Proceedings of the 53rd Hawaii International Conference on System Sciences | 2020

Active Loop Programming for Adaptive Systems

Dr. Christopher Landauer, Dr. Kirstie L. Bellman
Topcy House Consulting
Thousand Oaks, California
topcycal@gmail.com, bellmanhome@yahoo.com

Abstract

We describe a new approach to adaptive system construction, based on our belief that there are no one-way functions in biology (for example, no sensor is a one-way input device, and no effector is a one-way output device). We choose to mimic the fact that all biological systems have many active processing loops running at all times (at various different time and space scales), and all of them both produce and consume data. We wanted to see how far this notion can carry us towards highly adaptive computational systems, in combination with computational reflection and certain other biological principles of organization. We show that it carries us surprisingly far, by describing a system architecture that uses it as a fundamental organizing principle. We define what active loop programming is, show how it provides enormous flexibility in a software-intensive system, and show how it can be implemented with Wrappings.

Keywords: *Self-Adaptive Systems; Biological Principles; Active Loop Programming; Computational Reflection; Wrapping Infrastructure*

1. Introduction

Biological systems have remarkable robustness and adaptability, and software-intensive computing system designers have been trying to isolate and replicate the principles of organization for a very long time. It has emerged that computational reflection [33] is one important principle of organization, but there hasn't been much study of or even agreement about the others, and they don't seem to cover the computing design space sufficiently to inform a complete system design. The result is that many approaches are forced into using superficial architectural observations or computing system design methods instead of organizational principles derived from biology.

In this paper, we try a different approach to programming computational systems to provide the expected level of adaptability. We take feedback loops as the fundamental programming and processing unit, reflection as a fundamental organizing principle, and see how far those choices can be pushed towards implementation. We use Wrappings for the parts that seem to be more explicitly algorithmic, and identify the next set of important challenges in applying and extending the approach. We do not pretend to have all the answers here, just a set of choices and decisions that seems very different from the usual adaptive architectures, and very intriguing in its possibilities.

In this approach, sensors and effectors are always two-way communication, so none of them is passive or ballistic. System activity at this level of detail is continuous. Sensors always adjust themselves to ambient conditions (for the sensor), and effectors always provide feedback about their conditions, such as resistance to motion. Computational reflection is implemented by having *reflection sensors*, which collect the computational steps performed, not the corresponding data. There are also *reflection effectors*, which can change those computational steps.

All processing is performed in loops, and all interaction is performed by two-way usually synchronous connections between loops (for example, the reflection sensors have connections to the loops they sense, the reflection effectors have connections to the loops they change, etc.). Loops can occur at different levels of resolution (and therefore timing),

All parts of the system can have associated monitors, that examine the data or process at hand, perform context-based inference and situation modeling to build higher-level models of the situation or behavior, and then operate goal-based decision processes to determine what to do in a given situation. All of these elements will be described in the rest of the paper.

We start in Section 2 with the relevant biological background, then describe the Wrapping infrastructure that provides much of the flexibility and top-down organization in Section 3.

Then we describe in Section 4 how loop

URI: https://hdl.handle.net/10125/64544
978-0-9981331-3-3
(CC BY-NC-ND 4.0)

programming can lead to highly adaptive systems reflecting those biological and computational principles, and define our current approach to and progress in loop programming in Section 5.

Finally, in Section 6 we describe several of the important challenges that remain in defining loop programming as a suitable paradigm for implementing complex software-intensive systems, and in understanding how it can lead to systems that are more adaptive, more usefully autonomous, and that have better robustness properties than many other approaches.

2. Biological Considerations

In this Section, we describe some of the biological principles that can used [12] [34], and begin to describe how we expect to use them [31] [9].

The start of our biological background description is some results and speculations from theoretical neurobiology [8], that explain how to consider movement and motor processes as having essentially the same overall organization and structure as language processes. That is, from the very earliest and simplest animals, there are layers of representational mechanisms (i.e., symbol systems) that abstract the external situation into an internal situation that may be more stable, more persistent, and more easily processed internally, and in turn converting those internal processes into external behavior. These layers of symbol systems provide a very flexible scheme for representation of events, observations, decisions, intentions, and actions.

In fact, we go further than this notion. We believe that to be an animal on this planet requires some basic adaptability: the ability to sense the environment (to some distance), to choose a course of action (among several or many), and the ability to not persist in a course of action that is not succeeding. This simple kind of adaptability is to be able to choose a response or behavior from a given set of alternatives (we do not mean “choose” here as an explicit decision process, only as the existence of mechanisms that inhibit or prevent all courses of action but one).

We want a stronger kind of adaptability than that. We want the system to be able to create new choices to make the set of alternatives larger.

The next notion is that feedback loops occur at many time and space scales [6] [2] [47] [7], and that this activity powers essentially all local processing. It is this notion that we have taken as the fundamental organizing principle that defines our approach in this paper. The rest of the paper is about what else is needed to make this notion viable as a basis for highly adaptable systems.

The next issue is that of variation. Adaptation requires something that can be changed, and adaptability requires a wide enough set of choices to be available at decision time. These “variation spaces” [4] [34] need to be generated in context, based on the current situation and the capabilities of the resources that are available. Moreover, those capabilities will also be limited by the details of the situation and the specific resource limitations. These “controlled sources of variation” [12] [30] [10] underlie our approach to movement (i.e., motor control) in our autonomous systems. In particular, we do not model a behavior as a “course of action”, but instead as a fat trajectory (a swath through a space of allowed variation), tuned according to context and situation. These large families of related specialized movement configurations [34] are the key to another aspect of system behavior, merging [4]. When the system has multiple possible actions, these fat trajectories can often be partially or wholly merged, to produce a new, more restricted set of trajectories (a “thinner” fat trajectory), that can be refined into actual actions. This process requires arbitration and selection only when the merging trajectories conflict, and allows multiple parallel activities when they do not. Similar research is being conducted in cognitive science [18] [17] [19], and we expect to apply those methods here.

Another aspect of the use of explicit representational mechanisms is a set of theorems that we call the “Get Stuck” theorems [29], that basically imply that systems in complex environments will need to change their own representations of their environment (to emphasize certain aspects and downplay others), in order for their reasoning processes to keep up with persistent changes in the environment. Since the observed environment of these systems also includes the system internal behavior (to some level of detail), these symbol system change processes are also needed for the system to maintain currency with its own models of its own behavior.

Finally, all systems are constrained by what their sensors and internal processing can tell them [39], and all cooperative system architectures are about how the components make agreements on what subset of their capabilities will be used in the combination architecture [38].

These principles have been used in part or in combination for several kinds of system architecture and constructions using an integration mechanism called “Wrappings” [30] [34] [35] which we describe next.

3. Wrapping Software Infrastructure

The Wrapping integration infrastructure is a knowledge-based approach to integration of disparate

resources in a complex software-intensive system [5] [32] [33] [28]. Instead of defining the details here, we list a few of the properties; many of the references have more details.

The Problem Posing interpretation of programs is based on an important change of attitude in system design and implementation [33]. It is a declarative interpretation that can be applied to any programming or design language, and we believe that it affords a clearer way to interpret the expressions of all programs. The basic idea is to consider the code that usually gets written as defining a "resource" that provides some kind of "information service" in response to a "posed problem", and then keep the problems available in the code along with the solutions. This separation of clients from servers has become interesting and useful in larger units (clients and servers are typically entire programs), but we believe that it is important also for smaller units, as far down as one wants to gain the associated flexibilities.

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

Because we separate the problems from the applicable resources, we can use very much more flexible mechanisms for connecting them than simply using the same name. We have chosen in Wrappings to use a knowledge base [33] that maps problems in context to resource applications, and shown that this choice leads to some interesting flexibilities [36] [37], including such properties as software reuse without source code modification and system upgrades by incremental migration instead of version based replacement.

A Wrapping-Based system has two main conceptual components: the *Wrapping Knowledge Base* (WKB), which contains a set of context-dependent mappings from *problems* (information service requests) to *resources* (information service providers), and the *Problem Managers* (PMs), that organize the system by interpreting the Wrappings.

It is a fundamental assumption we make that ALL system activity occurs as responses to problems posed by users (if any) or parts of the system. The coordination of these activities is managed by the PMs using the WKBs. The PMs come in many flavors, so we describe just the simplest ones here.

The default generic *Coordination Manager* (CM) is the heartbeat that keeps the system running. In this simplest case, problems are assumed to be posed and treated sequentially, as shown in the loop on the left side of Figure 1: After some initial context is collected (generally from invocation parameters or design-time preset values), the CM cycles through three steps: "Pose Problem", to solicit or create a problem to consider, "Study Problem", to address the problem according to the current context, and "Assimilate Results", to map those results back into whatever context changes are warranted. The CM is analogous to the "read-eval-print" loop of almost all variants of LISP, and is clearly of the same class as the common activity loops in other applications [2] [47] [15] [25] [20]. The important difference with other activity loops is that these steps are not functions; they are posed problems, treated in exactly the same way through the Wrappings as any other, with different appropriate resources in different contexts (this is how a Wrapping-based system can choose different processing methods for the same problem in different contexts).

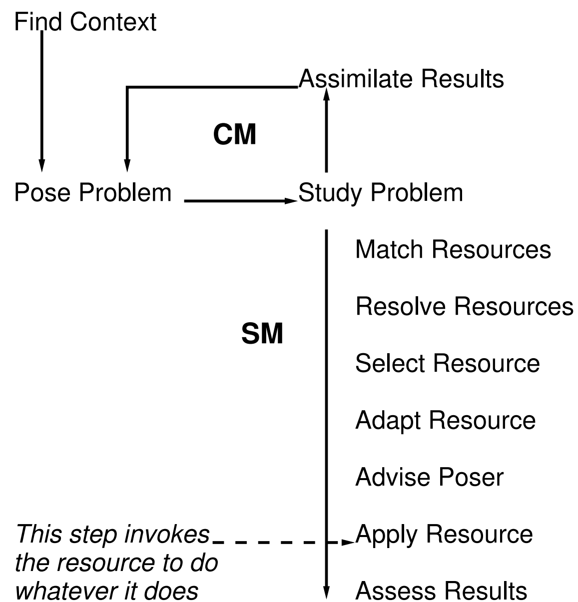

Figure 1. CM and SM Steps

The main part of the problem interpretation process is to map the posed problem into the application of a resource in the current context. The default generic resource that is to be selected for "Study Problem" is the *Study Manager* (SM), which is a planner that organizes the resource applications shown on the right side of the Figure. The CM and SM interact as shown schematically in the Figure.

The default generic SM steps are also posed problems, for which we expect the selected resources to identify appropriate resources for the current problem

in the current context (“Match Resources”), filter them through the context conditions and create a resource application via negotiation with the problem characteristics (“Resolve Resources”), choose one of the remaining applicable resources (“Select Resource”), adapt it as needed (“Adapt Resource”), and announce what is about to happen (“Advise Poser”). After all that, the next steps are to apply the resource (“Apply Resource”) and assess its success or failure (“Assess Results”). This is clearly a simple-minded planner, and if an application has a better notion of what kind of planning is needed, that planner can be added to the resources, with WKB entries that specify under what context conditions to use it. In this same way, every part of the infrastructure can be selected at run time, with the default PMs used only to get started.

It can be ssen that there are no privileged resources; any part of the infrastructure can be replaced (or superseded) at run time, all processing of the Wrappings is explicit, and those processes are also resources, also have Wrappings, and are therefore also replaceable. This is what makes Wrappings ideal for studying infrastructure [28]: it makes no assumptions about the processes and their purposes. Finally, any part of the decision processes that will not change (because, for example, the set of resources for a particular system is fixed) can be compiled out so that there is no performance hit.

There are a few guidelines for building a Wrapping-based system [32], one of which is to use a systematic system engineering method [27], such as the Scenario-Based Engineering Process [42] [43] [44], to go from expectations to responsibilities to activities, and then to identify the activities as problems.

We have also used Wrappings to implement self-modeling systems [36] [37] , that is, systems that have a model of their own behavior that they interpret to produce that behavior. That means they have access to their own processing mechanisms, and can change them.

4. Adaptive Systems Architecture

In this Section, we describe our approach to adaptive system architectures via active loop programming (and Wrappings). The next Section describes some of the actual loop programming attitudes and properties.

Because the environment of essentially any system is dynamic and uncontrollable, it seems to us that any kind of generic viability means that the system operation must include a cycle of interaction with that environment. Such a cycle will need to include determination of something about the environment, making some kind of responsive decision, and then acting within that environment. This ubiquity of loops (implicitly defined by chemical processes or explicitly defined in programming languages) is what stimulated this research investigation.

Similarly, any generic kind of adaptability means that the system is constructed of components of functionality, and that it can move and reconnect components in different configurations, possibly even create new components or component groups.

4.1. Activity Loops

There have been many proposals for fundamental activity in systems embedded in the world (autonomous systems, reflective computational systems, autonomic systems, etc.), of which we mention only a few (see [11] for further discussion). Each of them has some useful hints about the activity selection problem, and is completely capable of explaining some kinds of activity. Each of them describes a cycle of learning something about the environment and then doing something to it.

Perhaps the most common is the activity loop inherent in the operation of the programming language LISP

- READ = get input expression from user;
- EVAL = compute value of expression;
- PRINT = display value to user.

This loop is made quite explicit in LISP [1], but it is implicit in almost every programming language, with READ being generalized to “accept a command from the user or another source”, EVAL to “do what the command requires”, and PRINT to “change the display or output accordingly”. However, very few programming languages provide access to that loop so it can be changed.

Perhaps the most famous explicit loop not in a programming language is the OODA loop defined in military strategy writing [54] (which is actually derived from a much earlier source in manufacturing called the Shewhart or Deming cycle, which ultimately derives from the scientific method as described by Francis Bacon [3]), which partitions all activity into four steps:

- OBSERVE = determine what is around you that requires you to act;
- ORIENT = determine what is your situation;
- DECIDE = decide what to do;
- ACT = do it.

These are our own versions of the definitions; there are many others to be found in the military and business strategy literature. There are a number of issues with this and many other forms of the OODA loop (such as the adaptations called CADE = collect, analyze, decide, act, or MAPE = monitor, analyze, plan, execute for "autonomic" systems [14] [55] [13]), among which are that all steps require both decision and selection processes. We do not think this loop description is adequate for highly adaptable systems, primarily because it is too fixed in its expectations of activity at each step, even if it is stacked up recursively as the next one is. Similarly, while most kinds of adaptive control methods use multiple levels, with upper levels inferring models and changing lower level parameters, very few of them have any reflection at the top level.

A different approach is more suited to autonomous computing systems. The ELF = Elementary Loop of Functioning [2] [47] divides the cycle into the following steps:

- SP = sensory processing,
- WM = world modeling,
- VJ = value judgment,

  and
- BG = behavior generation,

with the interaction between the agent and the environment occurring between BG and SP (generated behavior affects the environment, which in turn is detected by the sensors, to complete the cycle). It is assumed that these processes are continuously active, and that there are different instances of the cycle with different scopes and different scales of interaction (different scope means different ranges of interaction, like different senses or boundaries of attention, and different scale means different resolutions, so different frequencies or minimal indistinguishable stimuli).

The GFACS model [45] [46] is not strictly an activity loop, but we mention it for its decomposition of decisions into three usefully distinguishable parts: G = Grouping, FA = Focussing Attention, and CS = Combinatorial Search (with no fixed assumption of ordering among them). Grouping is a kind of simplification to identify essential commonalities as features (or re-expressing the space in a way more conducive to the search). Focussing is a way to ignore all inessential features and regions of a large search space (reducing the scope and the resolution). Search is just looking through the remaining (reduced) set of choices.

With these notions as background, and the fact that a Wrapping-based system has a fundamental processing loop built into it (the "Coordination Manager" [33] [36] [37] , we decided to consider whether, instead of implementing this or that activity loop in a series of ordinary programming steps, perhaps we could implement all of the program in terms of a fundamental loop construction.

4.2. Reflective Systems

We still want to insist on reflection, and on making the system anticipatory, not merely reactive, since reactive systems are way too slow for a complex environment. Anticipation of environmental changes and necessary responses requires a kind of self-training via anticipatory exploration and response computation, and it means that response computation does not need to occur in real time, only situation recognition (mostly: unforeseen situations and unexpected failures can only be partly prepared for).

The anticipatory exploration is done via simulations for prospective action evaluation. It uses multiple parallel hypotheses, all evaluated and some abandoned (like "Intention movements" in animal behavior), and some possibly merged (when possible). This process may include superficial simulation for quick decisions; these can also be trained. Training is about elaborating responses to recognizable situations, with explicit recognition criteria, and then recording the responses as adaptive behavior units, so that they need not be recomputed when they are needed, only initiated and monitored. More extensive training will provide better and more effective responses (more contingencies accepted and addressed, quicker activity decision criteria, etc.).

4.3. Active Loops

A system having reflexes and reactions at various speeds means it has feedback loops at various levels of detail. When this is extended to include all processing, it is called a "loop" architecture. The question we are exploring here is whether loop architectures can provide sufficient flexibility to make them a good candidate for self-adaptive systems.

Sets of loops are inherently decentralized, since the only interactions are through connections (and the only interferences are through resources). The main difficulty with distribution is that coordination processes are required to bring things back together [33].

We are interested in applying the notion of merging to these systems, but this kind of conceptual dynamics, with fat trajectories and operational variation has a

simple problem: the software world structure is crunchy (completely discrete), whereas the real world is mostly smooth, with bifurcations and a few discontinuities. Moreover, chaos is a long-term phenomenon, not a short-term one. The result is that we can easily imagine merging of activity in the world, but it is much harder to organize it to operate internally.

A connection is not just a stimulus - response relationship. We have to include more in every connection: sequence modeling, sequence with attribute modeling processes (swamp), inference and improvement of model structure, surprises and model adjustment, sensor tuning, sensor replication and domain separation, then we can even consider the meta-system transition (a method for elaborating a complex organization in time [52] [53], though our description below is much more operational than is found in these references). There is an issue of how to identify and compute higher levels of abstraction (more complex structures derived from observation and reasoning), which involves continual monitoring of the environment (or any of the system components under observation) [16] [50].

Another aspect requiring continual monitoring [50] is unanticipated adaptation, which can take the form of partial behavioral reflection (which can be programmer specified at compile time, as in the meta-object protocol [23]), unanticipated partial behavioral reflection, which must be operator specified at run time (when there is an operator), and does not require any preparation of the code, and is correspondingly limited in capability.

What about goals? Satisfaction of goals needs to have a lot of flexibility to be adaptive.

How does the system map goals into actions? If we take top-level goals to be problems posed at the highest level of purpose for the system, and all goals to be problems posed at some level, then this is actually another context-dependent mapping (i.e., a Wrapping).

How do we describe what loops do, so the system can pick and choose the right ones for the task at hand? How does the system determine the task at hand? How does it activate a loop (or is that a contradiction, since the loops are always active, and in that case, what does it mean to choose one)?

For example, for GFACS [2] [47], grouping and combinatorial search are easy, but what is and how do we do focus of attention? Especially since lots of activities are always going on at the same time?

Like what kinds of activities? Maintenance and monitoring processes, integration, inference and modeling processes, alarm and warning processes (these cause a change of attention); these are the autonomic processes that are mostly (or entirely) independent of purpose.

Higher-level control of lower-level processes can also be implemented this way, but the control processes are chosen instead of autonomic; the selection preferences are carried out using context-dependent mappings. In our opinion, the best way to get this generality of processing is to use straightforward default Wrapping processes at the highest levels, and all the rest loops (though we are working on making the default Wrapping processes into active loops also).

Problems are used to select resources, which have inputs and outputs, or connections that include both. This is a classic Wrappings style of behavior. [33].

5. Active Loop Programming

In this Section, we start to define what loop programming needs to be, partly by analogy and partly by definitions. It contains many important questions about scope and semantics, which we are actively working to resolve.

All processing is based on active loops. Each loop has one or more connections, generally to other loops, through which data flows in both directions (in two specified and fixed data spaces). The insistence of two-way communication is much like the original Petri Nets [49] [22], but we allow a higher level of abstraction in these connections. We generally expect synchronous communication (between two communicating entities), but have provisions for the (presumably rare) case of asynchronous data movement (interrupts and open loop messages).

We also want all connections to be local and all data transfer across a connection instantaneous, so we need to consider what processes perform transport. We take the simplest road for now: every loop can be a transporter, that is, loops can have connections that are remote from each other.

The unit of activity is called a token, and represents a set of types, structures, and values. Several tokens may be active in the same loop.

For example, a function is like an activity loop with one connection, and a waiting protocol. A re-entrant function can have several tokens in different stages of execution.

The difference between a loop and an object is that the loop sequences the methods (only considers them in a certain order), while an object can get method calls in any order. This may seem to be a restriction, but it allows us to orchestrate sequences of operations that are extremely hard to identify in object oriented programs.

A loop is a simple closed curve in a conceptual space, with data traveling along the curve, and fixed

connection points marked. Sensors and effectors are just loops whose connections go to the outside environment, across the local system or subsystem boundary (it means ALL sensing is active, and ALL action has immediate feedback, even if it isn't very much data). Data that is input at one connection is processed with the local state to create data that may be output on the next connection.

The local state is what circulates in the loop. One loop can have several local state instances (called "tokens"), denoting active processing within the loop. It is assumed to have a locally fixed data type, which can change from one connection to the next (i.e., each segment of the loop has a fixed data type for the tokens that traverse it). It is affected by the input for persistence, and can be used to create the output.

There is no necessary relationship between the number of local states and the number of connections (though the number of states has to be positive for anything to happen), and there is no presumption that the tokens are in different parts of the loop. The loop is like a reentrant program; the states are like the process instances.

There also need to be rules for creating new tokens and combining them.

Each loop has a characteristic time scale, but they are not assumed to be periodic (though they can be with appropriate protocols at each connection).

5.1. Connections

Two loops can interact through a common connection (inputs and outputs connected like graph incidences, with specified protocols for data movement), or interfere through common resource requirements (common processing or storage resources, with arbitration and time-sharing rules).

It is especially important for programming that EVERY "wait for data" step in any of the protocols have a fairly short timeout with default data or a failure indicator, to prevent deadlocks, as well as a "how fast is this happening" meta-step with criteria for what is too fast, to prevent runaway event cascades.

The way connections are defined is based on an existing communication protocol definition notation called *com* [26] , that was in turn based on theoretical work on the semantics of concurrency [48] [21]. Each connection has a well-defined data type in each direction (there is no need for them to agree), and a notion that the connection is synchronous (both sides wait for connection offers) or asynchronous (neither side waits, and there are default values). Details can be found in the references.

5.2. Loop Operation

A loop is like the program source code, and the token is like the executing process. Among the available process steps are creating loop tokens (need to refer to the loop in which the token is expected to operate), and terminating the current token. The system can also create new loops by incorporating specified files (as can be done with Wrappings), and there are a few ways to create loops during execution time.

One process for spontaneously creating new loops is called factoring. when any state gets large enough (with a processing time based threshold derived from the first "Get Stuck" theorem) [29], The loop is split into two: the state is split into a quotient and remainder (the quotient is an abstraction of the large state, and the remainder a kind of coset).

The new loop is spun off with the abstracted state, and a longer time cycle, and the loops have connections to each other, so that when the "remainder" gets away from the corresponding quotient item (think of it as a kind of abstract counting), it will meet the new loop for a state update (in both directions remember). Also, any connections in the original loop are also factored into connections for the separate ones.

There are a number of operations on and in loops. We mentioned above the need for coordination. This is accomplished by explicit coordinating processes that are loops with multiple connections to the loops being coordinated. Then these coordinative loops can also have connections to provide other higher-level loops with summaries or analyses.

We have emphasized computational reflection in all of our discussions. This begins with a simple internal sensor that provides a sequence of operations in a "reflective" connection, not the usual sequence of data values (this process argues for simple operations within the loops). Then all other sensor monitoring and analysis processes can be applied to it.

This kind of reflection for processes us called "behavioral reflection" [50] [51], and we often make the engineering decision not to activate all of the reflection loops, which is called "partial behavioral reflection".

To make reflection useful, the system also requires various kinds of summaries to be provided to other processes. There are integration and inference processes that allow values to be considered at different scales, and with different smoothers. That means that there are always a whole stack of loops with different scales, related by integration or inference.

5.3. Other Mechanisms

Instead of describing the current state of our research in the other mechanisms we expect to use, we will list them and describe what we expect to get from them.

These systems will require processes for event identification and event pattern detection, especially in the context of a noisy environment. These systems will need to be able to compute some analog of "abstraction", to get the appropriate level of generality in their models of environmental behavior (it should be noted that even at this detail, flexibility and adaptation will be needed in the very representational mechanisms used). This expectation leads directly to the notion of semiotic boundary [9], a fundamental concept in semiotics [46] [19], which treats the creation and use of symbols in computing systems.

Designing these systems will require a good set of flexible models of conceptual domains [35], and different methods for adapting them in different directions in different contexts [52] [53].

The Wrappings approach has mappings from the abstract problems to the concrete resources, as managed by context conditions. For these systems to reach the level of adaptation we desire, they will also have to be able to make the reverse mapping (to a certain extent), abstracting the observed behavior into descriptive models.

These systems will need access to the models and to the model interpreters. Remember, declarative programs do not DO anything; they depend on the behavior of an interpreter to make them active (and different interpreters can lead to different behavior, even if the declarations are taken to be logical statements). The Wrapping approach was specifically designed with this requirement in mind.

The kind of programming notations that will be useful for active loop programming are different from most existing kinds of notation, since a component is not just a set of specific structures, but a behavior generated from cooperating loops (it is like the APIs are much more complex than in most programming languages). The compositional structure of these loops also needs to be defined, and the possibility of verifying a complex loop programs will lead to new techniques.

This section is a smattering of hard research and engineering questions that we are presently trying to address.

6. Challenges and Speculations

It should be clear that there are many unanswered questions in this approach. In this Section, we describe what we think is hard, and what we think is easy (actually, none of it is easy, except that the Wrapping infrastructure is already well developed for this kind of flexibility and adaptation [36] [37]).

According to [40], there are four essential attributes to a self-adaptive system roadmap:

- design space for adaptive solutions;
- processes development, deployment, operation, maintenance, evolution;
- decentralization (of specific research topics);
- practical run-time verification and validation.

The first requirement is to construct or adapt development methods, techniques, and tools, which will require different classes of decisions [14]:

- modeling dimensions (our conceptual domains);
- requirements (on behavior and resource utilization);
- make feedback control loops more explicit (we do not think these are the right kind of loop);
- assurance (of behavior).

Then there are four essential attribute challenges, some of which are addressed by Wrappings:

- design space for adaptive solutions: representation, observation, control, identification, adaptation;
- processes development, deployment, operation, maintenance, evolution: software system engineering;
- decentralization: MAPE = Monitor, Analyze, Plan, and Execute (this is a rehashed and redirected OODA loop [54], as almost all activity loops are), with interacting control loops ;
- practical run-time verification and validation .

These challenges are well-handled in Wrapping-based systems [10] [11], but this application context imposes more stringent expectations.

References

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

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

[3] Francis Bacon, *Novum Organum*, London (1620); several translations exist on the Web, and the Wikipedia source on the OODA loop has a nice history

[4] Kirstie L. Bellman, *The Conflict Behavior of the Lizard, Sceloporus Occidentalis, and Its Implication for the Organization of Motor Behavior*, Ph.D. Dissertation, U.C. San Diego (1979)

[5] Kirstie L. Bellman, "An Approach to Integrating and Creating Flexible Software Environments Supporting the Design of Complex Systems", pp. 1101-1105 in *Proceedings of WSC1991: The 1991 Winter Simulation Conference*, 08-11 December 1991, Phoenix, Arizona (1991)

[6] Kirstie L. Bellman, "Motion and Communication in Layers of Symbol Systems", (presented to) *SMC'98: The 1998 IEEE International Conference on Systems, Man, and Cybernetics*, 11-14 October 1998, San Diego, California (1998)

[7] Dr. Kirstie L. Bellman, "The Challenges of Integrating, Managing, and Analyzing Complex Systems With Sensors", *DASP'2006: The 5th Workshop on Defence Applications of Signal Processing*, 10-14 December 2006, Fraser Island, Queensland, Australia (2006)

[8] Kirstie L. Bellman and Lou Goldberg, "Common Origin of Linguistic and Movement Abilities", *American Journal of Physiology*, Volume 246, pp. R915-R921 (1984)

[9] Kirstie L. Bellman, Christopher Landauer, "Computational Embodiment: Biological Considerations", pp. 422-427 in A. M. Meystel (ed.), *Proceedings of ISAS'97: The 1997 International Conference on Intelligent Systems and Semiotics: A Learning Perspective*, 22-25 September 1997, NIST, Gaithersburg, Maryland (1997)

[10] Dr. Kirstie L. Bellman, Dr. Christopher Landauer, Dr. Phyllis R. Nelson, "System Engineering for Organic Computing: The Challenge of Shared Design and Control between OC Systems and their Human Engineers", Chapter 3, pp. 25-80 in Rolf P. Würtz (ed.), *Organic Computing*, Understanding Complex Systems Series, Springer (2008)

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

[12] Kirstie L. Bellman and Donald O. Walter, "Biological Processing", *American Journal of Physiology*, Volume 246, pp. R860-R867 (1984)

[13] Yuriy Brun, Ron Desmarais, Kurt Geihs, Marin Litoiu, Antonia Lopes, Mary Shaw, and Michael Smit, "A Design Space for Self-Adaptive Systems ", submitted to Software Engineering for Self-Adaptive Systems, II (2012)

[14] Yuriy Brun, Giovanna Di Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi Müller, Mauro Pezze, and Mary Shaw, "Engineering Self-Adaptive Systems through Feedback Loops", p.48-70 in Betty H.C. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee (eds.), *Software Engineering for Self-Adaptive Systems*, SLNCS 5525, Springer Verlag (2009)

[15] Arjun Chandra, Peter R. Lewis, Kyrre Glette, and Stephan C. Stilkerich, "Reference Architecture for Self-aware and Self-expressive Computing Systems", Chapter 4, p.37-49 in [41]

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

[17] Gilles Fauconnier, *Mappings in Thought and Language*, Cambridge U. Press (1997)

[18] Gilles Fauconnier, Eve Sweetser, George Lakoff, *Mental Spaces*, MIT Press (1985), Cambridge U. Press (1994)

[19] Gilles Fauconnier, Mark Turner, *The Way We Think:, Conceptual Blending And The Mind's Hidden Complexities*, Basic Books (2903)

[20] 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, p.237-275 in [24]

[21] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall (1985)

[22] Gabriel Juhás, Fedor Lehocki, Robert Lorenz, "Semantics of Petri Nets: a Comparison", Shane G. Henderson, Bahar Biller, Ming-Hua Hsieh, John Shortle, Jeff D. Tew, and Russell R. Barton (eds.) *Proc. WSC'07: the 2007 Winter Simulation Conference*, 09-12 December, Marriott Hotel, Washington D.C. (2007)

[23] Gregor Kiczales, Jim des Rivières, Daniel G. Bobrow, *The Art of the Meta-Object Protocol*, MIT Press (1991)

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

[25] 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, p.3-16 in [24]

[26] Christopher Landauer, "Network and Protocol Modeling Tools", pp. 87-93 in *Proceedings of the 1984 IEEE / NBS Computer Networking Symposium*, December 1984, NIST, Gaithersburg, Maryland (December 1984)

[27] 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)

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

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

[30] Christopher Landauer, Kirstie L. Bellman, "Computational Embodiment: Constructing Autonomous Software Systems", pp. 42-54 in Judith

A. Lombardi (ed.), *Continuing the Conversation: Dialogues in Cybernetics*, *Volume I*, *Proceedings of the 1997 ASC Conference*, American Society for Cybernetics, 08-12 March 1997, U. Illinois (1997); pp. 131-168 in *Cybernetics and Systems: An International Journal*, Volume 30, Number 2 (1999)

[31] Christopher Landauer, Kirstie L. Bellman, "Computational Embodiment: Software Architectures", pp. 205-210 in A. M. Meystel (ed.), *Proceedings of ISAS'97: The 1997 International Conference on Intelligent Systems and Semiotics: A Learning Perspective*, 22-25 September 1997, NIST, Gaithersburg, Maryland (1997)

[32] Christopher Landauer, Kirstie L. Bellman, "Lessons Learned with Wrapping Systems", pp. 132-142 in *Proceedings of ICECCS'99: The 5th IEEE International Conference on Engineering Complex Computing Systems*, 18-22 October 1999, Las Vegas, Nevada (1999)

[33] 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)

[34] Christopher Landauer, Kirstie L. Bellman, "Computational Embodiment: Agents as Constructed Complex Systems", Chapter 11, pp. 301-322 in Kerstin Dautenhahn (ed.), *Human Cognition and Social Agent Technology*, Benjamins (2000)

[35] Christopher Landauer, Kirstie L. Bellman, "Computational Infrastructure for Experiments in Cognitive Leverage", in *Proceedings of CT'2001: The Fourth International Conference on Cognitive Technology: Instruments of Mind*, 06-09 August 2001, Warwick, U.K. (2001)

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

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

[38] Christopher Landauer, Kirstie L. Bellman, "Integration by Negotiated Behavior Restrictions", *Proc. 4th SiSSy: Workshop on Self-Improving System Integrating*, Part of *SASO2017: The 11th IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, Part of *FAS*: Foundation and Applications of Self-* Computing Conferences*, 18-22 September 2017, U. Arizona, Tucson, Arizona (2017)

[39] Christopher Landauer, Kirstie Bellman, "Living in a Sensor-Limited World", *Proc. CogSIMA 2019*: *2019 IEEE Conference on Cognitive and Computational Aspects of Situation Management*, 08-11 April 2019, Las Vegas (2019)

[40] Rogério de Lemos, Holger Giese, Hausi A. Müller, Mary Shaw, "Software Engineering for Self-Adaptive Systems: A Second Research Roadmap", (Draft Version of November 9, 2011)

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

[42] Karen McGraw and Karan Harbison, *Knowledge Acquisition using the Scenario-Based Engineering Process*, Lawrence Erlbaum (1995)

[43] Karen McGraw and Karan Harbison, *User-centered Requirements: The Scenario-Based Engineering Process*, Lawrence Erlbaum (1997)

[44] E. Mettala, D. Cook and K. Harbison, "Application of the Scenario-based Engineering Process to the Unmanned Ground Vehicle Project", in O. Firschein and T. Strat (eds.) *Reconnaissance, Surveillance, and Target Acquisition for the Unmanned Ground Vehicle*, Morgan Kaufman (1997)

[45] Alex Meystel, "Multiresolutional Architectures for Autonomous Systems with Incomplete and Inadequate Knowledge Representations", Chapter 7, pp.159-223 in S. G. Tzafestas, H. B. Verbruggen (eds.), *Artificial Intelligence in Industrial Decision Making, Control and Automation*, Kluwer (1995)

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

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

[48] Robin Milner, *A Calculus of Communicating Systems*, Lecture Notes in Computer Science 92, Springer (1980)

[49] Carl Adam Petri, "Kommunikation mit Automaten", Ph.D. thesis, U. Bonn (1962); "Communication with Automata", Supplement I to *Tech. Rep. RADC-TR-65-377*, vol.1, Rome Air Development Center (January 1966)

[50] David Röthlisberger, Marcus Denker, Éric Tanter, "Unanticipated partial behavioral reflection: Adapting applications at runtime", Computer Languages, Systems and Structures, v.34, p. 46-65 (2008)

[51] Éric Tanter, Jacques Noyé, Dennis Caromel, Pierre Cointe, "Partial Behavioral Reflection: Spatial and Temporal Selection of Reification", *Proc. OOPSLA'03: 2003 Conference on Object-Oriented Programming, Systems, and Languages*, 26-30 October, Anaheim, California (2003)

[52] Valentin F. Turchin, *Phenomenon of Science*, Columbia Univ. Press (1977)

[53] Valentin F. Turchin, Cliff Joslyn, *The Metasystem Transition*, Principia Cybernetica Web, (1993); last revised 19 July 1999, on Web at URL `http://pespmc1.vub.ac.be/MST.html` (availability last checked 15 June 2019)

[54] David G. Ullman, "OO-OO-OO" the Sound of a Broken OODA Loop, *Crosstalk* (April 2007)

[55] Danny Weyns, Sam Malek, Jesper Andersson, "On Decentralized Self-Adaptation: Lessons from the Trenches and Challenges for the Future", *Proc. SEAMS'10: the 2010 Conference on Software Engineering for Self-Adaptive Systems*, 02-08 May, Cape Town, South Africa (2010)