EPiC Series in Computing

Volume 102, 2024, Pages 163–171

Proceedings of Conference on Cognitive and Computational Aspects of Situation Management 2023

Autonomous Computational Partners

Christopher Landauer

Topcy House Consulting, Thousand Oaks, California
topcycal@gmail.com

Abstract

If we are going to have computing systems that can be trusted to help us with difficult situations in difficult environments, then those systems are going to need much more capability, both for actions that conform to our goals for the systems, and for appropriate adaptations to unexpected or difficult conditions in their operational environment.

We may not be able to communicate with them in any timely or even useful way, so they will need to have strong autonomy in their action and adaptation decision processes.

This paper extends earlier work with further implications and expectations, along with some design notes for experiments that we are in the process of developing.

The first key finding of this investigation is that systematic language and the expressive and analytic properties of symbol systems are extremely important: abstractions cannot be computed without symbol systems; analogies cannot be discovered without symbol systems; models cannot be analyzed without symbol systems; and there are many other processes that we think are necessary that are greatly facilitated by explicit symbol systems.

The difficulty is that symbol systems cannot be indefinitely elaborated without a corresponding reduction process (by the "Get Stuck" Theorems in the field of Computational Semiotics, which studies the use of symbol systems by computing systems), so some kind of balance must be kept between what the system needs to know and how much that knowledge requires of its resources.

1 Introduction and Context

Our interest is in designing and deploying computing systems that can be trusted to help us with difficult situations in environments that may be remote, hazardous, possibly even denied or actively hostile. We expect to need much more autonomous capability in those systems, both for actions that conform to our goals for the systems, and for appropriate adaptations to unexpected or difficult conditions in their operational environment (which must often modify the original goals). In these difficult environments, we may not be able to communicate with them in any timely or even useful way, so they will need to have strong autonomy in their decision processes for both action and adaptation. They will have to be "Autonomous Computational Partners".

This paper extends [20] with further implications and expectations, along with some further design notes and some near term experiments.

We claim that the use of systematic languages and the expressive and analytic properties of symbol systems are essential to the flexibilities and adaptiveness we expect for building capable

K. Baclawski, M. Kozak, K. Bellman, G. D'Aniello, A. Ruvinsky and C. Da Silva Ferreira Barreto (eds.), CogSIMA 2023 (EPiC Series in Computing, vol. 102), pp. 163–171

autonomous computational partners, and we think that careful consideration of representations and the symbol systems that define them, and how a computing system uses and even creates them (this subject is called "Computational Semiotics" [29] [22]), will lead to advances in adaptive behavior. We have already shown the advantages of flexibility at the computational resource management level [23], [17], and this project is extending these advantages to the very symbols in which the system processes and data are written (to a certain level of detail, selected as an engineering design decision). In this paper, we describe the properties we deem essential for improving autonomous systems in this way, primarily in data and model construction.

There are really only two classes of requirements for effective autonomy, other than correctness (both are difficult): robustness and timeliness. Robustness means graceful degradation in increasingly hostile environments [19], which to us implies a requirement for adaptability, and timeliness means that situations are recognized "well enough" and "soon enough", and that "good enough" actions are taken "soon enough" [24], [6]. There is never any optimization here, since that usually takes too much time and produces non-robust solutions.

2 Partner Properties

This is a non-exclusive list of what we think are the important properties that should exist in any kind of computational partner [20], autonomous or not. They are *predictability* (we can know what the system is likely to do), *interpretability* (we can follow what the system is doing), and *explainability* (we can understand why the system did whatever it did, and why it did not do something else).

These properties allow us to build viable mental and computational models of what that system is and does, which we need to use for our own planning. One way to provide some of the information needed is our notion of an "audit trail", that is, a description of what was and was not done and why (this requires essential elements of the decision process that made the choice and the information used to support the decision, as well as the elements that precluded or discounted other possible choices), with the implications that would result from making alternative choices (this is much more than the usual kind of audit trail that records only what was done, when, and sometimes why).

Other important properties are easily seen to have significant overlaps in purpose and functionality. We choose only a few to elaborate further here (primarily for space reasons):

- *verification and validation*;
- *behavioral constraint management*;
- *dynamic efficiency management*;
- *continual improvement*; and
- *representation.*

We are certain that these are not all of the important properties, but we think they are enough to make system behavior more amenable to difficult applications.

The terms *verification* and *validation* refer to two different but related aspects of a system. Verification is about proving (or just demonstrating) that a system conforms to its specification ("is the system built right?"), while *validation* refers to showing that a system satisfies the user / owner / operator / other stakeholder expectations ("is it the right system?"). We expect all systems to undergo both processes during design time, before deployment, but we insist that

it can also be performed by the system itself at run time [12]. The issue is that very often (we claim almost always), the specifications for what is required are not complete or unambiguous (and far too often not even correct). They very often do not specify what is supposed to happen when hardware wears out or the environment does not cooperate (which we know is almost always, eventually).

The term *behavioral constraint management* refers to a set of guarantees of system behavior, each with context dependent gradations of these guarantees, ranging from "must do" to "must not do". It also includes graceful degradation in the face of failures or other problems (such as the environment not conforming to the system's expectations), with surprise, deficiency, and degradation announcements to the system decision processes, and to the system operators, when possible.

Any system can be overwhelmed by a sufficiently hostile environment; we cannot expect or even hope to provide complete protection in all situations. We can, however, make the application specific engineering judgments that decide how the system should to respond to system threats, especially ones that appear to be getting more likely or more dangerous. Sometimes the right answer is for the system to go into "safe-hold", shutting down most processes until conditions improve.

The term *dynamic efficiency management* means balancing appropriate efficiency with desired robustness, at run time according to ambient conditions, with "compiled-out" design decisions [7], [14] that can be revisited when conditions change. We call this "dynamic" efficiency as an analogy with dynamic range in signals and human perception, since for us it is a range (a region in an abstract space) of decision opportunities or action capabilities, while in signal processing it is a range of amplitudes for specific measurable variables. This process includes balancing flexibility and efficiency with assurance and robustness (more flexibility usually means assurance is harder to guarantee; more efficiency usually means less robustness); the "management" part of the phrase means being able to change those decisions at run time.

The phrase *continual improvement* means that the system is examining its own behavior, deciding where that is deficient (usually according to designer-provided criteria), and striving to improve the models that give rise to the deficient behavior. Our main approach here is Model Deficiency Analysis, which aims to assign blame for deficiencies and decide where new models need to be built, or old models need to be adjusted. This process involves explicit construction, selection, evaluation, and adaptation of models.

For us, *representation* is the key, since nothing can be computed without a representation. These are almost always fixed in advance by expert designers, and each time that happens, a large fraction of otherwise available adaptability is lost. We want our systems to be able to change their frame of reference, which means change their representations, at multiple scopes (ranges of considered concepts), and scales (levels of detail). We especially want our systems to be able to detect unknown phenomena (because there is an effect on their models that is not otherwise explained [31]), and devise experiments to understand their effect on the system, or just ignore them and complain (eventually, when possible) to the system operators.

We get another clue from considering commonly desired properties of adaptive systems. We defined a set of desired properties [24], and then devised a map of desired properties to underlying capabilities:

- speed and scope of adaptability to unforeseen situations;
- rate of effective learning of observations;
- accurate modeling and prediction of the relevant external environment;

- speed and clarity of problem identification and formulation;
- effective association and evaluation of disparate information; and
- identification of more important assumptions and prerequisites.

In [24], we showed that these capabilities can be provided by a system that has the following processes:

- methods of retention and retrieval;
- methods of comparison and association;
- methods of representation (conceptual spaces);
- methods of evaluation in context;
- methods of abstraction, analogy, and simplification; and
- creation and use of symbolic language, and, in particular, layers of symbol systems.

These properties are the subjects of intense studies in many fields of computing, and we do not believe that there is a widely supported consensus (and we are not sure that there will ever be one, because the applications are so different in scope and difficulty). We offer no better than notional descriptions here.

The first two are the simplest. **Retention and retrieval** is about saving and restoring information, which can be difficult in light of the Get Stuck Theorems [22] [19].

Comparison and association (equal or related) are two of many ways to relate different sets of information by content.

Representation underlies all of our theory about making information explicit so it can be analyzed and shared, applied and improved (what are these expressions, what do they mean, and how can they be analyzed).

Evaluation in context is an evaluation of significance of information (what does this information mean to me here and now?). It must represent the (relevant part of the) context and have some information about the significance of various phenomena. Of course, the level of significance can change for the same information in a different situation, and the system must manage that also.

An **abstraction** is a detail-independent description of a phenomenon, but it is not complete without being related back to the original phenomenon. **Making abstractions** involves identifying what roles those details play, and how to describe them so that other details can step in when necessary.

An **analogy** is a comparison of certain aspects of two situations or contexts that are expected to be similar, as a way of deciding what to do when there is no clearly applicable model.

Simplification just asks what aspects do not matter as much and what changes when we ignore them? It is often a first step towards abstraction, and is often way too drastic.

For our part the last one **creation and use of symbolic language** is the most important, since it is about how the system creates its own language to express what it has learned. This process clearly requires creating simpler overall languages that do not have decideability problems [10], so that common methods can produce analyzable languages [1].

Finally, [4] has argued that there is a common origin in biological systems of both language and movement (biology being tremendously conservative about basic processing methods), so that it seems important to examine layers of symbol systems for language applications, since it has been so useful in simplifying computing system architectures (the internet, structured programming, etc.).

3 Enablers

There is a set of enablers that we believe can supply or at least support many of these aforementioned properties. They have played a prominent role in recent work in Self-Aware and Self-Adaptive Systems (see [26], [5], [16], [3], [2]), and we are using several results and approaches from that area in this design study.

We described the most important three earlier in [20]:

- Computational Reflection,
- Model-Based Operation, and
- Speculative Simulation.

(there were others, but only these were discussed).

"Computational Reflection" [17] is when a system has effective access to all of its internal processes, so it can study them and change them. We have usually paired this with Continual Contemplation [23] in our Wrapping integration infrastructure, to provide the system continual access to its own operations. Here, the issue is how to represent and reason about the computational resources and the conditions for using them, so that they can be used in the appropriate contexts, and different resources can be used in different contexts for the same purpose (a simple version of this fact is that when you know that a finite matrix is symmetric, that greatly reduces the amount of computation time needed to produce its eigenvectors, and also improves the accuracy [11]). There are many examples of this algorithm improvement for special cases. This is the "Problem Posing Paradigm" [23], which separates information service requests (which we call "problems", hence "problem posing") from the information service providers (which we call "resources", generally meaning computational algorithms or structures that can be queried for data or applied to produce that data), and maps one to the other using a context-dependent Knowledge-Based System. In addition (and this is what makes our Wrapping approach unusual), all the resources that are involved in these maps are themselves resources, and selected using exactly the same mechanism. This commonality supports many kinds of flexibility in system operation.

"Model-Based Operation" is inherent in our "Self-Modeling" systems [25], since they are defined as consisting entirely of models that are interpreted to produce whatever decisions or actions are required, desired, expected, or just tried in experiments. Then, if a system can change its models, it will change its behavior. This kind of modeling extends Model-Based Engineering [32] to run time, and includes, not just the model definitions and interfaces, but precise declaration of the responsibilities of each model, and the conditions under which it is to be used. This approach clearly requires expressive representations of the models, their prerequisites, their interfaces, the algorithms for deciding applicability, and the algorithms for deciding that degradation has occurred and which replacement models to use.

In addition, explicit expression of all models and model processes allows "refactoring" [9], which is (for our purposes) a rearrangement of the models and a reassignment of their original responsibilities, generally intended to support better "separation of concerns" [8].

"Speculative Simulation" is the ability of a system to set up scenarios, evaluate them, and use those results to choose courses of action, including changes of goals. It may also involve refactoring of existing processes (computational resources) into new combinations. This one is a little harder, but it requires the system to have not only good models of the system processes, goals, and the expected environment, but also good models of phenomena that do not relate to the actual task at all (for example, in automatic driving algorithms, there needs to be a model

for detecting "squirrels" and accounting for what they might do; using a generic "anomalies" model is much harder). And again we are back to representations.

4 Problems

These considerations are not without their difficulties. There is a large literature in multiple communities that addresses some of these problems: Organic Computing [33], [30], which is largely about the system qualities needed to enable collections of largely autonomous systems to be effective in complex real-world environments, interwoven and self-integrating systems [3], which are concerned with the difficult and dynamic boundaries between cooperating systems, and how much a system needs to know about its own behaviors and capabilities to integrate effectively into a team, autonomic systems [15], which are about keeping the underlying processes running, in analogy with the autonomic nervous system, and even explainable artificial intelligence, though that is currently largely limited to explaining a few classes of learning algorithms.

We noted a few problems and questions earlier [20] that cannot be solved, only mitigated: bad models, hardware failures, unforeseen circumstances and consequences, lack of data, and reliability of human and other partners (which we do not think is easily handled by the computing system at all).

The problem with **bad models** is not just that they may not contain the required information [28] [27], but that they can use it inappropriately, and even spew enormous amounts of irrelevant or wrong data into the system's communication processes. To that end, we described "integrity fences" [18], which are interfaces between models that act as verifiers of correctness and appropriate volume at run time. These act much like bulkheads on a submarine (or other kind of ship): during normal operation, bulkheads do nothing. They are open. During heightened difficulties, such as subsystem failures or perceived threats, they can be partially or completely closed to prevent one problem from propagating to other subsystems. This is done with software organized into multiple levels of protection and even multiple styles of threat.

For **hardware failures**, which we know will eventually occur, the system should have multiple models of partially degraded subsystems, and mechanisms to identify which ones are most appropriate to use for the recognized situation. The same is almost true for **unforeseen circumstances and consequences**, which is a special (but harder) case of **lack of data**. The difference is that with hardware, we know many of the ways in which it can fail (we have been building and breaking things for at least thousands of years), but with software, our history is much shorter, at most since the Jacquard loom in 1804, and arguably mostly after Charles Babbage and Ada Lovelace in the 1840's, and moreover, software is not subject to most of the physical laws with which we are familiar, so we have little or no long-term experience with software errors in comparison.

Missing data can occur in many ways: complex numerical calculations [27] are the most prominent (and they can be easier to handle, with a number of mathematical techniques that are beyond the scope of this paper), but we are mostly considering other kinds of structured or non-numerical data, and what we think the system should do [20].

There are processes for completing analyses or explanations and representations of the resulting structures [21] that can identify that something is missing, such as a behavior offer from a resource satisfying almost all of the conditions in the request. We would like the system to ask for the missing data, the way many web servers already do, but in our applications, we are not easily available to ask. It is still therefore usually quite hard to determine what exactly is missing, but that will be somewhat easier in the limited application domains we expect to

consider first. The system can sometimes know that something is missing by examining these structures for missing data.

There are analysis or explanation frameworks [21] that are mostly complete, missing only one or two steps. The system can decide that it needs to know something it doesn't know because of the missing steps.

The system can decide that it needs to do an experiment when it has no models to apply exactly to the perceived situation (or it can just call home, and at some level of complexity, it must).

For each of the above enablers, and for most of the above problems, we have explained that we expect to reduce or at least mitigate the issue with explicit models, because that allows the issue to be studied directly, instead of only through its appearance in the software, and its effects on software errors. This is primarily a representation issue: building and using the right representations for the problem at hand. What the designers need to do is not only build initial models for the situations that are expected, but also provide mechanisms to improve them.

The first key finding of this investigation is that the use of systematic language and the expressive and analytic properties of symbol systems are extremely important: abstractions cannot be computed without symbol systems; analogies cannot be discovered without symbol systems; models cannot be analyzed without symbol systems; and there are many other processes that we think are necessary for effective autonomy that are greatly facilitated by explicit symbol systems. We are not trying to solve hard or impossible problems; we only need "good enough" responses computed "soon enough" [6].

The difficulty is that symbol systems cannot be indefinitely elaborated without a corresponding reduction process (the "Get Stuck" Theorems in the field of Computational Semiotics, which studies the use of symbol systems by computing systems, show this [22]), so some kind of balance must be kept between what the system needs to know and how much that knowledge requires of its resources [19]. Of course, changing the symbol system does not eliminate the problem; it just pushes it farther away. But that might be good enough for the application. How much of this balance to leave to run time is an important application specific design decision.

5 Conclusions and Prospects

We have described some important properties that should be available in every system that we use as an augmentation of our abilities "in the wild" [13], especially if we cannot go out into that wild. We have shown that most of them are difficult, which is why they are not common in such systems, but also that all of them can be addressed with the appropriate system and software architecture to varying levels of success.

The most prominent among them are, in decreasing order of difficulty, representation management, model creation, and integration infrastructure, with the last one being largely solved [23] [17], the second one having plausible approaches [21] [20], and the first one having no more than seemingly possible beginnings [22] [9]. But they all come down to careful management of representation (creation, comparison, application, evaluation, and improvement).

As a simple experiment to consider approaches to the first hard problem, that of representation management, we consider the application domain of patient monitoring in hospitals. The display devices may have dozens of active measurements taken quite often, and adjusting the representations used according to ambient conditions (both patient and room environment) is simple and constrained enough to be addressed. The purpose would be to determine what the measurements need under different conditions, and computing likely a greatly reduced data volume, which could be important in an emergency involving large numbers of patients. In ad-

dition, there is a large amount of information available about the necessary medical knowledge. This is not the main experiment on autonomy, since it could only be carried out under strict continuing medical supervision (and appropriate legal conditions), but it is a way to use real data to consider the representation problem.

References

[1] Dana Angluin and Carl H. Smith. Inductive inference: Theory and methods. *Computing Surveys*, 15(3):237–269, September 1983.

[2] K. Bellman, C. Landauer, N. Dutt, L. Esterle, A. Herkersdorf, A. Jantsch, P. R. Lewis, M. Platzner, N. TaheriNejad, and K. Tammemäe. Self-aware cyber-physical systems. *ACM Transactions on Cyber-Physical Systems (TCPS)*, 4:1–26, October 2020.

[3] Kirstie Bellman, Jean Botev, Ada Diaconescu, Lukas Esterle, Christian Gruhl, Christopher Landauer, Peter R. Lewis, Phyllis R. Nelson, Evangelos Pournaras, Anthony Stein, and Sven Tomforde. Self-improving system integration: Mastering continuous change. *FGCS: Future Generation Computing Systems, Special Issue on SISSY*, 117:29–46, April 2021.

[4] Kirstie L. Bellman and Lou Goldberg. Common origin of linguistic and movement abilities. *American Journal of Physiology*, 246:R915–R921, 1984.

[5] Kirstie L. Bellman, Christopher Landauer, Phyllis Nelson, Nelly Bencomo, Sebastian Götz, Peter Lewis, and Lukas Esterle. Self-modeling and self-awareness. In Samuel Kounev, Jeffrey O. Kephart, Aleksandar Milenkoski, and Xiaoyun Zhu, editors, *Self-Aware Computing Systems*, chapter 9, pages 279–304. Springer, January 2017.

[6] Kirstie L. Bellman and Phyllis R. Nelson. Developing mechanisms for determining "good enough" in sort systems. In *Proceedings SORT 2011: The Second IEEE Workshop on Self-Organizing Real-Time Systems*, March 2011.

[7] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Proceedings PoPL 1993: The 20th ACM Symposium on Principles of Programming Languages*, January 1993.

[8] Edsger W. Dijkstra. *On the role of scientific thought, Selected writings on Computing: A Personal Perspective*. Springer-Verlag, 1982.

[9] Martin Fowler and Kent Beck. *Refactoring*. Addison-Wesley, 2018.

[10] E. M. Gold. Language identification in the limit. *Information and Control*, 10:447–474, 1967.

[11] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, 1983.

[12] Klaus Havelund and Grigore Roşu. Monitoring programs using rewriting. In *Proceedings ASE 2001: The 16th International Conference on Automated Software Engineering*, November 2001.

[13] Edwin Hutchins. *Cognition in the Wild*. MIT, 1995.

[14] N. D. Jones. Partial evaluation. *Computing Surveys*, 28(3), September 1996.

[15] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer J.*, 36:41–50, January 2003.

[16] 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. In Samuel Kounev, Jeffrey O. Kephart, Aleksandar Milenkoski, and Xiaoyun Zhu, editors, *Self-Aware Computing Systems*, chapter 1, pages 3–16. Springer, January 2017.

[17] Christopher Landauer. Infrastructure for studying infrastructure. In *Proceedings ESOS 2013: Workshop on Embedded Self-Organizing Systems*, San Jose, California, June 2013.

[18] Christopher Landauer. Active system integrity fences. In *Proceedings M@RT 2017: The 12th International Workshop on Models@run.time*, Austin, Texas, September 2017.

[19] Christopher Landauer. Mitigating the inevitable failure of knowledge representation. In *Proceedings 2nd M@RT: The 2nd International Workshop on Models@run.time for Self-aware Computing Systems*, Columbus, Ohio, July 2017.

[20] Christopher Landauer. What i want in a (computational) partner. In *Proceedings LIFELIKE: Workshop on Lifelike Computing Systems*, July 2021.

[21] Christopher Landauer and Kirstie Bellman. Strands of memory. In *Proceedings CogSIMA 2020: IEEE Conference on Cognitive and Computational Aspects of Situation Management*, May 2020.

[22] Christopher Landauer and Kirstie L. Bellman. Situation assessment via computational semiotics. In *Proceedings of ISAS1998: The 1998 International MultiDisciplinary Conference on Intelligent Systems and Semiotics*, pages 712–717, September 1998.

[23] Christopher Landauer and Kirstie L. Bellman. Generic programming, partial evaluation, and a new programming paradigm. In Gene McGuire, editor, *Software Process Improvement*, pages 108–154. Idea Group Publishing, 1999.

[24] Christopher Landauer and Kirstie L. Bellman. Refactored characteristics of intelligent computing systems. In *Proceedings of PERMIS2002: Measuring Performance and Intelligence of Intelligent Systems*, Augusts 2002.

[25] Christopher Landauer and Kirstie L. Bellman. Self-modeling systems. In R. Laddaga and H. Shrobe, editors, *Self-Adaptive Software*, volume 2614 of *Lecture Notes in Computer Science*, pages 238–256. Springer, 2002.

[26] Peter R. Lewis, Marco Platzner, Bernhard Rinner, Jim Trresen, and Xin Yao. *Self-Aware Computing Systems: An Engineering Approach*. Springer, 1st edition, July 2016.

[27] R. J. A. Little and D. B. Rubin. *Statistical Analysis with Missing Data*. Wiley, 1987.

[28] Q. Ethan McCallum. *Bad Data Handbook*. O'Reilly, 2012.

[29] Alex Meystel. *Semiotic Modeling and Situation Analysis: An Introduction*. AdRem, Inc., 1995.

[30] C. Müller-Schloer, H. Schmeck, and T. Ungerer, editors. *Organic Computing - A Paradigm Shift for Complex Systems*. Springer, 2011.

[31] Phyllis Nelson, Kirstie Bellman, and Christopher Landauer. Self-modeling: a practical example of why it is hard. In *Proceedings 9th SiSSy: Workshop on Self-Improving System Integrating*, September 2022.

[32] Douglas C. Schmidt. Model-driven engineering: Introduction to the special issue. *IEEE Computer*, 39:25–31, February 2006.

[33] R. P. Würtz, editor. *Organic Computing*. Springer, 2008.