What I Want in a (Computational) Partner

Christopher Landauer

Topcy House Consulting, Thousand Oaks, California, 91362
topcycal@gmail.com

Abstract

This is a position paper about some stringent conditions that we should expect any computational partners to satisfy before we are willing to trust them with non-trivial tasks, and the methods we expect to use to build systems that satisfy them.

We have implemented many of the methods we describe to demonstrate some of these properties before, but we have not collected them into a single implementation where they can reinforce or interfere with each other. This paper is a preliminary description of a design of such an implementation, which we are in the process of defining and constructing.

Introduction

The structure of the paper is as follows: it will start with some context and a list of relevant tasks and locations for which computer assistance could be extremely helpful.

Then comes the first main part of the paper: the set of characteristics and responsibilities that should be expected, These include *predictability* (we can know what it is likely to do), *interpretability* (we can follow what it is doing), and *explainability* (we can understand why it did whatever it did), guarantees of behavior, graceful degradation, reputation and trust management, and continual improvement.

The second main part of the paper is about the "enablers", that is, the methods and approaches that we think will lead to systems that can satisfy all of these expectations to the extent required (there is, of course, an application-dependent engineering decision about how much of each of these properties is needed). The chief enablers are Computational Reflection, Model-Based Operation, and Speculative Simulation, each of which will be explained in that part of the paper.

The paper follows this description with a list of problems, that is, major difficulties with these application areas that we think every computing system will have to address if it is to be used in that environment. We are sure that this list is far from complete, and will likely never be complete, but we are expecting that it is complete enough to proceed.

Finally, the paper closes with a few notes about our conclusions and prospects. In the interests of space, it does not discuss the integration of models and components that is well-described in earlier papers on Self-Awareness and Self-Modeling Landauer and Bellman (1999) Landauer and Bellman (2002) Landauer (2013).

We are not at all suggesting that this is the only way to address these issues, or even that these are the only important issues; only that we think that all of these issues are important and must be addressed, and that this approach has gained enough coherence and completion, with supporting computational methods, to attempt an implementation.

There is a large literature in multiple communities that addresses some of these problems: organic computing Würtz (2008) Müller-Schloer et al. (2011), 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 Bellman et al. (2021), 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, and even explainable artificial intelligence, though that is currently largely limited to explaining a few classes of learning algorithms.

Context

The context of this work is a software-managed system (also called constructed complex system, cyber-physical system, technical system) in a difficult environment: complex, dynamic, remote, hazardous, malicious or even actively hostile, or all of the above. These systems will go places we cannot (or should not) go, or provide physical or computational assistance when we do.

Relevant Tasks and Locations

The kinds of tasks we are expecting to be relevant are quite varied:

- search and rescue (remember also that dogs and other animals may be helping);
- medical maintenance, prevention and intervention, emergencies;

- IED (Improvised Explosive Devices) and other bombs, also biological, chemical, or radiation threat detection;
- shelter / road / bridge / space station construction;
- scientific exploration; and
- cooperative / competitive games.

We are not so much interested in autonomous vehicles, not because it is not important or prominent today, but because we are waiting for that area to have a purpose beyond taking people where they want to go (unless that counts as a competitive game in a hostile environment).

The potential locations in which these activities might take place are also varied:

- inside a building or other structure, perhaps collapsed and / or unstable;
- outside buildings in an urban area;
- in a remote wilderness area (forest and jungle, hill and canyon, ice and rocks and tundra);
- airborne, surface maritime, or undersea;
- active conflict zone; and
- off-planet, either on another or in space.

Some of these tasks are obviously more relevant to some of these locations than others, and we do not expect any system to manage all (or even more than one or two) of them the way we generally expect humans to be able to do.

Characteristics and Responsibilities

The first main point in this paper is our current list of the desired and / or expected characteristics and responsibilities of such a system:

- predictability and interpretability and explainability;
- verification and validation;
- flexibility vs assurance;
- reputation and trust management;
- continual improvement; and
- power assist.

To our mind, some of the most important properties of a system are its *predictability* (we can know for our own planning what it is likely to do), *interpretability* (we can recognize and follow what it is doing), and *explainability* (we can understand why it did whatever it did and not something else). These properties allow us to build a viable mental model of what that system is, which we can (indeed, must) use for our own planning. An important aspect of these properties is the notion of an "audit trail", that is, a description of what was and was not done and why (essential elements of the decision process that made the choice and the information used to do so), with the implications of making alternative choices (this is much more than the usual kind of audit trail that records only what was done). It is also important to be able to know what can be expected in different situations, before those situations occur.

These explanation will necessarily involve multiple levels of detail, not only the usual multiple time and space scopes (range of consideration) and scales (resolution detail), but also multiplicity and granularity in the semantic domain detail (which could be a superficial summary or more detailed descriptive terminology; each is useful for different purposes).

We expect these systems to undergo verification and validation (see any Software Engineering book for these terms) as usual at design / test / deployment time, but also continually at run time (not just to satisfy requirements, which is verification, but also to satisfy expectations, which is validation).

A perennial trade-off in system design is flexibility vs. assurance. The issue is to balance the flexibility of adaptive behavior with the assurance of (semi-)predictable behavior. We are strongly on the side of emphasizing flexibility, with explicit processes that limit it when appropriate. This requires behavioral constraint management, to account for guarantees of behavior, both safety and liveness (these terms and related ones are defined in Alpern and Schneider (1986)).

If the environment stays within the specified constraints, then the system will behave as advertised (this is called an "assumption - guarantee" model in Kwiatkowska et al. (2010) and Chilton et al. (2013); this model has been studied for quite some time).

We want the system to exhibit graceful degradation, which means, that if the environment does not stay within the expected constraints, then the system will only gradually lose capability (up to a certain catastrophic level of pernicious behavior). In this sense, we want the system to be robust and resilient. A system is *robust* if it can maintain function in the face of disruptive efforts or effects. A system is *resilient* if it can restore function in the aftermath of a disruption. These definitions are of course dependent on the meanings of "maintain" and "restore" (and "disruption"), but those meanings can be turned into graded measurements, both for how much a system is disrupted and how much and how quickly and how well it recovers. They are also clearly task- and location-specific.

Another area where design engineering often fails is in the notion of "appropriate efficiency". We know that efficiency is the enemy of robustness, and we are expecting this system to be extremely robust. We are willing to spend a little

efficiency to gain that robustness.

Reputation and trust management are also essential. Not only do we want the system to do the right thing, it must be trusted that it does the right thing, that it will do the right thing under increasingly varied and difficult circumstances, and that it will tell us when it cannot, hopefully before that happens. The system should provide deficiency and degradation announcements, so we know what we can expect. There also needs to be a method for new role negotiation, based on a system's current or projected degraded or enhanced capability. The mechanisms and processes of establishing and maintaining trust are well-studied in the literature, but we are also expecting the system to have notions of trust of its own behavior, something like a self-assessment of reliability (e.g., if it tries to perform a certain action, will that action occur?).

There are some other aspects of the system that are not as important to us at first, but that will become important if we want to use these systems.

There should be a way for human operators to over-ride almost anything, with varying levels of justification effort required. Of course, that can become problematic in time- and life-critical decisions. There should be a kind of "power assist" mode (as in power steering and power brakes in automobiles), where the system lets its operators drive it under certain circumstances, with little or no decision making (just providing low level support to operator selected actions). This becomes even more useful if we instrument everything and analyze it later, to suggest operating improvements.

Finally, we expect the system to analyze itself for continual improvement along several paths. It will be improving current behaviors by streamlining behavior combinations: if decisions are made the same way every time, they can be compiled out of the code until the relevant part of the environment changes (this process is called "partial evaluation" Jones (1996); it can be very useful in conjunction with a process that watches for relevant environmental changes).

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

Enablers

The second main point of this paper is that we believe that all of this is feasible: there is a set of enablers that we believe can supply these aforementioned properties. They have played a prominent role in recent work in Self-Aware and Self-Adaptive Systems (see Lewis et al. (2016), Bellman et al. (2017), Kounev et al. (2017), Bellman et al. (2021), Bellman et al. (2020)), and we are using several results and approaches from that area in this design.

Among the most important ones we have used are Computational Reflection, Model-Based Operation, and Speculative Simulation (there are others, but these are the three we wish to discuss here).

A Computationally Reflective system has access to all of its own internal computation and decision processes, can reason about its capabilities and behavior, and can change that behavior when and as appropriate (see Kiczales et al. (1991) Buschmann (1996) for a description of reflection, or Landauer and Bellman (1999) for a description of our approach). In addition to whatever external expectations there are, it engages in a process we call "continual contemplation", examining its own activity for anomalies and potential improvements.

For our purposes, the easiest way to do this is through models (this is "Model-Based Operation"). The way we use the term, "model-based operation" is more than model-based design or engineering (as in Schmidt (2006)), which make extensive use of modeling during system design and development. All system knowledge and processes are maintained as models, which can be examined as part of the decision processes, and exercised or interpreted to produce the system's behaviors. That way, when the system changes the models, it changes its own behavior. In some applications requiring extreme flexibility, even the model interpreter can be one of the models, so the very notation in which the system is written can change.

In that sense, we call these systems "self-modeling" (as described in Landauer and Bellman (2002), Landauer et al. (2013)). While there are many ways such a system might be implemented, we have shown the efficacy of *Wrappings* as one way to do these things. Wrappings are described in Landauer and Bellman (1999), Landauer (2013), and in many other papers. We will not describe them here for lack of space, except to say that they provide a Knowledge-based integration infrastructure that is extremely flexible and expressive.

For us, modeling is pervasive throughout the lifetime of the system (we have written about this issue in Landauer and Bellman (2015a), Landauer and Bellman (2015b), Landauer and Bellman (2016a), for example). In fact, we expect the system itself to build models, as a way of coping with the vagaries and hazards of its environment, by retaining essential properties of it for analysis and planning, as well as perspective views of how those models change in time.

That puts model construction at the center of our considerations. The system will perform, as part of its continual contemplation, what we call "Behavior Mining", an examination of the event and action history, for the purpose of discovering persistent structures or event patterns that can be used for system improvement. This includes history maintenance and management, so the system has access to the activity. Machine Learning techniques can be valuable here, but they are just one of the possible approaches (e.g., grammatical and event pattern inference), and in any case, they do not usually address data in the form of partially ordered sets of multiple-resolution descriptions of events.

This process entails a continual identification of common

structures and behaviors, based on internal activity indicators. Our Wrapping integration infrastructure (see Landauer and Bellman (1999), Landauer and Bellman (2002), Landauer (2013)) facilitates such access, encapsulation of commonly co-occurring activities, and bottom-up evolution of empirical system structures (the structures change in response to behavior changes).

Another significant model process is Model Deficiency Analysis, based on the discovery of anomalies, such as noticing unusual or unexpected behavior (e.g., "that's peculiar"), the discovery of novelty, including the exploitation of side effects, and other model evaluations (e.g., for resource cost-effectiveness reliability).

To manage all of this complexity of knowledge embodied in models, we use processes we have called Dynamic Knowledge Management (see Landauer (2017) and Landauer and Bellman (2016b)), including knowledge refactoring and constructive forgetting.

The third enabler, Speculative Simulation, is a way for the system to try decisions out before committing to them, or just explore the space of possibilities (much like the "play" described in Bellman (2013)). This kind of analysis includes actions and adaptations, and clearly requires models of the effects of system choices. Most of the "what-if" scenario descriptions that we have seen are for risk management, usually from a business standpoint, but sometimes for engineering design. We have not seen any good ones for opportunity exploitation, that is, how to recognize that a certain process or resource could save time or improve accuracy, but this ability is clearly useful for systems in complex environments.

This is one of the hard parts, but also the most exciting for us. We are expecting the system to act as an experimental scientist, exploring and attempting to explain its environment. To make this effective, the system needs methods for hypothesis generation, experimental design, and experiment evaluation.

The most important and difficult questions to be answered here are:

- How does the system decide it needs to do an experiment? When it doesn't know something.
- How does it decide that it needs to know something it doesn't know? There are missing steps in an analysis or explanation.
- How does it know something is missing? There are processes for completing analyses or explanations that can identify that something is missing (it is still quite hard to determine what exactly is missing).

In some sense, this system is being constructed to explore approaches and potential answers to these questions.

Problems

There is of course a myriad of potential problems. The unfortunate part is that most of them cannot be overcome, only mitigated, and some not even that. On the other hand, it is our contention that almost all these same problems apply no matter what kind of system is constructed and deployed.

Bad models. When you live by your models, you die by your models: it is known that computer programs are easier to subvert when they are formally proven than otherwise (don't attack the object being protected, attack the protector by side-stepping the formal model). The only thing we can do here is call for help. This is the one of these problems that is caused by our emphasis on models, but we prefer the model-based approach anyway for its ability to be analyzed.

Lack of data. There are many things we can try to do: go get more, find workarounds (some workarounds can be planned in advance, in anticipation of certain kinds and levels of data unavailability). We can have the system make best guesses, using some kind of hazard-risk-consequence map, with a corresponding sensitivity analysis over potential decisions (which ones have the worst consequences, which ones can the system afford to treat in its current state).

Hardware failures are foreseeable from years of reliability studies, but specific instances are largely unpredictable. They are related to the second most difficult category. *Unforeseen circumstances and consequences*, about which there are only a few things we can do, none of which are guaranteed to work at all (we have only the barest minimum of available responses to this problem Landauer (2019)). We can design the system with numerous and varied backups (alternative ways to carry out some tasks) and failsafes (consistent levels of reduced functionality), that may provide enough time for a problem to be addressed or even solved.

And of course, the most difficult of all. *Reliability* of humans and other partners. We hope their training and knowledge suffices, as they hope ours does.

There are others, of course, but these are at least among the most pernicious and persistent ones.

Conclusions and Prospects

We hope this note contains enough description to explain why we think that an implementation can be constructed, what we intend its basic structure to be, and how we expect the system to satisfy the original expectations. We know that it does not explain how all of these properties will be achieved, because in many cases, the answer is not yet known. We expect that this kind of system architecture will allow us to study these (and other) hard questions. We think that systems with these capabilities could be acceptable as computational partners.

References

Alpern, B. and Schneider, F. B. (1986). Recognizing safety and liveness. *Distributed Computing*, 2:117–126.

Bellman, K., Botev, J., Diaconescu, A., Esterle, L., Gruhl, C., Landauer, C., Lewis, P. R., Nelson, P. R., Pournaras, E., Stein, A., and Tomforde, S. (2021). Self-improving system integration: Mastering continuous change. *FGCS: Future Generation Computing Systems, Special Issue on SISSY*, 117:29–46.

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

Bellman, K. L. (2013). Sos behaviors: Self-reflection and a version of structured "playing" may be critical for the verification and validation of complex systems of systems. In *CSD&M 2013: The Fourth International Conference on Complex Systems Design & Management*.

Bellman, K. L., Landauer, C., Nelson, P., Bencomo, N., Götz, S., Lewis, P., and Esterle, L. (2017). Self-modeling and self-awareness. In Kounev, S., Kephart, J. O., Milenkoski, A., and Zhu, X., editors, *Self-Aware Computing Systems*, chapter 9, pages 279–304. Springer.

Buschmann, F. (1996). Reflection. In Vlissides, J. M., Coplien, J. O., and Kerth, N. L., editors, *Pattern Languages of Program Design 2*, chapter 17, pages 271–294. Addison-Wesley.

Chilton, C., Jonsson, B., and Kwiatkowska, M. (2013). Assume-guarantee reasoning for safe component behaviours. In Păsăreanu, C. S. and Salaün, G., editors, *FACS 2012: Formal Aspects of Component Software*, volume 7684 of *Lecture Notes in Computer Science*. Springer.

Jones, N. D. (1996). Partial evaluation. *Computing Surveys*, 28(3).

Kiczales, G., des Rivières, J., and Bobrow, D. G. (1991). *The Art of the Meta-Object Protocol*. MIT Press.

Kounev, S., Lewis, P., Bellman, K. L., Bencomo, N., Cámara, J., Diaconescu, A., Esterle, L., Geihs, K., Giese, H., Götz, S., Inverardi, P., Kephart, J. O., and Zisman, A. (2017). The notion of self-aware computing. In Kounev, S., Kephart, J. O., Milenkoski, A., and Zhu, X., editors, *Self-Aware Computing Systems*, chapter 1, pages 3–16. Springer.

Kwiatkowska, M., Norman, G., Parker, D., and Qu, H. (2010). Assume-guarantee verification for probabilistic systems. In Esparza, J. and Majumdar, R., editors, *TACAS 2010: Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *Lecture Notes in Computer Science*. Springer.

Landauer, C. (2013). Infrastructure for studying infrastructure. In *Proceedings ESOS 2013: Workshop on Embedded Self-Organizing Systems*, San Jose, California.

Landauer, C. (2017). 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.

Landauer, C. (2019). What do I do now? Nobody told me how to do this. In *Proceedings M@RT 2019, The 14th International Workshop on Models@run.time*, Munich, Germany.

Landauer, C. and Bellman, K. L. (1999). Generic programming, partial evaluation, and a new programming paradigm. In McGuire, G., editor, *Software Process Improvement*, pages 108–154. Idea Group Publishing.

Landauer, C. and Bellman, K. L. (2002). Self-modeling systems. In Laddaga, R. and Shrobe, H., editors, *Self-Adaptive Software*, volume 2614 of *Lecture Notes in Computer Science*, pages 238–256. Springer.

Landauer, C. and Bellman, K. L. (2015a). Automatic model assessment for situation awareness. In *Proceedings CogSIMA 2015: The 2015 IEEE International Multi-Disciplinary Conference on Cognitive Methods in Situation Awareness and Decision Support*, Orlando, Florida.

Landauer, C. and Bellman, K. L. (2015b). System development at run time. In *Proceedings M@RT 2015: The 10th International Workshop on Models@Run-Time*, Ottawa, Canada.

Landauer, C. and Bellman, K. L. (2016a). Model-based cooperative system engineering and integration. In *Proceedings SiSSy 2016: The 3rd Workshop on Self-Improving System Integration*, Würzburg, Germany.

Landauer, C. and Bellman, K. L. (2016b). Self-modeling systems need models at run time. In *Proceedings M@RT 2016: The 11th International Workshop on Models@run.time*, Palais du Grand Large, Saint Malo, Brittany, France.

Landauer, C., Bellman, K. L., and Nelson, P. R. (2013). Self-modeling for adaptive situation awareness. In *Proceedings of CogSIMA 2013: The 2013 IEEE International Inter-Disciplinary Conference on Cognitive Methods for Situation Awareness and Decision Support*, San Diego, California.

Lewis, P. R., Platzner, M., Rinner, B., Trresen, J., and Yao, X. (2016). *Self-Aware Computing Systems: An Engineering Approach*. Springer, 1st edition.

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

Schmidt, D. C. (2006). Model-driven engineering: Introduction to the special issue. *IEEE Computer*, 39:25–31.

Würtz, R. P., editor (2008). *Organic Computing*. Springer.