System Development at Run Time

Dr. Christopher Landauer, Dr. Kirstie L. Bellman

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

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.

Keywords: Self-Modeling Systems, Computational Reflection, Wrapping Integration Infrastructure, Scenario-Based Engineering Process, Model Creation and Analysis

1 Introduction

In this paper, we strongly argue the notion that models are not only useful, but even essential, for defining, developing and even operating a system for a complex operational environment. We also argue that 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. 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 [19].

Our ability to build systems with this property depends crucially on Computational Reflection, and specifically on Wrappings [17] [15], an integration infrastructure for complex software-intensive systems. It was originally developed to support run-time decision-making and reconfiguration [18], 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 [19].

This work is to be clearly distinguished from systems that use parallel code and models [10], 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 [8] [9] can reduce all unchanging decisions to simple sequences (the partial evaluation methods have more information in a Wrapping-based system than in traditional software [17]).

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 [22], 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 [17]) 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 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 short overview of Wrappings, along with the Problem Posing Programming Paradigm [18] [17] [20]. These are the methods that allow us to study infrastructure [15] and build self-modeling systems [19]. 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.

It is clear that self-adaptive systems can make substantive changes at runtime. In one sense, this is already autonomous development. We extend this notion to the entire development cycle, using the Scenario-Based Engineering Process [22] 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 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 [17]. The Wrapping integration infrastructure is our approach to run-time flexibility [15], 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 [17] 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 [3] [4] 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 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 [11], 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 [8] [17] [9] 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 [4] [15]. 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 [2]. 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.

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 [24]. 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 [12] [25] [23], so it can be compared to the run-time activity [6], using models of inferred behavior [1]. An interesting parallel set of studies has been ongoing in the business process modeling community, regarding workflow models as process models [28], though typically producing external models of human processes.

However, it is also well-known that there are several challenges in using architectural models at run time (adapted from [24] [12] [23]):

- 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 [25]), 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 beyond the scope of this paper [16].

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 [25] [13] [7]. 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 [26] 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 [5] [23] [21]. 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 [21], 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 [14] [26] [27], but also others) explicitly use activity loops (such as the MAPE loop of autonomic computing) 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 [25], 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 Challenges

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". 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.

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, but it will do for a starting point.

5 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 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 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, 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)
2. Harold Abelson, Gerald Sussman, with Julie Sussman, *The Structure and Interpretation of Computer Programs*, Bradford Books, now MIT (1985)
3. 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)
4. 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)
5. 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)
6. 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)
7. 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)
8. 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)
9. 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)
10. 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)
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 Nov 1998, Lake Buena Vista, Florida (1998); also *SIGSOFT Software Engineering Notes*, vol.23, no.6, pp.209-221, ACM (1998)
12. David Garlan and Bradley Schmerl, "Using Architectural Models at Runtime: Research Challenges", *Proc. First European Wksh. 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)

13. 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)
14. Jeffrey O. Kephart, David M. Chase, “The Vision of Autonomic Computing”, *IEEE Computer*, Vol.36, Issue 1, p.41-50 (Jan 2003)
15. Christopher Landauer, “Infrastructure for Studying Infrastructure”, *Proc. ESOS 2013: Wksh. 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)
16. Christopher Landauer, “Advanced Mathematical Methods for Telemetry Analysis” (presentation), *2015 Spacecraft Flight Software Workshop*, 27-29 October 2015, JHU/APL, Laurel, Maryland (2015)
17. 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)
18. 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)
19. Christopher Landauer, Kirstie L. Bellman, “Self-Modeling Systems”, p. 238-256 in R. Laddaga, H. Shrobe (eds.), “Self-Adaptive Software”, LNCS 2614, Springer (2002)
20. Christopher Landauer, Kirstie Bellman, “Designing Cooperating Self-Improving Systems”, *Proc. 2105 SISSY Wksh.: 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)
21. 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)
22. Karen McGraw and Karan Harbison, *User-centered Requirements: The Scenario-Based Engineering Process*, Lawrence Erlbaum (1997)
23. 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)
24. 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)
25. 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)
26. 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)
27. 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)
28. Workflow Management Coalition, *Workflow Standard: Process Definition Interface; XML Process Definition Language*, Document Number WFMC-TC-1025, 03 Oct 2005 (2005)