Infrastructure for Studying Infrastructure

Christopher Landauer
Topcy House Consulting

Abstract

Self-adaptation in embedded self-organizing real-time systems pose stringent expectations on the performance of their system architecture. The flexibility required for self-adaptation argues for multiplicity or variability of processes, whereas the embedded real-time aspects argue for very fast and low power processes. These two expectations are in direct opposition to each other, and good engineering practice implies careful study of the implications of the trade-offs. In this paper, we show how to use our Wrapping approach to integration infrastructure as a base to study proposed infrastructure choices for these applications, and argue that the Wrappings approach is ideally suited to this endeavor, since it makes no *a priori* assumptions about the infrastructure (or about itself, as we shall explain), and therefore allows any such questions to be studied. To do so, we provide enough details about the Wrapping approach to support our claims, and show how it would be applied to some popular infrastructures.

1 Introduction

Implementing self-adaptation in embedded self-organizing real-time systems presents many difficulties. For embedded systems, there are complex and often poorly understood interfaces to the operational environment, to the hardware platform properties, and even to the styles of expected use of the system.

For self-organizing systems, there are the design problems of mapping from provided system knowledge to internal system information, and from provided system purposes to internal system goals, and then there is the whole issue of emergent properties. We want to design the system to avoid most of the issues, and encourage others. Even when we know which is which, we still have the problem of devising methods for recognizing and exploiting emergence in systems.

For real-time systems, there are perennial problems of resource constraints and the appropriate use of approximations in both computation and operation as a way of saving time.

We do not presume that these are the only hard problems in real-time systems, but each of these issues is difficult enough, and it is very difficult to separate them in a given system, to study the effects of possible alternative approaches. For more information about these and related problems, see [5] [28].

A further exacerbation of the difficulty is caused by the realization that any such system will have an underlying infrastructure that organizes the processes within the system and manages communication among them and the outside environment, and that it is very difficult to demonstrate that the infrastructure supports the necessary processes and provides the features we expect.

We therefore believe that we need a mechanism to study these system infrastructures, to help us understand how their various components and structures relate to the properties above.

But how do we study infrastructure? We can certainly make choices and implement a few of them, but we want to reach a deeper level of understanding, so that results may be shared. When we set out to develop a system with which to study infrastructure, we realized that any fundamental choice we make about the infrastructure is one that we cannot study within it. We therefore devised a knowledge-based approach that makes no privileged choices, that is, every choice we made can be superseded; every knowledge base and every process the approach uses can be replaced.

In Section 2, we introduce our Wrappings approach to infrastructure and show how it satisfies our first claim above (“no privileged choices”).

In Section 3, we describe approaches that seem to be similar to Wrappings, though we contend that none goes as far as we do in supporting flexibility with Computational Reflection.

In Section 4, we return to considerations of embedded systems to show how some popular infrastructure choices can be studied with Wrappings.

Finally, in Section 5, we present our conclusions and describe some prospects for this approach.

2 Wrapping Integration Infrastructure

We provide a short description of Wrappings in this Section, since there are many other more detailed descriptions elsewhere [21] [19], and especially the tutorials [25] [26]. The Wrapping integration infrastructure is our approach to run-time flexibility, 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 are flexible in their interconnections [20], 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 [24].

This Section is a very short introduction to the capabilities of Wrapping-based systems. Much more detail is available in the references, especially the tutorials.

2.1 Problem Posing Interpretation

The "Problem Posing" interpretation of programs [21] 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 interface from intended purpose.

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 an information request called 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, we can use more flexible mechanisms for connecting them than simply using the same name.

2.2 Wrapping Overview

Many styles of mapping from problems to resources exist, often using a mechanism called implicit invocation [11]. 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 [2] [3] [4] 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. 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 therefore have no privileged resource; anything can be replaced. Default PMs 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 implicit invocation process, allowing arbitrary processes to be inserted in the middle of the resource invocation process. This choice leads to very flexible systems.

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

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.

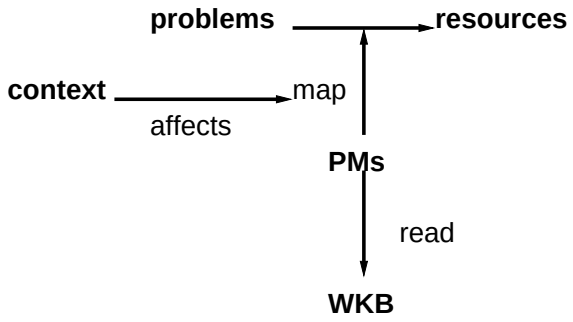

Figure 1: Wrapping Aspects

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 [23]:

   - *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 problems in context to appropriate resources. A Wrapping is not simply a coded interface "to" a resource; it is a conceptual interface to the "use" of a resource, for a particular problem in a particular context. This information is used to generate the appropriate invocation 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.

2.3 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.

If the Coordination Manager is the basic cyclic program heartbeat, then the Study Manager is a planner that organizes the resource applications. The CM and SM interact as shown schematically in Figure 2.

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,

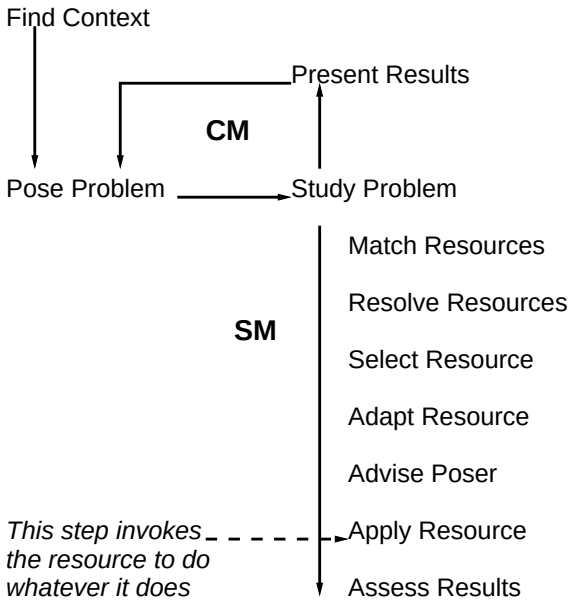

Figure 2: CM and SM Steps

- 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.

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 [1]. 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 therefore 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 [22] [23], 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 [3], but we think our CM / SM meta-recursive interaction subsumes all of them. The meta-interpretation style [1] of Wrappings 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 text only WKBs. 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 emphasis 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 [15]), 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 [6] [21] [8] 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.

2.4 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 [3], 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 defining object file).
- ND (this is an end marker, with no associated text).

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 and linux implementations), 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 function 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.

2.5 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 [6] [21] [8] 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. It is also more or less completely independent of the programming language used.

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 ab-

stractions 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, though often quite difficult, is essential for effective real-time or embedded system design, whether or not our Wrappings approach is used.

3 Similar Approaches

There are a number of approaches similar in spirit to Wrappings, since it has been widely recognized for some time that our modern computing system development processes are seriously deficient for the kinds and complexities of systems that we are now building. The "divide-and-conquer" / reductionist paradigm has worked extremely well for hundreds or even thousands of years, and we are only now attempting to address problems that are too complex or too large for it to work well enough soon enough. These papers are more about similar goals than similar approaches to addressing them, but they all share a number of features with Wrappings.

There isn't very much common terminology, though the different approaches usually take note of robustness, emergence (usually as an unpleasant phenomenon), unintended side effects, and a class of properties loosely called self-x, which varyingly includes self-adapting [31] [27] [32] [37] [36], self-configuring, self-evolving [30], self-healing [7], self-improving, self-managing [12] [16], self-monitoring, self-organizing, self-protection, and many more. Many of these discussions recognize that the goal of robustness in any systems is in direct contradiction to the goal of efficiency, since a robust system needs to retain functionality that is only rarely used, whereas an efficient system wants to retain only that functionality that is being used.

Every one of these papers illustrates, but does not always emphasize, Computational Reflection as a fundamental need for self-x systems. The systems in question need internal instrumentation to determine what they are actually doing, so they can adjust it to fit their current circumstances. The information so gathered is a kind of model of the system, and we have advocated that these models, as well as models of the behavior of the external environment and all interactions with it, all be explicit and available to the system for analysis. The strong emphasis on these two properties distinguishes our approach from the others.

Organic Computing [2] was a Priority Program of the German Research Foundation, primarily led by several German Universities in partnership with several manufacturing companies. The main goal is to understand and manage the unavoidable emergence that occurs in sufficiently complex systems. The idea is that spontaneous local interaction causes self-organization, leading to emergent behavior, and the goal is to learn how to manage the emergence that cannot be prevented, and possibly exploit some kinds of emergence if they are helpful. Small scale experiments have shown that flexible, adaptive and robust services can be produced and that the usual problems with design, management, and acceptance can be addressed. As one example, the Organic Traffic Control Collective at the University of Karlsruhe describes its aims as follows:

> Organic Traffic Control Collaborative (OTC2) aims at the realisation of an organic traffic control System capable of controlling and optimising traffic signals in urban road networks.
>
> The combination of decentralised control pursuing fine-grained goals with higher level observation and control having a more abstract point of view is expected to be applicable to a broad range of problems worked on in the Organic Computing community.

Another research movement, called Self-Adaptive Software [17] [18], was originally a DARPA/ITO project, with the following description:

> Self Adaptive Software evaluates its own behavior and changes behavior when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible.

Autonomic Computing is a large program initiated by IBM in the United States, and picked up by many other research organizations here and abroad [9] [10], that intends to produce the analog of the autonomic nervous system (which exists in most vertebrates, including all mammals, reptiles, amphibians, and at least some fish; there is some disagreement about whether it exists in the simplest fish). This work seemingly concentrates on low-level health maintenance capabilities, to construct computer systems capable of some local self-management. They also emphasize the instrumentation required for closed control loops to be effective.

Finally, we call our approach self-modeling software [22] [23] [24], since it is explicitly about Computational Reflection and systems defined by models of themselves.

4 Embedded Self-Organizing Systems

Of course, not every infrastructure can be changed. Many are constrained by a particular choice of platform, and others are not under the control of the software developers. When it is possible, however, we recommend performing some experiments to determine the best infrastructure for the problem at hand, and using Wrappings to do that. They developers can then verify (or refute) their infrastructure (and other) design choices. In this Section, we describe five self-adaptation papers, and show how their infrastructure experiments would be facilitated by using Wrappings. We chose these papers to illustrate the potential range of this application of Wrappings, not to represent the range of applications of self-modeling.

Transformer [13] is an architecture-based adaptation framework that uses encapsulated adaption strategies called *composable adaptation planners* to address three challenges: composition of adaptation modules, meta-adaptation, and conflict detection and resolution. They use an infrastructure of six cooperating modules, cooperating via an unspecified message passing mechanism. A Wrapping infrastructure for this project would allow multiple alternative choices for each of their six adaptation management modules, and experimentation and comparison of those alternatives.

A risk-aware efficiency improvement advisor [14] observes actual system behavior and attempts to make suggestions for improved efficiency. In this application, there is a multi-agent system that is assigned a set of timed tasks, and the question is which task to assign to which agent. The analysis uses data mining of event patterns by one designate agent that is not to be assigned any other task, and requires collection of each agent history by the analyzer. A Wrapping infrastructure for this project would allow alternative classes of assignment problems, alternative strategies for analysis (centralized, distributed, cooperative, competitive), and even methods for dynamic reconfiguration as problems change.

In [29], a system consists of a number of independent control loops that perform local tasks that contribute to a specified goal, and they describe mechanisms for mapping a provided goal model into a collection of control loops, including the possibility of changing them at runtime. Their notion of control loops is essentially the same as in [3], which in our interpretation are the same as PMs. A Wrapping infrastructure for this project would allow alternative classes of goal models, conflict resolution strategies, and control loop modification strategies to be compared and evaluated.

In [34], the issue is interaction among separately self-adapting embedded systems, in which none has access to the entire system state, so there is an issue of parallel incompatible adaptations, and for physical systems, an issue of uncontrolled dynamics. They use a second-order observer controller architecture, with different layers having different scopes of applicability. A Wrapping infrastructure for this project would allow alternative architectures, and alternative choices for the higher-level control methods. They could also tradeoff the control performance with its complexity.

The issues in [35] are exactly about integration of self-x properties in a single middleware module called an *Organic Manager*. Here again is an observer controller architecture, with the manager controlling a set of separate nodes, each running middleware to allow the necessary control. The Organic Manager itself runs a particular control loop, which makes it essentially a PM as in [3]. A Wrapping infrastructure for this project would allow alternative sets of tasks for the control loop, such as planners, analyzers, monitors, and conflict resolution strategies, to be compared and evaluated.

In all cases, the use of a Wrapping infrastructure facilitates experiments and evaluation, as of course most system simulations could. We claim that using a Wrappings infrastructure would make the entire process more effective, because many more choices can be examined and compared explicitly, and because the inherent Computational Reflection allows the system to gain much more knowledge about how its components are interacting, and react to that knowledge more flexibly. We are not trying to say that all or even any of these applications would be more complete using Wrappings, but we do think that their conclusions could be better supported by studying the infrastructure alternatives that were not chosen.

Finally, we emphasize that we are not advocating that Wrappings be used in all target implementations, since it can impose a serious computational overhead, only that it be used to determine an adequate infrastructure for a target implementation. On the other hand, we have actually used Wrappings for the target implementation in CARS [3] [4] [19].

5 Conclusions and Prospects

We have described a modeling framework that we believe is ideal for modeling the infrastructure of a self-adaptive system, since it makes no *a priori* assumptions about that infrastructure, thereby allowing the developers freedom of design space exploration. It also provides a way to analyze the interaction and timing behavior of the infrastructure, and thereby to compare different choices against different criteria before the final choices

have to be made. The explicit integration support properties of Wrappings allow the domain resources to be kept strictly separate from the infrastructure resources until the infrastructure is completely determined, thus greatly reducing domain resource rework.

With the instrumentation available in Wrappings based systems, we expect that other infrastructure studies and framework providers can perform the experiments that verify (or help improve) their choices.

Future work in this area will lie in providing more PM resources specifically intended for this and other aspects of self-adaptation, including the experimental framework and simulation mechanisms, the integration of physics modules, and the incorporation of more interfaces to external modeling notations and their processors, such as AADL, DEVS, Modelica, and SysML.

References

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

[2] 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)

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

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

[5] Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, Jeff Magee (eds.), "Software Engineering for Self-Adaptive Systems: A Research Roadmap", Dagstuhl Seminar 08031, 13-18 January 2008; *Software Engineering for Self-Adaptive Systems*, SLNCS 5525, Springer (2009)

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

[7] Eric Dashofy, Andre van der Hoek, Richard Taylor, "Towards architecture-based self-healing systems", pp.21-26 in *Proc. First Wksh. Self-Healing Systems*, 18-19 November, Charleston, S.C. (2002)

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

[9] Jim Dowling and Vinny Cahill, "The K-component architecture meta-model for self-adaptive software", pp.81-88 in Akinori Yonezawa, Satoshi Matsuoka (eds.), *Proc. Reflection'2001: Intern. Conf. Metalevel Architectures and Separation of Crosscutting Concerns*, 25-28 September, Kyoto, Japan, LNCS 2192, Springer (2001)

[10] Jim Dowling, Eoin Curran, Raymond Cunningham, Vinny Cahill, "Building Autonomic Systems using Collaborative Reinforcement Learning", Special Issue Autonomic Computing, Knowledge Engr. Rev. J., Vol.21, No.3, Cambridge U. Press (2006)

[11] 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 November 1998, Lake Buena Vista, Florida (1998); also *SIGSOFT Software Engineering Notes*, vol.23, no.6, pp.209-221, ACM (1998)

[12] Robert P. Goldman, David J. Musliner, Kurt D. Krebsbach, "Managing Online Self-Adaptation in Real-Time Environments", in [33]

[13] Ning Gui, Vincenzo De Florio, "Towards Meta-Adaptation Support with Reusable and Composable Adaptation Components", p.49-58 in *Proc. SASO 2012: The 6th IEEE Intern. Conf. Self-Adaptive and Self-Organizing Systems*, 10-14 October 2012, Lyon, France (2012)

[14] Jonathan Hudson, Jo:rg Denzinger, Holger Kasinger, Bernhard Bauer, "Dependable Risk-Aware Efficiency Improvement for Self-Organizing Emergent Systems", p.11-20 in *Proc. SASO 2011: The 5th IEEE Intern. Conf. Self-Adaptive and Self-Organizing Systems*, 03-07 October 2011, Ann Arbor, Michigan (2011)

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

[16] Jeff Kramer and Jeff Magee, "Self-Managed Systems: an Architectural Challenge", pp.259-268 in *Proc. FOSE'07: Wksh. on the Future of Software Engineering*, part of *Intern. Conf. Software Engineering*, 20-26 May 2007, Minneapolis (2007)

[17] R. Laddaga, H. Shrobe (eds.), "Self-Adaptive Software", LNCS 2614, Springer Verlag (2002)

[18] R. Laddaga, H. Shrobe (eds.), *Proc. Third Intern. Wksh. on Self-Adaptive Software*, 09-11 June, Ar-

lington, VA (2003)

[19] Christopher Landauer, “Abstract Infrastructure for Real Systems: Reflection and Autonomy in Real Time”, *Proc. SORT 2011: The Second IEEE Wksh. on Self-Organizing Real-Time Systems*, 31 March 2011, part of *ISORC 2011*, 28-31 March 2011, Newport Beach, California (2011)

[20] Christopher Landauer, Kirstie L. Bellman, “Lessons Learned with Wrapping Systems”, pp.132-142 in *Proc. ICECCS’99: The 5th Intern. Conf. Engr. Complex Computing Syst.*, 18-22 October 1999, Las Vegas, Nevada (1999)

[21] 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)

[22] Christopher Landauer, Kirstie L. Bellman, “Self-Modeling Systems”, pp.238-256 in [17]

[23] Christopher Landauer, Kirstie L. Bellman, “Managing Self-Modeling Systems”, in [18]

[24] Christopher Landauer, Kirstie L. Bellman, “Self Managed Adaptability with Wrappings”, *Proc. Evolve2005: Wksh. on Software Evolvability* 26 September 2005, part of *ICSM 2005*, 25-30 September 2005, Budapest, Hungary (2005)

[25] 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 October 2012, Lyon, France (2012)

[26] 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 February 2013, San Diego, California (2013)

[27] Ákos Lèdeczi, Gábor Karsai, and Ted Bapty, “Synthesis of Self-Adaptive Software”, in *Proc. Aerospace Conf.*, Big Sky, MT, March 2000 (2000)

[28] Rogério de Lemos, Holger Giese, Hausi A. Müller, Mary Shaw, “Software Engineering for Self-Adaptive Systems: A Second Research Roadmap”, *Software Engineering for Self-Adaptive Systems*, Dagstuhl Seminar 10431, 24-29 October 2010 (Draft Version of 09 November 2011)

[29] Hiroyuki Nakagawa, Akihiko Ohsuga, Shinichi Honiden, “Towards Dynamic Evolution of Self-adaptive Systems Based on Dynamic Updating of Control Loops”, p.59-68 in *Proc. SASO 2012: The 6th IEEE Intern. Conf. Self-Adaptive and Self-Organizing Systems*, 10-14 October 2012, Lyon, France (2012)

[30] Peyman Oreizy, Nenad Medvidovic, Richard N. Taylor, “Architecture-based runtime software evolution”, pp.177-186 in *Proc. Intern. Conf. Software Engineering*, 19-25 April, Kyoto, Japan (1998)

[31] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, Alexander L. Wolf, “An architecture-based approach to self-adaptive software”, *IEEE Intelligent Systems and Applications* (1999)

[32] Paul Robertson, Robert Laddaga, “The GRAVA Self-Adaptive Architecture: History; Design; Applications; and Challenges”, pp.298-303 in *Proc. ICDCS’04: Intern. Conf. Distributed Computing Systems*, 23-24 March, Tokyo, Japan (2004)

[33] Paul Robertson, Howie Shrobe, Robert Laddaga (eds.), *Self-Adaptive Software*, *Proc. First Intern. Wksh. on Self-Adaptive Software*, 17-19 April 2000, Oxford U., England, LNCS 1936, Springer Verlag (2001)

[34] Nils Rosemann, Werner Brockmann and Christian Lintze, “Controlling the learning dynamics of interacting self-adapting systems”, p.1-10 in *Proc. SASO 2011: The 5th IEEE Intern. Conf. Self-Adaptive and Self-Organizing Systems*, 03-07 October 2011, Ann Arbor, Michigan (2011)

[35] Julia Schmitt, Michael Roth, Rolf Kiefhaber, Florian Kluge, Theo Ungerer, “Using an Automated Planner to Control an Organic Middleware”, p.71-78 in *Proc. SASO 2011: The 5th IEEE Intern. Conf. Self-Adaptive and Self-Organizing Systems*, 03-07 October 2011, Ann Arbor, Michigan (2011)

[36] Danny Weyns, Sam Malek, Jesper Andersson, “FORMS: a FOrmal Reference Model for Self-adaptation”, pp.205-214 in *Proc. ICAC’10: The 7th Intern. Conf. Autonomic Computing*, 07-10 June 2010, Washington, D.C. (2010)

[37] Ji Zhang, Betty H. C. Cheng, “Towards Re-engineering Legacy Systems for Assured Dynamic Adaptation ”, pp.10 in *Proc. MSE’07: Intern. Conf. Modeling in Software Engineering (ICSE’07)*, 20-26 May, Minneapolis, Minnesota (2007)