A Design Proposal for Self-Adapting Analytic Systems: Properties and Transformation

Stephen Kaisler
SHK & Associates
skaisler1@comcast.net

J. Alberto Espinosa
American University
alberto@american.edu

William H. Money
The Citadel
wmoney@citadel.edu

Frank Armour
American University
fjarmour@gmail.com

Abstract

This paper continues the development of concepts, structures and transformations for developing self-adapting analytic systems. We propose five properties that such systems must have and present some examples of base structures and subsequent transformations. We suggest a software architecture based on a virtual machine which provides an analytic infrastructure on which domain analytics can be built. The structures and transformations for developing the analytic infrastructure and domain analytics are based on previous work by Greiner (1980), Lenat (1983), and Oresky, Clarkson, Lenat, and Kaisler (1989).

1. Introduction

In a previous paper (Authors 2024), an examination of analytic morphology and the process of transforming analytics to improve their properties was begun. Morphological structures and transformations were introduced to frame the focus of this research effort based on a literature search. We proposed that a self-contained general-purpose analytic system infrastructure should reason about and successfully evolve itself to solve new problems in a domain represented by the analytic system.

Therefore, it is important to describe the properties required by such a system as the world races pell-mell into an environment filled with LLMs that have great capabilities, controls, and tools that pick the next word based upon controls and probabilities that direct them to select from certain word vectors. We do not believe that an LLM is adaptive in the sense that its behavior is based on a direct transformation of the input or a transformation into machine language.

Like most intellectual domains, there have been many papers and books that have addressed theoretical and applied aspects of self-adaptivity. Our approach is to develop an experimental system that will allow us to try different representational and reasoning mechanisms to begin to understand how the underlying mechanisms exhibited by humans can be mechanized in a computer system

We define a *self-adapting system*, e.g., a self-modifying and evolving system, as one that responds to external and internal stimuli by modifying its behavior through changes to existing system code; its structures, including generating new structures and transformations via an embedded, self-Reflecting and self-assessing control model. Numerous models of self-adaption using these properties occur naturally in biological systems (Smith 1998). Cai, Shao, and Vaynberg (2007) assert that "self-modifying code broadly refers to any program that generally loads, generates, or mutates code at run-time". We assume a somewhat stricter definition as discussed below.

Many approaches to self-adaptive systems use dynamic modification of the system configuration. Such modifications retain the system's essential behavior, but change how it is evinced in dynamic situations. The behavior remains the same as many such systems are implemented through compiled code. Genetic algorithms (Goldberg 1989) use the selection, mutation, and recombination operators as a means of emulating biological system. Many systems self-modify, but do not self-evolve.

A notable example is Cyc (Guha and Lenat1990, Guha and Lenat 1991), which is a commonsense knowledge base implemented using the Cycl representation language. Cyc incorporates concepts, rules for reasoning and different types of inferencing, and the use of domain microtheories t. Cyc appeared to be self-adaptive, but there seemed to a human-in-the-loop reviewing its findings.

Our interest in self-adaptive systems is based upon the understanding that they have been partially available for over 50 years in various forms. None of the forms currently integrate all of the properties together. Early systems with limited memory capacity dynamically generated new code for certain operations. An early example was IBM's System/360 OS, which dynamically generated channel programs for peripheral devices. The literature reports that Malware programs are self-modifying (Bonfante, Marion, and Reynaud 2009). IARPA's STONESOUP

URI: https://hdl.handle.net/10125/108962
978-0-9981331-8-8

program (IARPA 2009) investigated self-modification as a way to detect, mitigate, and repair flaws in source code that could lead to intrusion and suborning of control in applications.

Over the past two decades, considerable attention has been focused on adaptive systems as system complexity overwhelmed the software engineering methods used to create them. Cost and schedule overruns are frequent and functionality is often reduced from that initially envisioned. The objective of adaptive systems research has been to allow the system to evolve from its initial implementation as it is operationally used as opposed to trying to plan and test for every contingency in system usage before deploying it (Oreizy, Gorlick, Taylor, and Heimbigner 1999; Karsal and Stzipanovits 1999). This approach led us to define five properties that a self-adapting analytical system must possess.

This project focuses on three problems. First, we will identify an initial set of properties and structures for self-adapting systems. Second, we will identify structures and transformations for describing and implementing dynamic modifications to a system. We will show how to apply the transformations to different types of analytic structures to yield modified or new structures in a particular problem domain. Third, we will develop and provide guidance for evaluating transformation results on structures and their properties to determine how they improve the analytic system's problem solving capability.

2. Background

Our literature search revealed previous work focused on algorithms for program modification by external applications with different intentions. For example, optimizing compilers modify source programs to eliminate code that is never accessed or move code out loops that should only be executed once. Compilers for parallel processing pursue *computational alignment* of loops and data arrays to parallelize sequential programs and improve performance. These modifications are performed on the intermediate representation of a program. (Kulkarni and Stumm 2007), Banerjee 1994).

The development of self-adaptive systems that generate new structures and transformations using appropriate domain knowledge and integrate the changes into applications systems can provide a mechanism to attain some degree of enhanced or new capability. But, these require extensive domain knowledge for all aspects of software engineering due to the large number of components and the massive number of configurations possible for the many components.

The transformations developed in this research must occur at the symbolic level as compilation often eliminates critical information in translating source representations into machine language.

The system we propose will have a self-adaptive analytic capability and a representation language that can change its behavior by assessing its current situation and environment, determining modifications necessary to affect behavioral change, including performance improvements, and modifying itself to implement such modifications semi-automatically, initially, and, semi-autonomously, eventually.

This objective yields the following research questions:

RQ1: What properties and their implementation should an analytic system possess to allow its evolution?

RQ2: What syntax and semantics should a representation language take to describe the fundamental structures of an analytic system. including rules, to dynamically modify structures at run-time to affect behavioral change?

RQ3: What mechanisms should be used to evaluate the evolution of a system?

3. RQ1: Analytic System Properties

A *self-adaptive analytic system* (SAAS) will support evolution of existing systems written in a representation language. We have been influenced by Greiner's research (Greiner 1980) and Lenat's experience (Lenat 1983) with AM and Eurisko.

Greiner's and Lenat's work focused on developing a general (possibly universal) descriptive language, *Representation Language – 1 (RLL-1)* capable of defining and manipulating any concept. RLL-1 was used to define some primitive, minimal concepts for Eurisko, and, later, influenced the design of STRADS (Kaisler 1989; Oresky, Clarkson, Lenat, and Kaisler 1990,; Kaisler 1991; Kaisler and Modjeski 1990) and Cyc (Lenat 1983). However, as it proposed a design for a representation language, it was never fully implemented in any system of note.

3.1 Self-representation

Self-representation means that the system is expressed in a language which is self-describing. Lisp and Prolog are examples of programming languages that are self-describing. Every Lisp data structure and function can be represented as a list. Similarly, every Prolog function can be represented as a logic statement. Both forms allow operators, that can be extended through the analytic system hierarchy.

Self-representation means that the SAAS can examine any of its elements and transform any of its elements without resorting to external functions or programs. Self-representation requires that a set of descriptive slots be defined for the analytic structure that can capture its properties (Brown and Palsberg 2015)

Self-representation also requires that these slots are amenable to transformative operations that can generalize and specialize structures and generate new structures that are useful for problem solving. Lenat (1983) explored some aspects of this requirement in AM and Eurisko (Lenat and Brown 1984).

The definition of an *Isa* slot is an example of self-representation. An *Isa* slot specifies that a structure is a subclass of a more general class. A suggested description for an Isa slot is shown below.

Name:	Isa
Domain:	System Infrastructure or Macrostructure
Isa:	Slot
ValueType:	Set
Inverse:	Examples
MakesSenseFor:	Structure, Transformations
Creator:	SHK
CreateDate:	06/12/2024
Generalizations:	AKindOf
Description:	Specifies the class an entity belongs to
UsedBy:	InheritanceThruIsas

[The *Isa* description is defined in the infrastructure domain?] *Isa* itself is a subclass of Slot, which describes all possible slots used in the system. Slot is a template for all subclasses, so the values of its slots are datatypes. A partial definition for Slot might be:

Name:	Slot
Domain:	System Infrastructure or Macrostructure
Isa:	Structure
ValueType:	Structure
Inverse:	NIL
MakeaSenseFor:	Structures, Transformations
Creator:	String
CreateDate:	Date
Generalizations:	NIL
Description:	The template for a slot which is a descriptor for an entity
UsedBy:	Transformations

As the system is defined, more slots will be identified and described. Infrastructure slots are foundational slots for the analytic system.

3.2 Self-reflection

Self-reflection is the property of an SAAS that enables it to generate feedback on its planning and reasoning processes to provide refinement guidance foor its internal structures and reasoning mechanisms, e.g., being aware of its own computational and reasoning state(s). Awareness must detect change in its environment and in its internal state in response to environmental changes as well as internally generated changes. This awareness will allow the system to determine weaknesses, determine how to correct them, select the appropriate transformation operators, make the changes to its components to improve its computational state, and deploy the changes to its system code in order to do so. The description of a State might be the following:

Name:	State
Domain:	System Infrastructure or Macrostructure
Isa:	Structure
ValueType:	String?, Structure?
Inverse:	State
MakesSenseFor:	System?
Creator:	SHK
CreateDate:	06/12/2024
Generalizations:	NIL
Description:	Specifies the current state of the system as a predefined term or a structure
Used by:	Transformations

The state of the analytic system will be assessed and used by rules to control the operation of the system.

3.2.1 Rules

Rules will be defined that detect changes, select an appropriate response, and perform modifications to structures and transformations. Rules will embed the control structure that drives the system during modification and evolution. For example, the following rules for the rule selection control structure are suggested by our study of Eurisko (Lenat 1983):

Rule: Given an assessment of the current state of the system that mandates a potential modification, select a specific rule over a more general rule with a shorter running time or requires a fewer number of user interactions to be initiated.

Rule: Given an assessment of the current state of the system that mandates a potential modification, select a rule with the fewest number of conditions satisfied that has a high potential worth to improve the system.

We will identify and define slots to support rule selection and executions of rules. Rules will be organized into a class hierarchy based on type. For the above rules, some slots might be expected running time, user interaction requirements, worth for improvement, and number of conditions satisfied.

3.3 Self-assessment

Self-assessment means that an SAAS should be able to determine if a current structure or operation (a) satisfies one or more predicates pertaining to its features and (b) if so, enables it to be modified to provide new structural and transformative capabilities. Predicates are the mechanism for assessing a feature of a structure and are defined as part of the infrastructure

An associated problem is the ability of the system to be aware of its physical and operational environment, and to respond to signals from the environment when required. These signals can be of two types: domain space changes and domain temporal changes.

A system that responds to its environment is often said to be *context-aware*. This corresponds to an open versus closed world model. However, the additional code and rules to handle an *open world* can be extensive and divert attention from the primary development process. Thus, we elect to focus initially on a *closed-world model*, but consider aspects of existing in an open world model during the initial development stages.

An example of a rule for self-assessment might be:

Name:	StateAssessment
Domain:	System Infrastructure or Macrostructure
Isa:	Rule
ValueType:	Rule
Inverse	NIL
MakesSenseFor:	System?, State?
Creator:	SHK
CreateDate:	06/12/2024
Description:	Determines if an assessment of system state has not been performed since (TBD), then set the system state to ”assessment” and add relevant rules to the agenda.

3.4 Self-modification

Self-modification allows operators, implemented as both structures and infrastructure functions, which are invoked by executing rules, to modify the definition of structures. A set of rules can manipulate a single-ended list to change its structure and functionality to, perhaps, a double-ended list with additional functionality given by new rules. Transformers, will be defined by predicates and rules that can modify existing structures to yield new structures. Self-modification requires meta-knowledge expressed as slots with values and rules that embody the reasoning and decision-making elements for modifying structures.

As noted above, while the eventual goal is a fully automatic capability, the initial goal is to implement a semi-automatic system that is semi-autonomous. As Laws, Taleb-Bendiab, Wade, and Reilly (2001) suggested, for a system to exhibit autonomy requires effectors, coordination, management, audit, future scenario analysis, and system identity. This is comparable to the Real-time Control System (RCS) reference model architecture, which describes similar building blocks (Randles, Taleb-Bendiab, Miseldine, and Laws 2005).

Following Lenat’s approach, we will define rules and predicates that assess the structures and create new structures in SAAS when certain conditions are met. For example, if a structure has no examples or no subclasses, then place a task on the agenda to try to define some subclasses for a structure by modifying

some of its features through restriction of values of those features. This action would subsequently queue up tasks on the agenda to assess those new structures.

3.5 Self-evaluation

Self-evaluation means that the SAAS should be able to generate tests, execute those tests against new structures, including transformations, and determine if a transformation was successful in extending a structure and/or modifying its behavior. Implicit in this property is that a structure consisting of a test, an execution protocol/environment, and a result evaluation will be defined relative to other structures as part of the SAAS.

4. RQ2: Infrastructure

Infrastructure arises from two perspectives. The base hardware and software on which a self-adaptive analytic system runs represents the physical execution environment. We assume that this infrastructure, e.g., a "virtual machine", is some virtual processor, its operating system, and a programming language which provides a virtual environment. The use of a virtual machine facilitates the transport of an analytic system to different physical environments as exemplified, for example, by the Java Virtual Machine (JVM) or Medley Interlisp's maiko (Interlisp.org 2024).

The SAAS provides an infrastructure that constitutes the application development environment. Applications are constructed using the capabilities and tools of the infrastructure. The basic structures and transformations of the analytic system will be described in a separate document??.

We will define the structures and transformations that are environment-free meaning they are independent of a particular hardware and only dependent on the Language Virtual Machine (LVM). This provides the application developer with flexibility in building domain specific analytics that could run in different computational environments.

Beyond the analytic infrastructure and macrostructure are domain-specific procedures and rules that enable the user to build a semi-automated evolving application systems. Table 1 depicts a proposed structure for the analytic system.

Table 1. Analytic System Architecture

Component	Description
Analytic Application	A particular domain application (ex: chemistry)
Domain Specific Infrastructure	Domain structures and rules defining the concepts and transformations of the domain.
Analytic System Infrastructure	Analytic structures and rules defining the concepts and transformations of the analytic system.
Langauge Virtual Machine	The implementation language for the analytic system.
Operating System	The virtual system environment
Hardware	A particular machine supported by the operating system.

4.1 Analytic Infrastructure Design

Analytic Infrastructure defines a set of procedures for manipulating analytic structure representations. Procedures are implemented using the LVM for efficiency purposes in computation time, basic transformation parsimony, evaluation mechanisms, storage utilization, and cognitive economy (Lenat, Hayes-Roth, and Klahr 1979). Infrastructure procedures will be documented in the AMT (Kaisler 2024c).

The Analytic Infrastructure defines a set of rules using the LVM capabilities which could be a rule-based system such as CLIPS (Giarratano and Riley or Medley LOOPS (Kaisler 2024a, Kaisler 2024b). The rules contain the logic for transforming the structures (including rules) of the analytic system. The rules invoke procedures in the infrastructure to make structural modifications – adding, deleting, or enhancing the structures.

5. Extended Example

This section presents an extended example to illustrate some of the structures and transformations of the proposed analytic system. The example used is chosen from chemistry. A more complex set of transformations for translating object-oriented to process-oriented programs can be found in Kaisler (1997).

5.1 Simple Graph

As defined in AMT, a Simple Graph is a list of lists G = (V,E), where G is the name of the list, V is a list of nodes and E is a list of edges. Edges are described by a list of the two nodes connected by an implied edge. A few examples are:

Hydrogen = ((H), NIL), since a hydrogen atom is an isolate. Deuterium = ((H1, H2), (H1,H2)). Water = ((H1, H2, O), ((H1,O), (H2,O))). We can continue this for molecules comprised of Hydrogen and other elements. This representation is applicable to any molecule or, indeed, to any structural description of a system. Figure 1 depicts the graph structure of H2O.

H1, H2, and O are atoms that can be described by simple structures, which have slots that have properties. H1 as a Hydrogen atom could be described as H1 = (<slots>), where <slots> is a list of slots describing a Hydrogen atom. A brief (incomplete) description of H1 and H2 might be:

```
H = (    (name Hydrogen)
         (isa Element) (phase gas)
         (weight 1.0080) (Z 1)
         (group 1) (period 1)
         (electron 1s1) (density 0.8988)
         (meltingpoint 13.99K)
         (boilingpoint 20.271K))
```

where the superior structure of H could be described as an Element (incomplete):

```
Element =
((slots name isa phase weight number group period electron
melting point boiling point density)
(name (type string))
(isa type structure))
(phase (type (one of gas metal liquid)))
(weight (type real) (units ??))
(Z (type integer) (description Atomic Number))
(group (type integer))
(period (type integer))
(electron (type string)))
```

We have omitted many slots in this example, which would be used to support behavior of the structure and transformations upon it. For example, we might define an AbstractSlot which serves as a template for concrete slots. Slots will also be defined for functions or procedures that may be used be applied to a particular structure. These are documented in the AMT.

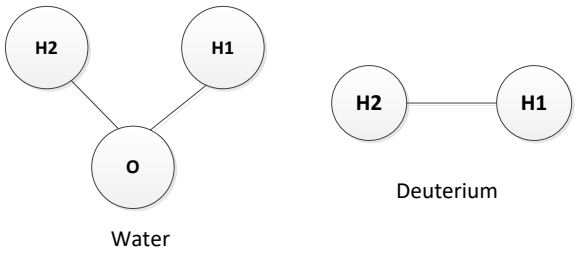

Figure 1. Simple Graph Examples

5.2 Simple Graph Transformations

Simple graph transformations can be described that modify the structure and/or descriptions of simple graphs. Here are some examples:

```
; Removing a node from a graph
(remove-node rule (graph node)
     (description “remove node from graph”)
     (if-graph graph)
     (if-node graph node)
     (then-remove-node graph node)
     (then-assign-name $graph)
)
```

A new graph, e.g., $graph, (or any other structure) is created as a result of an action, $<Structure> is a temporary structure representing the new graph. Provision must be made to provide a name for the new structure either through user intervention or as an element of the rule. The type of the new structure is the same as the source structure.

In Figure 2, removing the O(xygen) node, yields Deuterium. However, removing a node (or adding a node) could yield an invalid structure. The evaluation mechanism must ensure that new structures are valid, if it can, or alert the user with a message as to the reason for invalidation.

5.3 Modifying a Transformation

Transformations can be modified to yield new analytics, which, in turn, can lead to new capabilities and behaviors. A simplified example would take the rule above and modify it to replace node1 with a different element. EURISKO pioneered this approach in several applications.

```
; replace-node-rule (graph node1 node2)
(replace-node-rule (graph mode1 node2)
    (description "replace node1 with node2 in
    graph")
    (if-graph graph)
    (if-node graph node1)
    (then-replace graph node1 node2)
    (then-assign-name $graph)
)
```

In Figure 2, we could replace "O" with "S" which yields Hydrogen Sulfide, H2S. As with node deletion or addition, replacing a node with a different node could also yield an invalid structure. Rules to evaluate transformation results must check for invalid configurations. This will require access to external databases to assist in the evaluation.

6. Generating New Transformations

To generate the new transformation, replace-graph-node from remove-graph-node, we could define and apply the following rule:

```
; Generate a rule to replace a node
(generate-replace-node-rule (rule remove-node-clause)
    (description "generate replace-node rule from remove-
node rule")
    (if-rule rule)
    (if-then-clause remove-node-clause)
    (then-remove-clause remove-node-clause)
    (then-add-clause replace-node-clause)
    (then-assign-name $rule)
)
```

This is a simplified description of the transformation. The actual rule would be more complex and handle different cases.

6.1 Symmetry

Symmetry is a term that has been applied in many disciplines with different meanings, but generally means that an object has the property of symmetry if there are transformations that could change its structure but not the values contained within the structure or change the transformation operations without changing its structure. A *symmetric transformation* is one that is structure-preserving of an object, such as a matrix transpose. An inverse transformation, T-1(T(x)) -> x returns the original structure.

With the definitions, it is clear that not all sequences of transformations will be symmetric. However, identifying (sequences of) transformations that are symmetric will provide a new tool for automating transformations that maintain the integrity of the overall system

7. RQ3: Evaluating Transformations

This research also focuses on developing a semi-automated capability for evaluating new domain structures and transformations that will yield new analytic capabilities within the domain.

Semi-automation of evaluation mechanisms for assessing the validity of new structures and transformations may incrementally improve the performance of the system.

Assessing the results of the application of the transformations initially will be semi-automated in order to best utilize the systems as well as the expertise of the users. As we develop the set of rules for transformation and evaluation and observe the system's performance, we expect to be able to transition to an automated operation. We expect that some evaluation rules will not be able to determine a successful result and may not be able to provide a specific cause of failure, which will necessitate some user intervention.

When new structures are derived as an outcome of executing rules, several questions and issues need to be addressed:

- Is the structure just created a new structure, or just a variation of an existing structure?
- Even if it is a new structure, does it represent enough of an enhancement to retain it?
- How do we visualize the behavior of an evolving analytic systems and/or applications developed using it?

7.1 Visualizing Self-Modifying Code

Self-modifying system can exhibit behavior that is complex and difficult to understand. As Dux, Iyer, Debray, Forester, and Kobourov (2005) note, our standard software engineering models are based on the concept that a program is relatively static, particularly when it has been compiled into machine code. Visualization can be used in two aspects: (1) to see how a particular structure is modified over time, and (2) to visualize behavioral changes exhibited by a structure as it evolves.

7.2 Evaluating Structural Changes

Changes in morphological structure can lead to new insights in how to use the data structures. Some changes will be exploratory and may be discarded if careful evaluation shows little contribution to the analytic system. We expect that a general infrastructure for evaluation rules as well as specific rules for evaluating classes of structures will be developed.

Given the model presented here, semi-automation of the evaluation of new structures or modified structures as well as transformations is desirable. It removes the burden of requiring the user to inspect each new structure of which there may be many to determine if it matches an existing structure.

Given Figure 2, the system could ask the user if H1-H2 is a new structure in the current system. If the user answers No (in this case), the user would be asked to supply a name for the new structure

With semi-automation, given the new structure, the system will search its internal database to see if the new structure matches any existing structure. If so, it copies the relevant data from the structure in the repository to populate the new structure description.

The role of external databases, their accessibility, and the information they can provide is yet to be determined. We expect this will provide useful information for creating and refining new or existing structures and transformations.

7.3 Evaluating New Transformations

Similar issues arise when new transformations are created as a result of modifying existing transformations, including:

- Does the transformation just create a new transformation, or is it a variant of an existing transformation?
- Even if it is a new transformation, does it provide a significant new capability to advance the capabilities of the overall analytics ecosystem?
- Is the new transformation general-purpose – applying to many structures – or so narrow that applies only to special cases?

Changes in transformations can yield new transformations with enhanced behavior that provide adaptivity and evolution of knowledge embodied in the system. A key criterion is to maintain consistency and system integrity as evolutionary changes are made to the system. A set of regressive tests should be employed to ensure that new system behavior does not radically change as the system evolves.

8. Implementation

The initial implementation of the foundational structures and transformations will use Medley Interlisp/Medley LOOPS (Kaisler 2023, Kaisler 2024a, Kaisler 2024b). The software architecture for the M&T System is depicted in Table 1.

Table 1. M&T System Components

M&T System	The Analytic System
Infrastructure	M&T System Foundations
Medley LOOPS	Medley Lisp Object-Oriented Program System (LOOPS)
Medley Interlisp	The Interlisp Ecosystem
maiko	The Medley Interlisp virtual machine, provides the access to the underlying hardware.

The lower three component layers run on multiple machines: Macintosh, Windows 10 or later, Linux/WSL, and Raspberry PI. An online version runs through standard browsers. Several features of Medley Interlisp will facilitate adaptive transformations, including Masterscope, Gauges, the Break system, CLISP, and the debugging system. Masterscope is an interactive tool that allows the user to determine what Interlisp/LOOPS entities use and are changed by other entities. A key feature of Medley Interlisp is that new programming languages can be constructed with extended functionality (Emanuelson and Haraldsson 1980).

Medley LOOPS is an object-oriented programming system integrated with Medley Interlisp that provides a comprehensive object-oriented infrastructure. A key feature of Medley LOOPS is the set of gauges which can be attached to variables, structures, and transformation that can display the current status of these objects, which will allow the user to observe the dynamic changes to the system (Kaisler 2004a, 2004b).

Readers may learn more about Medley Interlisp/Medley LOOPS at www.Interlisp.org.

8.1 Agenda

STRADS implemented an agenda as a simple model for the control of tasks to be performed (Oresky, Clarkson, Lenat, and Kaisler 1990). We believe an agenda which records tasks to be performed, is the appropriate mechanism for directing the actions of the system. Tasks are placed on the agenda by predicates and transformations to perform the activities suggested herein. As each task is removed from the agenda and interpreted, it can invoke predicates and cause rules to be executed which modify, assess, and evaluate the system. The agenda model will be sorted by the importance or worth of the task.

8.2 Control Structure

Automation of generation and evaluation of new structures and transformations requires a control structure that can operate semi-autonomously. As noted above, research will be focused on these aspects of the system, but must also focus on when user intervention is required to resolve operational issues. Examples include user evaluation of structures and transformations, providing names for new entities, values for new slots to improve the efficiency of the system, and defining new rules to enhance the functionality of the system.

The system will initially be developed as an interactive semi-automated system which provides the user with a set of commands and tools to implement the operations mentioned in the previous sections as well as provide information and guidance for use by the rules of the system. As the system evolves to more automatic operation, we anticipate that substantial functionality can be automated as experience is gained with the system.

9. Conclusion

This paper has addressed three essential RQs that, we believe, will yield an analytic system that could exhibit a self-adaptive capability that allows a system to evolve itself. Its major contributions are:

- A morphology concept that will allow description of different domain structures that can support different transformations for evolving domain knowledge (Authors 2024).
- A set of transformations associated with the different structures and yield new knowledge to enhance our understanding of a domain.
- A control structure and mechanism for automating the process of generating and evaluating new structures and transformations.

We believe that the proposed system design will enable the creation of an infrastructure for an adaptive system that will allow self-evolving application to be developed for different domains.

9.1 Future Work

This research suggests many possible avenues for further exploration:

- Given the expected complexity of the analytic system, a decentralized control structure implemented as concurrent tasks may improve the time-to-adapt to new environmental conditions;
- Incorporating cost effectiveness as an evaluation criteria for generating new structures and transformations;
- If the system's operating environment significantly changes, modification and generation of structures and transformations could radically change the behavior of the system. Determining when to adapt a system versus creating a new system is an open research question;
- Developing a search capability to external structures to determine if a structure is similar to an existing structure;
- Developing a spatial and temporal visualization capability for understanding system evolution;
- Extending the system to an open world model in which new concepts can be developed

We thank the reviewers for their cogent suggestions highlighting issues with clarity, theory, and citations.

10. References

Banerjee, U. 1994. *Loop Parallelization*, Springer-Verlag.

Bonfante, G., J-Y Marion, and D. Reynaud. 2009. "A Computational Perspective on Self-Modifying Programs", retrieved on June 9, 2024 from: https://inria.hal.science/inria-00433472..

Brown, M. and L. Palsberg. 2015. "Self representation in Girard's System U". *ACM SIGPLAN Notices*, *50*(1), 471-484.

Cai, H., Z. Shao, and A. Vaynberg. 2007. "Certified Self-Modifying Code", ACM SIGPLAN Notices, 42(6):66-77.

Dux, B. A. Iyer, S. Debray, D. Forester, and S. Kobourov. 2005. “Visualizing the Behavior of Dynamically Modifiable Code”, Proceedings of the 13th International Workshop on Program Comprehension (WPC’05), pp.337-340.

Emanuelson, P. and A. Haraldsson. 1980. “On Compiling Embedded Languages in Lisp”, Proceedings of the 1980 ACM Conference on Lisp and Functional Programming, New York, NY.

Giarratano, J. and G. Riley.1994. *Expert Systems: Principles and Programming*, PWS publishing Co., Boston, MA.

Greiner R. 1980. *RLL-1: A Representation Language*, HPP 80-9, Stanford Heuristic Programming Project, Stanford University, CA.

IARPA (Intelligence advanced Research Projects Agency). 2009. Securely Taking on New Software of Unknown Provenance (STONESOUP) Program, IARPA-BAA 09-08. Washington, DC.

Kaisler, S. 2024a. *Medley LOOPS: The Basic System*, available at www.Interlisp.org.

Kaisler, S. 2024b. *Medley LOOPS: Tools and Utilities*, available at www.Interlisp.org in Fall 2024.

Kaisler, S. 2023. *Software Archaeology: Medley Interlisp: Resurrection, Restoration, and Modernization*, presented to the Computer Conservation Society, November 16, 2023, London, England.

Kaisler, S. 1989. A Knowledge Based System for Geopolitical Analysis, *57th MORS Symposium*, FT Leavenworth, KS, June 6-8, 1989 (Selected Best Paper of Working Group 26).

Kaisler, S. “Designing Geopolitical Simulations Using Knowledge-Based Systems”, *World Congress on Expert Systems*, Orlando, FL, December 16-19, 1991.

Kaisler, S. and R. Modjeski. “Knowledge-Based Simulation for Geopolitical Analysis”, *30th Joint ORSA/TIMS Meeting*, Philadelphia, PA, Oct. 29-31, 1990.

Kaisler, S. 1997. *Making Concurrency Explicit: Converting Object-Oriented to Process Oriented Programs*, D.Sc. Dissertation, George Washington University. [available on Researchgate.com].

Kaisler, S., J.A. Espinosa, W.H. Money, and F.J. Armour. 2024. *Analytics Morphology and Transformation*, 57th Hawai’i International Conference on System Sciences, Honolulu, Oahu, HI.

Karsal, G. and J. Sztipanovits. 1999. “A Model-based Approach to Self-Adaptive Software”, IEEE Intelligent Systems, 14(3):46-53.

Kulkarni, D. and M. Stumm. 2007. Computational Alignment: A New, Unified Program Transformation for Local and Global Optimization, University of Toronto, TR CSRI-292.

Laws, A.G., A. Taleb-Bendiab, S. J. Wade, D. Reilly. 2001. “From Wetware to Software: A Cybernetic Perspective of Self-adaptive Software”. IWSAS 2001, pp. 257-280.

Lenat, D. 1983. “EURISKO: A Program that learns New Heuristics and Domain Concepts”, *Artificial Intelligence*, 21(2):61-98.

Lenat, D. B. and J. S. Brown. 1984. “Why AM and Eurisko appear to work”, .*Journal of Artificial Intelligence*, 23:269-294.

Lenat, D., F. Hayes-Roth, and P. Klahr. 1979. Cognitive Economy, RAND Corporation, Santa Monica, CA, N 1185-NSF.

Interlisp.org. 2024. Implementation and documentation of Medley Interlisp.

Oreizy, P., M.M. Gorlick, R.N. Taylor, D. Heimbigner et al. 1999. “An Architecture-based Approach to Self Adaptive Software”, IEEE Intelligent Systems, 14(3):54-62.

Oresky, C., A. Clarkson, D. B. Lenat , and S. Kaisler 1990. *Strategic Automated Discovery System (STRADS)*, published in Knowledge Based Simulation: Methodology and Application, ed. by P. Fishwick and D. Modjeski, Springer-Verlag, December, 1990

Randles, M., A. Taleb-Bendiab, P. Miseldine, and A. Laws. 2005. “Adjustable Deliberation of Self-Managing Systems”, *Proceedings of the* 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS’05).

Smith, J.E. 1998. “Self-Adaptation in Evolutionary Algorithms”, Ph.D. Dissertation, University of the West of England, Bristol, England.