Towards semantics of self-adaptive software

Duško Pavlović

Kestrel Institute, Palo Alto, USA
dusko@kestrel.edu

Abstract. When people perform computations, they routinely monitor their results, and try to adapt and improve their algorithms when a need arises. The idea of self-adaptive software is to implement this common facility of human mind within the framework of the standard logical methods of software engineering. The ubiquitous practice of testing, debugging and improving programs at the design time should be automated, and established as a continuing *run time* routine.

Technically, the task thus requires combining functionalities of automated software development tools and of runtime environments. Such combinations lead not just to challenging engineering problems, but also to novel theoretical questions. Formal methods are needed, and the standard techniques do not suffice.

As a first contribution in this direction, we present a basic mathematical framework suitable for describing self-adaptive software at a high level of semantical abstraction. A static view leads to a structure akin to the Chu construction. An dynamic view is given by a coalgebraic presentation of adaptive transducers.

1 Introduction: specification carrying code

One idea towards self-adaptive software is, very roughly, to introduce some kind of "formalized comments", or Floyd-Hoare style annotations as first class citizens of programs. Together with the executable statements, they should be made available to a generalized, *adaptive* interpreter, extended by an automated specification engine (e.g., SPECWARE™-style), supported by theorem provers and code generators. This adaptive interpreter would not only evaluate the executable statements, but also systematically test how their results and behaviour satisfy the requirements specified in the formal annotations. Such testing data could then be used for generating improved code, better adapted to the specifications, often on the fly. On the other hand, the formal specifications could often also be refined, i.e. adapted to the empiric data obtained from testing a particular implementation.

1.1 Automated testing and adaptation

Coupling programs with their specifications in a uniform, automatically supported framework would, at the very least, allow monitoring correctness, reliability, safety and liveness properties of programs, with respect to the specified requirements, as well as the particular distribution of the input data, and any other aspects of the execution environment that may become available at run time.

In some cases, one could hope for more than mere monitoring of the relationship between an abstract specification and its implementation. Indeed, software can often be improved and adapted to its specifications in a predictable fashion, once its running behaviour can be observed on concrete inputs. This is, for instance, usually possible in the cases when the correctness criteria are not absolute, *viz* when the software only approximates its specification. Software that models physical systems, or stochastic processes, or even just computes genuine real numbers or functions, is usually of this kind: the infinitary nature of the output data precludes the exact solutions, which can only be approximated. But the approximations can, in principle, always be improved, on an additional cost. Comparing the actual runs of such software with its abstract specifications may suggest optimizing this cost, say, by adjusting the coefficients in the numeric formulas to the observed distribution of the input data. In some cases, different distributions may even justify applying different algorithms, which can be abstractly classified in advance, so that the adapted code can be synthesized on the fly.

Self-adaptive software can perhaps be compared with an engineer monitoring the results of his computations, updating the methods and refining the model. The point is here that a part of this process of adaptation can and needs to be automated.

1.2 Dynamic assembly and reconfiguration

Furthermore, in a complex software system, the adaptation cycle of a software component can take into account not only the run time behaviour of the component itself, but also the behaviour of the other components, and the changes in the environment at large.

In order to be combined, software modules must contain sufficient information about their structure and behavior. Conventional *application programming interfaces*, APIs, are intended to carry such information but APIs are usually under-specified (they contain just signature/type information), are often based on unfulfilled assumptions, and are prone to change. Conventional APIs are thus insufficient for the task of assured composition.

Ideally, APIs would completely capture all assumptions that influence behavior. However, formally verifying such completeness is usually infeasible — some degree of API partiality is inevitable in practical software development. Nevertheless, in dynamically adaptable software, API partiality should continually *decrease* as the interfaces evolve during the lifetime of a system, together with the specifications and implementations of its components and perhaps even its architecture.

We believe that the requirement of dynamically adaptable software naturally leads to the idea of specification-carrying code: an adaptable program must carry a current specification of its functionality, an adaptable specification must come with a partial implementation. Adaptability requires a simultaneous and interactive development of the logical structure and the operational behavior.

Moreover, specification-carrying code is the way to accommodate and support, rather than limit and avoid, the rich dynamics of *ever changing interfaces*, so often experienced in large software systems, with the unpredictable interactions arising from the ever changing environments. The fact that specifications, implementations, interfaces and architectures in principle *never stop changing* during their lifetime, should not be taken as a nuisance, but recognized as the essence of the game of software, built into its semantical foundation, and implemented as a design, and when possible a runtime routine.

Of course, adjusting the behaviour of autonomous software components to each other, tuning them in on the basis of their behaviour, or getting them to interact in a desired way, can be a very hard task. But if their abstract specifications in a generic language are maintained on a suitable formal platform, and kept available at run time, their possible interactions and joint consistency can be analyzed abstactly, e.g. using theorem provers, and their implementations can be modified towards a desired joint behaviour. This may involve reimplementing, i.e. synthesizing new code on the fly, and is certainly not a straightforward task. However, it seems unavoidable for the independently implemented components, necessary for the compositional development of systems — and we believe that it is also within the reach of the current software synthesis technologies.[1]

In any case, a software system that needs to adapt its code and behaviour the run time data, or to the changes of the environment, while maintaining its essential functionality, will surely need to carry and maintain a specification of this functionality in some form, perhaps including a history record, and a current correctness certificate. Since this task, and the structures involved, clearly go beyond the existing software methodologies, a careful analysis of the semantical reprecussions seems necessary. Building the suitable design and development environments for the specification carrying self-adaptive software will require a mathematical framework with some nonstandard features. In the present paper, a crude, initial picture of some of these features is outlined. Section 2 describes the structure and the intended interpretation of an abstract category of specification carrying modules. This can be viewed as a first attempt at denotational semantics of such bipartite modules, involving a structural and a behavioral component. Section 3 briefly outlines the structures needed for a dynamic view of the adaptation process, and the way to adjoin them in the described semantical framework.

[1] The idea of adding abstract logical annotations to code can be seen as a generalization of Necula and Lee's [12] combination of *proof-carrying-code* and *certifying compilers*. While mainly concerned with the security properties of mobile code, many of the ideas that arose in that work do seem to apply in general, and provide evidence for the imminent realizability of the present ideas.

2 Category of specification carrying programs

As the name suggests, a specification carrying program consists of a program P, a specification S, and a satisfaction, or model relation $\models$ which tells how they "carry" each other, i.e. establishes the sense in which P satisfies S. Formally, a specification carrying program is thus a triple $\langle P, \models, S\rangle$.

But what precisely are P, $\models$, and S? In principle, a formal specification S is a logical theory in a generic specification language (e.g., the higher-order predicate logic). A program P, on the other hand, here means a description of a computational behaviour in one of the available formalisms: it can be a transition system, an automaton, or simply a piece of code in a sufficiently general programming language. Finally, the satisfaction relation $q \models \psi$ tells which of the formulas ψ of S are satisfied at each of the states q of P.

Although diverse formalisms can be used for the particular presentations of P and S, they can always be uniformly represented as categories. The classifying categories $\mathbb{P}$ and $\mathbb{S}$ are derived respectively from the program P and the specification S using the known procedures of operational semantics and categorical model theory. The satisfaction relation $\models$ then becomes a functor from $\mathbb{P} \times \mathbb{S}$.

But we shall not attempt to present this abstract framework directly, but rather begin by motivating its main features and the underlying ideas, especially as they arise in extant frameworks. Some of the underlying mathematics will be outlined, but the details are beyond the scope of this presentation.

2.1 Contracts: the game of refinement and adaptation

Intuitively, an adaptive module $\langle P, \models, S\rangle$ can be thought of as a *contract* between a programmer and a client: the client specifies the requirements in S, and the programmer provides the program P in response. This intuition yields a conceptual basis for the discipline and theory of software *refinement* [1, 11].

The process of software adaptation can now be viewed as a *game* played between the client and the programmer: the former refines the specification S, say to S', and the latter tries to respond accordingly by adapting the program P to P' accordingly, i.e. in such a way that the satisfaction $\models$ is preserved. This means that a predicate φ from S should be satisfied in a state q of P if and only if the translation φ' of φ to S' is satisfied in all states q' of P' that simulate the state q.

In summary, an adaptation transformation of $\langle P, \models, S\rangle$ into $\langle P', \models', S'\rangle$ consists of

- a simulation $P \xleftarrow{f_P} P'$, and
- an interpretation $S \xrightarrow{f_S} S'$,

such that for all predicates φ in S and all states q' in P' holds

$$q' \models' f_S(\varphi) \iff f_P(q') \models \varphi \tag{1}$$

The pair $f = \langle f_P, f_S \rangle$, satisfying (1), is an *adaptation* morphism

$$\langle P, \models, S \rangle \xrightarrow{f} \langle P', \models', S' \rangle$$

An abstract semantical framework for software adaptation is thus given by the category $\mathcal{C}$ of specification carrying programs, viewed as contracts $C = \langle P_C, \models_C, S_C \rangle$, with the adaptation morphisms between them.

Contracts as intervals. Note that the morphism $f : \langle P, \models, S \rangle \longrightarrow \langle P', \models, S' \rangle$ is running concurrently with the specification refinement $f_S : S \longrightarrow S'$, but in the *opposite* direction from the simulation $f_P : P' \longrightarrow P$. An imprecise, yet instructive analogy is that a contract $\langle P, \models, S \rangle$ can be thought of as a real interval $[p, s]$. The desired software functionality, that S and P are approximating in their different ways, then corresponds to a point, or an infinitely small interval contained between p and s. The refinement/adaptation game of the client and the programmer now becomes an interactive search for greater lower bounds p, and smaller upper bounds s, i.e. for smaller and smaller intervals, nested in each other, all containing the desired point. This process brings the programs and their specifications closer and closer together, so that they better approximate the desired functionality from both sides, i.e. in terms of the behavior and the structure. Viewed in this way, an adaptation morphism becomes like a formal witness of the interval containment

$$[p, s] \supseteq [p', s'] \iff p \leq p' \wedge s' \leq s$$

The morphism $\langle P, \models, S \rangle \longrightarrow \langle P', \models', S' \rangle$ thus corresponds to the containment $[p, s] \supseteq [p', s']$, the interpretation $S \longrightarrow S'$ to the relation $s \geq s'$, and the simulation $P' \longrightarrow P$ to $p' \geq p$.

Of course, the analogy breaks down on the fact that a specification S and a program P are objects of different types. Nevertheless, the satisfaction relation $\models$ measures "the distance" between P and S, and the mathematical structures arising from refinement/adaptation remain similar to those encountered in approximating numbers by intervals. In spite of its imprecision, the metaphor remains instructive. Moreover, the view of S and P as the approximations of an ideal point where the program is optimally adapted to the specification, does seem conceptually correct. To better approximate this point, S prescribes a minimum of structure necessary for expressing some part of the desired functionality, whereas P provides the simplest behaviour sufficient for implementing it. The client's strategy is to refine S to S' along $f_S : S \longrightarrow S'$, to constrain the input/output requirements more, and make programmer's task harder. The programmer, on the other hand, must enrich the behaviour P to P', in order to better fit the task. This means that P' must at least be able to simulate P, along $f_P : P' \longrightarrow P$.

In any case, the client provides the input/output requirements S, whereas the programmer supplies in P the computation steps transforming the inputs to the outputs. Semantically speaking, the structural, "upper bound" descriptions of software are thus driven by its denotational aspects, *viz* the structure of the datatypes and the functions that need to be refined and implemented. This is summarized in a spec S. On the other hand, a “lower bound” description of a desired piece of software is driven by its operational aspects, and in our particular case by the behaviour of a given implementation P, that needs to be adapted, or optimized for better performance. While conceptually different, the structural/denotational and the behavioural/operational aspects can be captured in a uniform setting [22], which would perhaps make the idea of contracts as intervals more convincing for some readers, or at least give the “intervals” a more familiar appearance. However, as they stand, specification carrying programs can be represented and implemented using mostly the readily available semantical frameworks, on the basis of the extant specification and programming environments.

2.2 Example: adaptive sorting

To give the reader an idea of what a concrete specification carrying module looks like, and how it adaptats on the fly, we sketch an (over)simplified example, based on the material from [23, 24]. We describe how a sorting module can be automatically reconfigured in response, say, to the observed distributions of the input data.

Suppose that sorting is done by a Divide-and-Conquer algorithm, e.g. Quicksort, or Mergesort. The idea of the Divide-and-Conquer sorting is, of course, to decompose (“divide”) the input data, sort the parts separately, and then compuse (“conquer”) them into a sorted string. The abstract scheme is:

$$\begin{tikzcd}
 & 1 \arrow[dl, "i"'] \arrow[dr, "o"] & \\
I \arrow[rr, "s"'] \arrow[d, "d"', "|" marking] & & O \\
I \times I \arrow[rr, "s \times s"'] & & O \times O \arrow[u, "c"', "|" marking]
\end{tikzcd}$$

In words, there are two sorts, I for the inputs and O for the outputs, and the desired sorting function s maps one to the other. In principle, I should be the type of bags (multisets) over a linear order, whereas O are the ordered sequences, with respect to the same order. The constants $i : I$ and $o : O$ are used to denote a particular input and the induced output. The bars over the arrows for d and c mean that they are relations, rather than functions. They should satisfy the requirements that

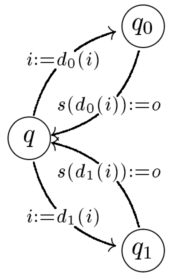
– if $d(x, y, z)$, then $x = y + z$, and
– if $c(x, y, z)$, then $|x| + |y| = |z|$

where $+$ is the union of bags, and $| - |$ maps sequences to the underlying bags. Although they are not functional, these relations are directed by the data flow. That is why they are denoted by the arrows.

The formal specification S_{DC} of the divide-and-conquer algorithms will thus look something like

```
spec Divide-and-Conquer[(S,<): Linear-Order]

   imports
      bag(S),
      ordered-seq(S)

   sorts
      I = bag(S),
      O = oredered-seq(S)

   operations
      s:I->O,
      d:I,I,I -> Bool,
      c:O,O,O -> Bool

   axioms
      d(x,y,z) => x = y + z,
      c(x,y,z) => |x| + |y| = |z|,
      d(x,y,z) /\ c(s(y),s(z),w) => s(x) = w
endspec
```

An abstract program P_{DC}, partially implementing S_{DC} can now be represented as the transition system



where $d_0(x)$ and $d_1(x)$ denote any bags satisfying $d(x, d_0(x), d_1(x))$. In a way, the state q hides the implementation of d and c, whereas q_0 and q_1 hide the implementation of the sorting of the parts.

The theorems of S_{DC} are satisfied at all states: they are the invariants of the computation. The transitions from state to state induce the interpretations of the specification S_{DC} in itself, mapping e.g. $i \mapsto d_0(i)$ in one case, or $s(d_0(i)) \mapsto o$ in another. They all preserve the invariants, *viz* the theorems of S_{DC}, of course.

The satisfaction relation $\models_{DC}$ tells, moreover, for each particular state, which additional predicates, besides the theorems of S_{DC}, have been made true by the executed computational steps, *viz* the substitutions in S_{DC}.

The suitable refinements of S_{DC} yield the specifications S_{QS} of Quicksort and S_{MS} of Mergesort. As explained in [23, 24],

- taking $d(x,y,z) \iff x = y + z$ implies that $c(x,y,z)$ must mean that z is a merge of x and y — which yields Mergesort, whereas
- taking $c(x,y,z) \iff x@y = z$ implies that $d(x,y,z)$ must mean that y and z are a partition of x, such that all elements of y are smaller than every element of z — which yields Quicksort.

Implementing S_{QS} and S_{MS}, one can synthesize executable programs P_{QS} and P_{MS}. While the specifications come with the interpretations $S_{DC} \longrightarrow S_{QS}$ and $S_{DC} \longrightarrow S_{MS}$, the programs come with the simulations $P_{QS} \longrightarrow P_{DC}$ and $P_{MS} \longrightarrow P_{DC}$, showing how the implementations hidden in P_{DC} have been realized.

All together, we now have three contracts, DC, QS and MS, with two adaptation morphisms between them.

$$\begin{tikzcd}
& \boxed{S_{DC}} \arrow[dl] \arrow[dr] & \\
\boxed{S_{QS}} & & \boxed{S_{MS}}
\end{tikzcd}$$

$$\begin{tikzcd}
& \boxed{P_{DC}} & \\
\boxed{P_{QS}} \arrow[ur] & & \boxed{P_{MS}} \arrow[ul]
\end{tikzcd}$$

This sorting module can thus run in one of the two modes: Quicksort or Mergesort, and adapt, when needed, between one and the other. At each point of time, though, only one of them needs to be present, since the other can be automatically generated when needed. The module can be set up to monitor its performance, and reconfigure when it falls below some treshold.

Suppose that the module is running as Quicksort, and the input data are coming in almost sorted, which brings it close to the worst-case behavior. The adaptive interpreter aligns P_{QS} and S_{QS}, observes that 98% of the computation time is spent on the divide routine d, and decides to simplify it. It generalizes from S_{QS} to S_{DC}, and chooses the simplest possible d, namely

$$d(x,y,z) \iff x = y + z$$

The theorem prover can now derive that c must be merge, and thus automatically refines S_{DC} to S_{MS}. Since S_{MS} completely determines the algorithm, the code generator can now synthesize P_{MS} in a chosen language. The adaptation path was thus

$$P_{QS} \longrightarrow S_{QS} \longrightarrow S_{DC} \longrightarrow S_{MS} \longrightarrow P_{MS}$$

Of course, in this simple case, the reconfiguration between the two modes could be achieved within a program, with a Quicksort and a Mergesort block. One could build in a performance monitor into the program, maintain its statistics, and then, depending on it, branch to one of the sorting blocks, more suitable for the observed input distributions.

However, the real-life examples, that genuinely require self-adaptation [21], often involve choice between modules too large to be loaded together. One can, furthermore, easily envisage situations when modules couldn't even be stored together, either because of their sizes, or because of their large numbers. With the advent of the agent technologies, there are already situations when there are *infinitely* many logically possible modes of operation, among which one might profitably choose on the fly. With the current level of the program synthesis techniques, of course, this approach would be very hard to realize. Conceptually, however, it seems to be well within reach, and developing the techniques needed for realizing it is a very attractive challenge.

2.3 Institutions, and satisfaction as payoff

The bipartite setting of specifications coupled with programs via a satisfaction relation will probably not appear unfamiliar to the categorically minded members of the software specification community. They will recognize the category $\mathcal{C}$ of contracts as conceptually related to *institutions*, although not in all details.

An institution is a very general model theoretic framework. introduced by Goguen and Burstall in [3], and pursued by many others in several different forms. Its main purpose was to fill a conceptual gap in semantics of software. While the formal methods of software engineering are in principle based on universal algebra and model theory, with specifications *statically* describing some computational structures, programs at large are *dynamic* objects, they change state, and behave differently in different states. And while the mathematical theories completely determine the classes of their static models, as well as the notion of homomorphism between them, the software specifications do not pin down the programs that realize them[2]. In model theory, the Tarskian satisfaction relation $\models$ is a fixed, primitive concept; in theory of software specifications, on the other hand, there are many degrees of freedom in deciding what does it mean for a program to satisfy a specification, in particular with respect to its operational behaviour. It is then reasonable to display an abstract satisfaction relation $\models$ as a structural part of an institution, that can be varied together with theories and models, while stipulating which models satisfy which theories.[3]

[2] Not in the sense that all programs implementing a specification can be effectively derived from the specification, like all mathematical models of a theory, and indeed the whole model category, are effectively determined by the theory.

[3] Institutions thus bridge the gap between static theories and dynamic models by allowing the abstract satisfaction relation to vary. Another way to bridge this gap

Following this conceptual lead, the satisfaction relation can be generalized from an ordinary relation, where $q \models \psi$ is evaluated as true or false, to an $\mathbb{M}$-valued relation, where $q \models \psi$ can be any element of a distributive lattice, or a suitable category, say $\mathbb{M}$, measuring the degree to which the condition ψ is satisfied at the state q.

In standard game theoretic terms, the relation $\models$ now becomes the payoff matrix, displaying the value of each programmer's response to each client's call. Indeed, if the formulas of S are understood as the set of moves (or strategies[4]) available to the client, and the moves available to the programmer are identified with the states of P, then the satisfaction $\models$ becomes an $P \times S$-matrix of the elements of $\mathbb{M}$, i.e. a map

$$\models\, : P \times S \longrightarrow \mathbb{M} \tag{2}$$

assigning the payoff to each pair $\langle q, \psi \rangle$. It can be understood, say, as displaying programmer's gains for each combination of the moves, and the game acquires the usual von Neumann-Morgenstern form, with the client trying to minimize and the programmer to maximize this gain. The intuitive and logical meaning of the pairs of arrows in opposite directions, like in the adaptation morphisms, has been analyzed in this context in [6], connecting games, linear logic and the Chu construction.

In any case, semantics of specification carrying programs must draw ideas and structures from sources as varied as institutions and game theory, although the goals and methods in each case differ essentially. On the level of abstract categories, both institutions and specification carrying programs can be analyzed along the lines of the mentioned Chu construction [2, 14, 19]. The lax version [13] is also interesting, capturing the situation when adaptation may not just preserve, but also *improve* the satisfaction relation between the program and the specification. This corresponds to relaxing in (1) the equivalence $\iff$ to the implication $\Longleftarrow$. If $\models$ and $\models'$ are taken to be general $\mathbb{M}$-valued relations, or payoff matrices, as in (2), a morphism $f : \langle P, \models, S \rangle \longrightarrow \langle P', \models, S' \rangle$ improving satisfaction will be a triple $f = \langle f_P, f_{\models}, f_S \rangle$, consisting of

- a simulation $P \xleftarrow{f_P} P'$,
- an interpretation $S \xrightarrow{f_S} S'$, and
- for each $q' \in P'$ and $\psi \in S$ an arrow

$$(f_P(q') \models \psi) \xrightarrow{f_{\models}} (q' \models' f_S(\psi)) \tag{3}$$

in $\mathbb{M}$, with the suitable naturality condition.

is to introduce dynamics into theories. This is one of the ideas behind Gurevich's Abstract State Machines (formerly known as evolving algebras) [4].

[4] the distinction is of no consequence here

2.4 Towards functorial semantics of contracts

In order to express the above naturality condition, or work out a generic representation of the category $\mathcal{C}$ of contracts, one needs to present the specifications, and the programs in terms of their respective classifying categories.

Given a specification S, say as a theory in a predicate logic, the objects of the induced classifying category $\mathbb{S}$ will be the well-formed formulas of S, modulo the renaming of variables (α-conversion). The arrows are the functional relations definable in S, modulo the provability. For instance, take formulas $\alpha(\boldsymbol{x})$ and $\beta(\boldsymbol{y})$ in S, as the representatives of objects in $\mathbb{S}$. By renaming, we can achieve that their arguments $\boldsymbol{x}$ and $\boldsymbol{y}$ are disjoint. An $\mathbb{S}$-arrow from $\alpha(\boldsymbol{x})$ to $\beta(\boldsymbol{y})$ will be a predicate $\vartheta(\boldsymbol{x}, \boldsymbol{y})$, such that

$$\begin{aligned} \vartheta(\boldsymbol{x}, \boldsymbol{y}) &\vdash \alpha(\boldsymbol{x}) \wedge \beta(\boldsymbol{y}) \\ \alpha(\boldsymbol{x}) &\vdash \exists \boldsymbol{y}.\ \vartheta(\boldsymbol{x}, \boldsymbol{y}) \\ \vartheta(\boldsymbol{x}, \boldsymbol{y}') \wedge \vartheta(\boldsymbol{x}, \boldsymbol{y}'') &\vdash \boldsymbol{y}' = \boldsymbol{y}'' \end{aligned}$$

can be proved in S. The arrows of $\mathbb{S}$ thus capture the theorems of S, whereas the objects capture the language. More details can be found in [17].

The point of presenting a theory S as a category $\mathbb{S}$ is that the models of $\mathbb{S}$ can be obtained as the functors $\mathbb{S} \longrightarrow \mathsf{Set}$, preserving the logical structure. This is the essence of functorial semantics [7], and the foundation of categorical model theory [8, 9]. The applications to software engineering are discussed in [17].

Related, more direct, but less uniform procedures allow deriving categories from programs. They usually go under the name of operational semantics [18, 25], and come in too many varieties to justify going into any detail here.

Assuming that a specification S and a program P have been brought into a categorical form, and presented as classifying categories $\mathbb{S}$ and $\mathbb{P}$, the satisfaction relation $\models: \mathbb{S} \times \mathbb{P} \longrightarrow \mathbb{M}$ will transpose to a structure preserving functor $\mathbb{S} \longrightarrow \mathbb{M}^{\mathbb{P}}$. When the category $\mathbb{M}$, measuring satisfaction, is taken to be the category Set of sets and functions, $\models$ will thus amount to a *model* of $\mathbb{S}$ in the universe $\mathsf{Set}^{\mathbb{P}}$ of sets varying from state to state in $\mathbb{P}$. A logically inclined reader may be amused to spend a moment unfolding the definition of adaptation morphisms in this model theoretic context, and confincing herself that such morphisms indeed preserve the given sense in which a program satisfies specification, or improve it along the suitable homomorphisms of models.

In any case, the naturality condition on the third component of the adaptation morphisms as defined the preceding section can now be expressed precisely, on the diagram displaying the involved composite functors.

$$\begin{tikzcd}
\mathbb{P}' \times \mathbb{S} \arrow[r, "id\times f_S"] \arrow[d, "f_P\times id"'] \arrow[dr, phantom, "\overset{f_{\models}}{\nearrow}"] & \mathbb{P}' \times \mathbb{S}' \arrow[d, "\models'"] \\
\mathbb{P} \times \mathbb{S} \arrow[r, "\models"'] & \mathbb{M}
\end{tikzcd}$$

3 Adaptive interpreter as coalgebra

While the described category of contracts, implemented and supported by suitable tools, provides the structural framework for software adaptation, it still does not provide a special handle for automated, on-the-fly adaptation and reconfiguration. The dynamics of self-adaptive software requires an additional dimension, to be added to in the actual implementation of the specification carrying modules. The main issue thus remains: *how to implement an adaptive interpreter, able to compute with self-adaptive, specification carrying modules?*

Given a contract $\langle P, \models, S\rangle$, the adaptive interpreter should be able to:

- evaluate P,
- test whether the results satisfy S,
- adapt P, or assist program transformation $P \longleftarrow P'$,
- support refinement $S \longrightarrow S'$.

In a standard setting, the denotation of a program P is a function $p : A \longrightarrow B$, where A and B are the types of the input and the output data, respectively. An adapted program P' will yield a function $p' : A \longrightarrow B$ (where we are ignoring, for simplicity, the fact that the data types A and B can be refined). If adaptation is viewed as a computational *process*, all the instances of an adapted function that may arise can be captured in the form

$$\widetilde{p} : \Sigma \times A \longrightarrow B$$

where Σ is the carrier of adaptation, probably a monoid, or a partial order. The *stages* of the adaptation $\sigma, \sigma' \ldots \in \Sigma$ can be associated with the successive refinements $S, S' \ldots$ of the specification of the adaptive program, and will be derived from them by one of the model theoretic methods developed for this purpose (i.e. as the "worlds", or the forcing conditions induced by the refinement).[5]

[5] The stages of adaptation should not be confused with the computational states, through which the execution of a program leads. The execution runs of a program from state to state are, in a sense, orthogonal to its adaptation steps from stage to stage.

All the instances $p, p' \ldots : A \longrightarrow B$ of the adaptive function will now arise by evaluating $\widetilde{p}$ at the corresponding stages $\sigma, \sigma' \ldots \in \Sigma$, *viz*

$$\begin{aligned} p(x) &= \widetilde{p}(\sigma, x) \\ p'(x) &= \widetilde{p}(\sigma', x) \\ &\cdots \end{aligned}$$

In this way, the process of adaptation is thus beginning to look like a rudimentary dynamic system. The process of *self*-adaptation will, of course, be a system with the feedback

$$\widehat{p} : \Sigma \times A \longrightarrow B \times \Sigma$$

computing at each stage $\sigma \in \Sigma$, and for each input $x \in A$ not only the output $y = \widehat{p}_0(\sigma, x) \in B$, but also the next stage $\sigma' = \widehat{p}_1(\sigma, x) \in \Sigma$, with a better adapted function $p'(x) = \widehat{p}(\sigma', x)$. Extended in this way along the coordinate of adaptation, the specification carrying programs viewed as contracts come in the form

$$\widehat{\models} : \Sigma \times \mathbb{S} \times \mathbb{P} \longrightarrow \mathbb{M} \times \Sigma$$

The predictable adaptation stages are structured in Σ and controlled by the resumption component of $\widehat{\models}$. Alternatively, automated adaptation steps can be encapsulated in specifications and programs themselves, by individually extending the particular functions from stage to stage.

Independently on the level on which it may be captured, the denotation of a self-adaptive function will in any case be a transducer $\widehat{p}$. Transposing it into a *coalgebra*

$$\widehat{p} : \Sigma \longrightarrow (B \times \Sigma)^A$$

brings with it the advantage that the behaviour preserving maps now arise automatically, as coalgebra homomorphisms. (Instructive examples and explanations of this phenomenon can be found, e.g. in [20].) But even more importantly, it allows a considerably more realistic the picture, since it also allows introducing various *computational monads* T on the scene. A coalgebra in the form

$$\widehat{p}_T : \Sigma \longrightarrow (T(B \times \Sigma))^A$$

captures an adaptive family of computations involving any of the wide range of features (nondeterminism, exceptions, continuations...) expressible by monads [10].

In any case, combining monads and coalgebra will ensure a solid semantical foundation not just for adaptive interpreters, but also for implementing the design environments for self-adaptive software. Explaining either of these theories is far beyond our scope here, but monads seem to have been established as a part of the standard toolkit of functional programmers, and the material about them abunds. Some coalgebraic techniques for implementing processes have been presented in [5, 15, 16].

References

1. R.-J. Back and J. Von Wright, *The Refinement Calculus: A Systematic Introduction.* (Springer 1998)
2. M. Barr, *$*$-Autonomous Categories.* Lecture Notes in Mathematics 752 (Springer, 1979)
3. J.A. Goguen and R.M. Burstall, Institutions: abstract model theory for specifications and programming. *J. of the A.C.M.* 39(1992) 95–146
4. Y. Gurevich, Evolving algebras 1993: Lipari guide. In: *Specification and Validation Methods*, ed. E. Börger, (Claredon Press 1995) 9–37
5. B. Jacobs, Coalgebraic specifications and models of deterministic hybrid systems. In: *Proc. AMAST*, ed. M. Nivat, Springer Lect. Notes in Comp. Sci. 1101(1996) 520–535
6. Y. Lafont and T. Streicher, Games semantics for linear logic. *Proc 6th LICS Conf.* (IEEE 1991) 43–49
7. F.W. Lawvere, *Functorial Semantics of Algebraic Theories.* Thesis (Columbia University, 1963)
8. M. Makkai and R. Paré, *Accessible Categories: The Foundations of Categorical Model Theory.* Contemporary Mathematics 104 (AMS 1989)
9. M. Makkai and G. Reyes, *First Order Categorical Logic.* Lecture Notes in Mathematics 611 (Springer 1977)
10. E. Moggi, Notions of computation and monads. *Information and Computation* 1993
11. C. Morgan, *Programming from Specifications.* (Prentice-Hall 1990)
12. G.C. Necula, *Compiling with Proofs.* Thesis (CMU 1998)
13. V.C.V. de Paiva, The Dialectica categories. In: *Categories in Computer Science and Logic*, J. Gray and A. Scedrov, eds., *Contemp. Math.* 92 (Amer. Math. Soc., 1989) 47–62
14. D. Pavlovic, Chu I: cofree equivalences, dualities and $*$-autonomous categories. *Math. Structures in Comp. Sci.* 7(1997) 49–73
15. D. Pavlovic, Guarded induction on final coalgebras. *E. Notes in Theor. Comp. Sci.* 11(1998) 143–160
16. D. Pavlovic and M. Escardo, Calulus in coinductive form. *Proc 13th LICS Conf.* (IEEE 1998) 408–417
17. D. Pavlovic, Semantics of first order parametric specifications. in: *Formal Methods '99*, J. Woodcock and J. Wing, eds., Springer Lect. Notes in Comp. Sci. 1708(1999) 155–172
18. G. Plotkin, *Structural Operational Semantics.* Lecture Notes DAIMI-FN 19(1981)
19. V. Pratt, Chu spaces and their interpretation as concurrent objects. In: *Computer Science Today: Recent Trends and Developments*, ed. J. van Leeuwen, Springer Lect. Notes in Comp. Sci. 1000(1995)
20. J.J.M.M. Rutten, Universal coalgebra: a theory of systems. To appear in *Theoret. Comput. Sci.*
21. J. Sztipanivits, G. Karsai and T. Bapty, Self-adaptive software for signal processing. *Comm. of the ACM* 41/5(1998) 66–73
22. D. Turi and G.D. Plotkin, Towards a mathematical operational semantics. In: *Proc. 12th LICS Conf.* (IEEE 1997) 280–291
23. D.R. Smith, Derivation of parallel sorting algorithms. In: *Parallel Algorithm Derivation and Program Transformation*, eds. R. Paige et al. (Kluwer 1993) 55–69
24. D.R. Smith, Towards a classification approach to design. In: *Proc. 5th AMAST*, Springer Lect. Notes in Comp. Sci. 1101(1996) 62–84

25. G. Winskel and M. Nielsen, Presheaves as transition systems. In: *Proc. of POMIV 96*, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* (AMS 1997) 129–140