18 in [11]

[5] Marco Autili, Davide De Ruscio, Paola Inverardi, Patrizio Pelliccione, and Massimo Tivoli, "ModelLAND: Where Do Models Come from?", p.162-187 in [11]

[6] Kirstie L. Bellman, "An Approach to Integrating and Creating Flexible Software Environments Supporting the Design of Complex Systems", pp. 1101-1105 in *Proceedings of WSC'91: The 1991 Winter Simulation Conference*, 08-11 December 1991, Phoenix, Arizona (1991)

[7] Kirstie L. Bellman, Christopher Landauer, Phyllis R. Nelson, "Systems Engineering for Organic Computing: The Challenge of Shared Design and Control between OC Systems and their Human Engineers", Chapter 3, p.25-80 in [81]

[8] 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 2010, Carmona, Spain (2010)

[9] 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 [43]

[10] Nelly Bencomo, Robert France, Sebastian Götz, Bernhard Rumpe (eds.), *Proceedings of the 8th Workshop on Models @ Run.time*, 29 September 2013, co-located with *MODELS 2013: the 16th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, 29 September - 04 October 2013, Miami, Florida, USA (2013)

[11] Nelly Bencomo, Robert France, Betty H. C. Cheng, Uwe Aßmann (eds.), *Models@run.time*, SLNCS 8378, Springer (2014)

[12] Amel Bennaceur, Robert France, Giordano Tamburrelli, Thomas Vogel, Pieter J. Masterman, Walter Cazzola, Fabio Costa, Alfonso Pierantonio, Matthias Tichy, Mehmet Akşit, Pär Emmanuelson, Huang Gang, Nikolaus Georgantas, and David Redlich, "Mechanisms for Leveraging Models at Runtime in Self-adaptive Software", p.19-46 in [11]

[13] J. Berstel, L. Boasson, "Context-Free Languages", Chapter 2, pp.59-102 in [58]

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

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

[16] 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 [43]

[17] D.G. Bobrow, R.G. Gabriel, J.L. White, "CLOS in Context - the Shape of the Design Space", p.29-61 in *OOP: the CLOS Perspective*, MIT (1993)

[18] Jürgen Branke, Hartmut Schmeck, "Evolutionary Design of Emergent Behavior", Chapter 6, p.123-140 in [81]

[19] Walter A. Carnielli, M.E. Coniglio and J. Marcos, "Logics of Formal Inconsistency", pp. 15-107 in [31]

[20] Walter Cazzola, "Evolution as ŀReflections on the Design", p.259-278 in [11]

[21] Walter Cazzola, Nicole Alicia Rossini, Phillippa Bennett, Sai Pradeep Mandalaparty, and Robert France, "Fine-Grained Semi-automated Runtime Evolution", p.237-258 in [11]

[22] Hans Chalupsky and Tom Russ, "WhyNot: Debugging Failed Queries in Large Knowledge Bases", pp. 870-877 in Proceedings IAAI-02: the 14th Innovative Applications of Artificial Intelligence Conference, 28 July - 01 August 2002, Edmonton, Alberta, Canada (2002)

[23] Arjun Chandra, Peter R. Lewis, Kyrre Glette, and Stephan C. Stilkerich, "Reference Architecture for Self-aware and Self-expressive Computing Systems", Chapter 4, pp. 37-49 in [59]

[24] Franck Chauvel, Nicolas Ferry, Brice Morin, Alessandro Rossini and Arnor Solberg, "Models@Runtime to Support the Iterative and Continuous Design of Autonomic Reasoners", in [10]

[25] Betty H. C. Cheng, Kerstin I. Eder, Martin Gogolla, Lars Grunske, Martin Litoiu, Hausi A. Müller, Patrizio Pelliccione, Anna Perini, Nauman A. Qureshi, Bernhard Rumpe, Daniel Schneider, Frank Trollmann, and Norha M. Villegas, "Using Models at Runtime to Address Assurance for Self-Adaptive Systems", p.101-136 in [11]

[26] Noam Chomsky, Marcel-Paul Schützenberger, "The Algebraic Theory of Context-Free Languages", pp. 118-161 in P. Braffort and D. Hirschberg (eds.), *Computer Programming and Formal Systems*, North-Holland (1963)

[27] Hal Draper, "Ms Fnd in a Lbry", *The magazine of Fantasy and Science Fiction* (Dec 1961); reprinted in Groff Conklin (ed.), *17 × Infinity*, Dell (1963)

[28] Philippe Flajolet, Robert Sedgewick, *Analytic Combinatorics*, Cambridge University Press (2009)

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

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

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

[32] Holger Giese, Nelly Bencomo, Liliana Pasquale, Andres J. Ramirez, Paola Inverardi, Sebastian Wätzoldt, and Siobhán Clarke, "Living with Uncertainty in the Age of Runtime Models", p.47-100 in [11]

[33] Holger Giese, Thomas Vogel, Ada Diaconescu, Sebastian Götz and Kirstie L. Bellman, "Generic Architectures for Individual Self-aware Computing Systems", Chapter 6, pp. 149-189 in [43]

[34] Holger Giese, Thomas Vogel, Ada Diaconescu, Sebastian Götz, Nelly Bencomo, Kurt Geihs, Samuel Kounev and Kirstie L. Bellman, "State of the Art in Architectures for Self-aware Computing Systems", Chapter 8, pp. 237-275 in [43]

[35] Holger Giese, Thomas Vogel, Ada Diaconescu, Sebastian Götz and Samuel Kounev, "Architectural Concepts for Self-aware Computing Systems", Chapter 5, pp. 109-147 in [43]

[36] Sebastian Götz and Thomas Kühn, "Models@run.time for Object-Relational Mapping Supporting Schema Evolution", in [37]

[37] Sebastian Götz, Nelly Bencomo, Gordon Blair, Hui Song (eds.), *Proceedings of the 10th International Workshop on Models@run.time*, 29

September 2015, co-located with *MoDELS 2015: the 18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, 27 September - 02 October 2015, Ottawa, Canada (2015)

[38] Sebastian Götz, Nelly Bencomo, Kirstie Bellman, Gordon Blair, *Proceedings of the 11th International Workshop on Models@run.time*, 04 October 2016, co-located with *MODELS 2016: the 19th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, 02-07 October 2016, Palais du Grand Large, Saint Malo, Brittany, France (2016)

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

[40] Christian Igel, Bernhard Sendhoff, "Genesis of Organic Computing Systems: Coupling Evolution and Learning", Chapter 7. p.141-166 in [81]

[41] Christophe Jacquet, Ahmed Mohamed, Frédéric Boulanger, Cécile Hardebolle, and Yacine Bellik, "Building Heterogeneous Models at Runtime to Detect Faults in Ambient-Intelligent Environments", in [10]

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

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

[44] 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 [43]

[45] Christopher Landauer, "Non-Deterministic Distributions", Paper mtlmd05 in *Proceedings of HICSS'99: The 32nd Hawaii Conference on System Sciences (CD)*, *Track VI: Modeling Technologies and Intelligent Systems*, *Logic Modeling Mini-Track*, 5-8 January 1999, Maui, Hawaii (1999)

[46] Dr. Christopher Landauer, "Problem Posing as a System Engineering Paradigm", *Proc. ICSEng 2011: The 21st International Conference on Systems Engineering*, 16-18 August 2011, Las Vegas, Nevada (2011)

[47] Christopher Landauer, "Infrastructure for Studying Infrastructure", *Proc. ESOS 2013: Workshop on Embedded Self-Organizing Systems*, 25 June 2013, San Jose, California (2013)

[48] 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)

[49] 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)

[50] 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)

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

[52] 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)

[53] Dr. Christopher Landauer, Dr. Kirstie L. Bellman, "Programming Paradigms for Real-Time Systems", *Proceedings SORT 2014: The Fifth IEEE Workshop on Self-Organizing Real-Time Systems*, 09 June 2014, Reno, Nevada (2014)

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

[55] Christopher Landauer, Kirstie L. Bellman, "Self-Modeling Systems Need Models at Run Time", in [38]

[56] 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)

[57] 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, Paderborn, Germany (2013)

[58] Jan van Leeuwen (Managing Editor), *Handbook of Theoretical Computer Science*, *Volume B: Formal Methods and Semantics*, MIT Press (1990)

[59] Peter R. Lewis, Marco Platzner, Bernhard Rinner, Jim Tørreson, Xin Yao (eds.), *Self-Aware Computing Systems: An Engineering Approach*, Springer (2016)

[60] Peter R. Lewis, Marco Platzner, Bernhard Rinner, Jim Tørreson, Xin Yao, "Conclusions and Outlook", Chapter 15, pp. 297-300 in [59]

[61] Maksym Lushpenko, Nicolas Ferry, Hui Song, Franck Chauvel, Arnor Solberg, "Using Adaptation Plans to Control the Behavior of Models@runtime", in [37]

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

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

[64] 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)

[65] Christian Müller-Schloer, Bernhard Sick, "Controlled Emergence and Self-Organization", Chapter 4, p.81-104 in [81]

[66] Hiroyuki Nakagawa, Kento Ogawa, Tatsuhiro Tsuchiya, "Caching Strategies for Run-time Probabilistic Model Checking", in [38]

[67] Luis H. Garcia Paucar, Nelly Bencomo, Kevin Kam Fung Yuen, "Runtime Models Based on Dynamic Decision Networks: Enhancing the Decision-making in the Domain of Ambient Assisted Living Applications", in [38]

[68] Christian Piechnick, Maria Piechnick, Sebastian Götz, Georg Püschel and Uwe Aßmann, "Managing Distributed Context Models Requires Adaptivity too", in [37]

[69] Nicholas Rescher, Alasdair Urquhart, *Temporal Logic*, Springer-Verlag (1971)

[70] A. Salomaa, "Formal Languages and Power Series", Chapter 3, pp. 103-132 in [58]

[71] Simon Spinner, Samuel Kounev, Xiaoyun Zhu, and Mustafa Uysal, "Towards Online Performance Model Extraction in Virtualized Environments", in [10]

[72] Michael Szvetits and Uwe Zdun, "Enhancing Root Cause Analysis with Runtime Models and Interactive Visualizations", in [10]

[73] 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)

[74] Thomas Vogel and Holger Giese, "On Unifying Development Models and Runtime Models (Position Paper)", in Sebastian Götz, Nelly Bencomo, Robert France (eds.), *Proceedings of the 9th Workshop on Models@run.time*, 30 September 2014, co-located with *MODELS 2014: the 17th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, 28 September - 03 October 2013, Valencia, Spain (2014)

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

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

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

[78] James R. Williams, Simon Poulding, Richard F. Paige, Fiona A. C. Polack, "Exploring the Use of Metaheuristic Search to Infer Models of Dynamic System Behaviour", in [10]

[79] Pierre Wolper, "Temporal Logic can be More Expressive", pp. 340-348 in *Proceedings of FoCS 1981: The 22nd Annual IEEE Symposium on the Foundations of Computer Science*, 28-30 October 1981, Nashville, Tennessee, IEEE (1981)

[80] Pierre Wolper, "Specification and Synthesis of Communicating Processes using an Extended Temporal Logic", pp. 20-33 in *Proceedings of PoPL 82: The Ninth Annual ACM Symposium on Principles of Programming Languages*, January 1982, Albuquerque, New Mexico, ACM (1982)

[81] Rolf P. Würtz (ed.), *Organic Computing*, Springer (2008)

[82] Yijun Yu, Thein Than Tun, Arosha K Bandara, Tian Zhang, and Bashar Nuseibah, "From Model-Driven Software Development Processes to Problem Diagnoses at Runtime", p.188-207 in [11]