Towards a Programming Paradigm for Control Systems with High Levels of Existential Autonomy

Eric Nivel[1] and Kristinn R. Thórisson[1,2]

[1] Center for Analysis and Design of Intelligent Agents / School of Computer Science, Reykjavik University, Venus, Menntavegur 1, Reykjavik, Iceland
[2] Icelandic Institute for Intelligent Machines, 2.h. Uranus, Menntavegur 1, Reykjavik, Iceland
{eric.nivel,thorisson}@gmail.com

Abstract. Systems intended to operate in dynamic, complex environments – without intervention from their designers or significant amounts of domain-dependent information provided at design time – must be equipped with a sufficient level of *existential autonomy*. This feature of naturally intelligent systems has largely been missing from cognitive architectures created to date, due in part to the fact that high levels of existential autonomy require systems to program themselves; good principles for self-programming have remained elusive. Achieving this with the major programming methodologies in use today is not likely, as these are without exception designed to be used by the human mind: Producing self-programming systems that can grow from first principles using these therefore requires first solving the AI problem itself – the very problem we are trying to solve. Advances in existential autonomy call for a *new programming paradigm,* with self-programming squarely at its center. The principles of such a paradigm are likely to be fundamentally different from prevailing approaches; among the desired features for a programming language designed for automatic self-programming are (a) support for autonomous knowledge acquisition, (b) real-time and any-time operation, (c) reflectivity, and (d) massive parallelization. With these and other requirements guiding our work, we have created a programming paradigm and language called *Replicode*. Here we discuss the reasoning behind our approach and the main motivations and features that set this work from apart from prior approaches.

1 Introduction

Future artificially *generally* intelligent (AGI) systems, to deserve the label, must be able to learn a wide variety of tasks and adapt to a wide range of conditions, none of which can be known at design time. This requires some minimum level of e*xistential autonomy* – the ability of a system to act without dependence on explicit outside assistance, whether from a human teacher or, in the case of AGI, the system designer. Achieving existential autonomy calls for unplanned changes to a system's own cognitive structures. Too be capable of cognitive growth – whether measured in minutes, days, years, or decades – such systems must thus ultimately be able to *program themselves*. Provided with (minimal) bootstrap knowledge, we target systems that operate in in a way that, while initially limited, are capable of facing novel situations in their environment – simple at first – and grow their intelligence as experience accumulates. Given scalable principles, a system will continuously grow to ever-increasing levels of cognitive sophistication.

In light of this goal, do any existing programming environments, paradigms, or languages allow us to get started on building such systems? After thorough investigation, having carefully considered an array of existing alternatives (for example Schiffel & Thielscher 2006, Schmidhuber 2004), our conclusion is that some, sometimes many, features of all prevailing programming paradigms and languages makes them unsuited to achieve the level of self-programming abilities required for the kind of AGIs we envision. Drescher (1991) proposes a paradigm that at first glance has some similarities with ours. Important and fundamental differences exist, however, which prevent us from building directly on his work – three of which we will mention here. First, operational reflectivity is not enforced in his approach, and thus his system cannot model its own operation. Second, as Drescher does not assume resource boundedness, no mechanisms for systems adapting to resource scarcity is provided. Third, his proposed schema control mechanism is inappropriate for real-time parallel operation. In fact, because all mainstream programming languages are created for human beings, their *semantic complexity is too high* to allow the kind of low-level self-programming needed for the kind of cognitive growth necessary for realizing truly adaptive systems (Thórisson 2012). For this reason no human-targeted programming languages provide a suitable foundation for systems capable of cognitive growth, which means a new paradigm must be developed.

This paper introduces a new programming paradigm and language – *Replicode* – for building control systems that can autonomously accumulate and revise knowledge from their own experience, under constraints of limited time and computational resources (Wang 2006), through *self-programming*. Departing from traditional development methodologies that rely on human-crafted code, we follow the path of constructivist development (Thórisson 2012), which delegates the construction of the system in large part to the system itself. In our approach knowledge consists thus primarily of learned executable code. Replicode is an interpreted language designed to acquire, execute and revise vast amounts of fine-grained models in parallel and in (soft) real-time. The interpreter of Replicode – the *executive* – is a distributed virtual machine that runs on various hardware configurations, from laptops to clusters of computers. A thorough description of the language is beyond our scope here (see Nivel & Thórisson 2013 for the full language specification). Here we focus on *how three key requirements* affect the design of the Replicode programming language, namely *automatic acquisition of knowledge*, *real-time any-time operation*, and *adaptation*. First we present key motivations and requirements, alongside the resulting design decisions. Then follows a quick overview of the main Replicode features, memory organization, and rules for governing model execution. The two last sections describe briefly how learning is implemented in Replicode-based systems and the control mechanisms for supporting adaptation to resource scarcity.

2 Key Requirements for Existential Autonomy

Autonomous expansion of a system's skill repertoire means the system must be outfitted with a principled way to govern the integration of new knowledge and skills. The rationale is threefold.

First, as informationally rich open-ended complex environments cannot be axiomatized beforehand, very few assumptions can be made about the knowledge that a system may have to acquire: The information available to be perceived and the behavioral challenges that the system's future environment(s) may present, can be of potentially multiple types, and the specifics of these types cannot be known beforehand. The generality of the system is thus constrained by its ability to deal with this potentially large set of information and skills in a uniform manner: the less specific to particular types of information the knowledge representation is, the greater the system's potential for being general. This is the requirement of *knowledge representation uniformity*.

Second, systems that improve all the time, while "on the job", must maintain and expand their knowledge incrementally and continuously, in order to discard faulty models as early as possible (before knowledge is built up on top of them – which by extension would also be faulty). To do this the system must perform frequent reality checks on its modeling (understanding) of the world, as the opportunities may arise, at any time, as the system steadily builds up its knowledge. This requirement directs us towards *low knowledge representation* ("peewee") *granularity*; fine-grained knowledge permits higher "clock rate", with smaller incremental checks and changes.

Third, as the systems we envision should perform in (soft) real-time and any-time, knowledge integration speed is of the essence – in other words, the processes that perform integration shall be as simple and efficient as possible. What we aim for here is *knowledge representation plasticity*: A system must be able to add and remove knowledge very quickly and very often, irrespective of knowledge semantics. The knowledge must also have a high degree of composability, giving the system an ability to easily construct knew knowledge from existing knowledge.

Real-time Control. To act meaningfully in real-time, control systems must anticipate the behavior of the controlled entities. In other words, the system must make predictions, based on its knowledge acquired to date. To apply continuously to any and all actions of the system, at any level of detail, predictions must essentially be produced all the time, as an integral part of the system's cognitive operation, and as quickly as possible, to be ahead of the reality to which they apply. As the system is doing this, however, it must also keep acting on its environment to satisfy its goals and constraints. This means that sub-goals should be produced as soon as top-level goals are produced. This gives us two sub-requirements.

First, reality checks can only be performed by monitoring the outcome of predictions: this is how the reliability of the system's knowledge can be assessed – the quality cannot be judged based solely on the results of internal abduction; abduction can only produce a set of possible sub-goals, from which the system must then select, and discard the rest. As it does so, a potential problem is that the search (which sub-goal to choose) may be faulty – not the knowledge. Therefore if we had two kinds of models for representing predictive knowledge and prescriptive knowledge, only the predictive models could be maintained at any point in time. Thus, it follows that both deduction and abduction should be supported in a single model. This is the requirement for a *unified representation for abduction and deduction processes*.

Second, deduction and abduction should proceed concurrently at runtime: the control system cannot be put on hold achieving its goals while predictions are produced, because then it will lag behind its controlled entities; reciprocally, the system cannot wait to monitor the outcome of predictions to act in due time. This is the requirement of *simultaneous execution of abduction and deduction.*

Real-time Learning. We target systems to learn as they act, and to use newly-acquired knowledge as soon as possible, as the need may arise during the system's operation. A programming language must not restrict these in such a way that they are mutually exclusive at runtime. Here again, a *fine granularity* is of the essence. If the granularity of knowledge representation was so coarse-grain as to encode large complex algorithms, instead of e.g. a single simple rule, then it will more difficult for the system to assess the model's performance, as increased complexity would mean that the encoded knowledge would cover broader and richer situations – which would increase the complexity of reality checks, thus degrading the system‘s responsiveness.

Adaptation. In informationally rich open-ended environments conditions may arise at any time that an intelligent system is not equipped to handle, or that the system may only partially be able to address. A system which cannot prioritize its tasks according to the time and CPU power available is doomed in such conditions. As this kind of adaptability is critical for achieving experiential autonomy, methods for controlling resource expenditure is a hard requirement for such systems. Advanced levels of such resource control call for fully-fledged introspective capabilities; this is what we aim for in our work. We propose the following four principles to achieve this.

First, the system's executive should periodically publish assessments of its own performance (for example, the time it takes to execute a unit of knowledge, or the average lag behind deadlines).

Second, the executive should expose several control parameters that allow a system-wide tuning of its various computation strategies.

Third, operational reflectivity should be supported at every level of abstraction in the language, which means that every operation the system performs is reflected as a first-class (internal) input, allowing the modeling of causal relationships between strategy tuning and its effects (performance assessment).

Last, a programming language should also provide a way to reduce the amount of inputs to process (i.e. attention mechanisms) that discards irrelevant inputs.

3 Overview of Replicode

Taking a symbolic approach, Replicode[1] is based on pattern-matching and is data-driven: as input terms become available the executive continually attempts to match

[1] The Replicode language source code is available from `http://cadia.ru.is/svn/repos/replicode`

them to patterns; and when succeeding some computation is scheduled for execution, possibly resulting in the production of more input terms. Replicode is operationally reflective; the trace of every operation it performs is injected in memory as an internal input to allow a system to model its own operation, a necessary prerequisite for enabling some degree of self-control – and thus of adaptation. Replicode is goal-driven: the programmer defines fixed top-level goals (called drives) that initiate abduction and eventually the effective acting on the environment (when the system commits to a goal containing a command on an effector, this command is executed).

Replicode meets the requirement of uniform knowledge representation by encoding all knowledge as executable models, forming a highly dynamic hierarchy that models the behavior of entities in the environment – including the system's internal behavior. The hierarchy is expanded incrementally: fine-grained models are added continuously as the system interacts in its environment, and said models are also deleted as soon as enough counter-evidences of their reliability is observed.

The execution of a single model produces predictions, given some observed facts, and at the same time generates sub-goals, given some top-level goal(s). In essence, the model hierarchy is thus traversed by two simultaneous flows of information: a flow of predictions, bottom-up (assuming the inputs come from the bottom) and a flow of goals, top-down (assuming the super-goals come from the top and the commands to the effectors are located on the bottom). This paves the way for a system implemented in Replicode to drive its behavior in an anticipatory fashion to learn and act simultaneously, and achieve real-time and any-time performance.

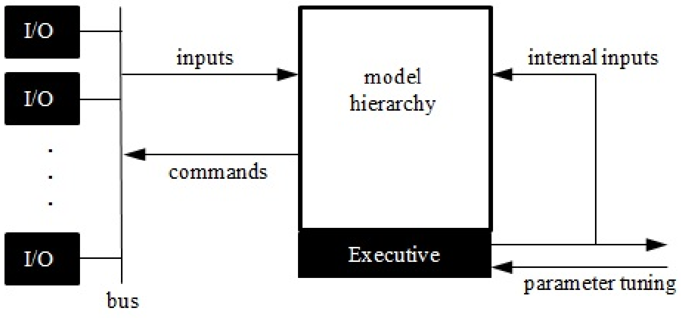

Fig. 1. Overview of a Replicode-based System

A system controls its environment or entities situated therein via dedicated sub-systems (I/O boxes) such as machine vision equipment or any kind of particular device driver. Notice that a system can be controlled by another one by means of the former's internal inputs and control parameters.

Replicode relies on a *real-valued temporal term logic*. Terms encode facts (or absence thereof) that hold within some time interval (specified in microseconds) and with a certain confidence value (in [0,1]). Goals and predictions are also encoded as facts: the confidence value carried by a goal stands for its likelihood to succeed, whereas the confidence value of a prediction stands for its likelihood to come true.

For goals, the time interval specifies the lower and upper deadlines in between which the goal shall be achieved, whereas for predictions, the time interval defines the time segment when evidences or counter-evidences shall be checked to assess the accuracy of the predictions.

4 Model Execution and Memory Management

The memory in Replicode is organized in groups of objects (for example models and facts) used to isolate subsets of knowledge. Each object in a group is qualified by three control values, saliency, activation and resilience. The saliency determines the eligibility of an object to be an input for some executable object (for example models[2]), the activation determines the eligibility of some executable object to process any input and the resilience defines the object‘s time to live. A group defines several control parameters, of which the most essential are (a) a *saliency threshold* (any object whose saliency is below the threshold becomes invisible to executable objects) and, (b) an a*ctivation threshold* (any executable object whose activation is below the threshold will not be allowed to process any input).

- Replicode models consists of two patterns (a left-side pattern, or l-pattern and a right-side one, or r-pattern). When an input matches an l-pattern, a prediction patterned after the r-pattern is injected in the memory (deduction, or forward chaining). Reciprocally, when a goal matches an r-pattern, a sub-goal patterned after the l-patterned is injected in the memory (abduction, or backward chaining). A system can thus be considered a dynamic model hierarchy based on pattern affordances.
- Models carry two specific control values, a success rate and an evidence count. The success rate is the number of positive evidences (the number of times the model predicted correctly) divided by the total number of evidences (the number of times the model tried to predict). When a model produces an output from an input it computes the confidence value of the output as the product of the confidence value of the input and the success rate of the model. The confidence value carried by an object is assigned to its saliency. It follows that, in the model hierarchy, information resulting from traversal of many models will likely be less salient than information resulting from traversing fewer models (assuming identical success rates in both cases). If we picture predictions flowing bottom-up and goals top-down (see figure 2 below), then only predictions produced by the best models will reach the top of the hierarchy, whereas only goals produced by the best models will end up at the bottom where commands are issued to the actuators of the system. In addition, the success rate of a model is used as its activation value: as a result bad performers will be eventually deactivated.
- The programmer initially defines top-level goals (drives) that subsequently are periodically injected into the hierarchy by the system. These drives are fixed and

[2] Replicode defines other types of executable code (programs). These are mere infrastructure constructs and are not essential for the present discussion.

encode the objectives and constraints of a given system – representing its reasons for being, as it were. By tuning their saliency values, the designer or the system itself can activate or deactivate some of these drives. For example, when the resources are becoming dangerously deficient, the system can choose to ignore the least essential (salient) drives – provided it has been given (or has acquired) the knowledge to do so.

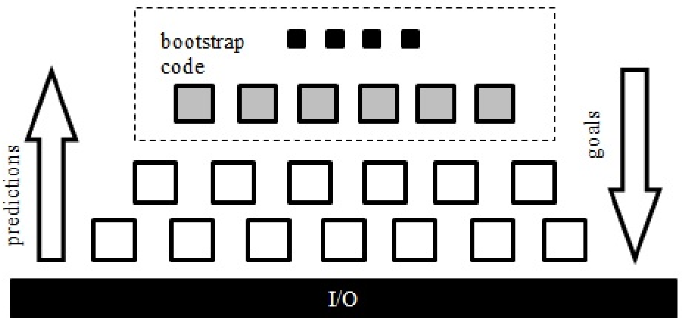

Fig. 2. Model Hierarchy

The bootstrap code is composed of drives (black boxes) and top-level models (grey boxes) that give the system the initial knowledge to satisfy its drives. New models are dynamically added to the hierarchy (white boxes) when the system learns how to predict inputs for the top-level models or to solve goals produced by said top-level models - or deleted from it if they turn out to be poor predictors. Only the best models will propagate predictions bottom-up from the inputs received from the I/O devices and only the best models will generate sub-goals top-down, eventually issuing commands to the I/O devices.

In Replicode backward chaining provides a way to perform abductions, i.e. to derive sub-goals given an input goal. It may turn out that given such an input goal, several sub-goals can be produced, each representing a particular way to achieve the super-goal. In addition, several sub-goals resulting from several super-goals may target conflicting states. These situations call for a control policy over the search: before committing to any sub-goal, the executive simulates their respective possible outcomes, ranks them and commits to the best ones. The simulation phase is akin to a parallel breadth-first search and proceeds as follows. The executive defines a parameter called the simulation time horizon. When a goal matches an r-pattern a simulated sub-goal is produced which triggers the production of more sub-goals. At half the time horizon, backward chaining stops and simulated predictions are produced (these predict states targeted by the deepest simulated sub-goals). Such predictions flow up in the hierarchy for another half the time horizon. At the time horizon, and on the basis of the simulated predictions, branches of sub-goal productions are evaluated and the best ones selected for commitment (see Figure 3 below for an example).

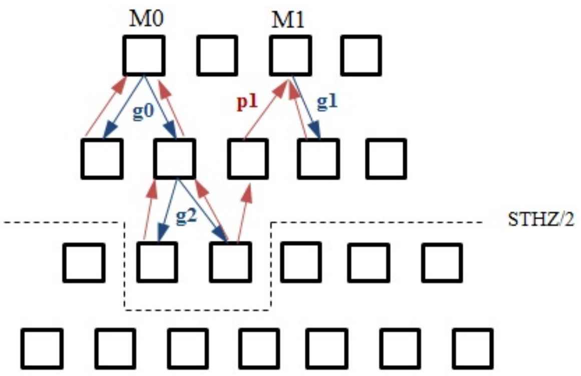

Fig. 3. Simulation

Consider a model M0 producing a goal g0 at time t0.Simulated sub-goals (blue arrows) will be produced by relevant models until time t0+STHZ/2 where STHZ stands for simulation time horizon. At this time, simulated predictions will start to flow upward (red arrows) and at time t0+STHZ, models having produced goals (like M0 or M1) will assess the simulated predictions for committing to the best sub-goal(s). For example, it can turn out that committing to g2 for achieving g1 will predictably (p1) conflict with the resolution of another goal (g1). In that case, and assuming g0 is less important than g1, M0 will not commit to g2 and will chose another sub-goal (if available) whereas M1 will commit to g1.

5 Learning

The formation of new models relies on a basic heuristic: *temporal precedence means causation*. The good news is that, at least in our pilot tests, temporal precedence does actually indicate causation, and some models will capture such causal relationships correctly. The bad news is that this approach leads to the construction of many faulty models. To address this we have implemented a model revision process that manages faulty model deletion. Learning in a Replicode-based system results from the interplay of the continuous and concurrent processes of model acquisition and revision. In that sense, Replicode implements learning that is *incremental* (the model hierarchy is built progressively as experience accumulates), *continuous* (the system never stops learning as long as it faces novelty), and *real-time* (the system learns on the fly while acting on its environment).

New models are built based on the exploitation of time-bound input buffers, which can be considered a kind of short-term memory. The buffers are allocated for each goal and predictions the system produces: the executive will attempt to model the success of a goal or the failure of a prediction provided these events have not been predicted by existing models. In addition, the executive will also attempt to model the change of a given state, provided said change has not been predicted. In any case, the executive takes every input from a buffer and turns it into an l-pattern paired with the target (the r-pattern), that is the goal, prediction or state it focuses on.

Each time a model produces a prediction, the executive monitors all inputs for a confirmation of the predicted fact (with respect to the predicted time interval). When the outcome is positive the success rate of the model will increase. If the predicted fact is not observed in due time or if a counter evidence is observed, then the success rate will decrease. If the success rate gets under the group‘s activation threshold, then the model is deactivated.[3]

6 Control Mechanisms

Given its limited resources, an existentially autonomous system must direct its computation at the most interesting inputs. Some advanced designs have been proposed for such an attention mechanism (see Helgason et al. 2012, for example), however Replicode uses a simpler scheme. An input is said interesting if (a) it shares at least one variable with a goal the system currently pursues, (b) it shares at least one variable with a prediction the system is monitoring or, (c) it indicates a change of a state. This is to say that the focus is driven top-down (triggered by goals and predictions) and also bottom-up (detection of state changes).

There are several other ways to control the processing in Replicode. These are:

- *Adjusting the thresholds of the primary and secondary groups*: this has an immediate effect on the number of goals and predictions that constitute inputs for models, and thus affects the processing load.
- *Adjusting the saliency of drives*: discarding less important drives will prune the (top-down) flow of goals.
- *Adjusting the time horizons for simulation*: this will narrow the breadth of the search, leading to more responsiveness, at the expense of discarding valuable alternatives perhaps too early – the system will prefer well known and reliable solutions over less proven ones that might have been more efficient.
- *Adjusting the time horizon for the acquisition of models*: by doing so, less model candidates will be inferred, thus reducing the size of the model hierarchy, at the expense of making the system more "short-sighted".

7 Conclusion and Future Work

Replicode is a programming paradigm designed to meet a stringent set of requirements derived from the goal of producing systems exhibiting high levels of existential autonomy. Replicode is intended for creating model-based control systems that control other systems in a general fashion. Given (minimal) bootstrap code, Replicode systems are meant to continuously improve their understanding of their operating environment through incremental knowledge accumulation via the generation of models that describe observed causal relationships, both in the environment and within themselves.

[3] Some advanced mechansims to reactivate models have been implemented, but these are out of the scope of this paper.

The language has already been implemented and tested on pilot systems, and proven to solve all the requirements; it nevertheless is still in early phases of development. Features that we intend to address in the coming years include more sophisticated inferencing, like similarity and equivalence identification, and the ability to make analogies. A lack of analogy capabilities makes a system singularly dependent on observation of explicit causation in its environment; advanced inferencing abilities would allow the system to extract more knowledge from that same information.

Acknowledgments. This work has been supported in part by the EU-funded project HUMANOBS: Humanoids That Learn Socio-Communicative Skills Through Observation, contract no. FP7-STReP-231453 (www.humanobs.org), and by grants from Rannís, Iceland. The authors would like to thank the HUMANOBS Consortium for valuable discussions and ideas, which have greatly benefited this work.

References

Drescher, G.L.: Made-up minds - a constructivist approach to artificial intelligence, pp. I–X, 1–220. MIT Press (1991) ISBN 978-0-262-04120-1

Helgason, H.P., Nivel, E., Thórisson, K.R.: On Attention Mechanisms for AGI Architectures: A Design Proposal. In: Bach, J., Goertzel, B., Iklé, M. (eds.) AGI 2012. LNCS, vol. 7716, pp. 89–98. Springer, Heidelberg (2012)

Nivel, E., Thórisson, K.R.: Replicode: A Constructivist Programming Paradigm and Language. Reykjavik University Tech Report RUTR-SCS13001 (2013)

Schiffel, S., Thielscher, M.: Reconciling Situation Calculus and Fluent Calculus. In: Proc. of the 21st National Conference on Artificial Intelligence and the 18th Innovative Applications of Artificial Intelligence Conference, AAAI 2006 (2006)

Schmidhuber, J.: Optimal Ordered Problem Solver. Machine Learning 54, 211–254 (2004)

Thórisson, K.R.: A New Constructivist AI: From Manual Construction to Self-Constructing Systems. In: Wang, Goertzel (eds.) Theoretical Foundations of Artificial General Intelligence. Atlantis Thinking Machines, vol. 4, pp. 145–171 (2012)

Wang, P.: Rigid Flexibility – The Logic of Intelligence. Applied Logic Series, vol. 34. Springer (2006)