System Development at Run Time

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

last changed 13:50 PDT, 08 September 2015 at nguyen

Abstract

Models are essential for defining and developing systems that support run-time decision-making and reconfiguration, and for implementing autonomous and adaptive systems for remote, hazardous, and largely unknown external environments. We show that they can also be used as the operational code throughout the development process, including deployment. Our ability to build systems with this property depends crucially on Computational Reflection, and our implementation thereof, an integration infrastructure for complex software-intensive systems called *Wrappings*.

It is inherent in a Wrapping system that all activity (down to a specified level of detail) can be recorded as sequences of events with associated context. The system can consider these event elements as points in a "behavior trajectory" space, and use recent advanced mathematical analysis methods to discover hidden relationships in the environment and system behaviors. These relationships can be used to improve the system models and therefore the corresponding behavior.

In this paper, we show that the Wrapping approach provides a powerful organizing principle for designing and building self-modeling systems. We also describe some dvanced mathematical methods that can be used by the system to construct models of its own behavior.

Key Phrases: Self-Modeling Systems, Computational Reflection, Wrapping Integration Infrastructure, Scenario-Based Engineering Process, Model Creation and Analysis, Active Control Loops, Advanced Mathematical Methods, Computational Semiotics

Contents

List of Figures

1 Introduction

Models are not only useful, but even essential, for defining, developing, and even operating systems for a complex operational environment, because they support run-time decision-making and reconfiguration. In this paper, we show how they can also be used as the operational code, in which the models are written early and refined throughout the development process, until they are deployed and afterwards. The system development process becomes the construction of a series of models that gradually conform to the original behavior expectations, and the result is a "self-modeling" system, in which the deployed code is the model of the deployed system, and interpreting that model is the behavior of the deployed system [45]. Similarly, further system development can occur at run time, with the system proposing changes or evaluating externally proposed changes.

Our ability to build systems with these properties depends crucially on Computational Reflection, and specifically on Wrappings [41] [38], an integration infrastructure for complex software-intensive systems. It was originally developed to support run-time decision-making and reconfiguration [42], as a way of implementing autonomous and adaptive systems for remote, hazardous, and even largely unknown external environments. This approach was also used to show how self-modeling systems can be built [45] [46].

This work is to be clearly distinguished from systems that use parallel code and models [20], since for us the models are the code, as interpreted to produce the intended behavior. These systems *are*, not just *use*, models at run time. There is also a reasonable expectation that the use of models need not be a performance problem, since partial evaluation methods [18] [19] can reduce all unchanging decisions to simple sequences (the partial evaluation methods have more information in a Wrapping-based system than in traditional software [41]), and many modern languages, such as Python and Java, are currently interpreted reasonably efficiently.

In this paper, we focus on a development process that we can use to build these systems, and also consider other ways for them to adapt themselves (i.e., their models of themselves and their behavior) to changing circumstances. We start with the Scenario-Based Engineering Process [51] [52], in which development begins with a collection of stakeholder expectations, embodied in a set of scenarios for the external environment and desired results for the behavior of the system. We write these as basic models of what happens outside and inside the system. As external constraints and interactions are better understood, these models are gradually changed from what should happen to how it should happen, refining functionality into more localized activity. We build these models as self-modeling systems using Wrappings, to provide a deep level of reflection, and we generally use *wrex* (our "Wrapping expression" notation [41]) to write the computational resources. The choice of *wrex* is a matter of convenience; other notations could be (and have been) used (e.g., Common Lisp, Python, C).

It is inherent in a Wrapping system that all activity (down to a level of granularity chosen via engineering judgment) can be recorded as sequences (or partially ordered sets in more complex concurrent applications) of events with associated context. We can have the system consider these event elements as points in a "behavior trajectory" space, and use recent advanced mathematical analysis methods to discover hidden relationships in and among the environment and system behaviors. These relationships can be used to improve the system models and therefore the corresponding behavior.

The rest of this paper begins, in Section 2, with some background history and a detailed overview of Wrappings, along with the Problem Posing Programming Paradigm [42] [41] [49]. These are the methods that allow us to study infrastructure [38] and build self-modeling systems [45] [46]. For us, a system is *self-aware* if it can use models of its own behavior, and it is *self-adaptive* if it can use those models to change that behavior. It is *self-modeling* if it also interprets these models of its own behavior to generate that behavior, and *self-developing* if it can use the models to further its own development (within general guidelines provided by developers).

It is clear that self-adaptive systems can make substantive changes at run-time. In one sense, this is already autonomous development. We extend this notion to the entire development cycle, using the Scenario-Based Engineering Process [51] [52] [36] [37] to build systems from stakeholder expectations in scenarios to requirements, and making models as soon as possible in the process. In Section 3, we describe how models can be provided or created, expanded and extended. In Section 4, we describe some of the many mathematical modeling and analysis methods that we think are applicable. In Section 5, we describe some of the many difficult challenges that remain. Finally, we describe our conclusions.

2 Wrappings

We provide a short description of Wrappings in this Section, since there are many other more detailed descriptions elsewhere [41], and especially the tutorials [47] [48], and a summary description is in the Appendix of this paper. The Wrapping integration infrastructure is our approach to run-time flexibility [38], with run-time context-aware decision processes and computational resources. It is defined by its two complementary aspects, the Wrapping Knowledge Bases (WKBs) and the Problem Managers (PMs). The WKBs contain Wrappings, which are Knowledge-Based interfaces to the uses of computational resources in context, and they are interpreted by the PMs, which are processes that are themselves resources.

We use the "Problem Posing" interpretation of programs [41]

to change our focus in programming these systems, separating code that computes something, called a "resource" from its purpose, called a "posed problem", and then keeping the problems available to the code along with the resources. 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 "methods" 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, and make context an essential part of reconnecting them, we can use very much more flexible mechanisms for connecting them than simply using the same name. We have shown that our choices lead to some interesting flexibilities, when combined with the "meta-reasoning" approach [8] [9] [10] including such properties as software reuse without source code modification, delaying language semantics to run-time, and system upgrades by incremental resource and infrastructure migration instead of version based replacement.

The WKBs define the entire set of problems that the system knows how to treat. The mappings are problem-, problem parameter-, and context-dependent, and identify the resources that can address each specific problem in a given context (this information is provided by the developers; it is not inferred by the system).

The PMs are the programs that read WKBs and select and apply resources to problems in context. The PMs are Wrapped in exactly the same way as other resources, and are therefore available for the same flexible integration as any resources. These systems have no privileged resource; anything can be replaced. Default Problem Managers are provided with any Wrapping implementation, but the defaults can be superseded in the same way as any other resource. These are the processes that replace the usual kind of implicit invocation [22], allowing arbitrary processes to be inserted in the middle of the resource invocation process. This flexibility does come with a cost, but there are also mechanisms based on partial evaluation [18] [41] [19] for removing any decisions that will be made the same way every time, thus leaving the costs where the variabilities need to be.

One of the keys to the flexibility of Wrappings is making these PM processes as important and as explicit as the WKB descriptions. The basic process notion is the interaction of one very simple loop, called the "Coordination Manager" (CM), and a very simple planner, called the "Study Manager" (SM). These are both examples of PMs.

The default CM is responsible for keeping the system going. It has only three repeated steps, after an initial one.

- FC = Find Context (establish a context for problem study.);
- loop:
  - PP = Pose Problem; (get a problem to study from a problem poser, who could be the user or the system);
  - SP = Study Problem (use an SM and the WKBs to study the posed problem in the current context);
  - AR = Assimilate Results (use the result to affect the current context).

It is therefore an activity loop of a sort that is common in autonomic computing and other self-adaptive system developments [9] [38]. Activity loops are not the focus of this paper, but they go a long way towards improving the flexibility of systems that use them.

We have divided the "Study Problem" process into a sequence of basic steps that we believe represent a fundamental part of problem study and resolution. These are implemented in the default SM:

- INT = Interpret Problem (find a resource to apply to the posed problem in the current context):
  - MAT = Match Resources (find a set of resources whose Wrappings say they might apply to the current problem in the current context);
  - RES = Resolve Resources (eliminate those that do not apply, via negotiations between the posed problem and each Wrapping of the matched resources to determine whether or not it can be applied, and make initial bindings of formal resource parameters to actual problem parameters);
  - SEL = Select Resource (choose which of the remaining candidate resources, if any, to use);
  - ADA = Adapt Resource (set it up for the current problem and problem context, by finishing all required bindings);
  - ADV = Advise Poser (tell the problem poser what is about to happen, that is, what resource was chosen and how it was set up to be applied);
- APP = Apply Resource (use the resource for its information service, to compute or present something, or provide some other information or service);
- ASR = Assess Results (determine whether the application succeeded or failed, and to help decide what to do next).

Finally, every step in the above sequences is actually a posed problem, and is treated in exactly the same way as any other, which makes these sequences "meta"-recursive [3]. This makes the system completely Computationally Reflective. That means that if we have any knowledge at all that a different planner may be more appropriate for the context and application at hand, we

can use it (after defining the appropriate context conditions), either to replace the default SM when it is applicable, or to replace individual steps of the SM, according to that context (which can be selected at run time).

3 Models

In this Section, we start with a discussion of many current approaches to using models at run time, and show how the Wrapping infrastructure, with its flexible selection and application of computational resources, supports the building, using, evaluating, and adapting models at run time. In a way this is cheating, since the Wrappings approach does not provide new methods of performing these functions. Rather, since it relegates all of the actual computational effort to the resources, its strength lies in the organization and interoperation of those resources, which in turn provides the flexibility to use any of the many mechanisms for specific kinds of model building and adapting (even using different mechanisms at different times in the same program).

We start with the models that are least like running code. It has long been recognized that a system with an explicit architecture model available at run time has access to more information about the running system, thus facilitating its management of its own adaptation [56] [57]. This is most useful when the model includes explicit representations of software components and connectors, or when it mimics the behavior of the system implementation [23] [1] [61] [54], so it can be compared to the runtime activity [14], using models of inferred behavior [2]. An interesting parallel set of studies has been ongoing in the business process modeling community, regarding workflow models as process models [71] [2], though they are typically producing external models of human centric processes.

However, it is also well-known that there are several challenges in using architectural models at run time (adapted from [56] [57] [23] [54]):

- Monitoring: how to select and collect necessary information from the system;
- Interpretation: how to process the event data;
- Resolution: how to determine changes;
- Adaptation: how to select and effect changes.

Monitoring is about how to select and collect necessary information from system internals, system behavior, detectable environment behavior, and interactions between system and environment to provide an adequate picture of the current behavior. Wrapping systems have an advantage of having a ready made language for events (the resource applications and context descriptions) that encompasses all system activity (to whatever level of detail has been selected for the designed variabilities in the system), and a built-in hook for measurements (the "Advise Poser" resources), already in the form of sequences of events.

Interpretation is about how to process the event data to make it usable. First, to convert event data into forms that support model building or analysis (this is complex event processing, with *a priori* event patterns or pattern discovery rules); then to build models from the event traces (this is called the *semantic gap* between low-level event traces and higher-level system concepts [61]), and finally, to evaluate model consistency. Here is where some of the advanced mathematical methods, such as Grammatical Inference and its generalizations, are used to build syntactic descriptions of the sequences, and either dimension reduction or manifold discovery to find behavioral manifolds that simplify the expressions. These mathematical subjects are generally beyond the scope of this paper [39], but there is a short summary in Section 4 below.

Resolution is about how to determine appropriate changes: how to specify and identify adaptation triggers, how to identify and resolve discrepancies between model and specification, how to specify resolution goals and policies, and how to decide what other data is needed to resolve a discrepancy. These are hard questions not always solvable *a priori*, but a Wrapping-based approach allows a system to contain many alternative analysis methods and compare their effects.

Adaptation is about how to select and effect changes: how to invent or select potential improvements, how to decide whether they are improvements, how to cause system changes, how to avoid thrashing (oscillations in adaptation usually due to fluctuations in environmental behavior). Once the replacement (or retuned) resources are available, changes in the Wrappings automatically make them selectable in the system.

One of the reasons that using architectural models is so difficult is that they are just scaffolding, not part of the system operation; they only define its structure that enables that operation. Devising the processes that can convert architecture changes into system changes at run time is the heart of making these systems effective. Here the Wrapping integration infrastructure allows any of the many methods currently in use to be applied and evaluated.

Our issue with many of these approaches is that they add adaptation to the system as an external feature, so only the combined system is partially self-adapting, and even then the adaptation mechanism itself is not usually subject to its own analysis and improvement [16] [61] [24] [17]. These approaches consider architectural models of the system as a control layer that has access to the components and connectors that define the system, and use effective control theory strategies for them. Some approaches [67] add another control layer to manage adapting the adaptation layer. Of course, this kind of add-on style is necessary when you start with an existing system, but it does leave a large part of the system non-adaptable.

Another approach is to organize the modeling using meta-models [11] [54] [50]. All of these approaches also seem to take the object system as a separate part, at the lowest level of abstraction, and include a varying number of distinct levels or layers of control:

- Original system, including source code, configuration files, reconfiguration policies (via automatic code and configuration file generation);
- Prescriptive part of the model (specifying how the system should behave);
- Descriptive part of the model (specifying how the system actually is by inferring models of its behavior).

Then the process infers a descriptive model of system behavior, using any of a number of methods, and compares the model to the prescription. Most of the approaches also have a separate layer for managing configuration variants, along with compatibility and transition rules. These are often written as a state machine for configurations (but only for systems with a small number of variations), with models of components and frameworks or configurations, a planner to construct configurations from conditions, and models of acceptable modifications. This is the layer that compares the description to the prescription. Finally, some of the approaches have a separate decision layer for mapping environmental behavior into configuration choices or conditions. For our purposes, the most important part of these descriptions is the causal connection [50], in the form of information flows between the system and its models. This looks like the closest to our self-modeling systems, though we make the causal connection the central feature (the model is the system).

In all of these approaches, there is the fundamental difficulty that the inferred models of behavior are not the way that behavior is generated, so they are always somewhat external to the system operation, and the interpretation step above is essential and difficult. Similarly, architectural models are not usually used to put the architectures together in the first place. They are more like structure or scaffolding, used until the system is ready to run and then discarded, or put in abeyance until something needs to be changed.

Some of the applications (mainly autonomic and self-adaptive systems [33] [67] [68], but also others) explicitly use activity loops (such as the MAPE loop of autonomic computing, the OODA loop in military strategic planning [66], the ELF = Elementary Loop of Functioning in intelligent system design [7] [53], and others) to organize their operation. For an activity loop based system, monitoring is automatically available in the step descriptions, and when the activity loop is recursive, as in Wrappings, the events are already expressed in system-relevant terms (resource application, relevant context). There are still the challenges of unravelling temporal overlaps (many higher-level events are interleaved), and the multiple differences in run time patterns [61], but using an activity loop greatly simplifies the interpretation process.

The control loop defined by the CM/SM in Wrappings is internally directed, looking only at internal problems and resources, unlike essentially every other loop, which seems to be directed externally. These other activity loops seem to be designed to perform the specified actions, not decide on the actions to be performed by other resources. If they were to be treated recursively, and separated from the performing resources, they could have the same flexibility as Wrappings.

Our conclusion from this discussion is that many of these approaches would likely benefit from using an explicit activity loop for their control process, separating their methods of adaptation and evaluation from the control thereof, and adding many more methods for construction, comparison and evaluation of models (of course, some of these approaches already do some of these things).

4 Mathematical Methods

There are several advanced mathematical analysis methods that are mentioned in this paper. In this Section, we describe them very briefly.

1. Robust Statistics;
2. Grammatical Inference and Event Pattern Correlation;
3. Fractal Dimension;
4. Dimension Reduction and Manifold Discovery;
5. Topological Data Analysis.

Some are old and have been revitalized, and some are very new. Other mathods can also be sued, but this is a representative set.

4.1 Robust Statistics

These are methods for analyzing non-traditional distributions (i.e., not *Gaussian*). The reason this matters is that most statistical or statistically based techniques assume that a random distribution is *Gaussian* (also called *normal* distributions), and those methods have been widely successful. However, for reasons we think follow from the complexity of modern applications, those assumptions are no longer adequate.

Data can vary widely in its quality: its noisiness, its errors, its missing values, etc.. A new movement within data analytics is combining decades-old **robust statistical methods** (from the 1970's and 80's [69] [29]) with new data mining and analysis methods [4]. **Robust statistics** have good performance for data drawn from a wide range of probability distributions, especially

for distributions that are not normally distributed or other departures from the model assumptions underlying classical statistical analyses. Especially, **robust statistics** provide methods that are less impacted by statistical *outliers*, and measures for how much a given statistic is impacted by outliers or other unusual characteristics of the data set distribution.

4.2 Grammatical Inference and Event Pattern Correlation

This is a collection of event pattern detection and identification methods for learning internal hierarchical structure of sequences and sequential patterns of events. Grammatical Inference usually concerns itself only with sequences, but we know that in more compex concurrent environments, activities will overlap and the right event set model is partially ordered sets. Event pattern correlation methods are for comparing different event patterns (temporally ordered sequences are a particularly simple form of event pattern). The move to partially ordered sets makes the identification processes much harder.

Grammatical Inference methods are currently used to automatically detect recurring patterns in data sequences, which includes mathematical methods for the discovery of sets of rules that define the internal structure of recurring patterns. These analyses can be used to find repetitive activities, recurring patterns of activity, and overlapped sequences of activities, whether or not the time intervals are the same.

It is relatively straightforward to imagine detecting repeated instances of the same action, and even repeated sequences of related actions. It is harder to allow both sequences and alternatives (e.g., these two sequences are the same, except that the fourth step is one of these two possibilities), and a little harder yet to identify iterations (e.g., these sequences are the same except that step two is repeated a variable number of times).

More complicated structures have another phenomenon called recursion, which is much harder to recognize, but still possible. The recognition of recursion depends on effective noticing of repeated structures that may not be easily seen as sequences or alternations. Algorithms for this process are still under development [28].

When the event sets are less definite, or are known to potentially contain errors, then *probabilistic* grammatical inference can be used [27], with corresponding increases in the time required to identify regularities.

Any example we have of a contingent set of dependent events can be modeled as a partially ordered set. These complex sets of events cannot be generated by grammars for string languages, so new methods are needed. There have been some excellent beginning approaches to this problem [5] [6], but no definitive results as yet.

4.3 Fractal Dimension

These are methods for measuring intrinsic complexity in complex phenomena, particularly dynamic sets (sequences of values presumed to be generated by some dynamic process, either *discrete*, with some potentially varying time interval, or *continuous* and therefore *sampled*, also with some potentially varying sample interval).

The **fractal dimension** of a set is an unusual characteristic that is hard to spoof, used for detecting some kinds of obfuscation and unnecessary associated data. It is an intrinsic property of the set. It is not dependent on the set's embedding into Euclidean space, provided the space has enough (ordinary) dimensions.

The *tuple map* takes any sequence of points and produces a new sequence, whose elements are consecutive m-tuples from the original sequence for some (usually fairly small) integer m. What is illustrated in [58], discussed also in the Theiler papers [64] [65], is that if the original sequence defines the dynamics of a chaotic system for which an *attractor* has **fractal dimension** d, then if $m > 2 * d + 1$, the tuple sequence almost always has the same dynamics, and if $m > d$, then it almost always has the same **fractal dimension**. This property is why we use the method.

4.4 Dimension Reduction and Manifold Discovery

These are methods from computational data analysis for reducing difficult computations from thousands of dimensions to a few. The techniques reduce the number of dimensions while preserving essential structures. There are different computational methods that differ primarily on what aspect of the structure is deemed to be essential. The hard part in any application is to pick the most effective essential structure for the application question at hand. In addition, all of these methods are iterative, with weak stopping rules (there is no simple way to tell if another iteration will improve the result enough to be worth the time and effort). These are among several performance issues here that still need research.

A good survey of this entire field can be found in [12].

Another intriguing and surprising possibility is called *random projection*, in which the points in a very high dimensional space are mapped to a medium dimensional space (e.g., millions to hundreds), using random linear projections [13]. This work stems from a result of [31], who showed that up to N points in an N-dimensional space can almost always be projected into a space of dimension $C \times \log N$ for some constant C, when N is large, with control on the ratio of distances and the error (also called *distortion*). For large N, this can be a great reduction (and fewer points can often be mapped with even less distor-

tion).

Manifold Discovery is a particular class of dimension reduction methods, also called "Manifold Learning", that applies computational differential geometry to discover structure in high dimensional data.

Knowing that a set of points is in a manifold (or is nearly in a manifold because of noise) allows us to perform many operations not available for general point sets. We can compute boundaries and curvatures, which allows us to imagine *trajectories* within the set of points and also allows us to "unroll" the manifold into a simpler space for further analysis. In this context, a trajectory is any sequence of points that may represent a path or a motion. It is a useful generalization of the usual notion of a vehicle trajectory. This is an intuitive description of straightening out the trajectories, and bending the space to accommodate. For example, if we take a circle on a sphere and straighten it out, we get a straight line with a common point at both ends, called the *point at infinity*, and if we unroll the sphere to match, we get a plane with a common point in each direction, also called the *point at infinity*, since it is the same point. This map does not preserve distances (the Mercator map projection is a similar example, mapping a sphere to a cylinder), but it does preserve some other properties.

There are many methods in common use, including ISOMAP [63] and LLE [59] [60].

4.5 Topological Data Analysis

These methods can be used to detect persistent structure in complex phenomena. These are the newest methods we have studied. *Topological* methods [25] [26] have some promise for a completely different way to view the *multi-scale behavior*, that is, the behavior at many time- and space-scales, of complex phenomena, and perhaps even efficient computational methods to do so. They seem to compute a kind of "topological signature" that is invariant under many kinds of changes, and reflects the structure at all scales simultaneously. The algorithms have recently been greatly improved [72] [73], but they are still time-consuming and use difficult data structures.

5 Challenges

Many difficult challenges remain, especially in the basic event identification and processing (the interpretation step above).

We have described a development methodology to build self-adaptation into systems from the beginning, but of course, that is the easy case, since we have control over the entire process. The more interesting and difficult case is to add new adaptation capabilities to an existing system. There are difficulties in choosing instrumentation points, and in usefully inserting them without disrupting their operation, as mentioned above in Section 3. The existing methods seem to be most effective when they add a control layer or two onto an existing program or system.

However, one can also use our approach to this issue, which we describe as "reuse without modification". [47] [48] Using Wrappings, and at the cost of writing a compiler for the source code language(s) (which is far less now than it used to be), we can use the old code without changing at at all, inserting function call diversions into the generated code using Wrappings. Then we can add whatever adaptations are useful, so the program has them available at run time.

There are also some mathematical challenges in applying the advanced techniques to and in these systems. We need better algorithms for event pattern correlations for partially ordered sets, since the ones that exist are too weak or too time consuming. These will be used to create structural models of actual behavior. Other mathematical methods from geometry and topology also need to be examined, such as homological algebra and persistence methods.

As a practical matter, we also need to investigate some simplifications of advanced mathematical methods, especially the manifold discovery and other geometric methods, to determine how much we can do in a useful amount of time. We also need better methods for handling non-stationary environments, and measurements of how effectively different algorithms can keep up with changes.

This isn't nearly the full list of difficulties we have in implementing and improving these systems, and especially their ability to improve themselves, but it will do for a starting point.

However, perhaps the most difficult is in the realm of Computational Semiotics. The "Get Stuck" Theorems [40] provide fundamental limits on the capabilities of a system with a fixed set of representational mechanisms. We must therefore consider three difficult questions:

- What are symbols? [15]
- What are objects? [62]
- What are events?

There are non-trivial processes involved in symbol identification, symbol system assessment, and even symbol system revision (which is the analog of "refactoring" in object-oriented systems [21] [34] [55]).

These Semiotic issues have been addressed in other work [43] [44], but are not yet solved.

6 Conclusions

We have described some important issues for systems that will undergo (at least) some part of their system development at run time, primarily because the run-time environment is not well enough known at design time to decide certain optimization or implementation questions.

We have also described an approach (Wrappings) that is known to support defining and building such systems with adequate flexibility and available information at run time. We have explained how the Wrapping integration infrastructure provides the required flexibility, through its separation of control processes from computational resources, and the Computational Reflection that ties them back together. In this development approach, system models exist from very early in development time, and together with the environment and scenario models, system behavior can be examined and improved by the developers until it is deployed, and to some extent by the system itself afterward.

Self-modeling (and especially self-developing) systems must have many mechanisms for modeling, model comparison, and model adjustment, that allow the systems to manage their own behavior, behavioral evaluations and changes, and the suite of varyingly detailed models that are necessary.

The point of this paper is not so much to describe specific modeling techniques (inference for observation models, optimization adjustments, consistency and discrepancy analyses, simulation and hypothesis testing, and other applications of the advanced mathematical methods) that can be used to build and evaluate models of behavior as it is to show that using Wrappings helps a system organize its modeling processes and coordinate its experimentation and evaluation of multiple methods and models in an extremely flexible way.

References

[1] W.M.P. van der Aalst, A.K. Alves de Medeiros, A.J.M.M. Weijters, "Genetic Process Mining", p.48-69 in Gianfranco Ciardo, Philippe Darondeau (Eds.), *Proc. ICATPN 2005: 26th Intern. Conf. on Applications and Theory of Petri Nets*, 20-25 Jun 2005, Miami, Florida (2005)

[2] W.M.P. van der Aalst, M. Pesic, H. Schoenberg, "Declarative workflows: Balancing between flexibility and support", Computer Science - Research and Development, Vol. 23, Issue 2, p. 99-113 (10 Mar 2009)

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

[4] Fatemah Alqallaf, Kjell Konis, R. Douglas Martin, and Ruben Zamar, "Scalable Robust Covariance and Correlation Estimates for Data Mining", *Proc. SIGKDD 02*, Edmonton, Alberta, Canada (2002)

[5] Dana Angluin, "Finding Patterns Common to a Set of Strings", J. Computer and System Sciences, v.21, p.46-62 (1980)

[6] Roger S. Barga, Hillary Caituiro-Monge, "Event Correlation and Pattern Detection in CEDR", *Proc. Middleware 2005* (2005)

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

[8] Dr. Kirstie L. Bellman, Dr. Christopher Landauer, Dr. Phyllis R. Nelson, "System Engineering for Organic Computing", Chapter 3, pp.25-80 in Rolf P. Würtz (ed.), *Organic Computing*, Understanding Complex Systems Series, Springer (2008)

[9] Dr. Kirstie L. Bellman, Dr. Christopher Landauer, Dr. Phyllis R. Nelson, "Managing Variable and Cooperative Time Behavior", *Proc. SORT 2010: The First IEEE Workshop on Self-Organizing Real-Time Systems*, 05 May 2010, part of *ISORC 2010*, 05-06 May 2010, Carmona, Spain (2010)

[10] Dr. Kirstie L. Bellman, Dr. Phyllis R. Nelson, "Developing Mechanisms for Determining G̈ood Enoughïn SORT Systems", *Proc. SORT 2011: The Second IEEE Workshop on Self-Organizing Real-Time Systems*, 31 Mar 2011, part of *ISORC 2011*, 28-31 Mar 2011, Newport Beach, California (2011)

[11] Nelly Bencomo, Paul Grace, Carlos Flores, Danny Hughes, Gordon Blair, "Genie: Supporting the Model-Driven Development of Reflective, Component-Based Adaptive Systems", p.811-814 in *Proc. ICSE08: The 30th Intern. Conf. on Software Engineering*, 10-18 May 2008, Leipzig, Germany (2008)

[12] Christopher J. C. Burges, *Dimension Reduction: A Guided Tour*, Now Publishers Inc (2010)

[13] Emmanuel J. Candes and Terence Tao, "Near-Optimal Signal Recovery From Random Projections: Universal Encoding Strategies?", IEEE Trans. Inform. Theory, Vol.52, No.12, p.5406-5425 (December 2006)

[14] Paulo Casanova, David Garlan, Bradley Schmerl, and Rui Abreu, "Diagnosing architectural run-time failures", *Proc.SEAMS'13: The 8th Intern. Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 20-21 May 2013, San Francisco, California (2013)

[15] Daniel Chandler, "Semiotics for Beginners", on the Web at URL `http://visual-memory.co.uk/daniel/Documents/S4B` (last checked 19 Jul 2015)

[16] Shang-Wen Cheng, David Garlan, Bradley Schmerl, "Making Self-Adaptation an Engineering Reality", in Ozalp Babaoglu, Mrk Jelasity, Alberto Montresor, Christof Fetzer, Stefano Leonardi, Aad van Moorsel, and Maarten van Steen (eds.), *Self-Star Properties in Complex Information Systems*, LNCS 3460, Springer (2005)

[17] Shang-Wen Cheng and David Garlan, "Stitch: A Language for Architecture-Based Self-Adaptation", in Danny Weyns, Jesper Andersson, Sam Malek and Bradley Schmerl (eds.), Special Issue on State of the Art in Self-Adaptive Systems, *J. Systems and Software*, Vol.85, No.12 (Dec 2012)

[18] C. Consel, O. Danvy, "Tutorial Notes on Partial Evaluation", *Proc. PoPL 1993: The 20th ACM Symposium on Principles of Programming Languages*, Charleston, SC (Jan 1993)

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

[20] Mahdi Derakhshanmanesh, Jürgen Ebert, Thomas Iguchi, and Gregor Engels, "Model-Integrating Software Components", p.386-402 in Juergen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahão, and Emilio Insfran, *Model-Driven Engineering Languages and Systems*, LNCS 8767, Springer (2014)

[21] Martin Fowler, *Refactoring*, Addison-Wesley (08 July 1999)

[22] D. Garlan, S. Jha, D. Notkin, J. Dingel, "Reasoning About Implicit Invocation", pp.209-221 in *Proc. SIGSOFT'98/FSE-6: The 6th ACM SIGSOFT Intern. Symposium on Foundations of Software Engineering*, 03-05 Nov 1998, Lake Buena Vista, Florida (1998); also *SIGSOFT Software Engineering Notes*, vol.23, no.6, pp.209-221, ACM (1998)

[23] David Garlan and Bradley Schmerl, "Using Architectural Models at Runtime: Research Challenges", *Proc. First European Workshop on Software Architectures*, 21-22 May 2004, St. Andrews, Scotland, p.200-205 in Flavio Oquendo, Brian Warboys, Ron Morrison (eds.), *Software Architecture*, LNCS 3047, Springer (2004)

[24] David Garlan and Bradley Schmerl and Shang-Wen Cheng, "Software Architecture-Based Self-Adaptation", Chapter 1 of Mieso Denko, Laurence Yang and Yan Zhang (eds.), *Autonomic Computing and Networking*, Springer (2009)

[25] Robert Ghrist, "Three Examples of Applied and Computational Homology", Nieuw Archief voor Wiskunde 5/9 no.2 (June 2008)

[26] Robert Ghrist, *Elementary Applied Topology*, CreateSpace Independent Publishing Platform (01 September 2014)

[27] Mattias Gybels, Francois Denis, and Amaury Habrard "Some improvements of the spectral learning approach for probabilistic grammatical inference", JMLR: Workshop and Conference Proceedings v.34, p.64-78 (2014)

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

[29] Peter J. Huber, *Robust Statistics*, Wiley (2009)

[30] R. John M. Hughes, "Lazy Memo Functions", p.129-146 in *Proc. Conf. Functional Programming Languages and Computer Architecture*, 16-19 Sep 1985, Nancy, France (1985); LNCS 201, Springer Verlag (1985)

[31] W. B. Johnson and J. Lindenstrauss, "Extensions of Lipshitz mapping into Hilbert space", Contemporary Mathematics, vol.26, p.189-206 (1984)

[32] Holger Kantz, *Nonlinear Time Series Analysis*, Cambridge U. Press (2003)

[33] Jeffrey O. Kephart, David M. Chase, "The Vision of Autonomic Computing", IEEE Computer, Vol.36, Issue 1, p.41-50 (Jan 2003)

[34] Joshua Kerievsky, *Refactoring to Patterns*, Addison-Wesley (15 Aug 2004)

[35] Janet Kolodner, *Case-Based Reasoning*, Morgan Kaufmann (1993)

[36] Christopher Landauer, "Systems Engineering of Software", *Proc. CSER 2011*: *The 9th INCOSE Conf. on Systems Engineering Research*, 15-16 Apr 2011, Crown Plaza, Redondo Beach, California (2011)

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

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

[39] Christopher Landauer, "Advanced Mathematical Methods for Telemetry Analysis" (presentation), *2015 Spacecraft Flight Software Workshop*, 27-29 October 2015, JHU/APL, Laurel, Maryland (2015)

[40] Christopher Landauer, Kirstie L. Bellman, "Situation Assessment via Computational Semiotics", pp. 712-717 in *Proc. ISAS'98: The 1998 Intern. MultiDisciplinary Conf. on Intelligent Systems and Semiotics*, 14-17 Sep 1998, NIST, Gaithersburg, Maryland (1998)

[41] 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)

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

[43] Christopher Landauer, Kirstie L. Bellman, "Some Measurable Characteristics of Intelligence", Paper WP 1.7.5, *Proc. SMC'2000: The 2000 IEEE Intern. Conf. on Systems, Man, and Cybernetics (CD)*, 08-11 Oct 2000, Nashville, Tennessee (2000)

[44] Christopher Landauer, Kirstie L. Bellman, "Refactored Characteristics of Intelligent Computing Systems", *Proc. PERMIS'2002: Measuring of Performance and Intelligence of Intelligent Systems*, 13-15 Aug, NIST, Gaithersburg, Maryland (2002)

[45] Christopher Landauer, Kirstie L. Bellman, "Self-Modeling Systems", p. 238-256 in R. Laddaga, H. Shrobe (eds.), "Self-Adaptive Software", LNCS 2614, Springer (2002)

[46] Christopher Landauer, Kirstie L. Bellman, "Managing Self-Modeling Systems", in R. Laddaga, H. Shrobe (eds.), *Proc. Third Intern. Workshop on Self-Adaptive Software*, 09-11 Jun 2003, Arlington, Virginia (2003)

[47] Dr. Christopher Landauer, Dr. Kirstie L. Bellman, Dr. Phyllis R. Nelson, "Wrapping Tutorial: How to Build Self-Modeling Systems", *Proc. SASO 2012: The 6th IEEE Intern. Conf. Self-Adaptive and Self-Organizing Systems*, 10-14 Oct 2012, Lyon, France (2012)

[48] Dr. Christopher Landauer, Dr. Kirstie L. Bellman, Dr. Phyllis R. Nelson, "Wrapping Tutorial: How to Build Self-Modeling Systems", *Proc. CogSIMA 2013: 2013 IEEE Intern. Inter-Disciplinary Conf. Cognitive Methods for Situation Awareness and Decision Support*, 25-28 Feb 2013, San Diego, California (2013)

[49] Christopher Landauer, Kirstie Bellman, "Designing Cooperating Self-Improving Systems", *Proc. 2105 SISSY Workshop: Self-improving Systems of Systems*; 07 Jul 2015, Grenoble, France part of *ICAC 2015: the 2015 Intern. Conf. on Autonomic Computing*, 07-10 Jul 2015, Grenoble, France (2015)

[50] Grzegorz Lehmann, Marco Blumendorf, Frank Trollmann, Sahin Albayrak, "Meta-Modeling Runtime Models", p.209-223 in Juergen Dingel, Arnor Solberg (eds.), *Proc. MODELS'10: the 2010 Intern. Conf. on Models in Software Engineering*, 02-08 Oct 2010, Oslo, Norway (03 Oct 2010)

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

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

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

[54] Brice Morin, Olivier Barais, Jean-Marc Jézéquel, Franck Fleurey, Arnor Solberg, "Models@Run.time to support dynamic adaptation", IEEE Computer, p.44-51 (Oct 2009)

[55] Emerson Murphy-Hill and Andrew P. Black, "Breaking the Barriers to Successful Refactoring: Observations and Tools for Extract Method", *Proc. ICSE'08: Intern. Conf. on Software Engineering*, 10-18 May 2008, Leipzig, Germany (2008)

[56] P. Oreizy, N. Medvidovic, R. Taylor, "Architecture-Based Runtime Software Evolution", *Proc. ICSE 1998: Intern. Conf. Software Engineering*, 19-25 Apr 1998, Kyoto, Japan (1998)

[57] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf, "An Architecture-Based Approach to Self-Adaptive Software", *IEEE Intelligent Systems*, p.54-62 (May 1999)

[58] Norman H. Packard, James P. Crutchfield, J. Doyne Farmer, and Robert S. Shaw, "Geometry from a Time Series", Phys.Rev.Lett. Vol.45, No.9, p.712-716 (1 September 1980)

[59] Sam T. Roweis and Lawrence K. Saul, "Nonlinear Dimensionality Reduction by Locally Linear Embedding", Science, v.290, p.2323-2326 (22 December 2000)

[60] Lawrence K. Saul and Sam T. Roweis, "Think Globally, Fit Locally: Unsupervised Learning of Low Dimensional

Manifolds", J. of Machine Learning Research, v.4, p.119-155 (2003)

[61] Bradley Schmerl, Jonathan Aldrich, David Garlan, Rick Kazman, and Hong Yan, "Discovering Architectures from Running Systems", IEEE Trans. Software Engineering, Vol.32, No.7, p.454-466 (Jul 2006)

[62] Brian Cantwell Smith, *On the Origin of Objects*, MIT (1996)

[63] Joshua B. Tenenbaum, Vin de Silva, John C. Langford, "A Global Geometric Framework for Nonlinear Dimensionality Reduction", Science, v.290, p.2319-2323 (22 December 2000)

[64] James Theiler, "Estimating Fractal Dimension", J.Opt.Soc.Am.A, Vol.7, No.6 (June 1990)

[65] J. Theiler, "Estimating the Fractal Dimension of Chaotic Time Series", The Lincoln Laboratory Journal, Vol.3, No.1 (1990)

[66] David G. Ullman, "OO-OO-OO" the Sound of a Broken OODA Loop, *Crosstalk* (April 2007); `http://www.crosstalkonline.org/storage/issue-archives/2007/200704/200704-0-Issue.pdf` (availability last checked 08 September 2015)

[67] Thomas Vogel and Holger Giese, "A Language for Feedback Loops in Self-Adaptive Systems: Executable Runtime Megamodels", In *Proc. SEAMS 2012: the 7th Intern. Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 04-05 Jun 2012, Zurich, Switzerland, p.129-138 (June 2012)

[68] Thomas Vogel and Holger Giese, "Model-Driven Engineering of Self-Adaptive Software with EUREMA", ACM Trans. Autonomous and Adaptive Systems, vol.8, no.4, p.18:1-18:33 (Jan 2014)

[69] L. Wasserman, *All of Nonparametric Statistics* Springer Texts in Statistics, Springer (2007)

[70] Andreas S. Weigend, and Neil Gershenfeld, *Time Series Prediction: Forecasting The Future And Understanding The Past*, Santa Fe Institute (1993)

[71] Workflow Management Coalition, *Workflow Standard: Process Definition Interface; XML Process Definition Language*, Document Number WFMC-TC-1025, 03 Oct 2005 (2005)

[72] Afra Zomorodian, "The Tidy Set: A Minimal Simplicial Set for Computing Homology of Clique Complexes", p.257-266 in Afra Zomorodian (ed.), *Proceedings SoCG10: 26th Symposium on Computational Geometry*, 13-16 June 2010, Snowbird, Utah (2010)

[73] Afra Zomorodian, *Topological Data Analysis: Proceedings of 2011 AMS Short Course on Computational Topology*, 04-05 January 2011, New Orleans, Louisiana, Symposia in Applied Mathematics, v.70, AMS (2012)

7 Appendix: Wrapping Description Details

We provide a combined description of Wrappings in this Appendix Section, as derived from many other more detailed descriptions elsewhere [41], and especially the tutorials [47] [48]. The Wrapping integration infrastructure is our approach to run-time flexibility [38], with its run-time context-aware decision processes and computational resources. The basic idea is that Wrappings are Knowledge-Based interfaces to the uses of computational resources in context, and they are interpreted by processes that are themselves resources.

Systems built with Wrappings can be very flexible in their interconnections [42], since different contexts can produce different connection networks (both components and interactions), with different sets of resources selected and applied, and these decisions can all be made at run-time.

7.1 Problem Posing Interpretation

The "Problem Posing" interpretation of programs [41] is based on an important change of attitude in system design and implementation. It extends the "what from how" separation of interface from implementation to a "why from what" separation of interfaces from intended purposes.

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 "methods" 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, and make context an essential part of connecting them, we can use very much more flexible mechanisms for connecting them than simply using the same name.

7.2 Wrapping Overview

Many styles of mapping from problems to resources exist, often using a mechanism called implicit invocation [22]. Our reason for not using that process is exactly the implicity of the mapping process. We want to be able to replace the mapping process at any time, to intercept the invocation with user-defined processes.

We have chosen in Wrappings to use a knowledge base that defines maps from problems in context to resource applications, and shown that this choice leads to some interesting flexibilities, when combined with the "meta-reasoning" approach of Wrappings [8] [9] [10] including such properties as software reuse without source code modification, delaying language semantics to run-time, and system upgrades by incremental migration instead of version based replacement.

The Wrapping integration infrastructure is defined by its two complementary aspects, the Wrapping Knowledge Bases and the Problem Managers.

The Wrapping Knowledge Bases (WKBs) contain the Wrappings that map problems to resources in context. They define the entire set of problems that the system knows how to treat (there are usually also default problems that catch the ones otherwise not recognized). The mappings are problem-, problem parameter-, and context-dependent.

The Problem Managers (PMs) are the programs that read WKBs and select and apply resources to problems in context. We get Computational Reflection because they are also resources, and are Wrapped in exactly the same way as other resources, and are therefore available for the same flexible integration as any resources. These systems have no privileged resource; anything can be replaced. Default Problem Managers are provided with any Wrapping implementation, but the defaults can be superseded in the same way as any other resource. These are the processes that replace the usual kind of implicit invocation [22], allowing arbitrary processes to be inserted in the middle of the resource invocation process. This choice leads to very flexible systems; more details can be found in our references previously cited, especially the tutorials.

Five essential properties underlie the simplicity and power of Wrappings. They are related as shown in Figure 1.

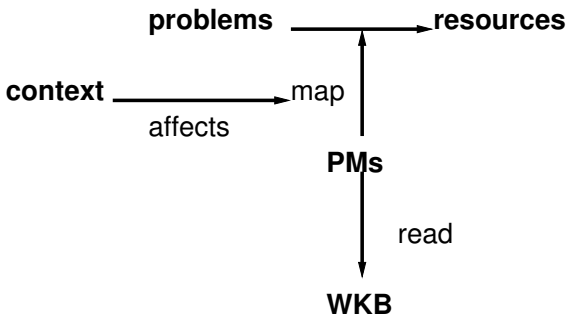

Figure 1: Wrapping Aspects

1. ALL parts of a system, at all levels of detail, are *resources* that provide some kind of *information service* or *computation service*. Everything that does anything is a resource.

2. ALL activities in the system are *problem study*, that is, all activities *apply* a resource to a *posed problem* in a *problem context*). Posed *problems* are computation or information service requests.

3. ALL maintenance of relevant system state is done with **context**. The invocation environment provides the initial context, and system operation updates the dynamic context from internal and external sources (as part of various resource applications).

4. *Wrapping Knowledge Bases* (or WKBs) contain *Wrappings*, which are explicit machine-interpretable descriptions of all of the ways resources can be applied to problems in contexts that are relevant to the system. ALL information connecting posed problems to applicable resources is maintained in WKBs, which define the mapping in a context- and problem-parameter-dependent way. The Wrappings are generally defined by developers and provided with the resources. The Wrappings provide what we have called the *Intelligent User Support* (IUS) functions [46]:

   - *Discovery* (which new resources can be inserted into the system for this problem),
   - *Selection* (which resources to apply to a problem),
   - *Assembly* (how to let them work together),
   - *Integration* (when and why they should work together),
   - *Adaptation* (how to adjust them to work on the problem),
   - *Explanation* (why certain resources were or will be used), and
   - *Evaluation* (what is the impact or effect of this use of this resource).

   Wrappings therefore contain much more than "how" to use a resource, as many computing libraries do. They also provide information to help decide "when" it is appropriate, "why" it might be the right one for the problem, and "whether" it can be used in this current problem and context.

5. *Problem Managers (PMs)*, including the *Study Managers (SMs)* and the *Coordination Managers (CMs)*, are algorithms that interpret the Wrapping descriptions to collect and select resources and apply them to problems. ALL

interpretation and performance activities are managed by PMs, which are themselves also resources, and are therefore also Wrapped and selectable, just like any other resource.

Thus, a system built with Wrappings uses what we have called *Knowledge-Based Polymorphism* to connect each problems in context to appropriate resources. It is therefore an example of the Problem Posing Paradigm.

A Wrapping is not simply a coded interface "to" a resource, the way the usual "wrappers" are; it is a conceptual interface to the "use" of a resource, for a particular problem in a particular context (the selection of the appropriate resource to apply for a given problem in a given context is a kind of case-based reasoning [35]). This information is used to generate the appropriate invocation wrapper interfaces on the fly. We Wrap "uses" of resources instead of resources in and of themselves, since many analysis tools have grown by accretion over the years, and common ways to use them have developed their own style.

This non-correspondence between problems and resources is one of the important normalizing features of the Wrapping approach, since it allows the uses of resources to be much more simply described than trying to describe the entire resource at once.

This allows us, for example, to map a series of posed problems representing a prospective analysis into a form suitable for execution on some computer in the actual system, and into a form suitable for use in a simulation. The different contexts mean we can take the same code (expressed as "problem"s in a nested structure) and map to different resources.

7.3 Wrapping Processes

One of the keys to the flexibility of Wrappings is making the processes as important and as explicit as the descriptions. The basic notion is the interaction of one very simple loop, called the "Coordination Manager", and a very simple planner, called the "Study Manager".

The default Coordination Manager (CM) is responsible for keeping the system going. It has only three repeated steps, after an initial FC = Find Context step:

- PP = Pose Problem,
- SP = Study Problem,
- AR = Assimilate Results

To "Find Context" means to establish a context for problem study, possibly by requesting a selection from a user, but more often getting it explicitly or implicitly from the system invocation. It is our placeholder for conversions from that part of the system's invocation environment that is necessary for the system to represent to whatever internal context structures are used by the system. To "Pose Problem" means to get a problem to study from the problem poser (a user or the system), which includes a problem name and some problem data, and to convert it into whatever kind of problem structure is used by the system (we expect this is mainly by parsing of some kind). To "Study Problem" means to use an SM and the wrappings to study the given problem in the given context, and to "Assimilate Results" means to use the result to affect the current context, which may mean to tell the poser what happened. Each step is a problem posed to the system by the CM, which then uses the default SM to manage the system's response to the problem. The first problem, "Find Context", is posed by the CM in the initial context of "no context yet", or in some default context determined by the invocation style of the program.

The main purpose of the default CM is cycling through the other three problems, which are posed by the CM in the context found by the first step. This way of providing context and tasking for the SM is familiar from many interactive programming environments: the "Find context" part is usually left implicit, and the rest is exactly analogous to LISP's "read-eval-print" loop, though with very different processing at each step, mediated by one of the SMs. In this sense, this CM is a kind of "heartbeat" that keeps the system moving. It is therefore an activity loop of a sort that is common in autonomic computing and other self-adaptive system developments [38].

If the Coordination Manager is the basic cyclic program heartbeat, then the Study Manager is a planner that organizes the resource applications.

We have divided the "Study Problem" process into three main steps: "Interpret Problem", which means to find a resource to apply to the problem; "Apply Resource", which means to apply the resource to the problem in the current context; and "Assess Results", which means to evaluate the result of applying the resource, and possibly posing new problems. We further subdivide problem interpretation into five steps, which organize it into a sequence of basic steps that we believe represent a fundamental part of problem study and solution. These are implemented in the default Study Manager (SM):

- INT = Interpret Problem:
  - MAT = Match Resources,
  - RES = Resolve Resources,
  - SEL = Select Resource,
  - ADA = Adapt Resource,
  - ADV = Advise Poser,
- APP = Apply Resource, and
- ASR = Assess Results.

To "Match Resources" is to find a set of resources that might apply to the current problem in the current context. It is intended to allow a superficial first pass through a possibly large collection of Wrapping Knowledge Bases. To "Resolve Resources" is to eliminate those that do not apply. It is intended to allow negotiations between the posed problem and each wrapping of the resource to determine whether or not it can be applied, and make some initial bindings of formal parameters of resources that still apply. To "Select Resource" is simply to make a choice of which of the remaining candidate resources (if any) to use. To "Adapt Resource" is to set it up for the current problem and problem context, including finishing all required bindings. To "Advise Poser" is to tell the problem poser (who could be a user or another part of the system) what is about to happen, i.e., what resource was chosen and how it was set up to be applied. To "Apply Resource" is to use the resource for its information service, which either does something, presents something, or makes some information or service available. To "Assess Results" is to determine whether the application succeeded or failed, and to help decide what to do next.

The CM and SM interact as shown schematically in Figure 2.

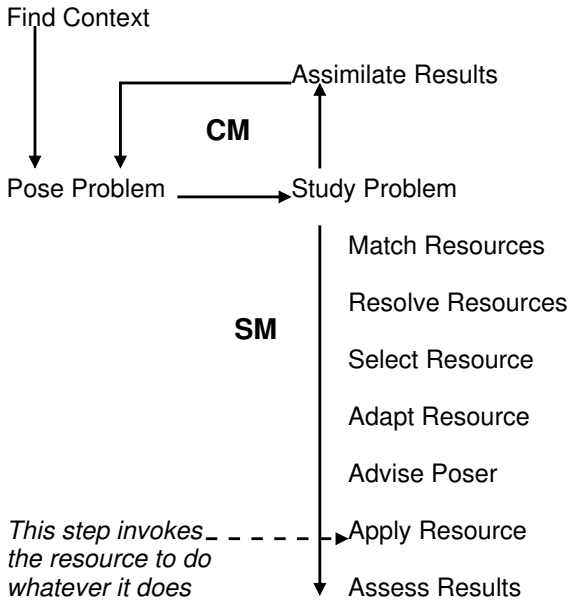

Figure 2: CM and SM Steps

Finally, we insist that every step in the above sequences is actually a posed problem, and is treated in exactly the same way as any other, which makes these sequences "meta"-recursive [3]. This makes the system completely Computationally Reflective. That means that if we have any knowledge at all that a different planner may be more appropriate for the context and application at hand, we can use it (after defining the appropriate context conditions), either to replace the default SM when it is applicable, or to replace individual steps of the SM, according to that context (which can be selected at run time).

This meta-recursive choice shows how Wrappings satisfies our claim above that there is "no privileged choice"; any part of the system may be replaced or superseded.

Of course, we also have to have something to replace or supersede. We have also provided default resources for each of the CM and SM steps, to be used when no other is selected to supersede it (as the above SM is the default resource for the problem "Study Problem"). A simple complication occurs with the default among many possible resources for the "Select Resource" problem: we want to allow other resources to be used, so we insist that the default resource (which otherwise might just pick the first resource on the list) not pick itself if there is another choice when it is addressing the "Select Resource" problem.

We have used these algorithms many times to explain and implement autonomous and reflective agents and systems [45] [46], and shown that they provide the appropriate level of manageable flexibility and auditable integration. The advantage in flexibility this approach provides over other activity loops that have been proposed is that the SM and CM steps are "meta"-steps, with posed problems for the activities, allowing one further level of abstraction and indirection when it is useful. There are a number of other activity loops that we have seen described in various places [9], but we think our CM / SM meta-recursive interaction subsumes all of them. The meta-interpretation style [3] can of course be applied to any of them to make them much more flexible.

We have implemented several different kinds of CMs in addition to the simple default CM defined above. There are CMs that short cut the reflection by calling the default step resources directly, and fully recursive versions that have extra levels of problem posing. Some of them are described in other papers in the references.

We have also used different SMs, beyond the default one that tries only one resource: one SM tries all applicable resources and returns with the first success, another tries them all and evaluates them to return the best success, and one collects all successes and summarizes. There are also different kinds of SM steps. The Match and Resolve resources that read XML WKBs are different from the ones that read te text only form. A different Match or Resolve might invoke a more sophisticated planner if there are no matches. A different Select might choose all compatible resources, then negotiate among them. Different versions of apply, beyond the default function call, might send a request message, or invoke an interpreter or other process. Another one might simply add the resource to a configuration, instead of invoking it.

Finally, for studying the timing characteristics of an infrastructure, we want to use a simulation style of analysis, but we want to use the same Wrapping infrastructure. In this case, we want to use a CM and SM that includes a simulation engine. This can be done easily, with exactly the same default CM and SM, with different CM and SM steps (this is one of the results of our em-

phasis on Computational Reflection). Essentially, the new Find Context resource initializes the future events set using a provided scenario. The new Pose Problem resource determines the next relevant event as a posed problem, using whatever dynamic knowledge there is about the system. The new Study Problem resource runs that event element as a posed problem, possibly scheduling new event elements. Running the event element includes keeping track of time requirements. Then the new Assimilate Results resource displays the recent movements. Then running the CM is running the simulation.

We have also developed a few mitigations for run-time decision time issues. A system that does not change its resources quickly can save processing time by using what are called memo functions (as defined for the Haskell Memo library and other places [30]), which are essentially collections of already computed values of a function. This memory can be extremely useful when the computation is time consuming. In our case, the resources used for the "Match Resources" problem could keep a record (which we presume would be a reverse index mapping problems to Wrappings. Even more useful might be a memo function for the "Study Problem" resources, which would keep the entire problem to resource in context values, and only compute them once. For our embedded system applications, however, we expect the context to be changing more rapidly, so this memo function may not be as useful.

In this case, we recommend using partial evaluation [18] [41] [19] to eliminate the SM altogether for some problems. If the set of resource Wrappings implies that there is only one resource that can apply to a problem, then we can sidestep the SM processing almost entirely, and replace the problem posing step with a test of the applicability conditions and an invocation of the resource.

7.4 Wrapping Knowledge Bases

The Wrappings in the WKBs are used by PMs to map problems to resources. The first implementations used a very simple keyword value format, intended for use by the default CM and SM.

Now, however, we usually write our Wrappings in XML for generality and simplicity, since XML parsers exist for many implementation languages. It should also be remembered that the format of the WKBs can differ for different SMs and other PMs, so it need not be the same throughout a system. Wrapping applications can use different knowledge representations (any of the popular Knowledge Representation Languages can be used, but we usually avoid them because they place too much power in processes not subject to change, and therefore not subject to study). The only real constraint is to support the CM/SM or whatever PMs are being used.

Also, for some applications, qualitative information and qualitative distinctions are important, considerations such as best practices, preference indications, and performance expectations. For example, two optimization methods might be distinguished by such information as "slow optimization method that requires a mathematician to interpret" versus "fast, inexact so only use it as a preliminary indicator".

With this in mind, we present a sample Wrapping Knowledge Base format used in a recent application [9], which was implemented in a keyword value style::

RS resource name as a sequence of symbols.
PB problem name as a sequence of symbols, with list of problem parameter names.
NF problem parameter conditions: Each of these is a boolean parameter conditions, assumed to apply conjunctively. A condition can test for existence or not, or for specific value range.
XC context conditions: Each of these is a boolean parameter condition, assumed to apply conjunctively. A condition can test for an attribute's existence or not, or for specific value range.
PM map from problem parameters to resource parameters, assumed to be in resource parameter order. Fancier versions might allow arithmetic expressions in the map.
XH context condition changes: Each of these is a context variable assignment. Fancier versions might allow arithmetic expressions in the assignment.
SY symbol (symbolic name of resource in object file)
FL source file (path name of object file that contains the symbol)
ND

Most of these entries are optional, and several may occur more than once. There must be exactly one RS, PB, and ND entry to define the map, exactly one SY and FL entry to define the compiled resource code (in our unix/linux implementation), and there may be zero or more of any of the others.

The WKB entries support information needed by the default SM steps. Match expects to find resources that claim to address the posed problem in the current context, by filtering on the parameter and context conditions. Resolve expects to find conditions that guarantee that a resource can address the posed problem in the current context, also by filtering on the parameter and context conditions (we usually expect the match conditions to be more superficial, as a preliminary filter, and the resolve conditions to be a negotiation between the specific Wrapping and the problem in context. Select expects to find resource preference information, qualitative or otherwise. Adapt expects to find specialized methods for adapting the selected resource, which can range from nothing to complex resource setup programs. Advise expects to find methods for presenting these decisions to the problem poser. Apply expects to find methods for applying a selected and adapted resource, from simple resource func-

tion invocation to the invocation of an interpreter for a domain-specific notation or other source code. Assess expects to find specialized methods for assessing the results (these are resource dependent). Match, Resolve, and Apply are the only necessary ones in the simplest cases. The others are optional, and sensible defaults exist, as described above.

7.5 Wrapping Summary

Wrapping-based systems support run-time decisions about which resources to apply in the current context, both at the application level (the resources that perform the task at hand) and at the meta-level (the resources that are used to select and organize the application level resources). This flexibility does come with a cost, but there are also mechanisms based on partial evaluation [18] [41] [19] for removing any decisions that will be made the same way every time, thus leaving the costs where the variabilities need to be.

The Wrapping approach makes infrastructure experimentation simpler and more effective because of its separation of problems and resource uses from resources. Such a system can have "macro-resources" that are combinations of resources applied together, and also "micro-resources" that are particular usage styles of resources packaged and treated separately for different contexts. The Wrapping infrastructure does not restrict mixing and matching these styles.

In summary, there are several advantages of using Wrappings that are also conducive to good system design practice:

- using Wrappings allows (requires) careful definition of the modeling spaces, especially the problem spaces that drive the whole process (a problem can be considered to be a generic activity within a model or modeling space);
- using Wrappings encourages (requires) good abstractions to facilitate experimentation with various strategies, to decide which ones can be done in real-time in the application at hand, and which ones can only be done in simulation;
- using Wrappings allows (requires) generic integration strategies that are explicit and therefore sharable and reusable;
- using Wrappings allows (requires) careful definition and decomposition of the expectations for the system, since the system design is done entirely in terms of posed problems and responsibilities, instead of components and requirements.

We believe that this up front modeling is essential for effective system design, whether or not Wrappings are used.