Enzyme-like replication *de novo* in a micro-controller environment

Uwe Tangen

June 10, 2008

Biomolecular Information Processing, Ruhr-Universität Bochum, c/o BMZ Otto-Hahn-Str. 15, 44227 Dortmund Germany, tel. +49 9742 6422

uwe.tangen@biomip.ruhr-uni-bochum.de

[http://www.biomip.de](http://www.biomip.de)

Keywords:

replication system; emergence of replication; evolving software; evolvable micro-controller; evolutionary robustness

Abstract

The wish to start evolution from scratch inside a computer-memory is as old as computers (5). Popularized by the software Tierra (42) and Avida (2) where a Cambrian-environment was subsumed few tried to evolve replicators from scratch.

The article presented, shows the creation of viable computer programs *de novo* without specific instantiation, simply starting with random bit-sequences. In addition, for the first time these programs are no self-replicators but much more physically catalyst-like. The micro-controllers used are the end-point of a long series of simplifications starting with commercial micro-controllers, still universal, and with an assembler translating man-made software into machine-code allowing to program the evolving system. Furthermore, the instruction-set of the software is sufficiently powerful to allow external signals to be processed.

This work now bridges the gap between computer-simulations of abiotic-environments, the development of evolvable hardware, the creation of artificial cell systems on the one hand and the experimental research of creation of life from biological precursors on the other hand.
1 Introduction

The transition between the prebiotic world and the advent of the first living cells still is a miracle not at all understood. Today after more then thirty years of high-performance computing not even computer-simulations can conceivably demonstrate such a transition. Though we now have considerable knowledge of the molecular details of life as it is today, experimental studies (35; 28), which were not able to create life from scratch in a laboratory – still have not been superseded. Several initiatives and programs are now underway to show experimentally how the first cellular entities could have been possible. Apart from the project PACE (PACE http://www.istpace.org) most of these initiatives require the existence of enzymes. Minimal cells (19) or synthetic biology (c.f. Synthetic Biology http://en.wikipedia.org/wiki/Synthetic_biology) can not ask the question of the transition between the prebiotic and biotic world, enzymes have not been part of the prebiotic world. Even the artificial cells envisaged in the PACE-project and follow-ups are dependent on specially designed (electronic-) environments to allow the creation of these chemical artifacts. Worse, theoretical studies had to make a lot of high-level assumptions which could not help the experimentalists to make better decisive experiments.

The first serious scientific modeling of self-reproducing systems has been devised with replicating numbers (4) and trying to deal with the problem of an automaton being able to self-replicate (53). Life as a game marked the simplest model in using cellular automaton platforms concerning artificial self-reproducing systems (7). In the seventies Holland, the inventor of Genetic Algorithms and Classifier Systems (22; 24), created $\alpha$-universe (23) to model the genetic replication apparatus in a one-dimensional simple string processing system. McMullin later could show that side-reactions destroy the self-reproducing capabilities of entities in this model-system (34).

The first observed computer-viruses sparked research in evolving software which became prominent with the game Corewars (10), a collection of programs working in parallel on a one-dimensional circular string as was the case with Holland’s $\alpha$-universe. Coreworld (40) and Tierra (42) followed and could at least show phenotypic behaviors found in the Cambrian Explosion.

From a biological point of view, experimental studies in the sixties and seventies (45) led to the development of the quasi-species theory (13), showing that replication errors drastically limit
the amount of information which can be stabilized over generations in replicating systems (error
threshold). Furthermore, Artificial Chemistry took a more operator-based view of chemistry on the
self-replicating computer entities \((32; 14)\), see \((11)\) for a review.

All these experimental and theoretical studies on replication and evolution culminated in the
endeavor to really build artificial cells, see ProtoCells \(\text{http://protocell.org}\) for an overview on
some of these projects. Especially the PACE project asks the question how computer-science can
contribute to information processing in primitive artificial cellular organisms. One avenue in getting
artificial cells realized is described in \((41; 39)\). The central question in artificial cell research is how
hereditary information in these cells can be conserved over time and make an essential contribution
to the survival and robustness of these primitive entities.

The work presented here will show three achievements: a set of universal micro-controllers,
though simple, showing the software executed by these controllers being able to stably sustain a
software-replication system. The second and most important achievement concerns the emergence
of these replicator-programs without seeding the system other than with random-numbers. The
third achievement is more of an indirect nature, namely, the simplification of the system – which
only was possible using reasonable physical assumptions (see embodiment literature, e.g. \((38; 44)\) –
now making the mapping to real biochemical experiments easier. Especially the dependency of the
dynamic properties from the used simulation-parameters give important insights into the molecular
environment.\(^1\)

2 The evolutionary setup

The work is a follow up of \((48)\), which shows and analyze a series of further simplified micro-
controllers. What is not clear, is, at which level of simplification, the number of bits necessary for
the emergence of replication is minimal. Our desire to utilize this micro-controller based model as
an abstract description of (organic-) chemistry when experimenting with prebiotic systems hinders

\(^1\)Concerning the nomenclature of this article: technico termini from computer-science and biochemistry are used
throughout, e.g. a genome here means a piece of software if not otherwise noted. This mixing should make clear the
similarity of concepts and ease the bridging between the two fundamentally different disciplines. In any ambiguous
case the connotation meant is specifically mentioned.
us to take arbitrary commercial micro-controllers - they are way too complex. The creation of one instruction \texttt{REPLICATE} in the instruction set on the other hand is too simple an approach, for a further discussion see (46). Another question is how many instructions are needed to execute a single replication cycle. More steps per cycle increase the probability that perturbations introduce errors in the replication process. A further point of consideration is ambiguity. The more different views on a program exist (see the frame-shift problems) the higher the dimensionality of the search space. Though in principle the number of possible solutions should increase also, the number of non-solutions increases much faster (curse of dimensionality, see the $2^n$- increase in search space and only $\sim 2^n$- increase of useful functionality when evolving digital functions (50)).

2.1 The model in a nut-shell

Micro-controllers can interact with each other, see section 2.2 for the spatial setup. Interaction is done via a recognition procedure. Each micro-controller can exhibit a recognition pattern, which currently is the first occurring concatenated sequence of \texttt{Site}-instructions in the program, see table 2. The attachment procedure tries to find a micro-controller in the vicinity with an appropriate recognition-site and puts that micro-controller’s address into the reading- or writing-slot, see figure 1, depending on the instruction which initiated the attachment procedure. The second recognition-based interaction is realized when transferring program control from the active micro-controller towards another micro-controller (equivalent to a sub-routine call, see instruction \texttt{Call} in table 2). The third recognition event is a register access where the foreign micro-controllers’ accumulator acts as a local register, see section 2.3.7.

The micro-controller has a Harvard-architecture, with input-ports – registers or an read-attached program and with output-ports – registers or the write-attached program of a further micro-controller. Each instruction has three parts, the cargo-, conditional- and special-part. The cargo-part is the parameter for the instruction in the special-part, which is executed if the conditional-part allows it, see figure 2. Several instructions are available and described in table 2.

A further bit is needed to allow for conditional execution. The instructions in row J1 of table 6 are also executed if row J2 is specified and the ZF-flag (accumulator value 0) active or if row J3 is
Figure 1: How micro-controllers interact with each other. Each interaction is realized via a recognition procedure with s concatenated bases (see Site instruction in table 2 and section 2.3.3). Two attachment-slots are available per micro-controller. Micro-controllers attached to the reading-slot (see Load instruction in table 2) serve as templates and micro-controllers attached to the writing-slot (see Store instruction in table 2) serve as products. Flags in other micro-controllers can be set if they are attached to the reading-slot. Standard registers are accumulators from other micro-controllers. The register-address is the recognition-site a neighboring micro-controller exhibits.
Figure 2: Structure of the micro-controller. The micro-controller uses 2 bits of the special instruction section (SP). The condition-part (C) is 2-bits wide. The width of the cargo depends on the experiments done. Usually $n = 2$ using quaternary encoding, see section 2.3.5. This gives a 6-bit micro-controller in the simplest case. The flags available are described in table 4. The zero-flag (ZF) and PF1-flag (PF1) are used for conditional execution, see table 3. Input and output stems from or is sent to other micro-controllers.
Table 1: Instruction set used. The possible flags are explained in table 4. The left instruction table concerns constant program lengths and the right instruction table is used with variable length sequences. The default instruction for (J0:S0, left) is Cycle. This instruction sets the instruction pointer to address zero. The instruction (J0:S0, left) can be changed by toggling the _CYCLE_-flag with the SetFB instruction. In some variants of the instruction set the instruction SetFB has been replaced by a Goto-instruction.

Table: Instruction set

<table>
<thead>
<tr>
<th>Execute</th>
<th>S0</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
</tr>
</thead>
<tbody>
<tr>
<td>J0</td>
<td>End</td>
<td>Cycle</td>
<td>Site</td>
<td>Load</td>
</tr>
<tr>
<td>J1</td>
<td>SetAccu</td>
<td>Call</td>
<td>SetFA</td>
<td>SetFB</td>
</tr>
<tr>
<td>J2 (ZF)</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
</tr>
<tr>
<td>J3 (PF1)</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
</tr>
</tbody>
</table>

Figure 3: The mutation landscape is shown. The numbers in the table on the left resemble the number of bit-mutations necessary to change one instruction into a new instruction. The graph on the right side show the mutational neighborhood of each instruction.

Figure 3: The mutation landscape is shown. The numbers in the table on the left resemble the number of bit-mutations necessary to change one instruction into a new instruction. The graph on the right side show the mutational neighborhood of each instruction.

specified and PF1-flag active, see also section 2.3.8.

Only a few instructions do have side-effects during execution, namely, Action, SetFA, SetFB and Site, see table 2 and section 2.3.3. With the instruction SetFB the meaning of instruction End can be changed from stopping the program immediately to setting the instruction pointer to 0 (Cycle, endless loop), see sections 2.3.6 and 2.3.10.

In figure 3 an evaluation of the likelihood of evolving certain instructions under mutational pressure is shown. Each instruction can evolve via single-bit-mutations into three different instructions (which is obvious). A slight asymmetry in relation to the Load and Store-instructions concerning the End-instruction have to be conceded. Whether this has an effect on evolution or not has to be investigated.
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>End</strong></td>
<td>The program stops until it is relaunched again. Only available with fixed length programs.</td>
</tr>
<tr>
<td><strong>Cycle</strong></td>
<td>The program jumps back to address zero and continues executing. Only available with fixed length programs.</td>
</tr>
<tr>
<td><strong>Load</strong></td>
<td>Load a value from a register into the accumulator. The cargo specifies the address of the register. Register 0 points to the micro-controller attached at the reading slot. Register 1 points to the micro-controller attached at the writing slot. With no micro-controller attached a search is ignited. Prepending Site-instructions increase the specificity of register-addressing. With no Site-instructions before and accessing register 0 or 1 a random search is done. If no suitable micro-controller is found this instruction is ignored.</td>
</tr>
<tr>
<td><strong>Store</strong></td>
<td>Store the accumulator in a register. The cargo specifies the address of the register. Register 1 points to the micro-controller attached in the writing slot. Register 0 points to the micro-controller attached at the reading slot. With no micro-controller attached a search is ignited. Prepending Site-instructions increase the specificity of the register-addressing. With no Site-instructions before and accessing register 1 or 0 a random search is done. If no suitable micro-controller is found and address 1 is accessed the program is stopped to save time-slices.</td>
</tr>
<tr>
<td><strong>Call</strong></td>
<td>Transfer execution to the micro-controller specified in the cargo-part of this instruction. Accumulator and attachment slots are transferred to the new micro-controller. The current program is stopped after this call. Prepending Site-instructions increase the specificity of the micro-controller addressing, these Site-instructions are combined with the cargo-part of the Call-instruction to one big virtual recognition-site. If no appropriate micro-controller found the instruction is ignored.</td>
</tr>
<tr>
<td><strong>SetAccu</strong></td>
<td>Preset the accumulator with the value provided by the cargo-part.</td>
</tr>
<tr>
<td><strong>Site</strong></td>
<td>Define a recognition-site, either to be recognized by others or to actively attach to other micro-controllers. Used with instructions Call, Load and Store.</td>
</tr>
<tr>
<td><strong>SetFA</strong></td>
<td>If a machine is attached at the reading-slot (e.g. after accessing register 0) then certain flags can be set in this machine, see description of table 4, left part.</td>
</tr>
<tr>
<td><strong>SetFB</strong></td>
<td>Set flags in the executing machine. See table 4, right part, for a description of these flags.</td>
</tr>
<tr>
<td><strong>Action</strong></td>
<td>This command is only available when using variable length program sequences, see section 2.3.11. See table 7 for a description of the possible actions.</td>
</tr>
<tr>
<td><strong>Goto</strong></td>
<td>This instruction was superseded by the SetFB instruction. It set the instruction pointer counter (IPC) to the address given in the cargo-part of the instruction.</td>
</tr>
</tbody>
</table>

Table 2: With a special command (SP) width of two bits (and borrowing a case from the conditional part) these special instructions can be encoded.
<table>
<thead>
<tr>
<th>Bit 1</th>
<th>Bit 0</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Instructions are executed always</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Instructions are executed always and do have conditional equivalents</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Only executed if the ZF-flag (ACCU with value 0) is set</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Only executed if the PF1-flag (via other controllers or from outside) is set</td>
</tr>
</tbody>
</table>

Table 3: Conditional part of an instruction. Instructions can be executed if certain conditions are fulfilled, like a zero-value of the accumulator (ZF-flag) or the flag PF1 set in the status-register of a micro-controller.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>PF1</em></td>
<td>general communication</td>
</tr>
<tr>
<td><em>PF2</em></td>
<td>set if this micro-controller is invisible (protection)</td>
</tr>
</tbody>
</table>

Setting flags A at a foreign micro-controller, using the special command SetFA. If no micro-controller is attached the instruction is ignored.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>CYCLE</em></td>
<td>End-instruction means jumping to address 0, only valid with fixed length programs</td>
</tr>
<tr>
<td><em>REVERSE</em></td>
<td>reading and writing goes backwards</td>
</tr>
</tbody>
</table>

These flags B can be set with the special command SetFB.

Table 4: All flags available for a cargo-width of two are shown here. Of course, with a cargo-width larger then two, further flags could be defined. Flags A (left table) are always available at a foreign micro-controller which is attached via the reading-slot and Flags B (right table) are situated in the own micro-controller.
2.2 Topology

It is now common sense that spatially resolved systems are important for evolution to escape parasitism, to maintain sensible information and to increase diversity in the population, see e.g. (8; 3; 33) or experimental work (55; 6; 12; 54). A question remaining is the optimum spatial dimensionality of the system. Experiments showed that the length of evolutionary avalanches and thus the diversity in the system is best in the one-dimensional case, (1), meaning space is spanned along a ring-topology, see Figure 4. Currently two-, one- and half-dimensional spaces can be investigated. We speak of half-dimension if the number of neighbors is one, communication along the ring can only proceed in one direction.

In the simulations reported the number $N$ of containers is between 16 and 4096 and the number $m$ of processors per container between 8 and 32. Communication is restricted to neighboring containers. The communication between containers is organized such that the micro-controllers do have read- and write-access to the micro-controllers of the neighboring containers as well as to the micro-controllers of the own container. This, for example, gives with two neighboring containers and 16 micro-controllers per container 47 micro-controllers to exchange information with. Information exchange means access to the others program-code, accumulator or status flags-section.

A container is a well-stirred reaction vessel with no further spatial organization. Each processor is able to issue arbitrary many Site instructions, see Table 1. Several Site instructions are concatenated, if they are issued in direct sequence, to create a larger and thus more specific recognition site. These extracted recognition sites are stored in a content-addressable memory to allow for a rapid attachment procedure. Of course, several other possibilities to distribute the recognition-sites do exist and even a cyclic redundancy check fingerprint (CRC16) could be used to identify the micro-
In addition to direct communication, micro-controllers are picked at random from time to time and are exchanged with micro-controllers from neighboring containers. This resembles a diffusive process in the system. In the experiments reported though no diffusive exchange was undertaken.

2.3 The environment and physics of the simulations

In principle more or less every aspect of the physical environment can be adjusted or implemented via the execution of special instructions.

The biggest issue with these special physics-emulation instructions is the number of bits needed to specify a replicating program. The more instructions needed to specify details which could otherwise be handled by physics the larger the search space is. Sequence-space sizes larger than $2^{24}$ are already too big for being exhaustively searched by current day computers. Also, utilizing the dynamic properties of all the none-replicative micro-controllers will not solve the problem. It might be the case that a cascade of 10 different micro-controllers in a network each doing some particular job will in the end result in a replicator-system. But it might also be that this cascade of 10 different micro-controllers actually tries to avoid the development of a replicator because the intermediate steps are more sensitive to perturbations for the system than doing nothing – indeed this is what almost always happens. Robustness is one of the most severe selection pressures in these dynamical systems. Avoiding disturbance is much more rewarded than trying to change something which of course in the beginning might be bad for the whole system. So far, this striving for becoming robust
has been the major bottleneck for explaining how non-replicators eventually would evolve or better
develop into enzyme-like replication systems.

Intriguing enough, this need for simplification – minimizing the number of bits needed – could
be best accomplished by assuming physical behavior and removing the according physics’ specifying
instructions from the instruction-set. Thus it was possible to come to an instruction-set which still
is universal, man-programmable and which only needs 22 significant bits in the minimum-replicator
case, see table 5 (left part). With these 22 bits evolution from scratch is possible though still not
trivial.

2.3.1 Specific versus unspecific attachment

What turns out to be an important physical property is the distinction between specific attachment
(a communication shall only be realized with a specified other micro-controller) or any unspecific
connection, see (48). This distinction is needed because otherwise micro-controllers either simply
replicate their genomes (programs) in small cycles without being able to proliferate the sequences
or they are replicating everything and become vulnerable to information dilution, see section 2.3.2.
Specificity can be realized in several ways: either with a fixed-recognition-site and an additional wild-
carding, see classifier systems and building-block hypothesis (24) or a composition of the recognition-
site from smaller modules. Both variants allow to adjust specificity and both variants have been
studied, see section 2.1. A structural drawback of the wild-carding approach is the length of the
recognition-site. This means that one recognition site can at most have the length of the width of
the cargo-part, see figure 2, which in the simplest case is two bits. This problem is not existent
with the modular-approach where it is easy to concatenate many modules (cargo-parts of the Site-
instructions) to one large recognition-site.

2.3.2 Containment or avoiding the parasites

Containment can be done in two aspects: either to introduce space, in our case the containers con-
ected like beads on a ring, or, in sequence-space to communicate only via specific recognition. Both
aspects have beneficial consequences in reducing the number of foreign machines to communicate
with but exert the drawback of not letting new information, resources and energy diffusing into the container or the occupied sequence-space domain. A vesicle which has no channel-proteins protects the inside pretty well from perturbations from the outside but makes it also extremely difficult to exchange educts and products of chemical reactions in the vesicle. This dichotomy was the rational for the open-container-concept. A further point is the need to somehow specify the compartment in the heritable part of the information constituting the replication system. This can be a huge amount of information which has to be stabilized against perturbation events and which is not available right from the beginning, thus it would have to be created \textit{de novo} by evolution.

\subsection*{2.3.3 Recognition sites and protection}

From a computer-science point of view software with many existing recognition sites per program should be benign for evolution because contact can be made easily. On the other hand, exactly this high connectivity might be a curse, because everyone can interfere. Only one recognition site per program turned out to be a good choice for the moment. For example, molecules in the RNA-world \cite{26} have to be small and probably will not be able to maintain several recognition sites per molecule. Similar with protection, in chemistry it is quite standard to work with protecting groups. It is easy to implement protection in the micro-controller, e.g. if the PF2-bit (see table \ref{tab:4}) of a micro-controller is set then this micro-controller becomes invisible. Unfortunately, being protected also hinders others to do the necessary changes. Protection without control of the protection is an evolutionary dead-end.

An important problem concerning specificity, see section \ref{sec:2.3.1}, is the specificity of recognition sites. In \cite{48} a special wild-carding system was used but handling the wild-cards is computationally expensive. Furthermore, wild-cards are somehow difficult when mapping them to biochemistry. In this work a more promising approach was considered, the concatenation of Site-instructions to form a larger recognition-site, see also section \ref{sec:2.3.1}. This makes handling easy, solves the problem of specificity and is biologically more convenient. If no Site-instructions precede, for example a \textit{Load}-instruction, then only the bits in the cargo-part of the \textit{Load}-instruction are used for recognizing a neighboring micro-controller. With several Site-instructions before \textit{Load} and the cargo-part not
pointing to one of the attachment-slots the bits of the cargo-part are appended to the already existing virtual recognition-site, giving a larger recognition-site. If the load-instruction, see table 2, wants to access the program section of a neighboring micro-controller (address 0 or 1) then a micro-controller is searched for with the appropriate virtual recognition-site, address 0 or 1 bits are not appended to this virtual recognition-site. If no site-instructions are preceding the load-instruction a micro-controller is chosen at random – yielding unspecific recognition. It is assured that this neighboring micro-controller is not already attached to a different slot. Self-attachment is not possible. If no appropriate micro-controller was found nothing is done and the next instruction is executed. A small difference holds for the write-instruction (store to address 0 or 1). If no other micro-controller was found the program is stopped, because finding a suitable micro-controller at a later stage lacks synchronization anyway. This attempt to reduce computer utilization is arguable and can of course be changed at a later stage. Another solution would be to search for the next smaller recognition-site but this again will be postponed to future research.

2.3.4 Bi-molecular versus tri-molecular reactions and the instruction-pointer counter (IPC)

This point is probably the most controversial one: in nature bi-molecular reactions are abundant. Only rarely tri-molecular chemical reactions will occur. And thus the experiments before have been of the bi-molecular type, (48).

In general, replication requires three different algorithmic steps: 1) take the information to be replicated, e.g. a nucleotide, 2) process it somehow, e.g. via a synthetase and 3) store it at the destination place, e.g. create the peptide-bond. Translated into computer-science language this means a load-operation, writing the data into a local register (ACCU) and exert a subsequent store-operation. Usually load- and store-operations require a register-address as well. Now, with only bi-molecular processing this means for the load-operation to attach prior the information donor (template) – typically a micro-controller of the same kinship, read out a specific instruction from a specific location in the genome (program) and store this value in the local accumulator (ACCU). Following this, the template-molecule (micro-controller) has to be released and a new
micro-controller, preferably not from the same kinship, has to be attached and the value of the
ACCU be stored into a specific location of the destination genome (program, prey). This means
for each single instruction to be copied, two attachment processes have to be made – reliably.
Of course, with any attachment event a completely different micro-controller could be attached,
destroying the synchronization between template and offspring. Not using the double-attachment
procedure strongly requires a tri-molecular reaction. Biomolecules must be specially designed to
allow for this.

For the replication of a full program, e.g. with 50 instructions, the pointers of where to read
in the foreign instruction and where to write into the other foreign program (prey) have to be
maintained. Experiments showed even slight errors with the replication counters made emergence
of replication impossible. Usually these counters are realized via registers which are incremented
and wrapped around after reaching the end of the template. This requires not only two additional
registers which must be addressed, it also requires the incrementing or better decrementing of the
values in these registers. Counter operations require pretty expensive Boolean hardware (typically
done with arithmetic instructions). All these instructions and addresses require many bits with the
according explosion of the search space.

The only solution to this coding-dilemma is to assume that the micro-controllers somehow do
have physical contact when attached and that one processor is sliding along the other during the
copy-process. This sliding, well known in protein-chemistry, makes complex counter arithmetic
superfluous. In the current model sliding is realized with an auto-increment feature. After reading
(Load) a value from a foreign program the read-counter is incremented automatically, the same
holds for write-operations (Store). What happens when the copying process is finished is described
in section 2.3.6. Each micro-controller has “physically realized” the Load- and Store-counters. These
counters are initialized with a value of zero. They might as well be preset randomly or by the
position of the recognition-site. These two options are currently not studied because preliminary
experiments showed a drastic decrease of the emergence-probability for stable replication.
2.3.5 Two-bit or quaternary encoding

The canonical encoding of instructions in micro-controllers takes 4, 8, 16 or 32 bits per instruction. In a Harvard architecture, especially with 8 and more bits, controllers register-addresses are a fixed part of the bits available. In principle, the register width might be arbitrary. But it is feasible if one register, as well as the accumulator, can be preset by one instruction, otherwise, many instructions must be sequentially executed, with appropriate counters set accordingly. Because of this problem the instruction-structure used here is divided into three parts, the cargo-part (magenta), a special-instruction part (SP, green) and a conditional part (C, orange), see figure 2.

A structural problem is apparent: if the program wants to write a full instruction into a neighboring micro-controller the width of the cargo-part has to be the instruction-width. This of course is only possible if the register width is of instruction-width size but then a register cannot be preset by one single SetAccu-instruction. The only solution to this problem is a sub-coding of a full instruction. In principle, blocking an instruction can be chosen freely. Bit-wise or two-bit-wise encoding is a natural choice here. Because of nature using a quaternary encoding (AUGC-alphabet with RNA) and the work of (15) who showed that the quaternary encoding has the best evolutionary properties we have chosen to encode an instruction with blocks of two-bits each. This means copy-operations which are the combined action of Load- and Store-instructions copy only blocks of two-bits per operation. The consequence is that a full genome copy in the simplest case takes $3 \times n$ copy-operations with length $n$ of the genome or program to be copied and 6-bit instruction width, see table 1.

2.3.6 Dissociation or detachment and loops

In origin of life studies the major problem after copying a sequence is the dissociation of the product from the template, also called product inhibition (43). This also is a problem in the model presented here because there is no obvious choice on what to do after copying is finished, see section 2.3.6. Only a continuous copying and wrapping around after the End-instruction results in the emergence of stable replication systems. The maximum program length currently is 128 quaternary blocks with a maximum program length 42 instructions in the 6-bit case. The copying loop can either be realized via recursively calling a sub-program or via a Goto-instruction, see 2.1 for more details. In the case
of an infinite long loop the acting micro-controller runs out of energy, reaches a maximum age or is overwritten by another micro-controller. Energy and age parameters are useful knobs to adjust the dynamics of the whole system, see section 2.3.10.

### 2.3.7 Registers and special registers

Micro-controllers have and do require registers. The question is how many and for what purpose. When trying to emulate biochemistry or molecular biology registers provide a real problem. There is no easy mapping from a storage-place like a register towards an equivalent molecule. For example, a one-bit-register could be realized via a conformation change of a biomolecule, e.g. two different secondary structures of an RNA-molecule due to some reversible chemical modification, for example a disulphide-bridge. Hybridization of co-factors also can mimic such register accesses. Because of this difficulty the number of registers used by the micro-controllers must be as small as possible. We have already seen in section 2.3.4 that we had to introduce two further registers to avoid lengthy and improbable pointer arithmetic during the copying process. It is also obvious that we need an accumulator, somewhere we have to store the intermediate values from the template-program. The problem of accessing foreign micro-controllers is solved here by defining the register address 0 or 1 as the access to the program of the attached micro-controllers when reading (Load) from, or writing (Store) into, the attached micro-controller. Furthermore a status-register, see table 4, is defined. The instruction pointer as such is a register which again is subsumed in the physics of the system.

The register-address is used for the recognition-procedure, with prepending Site-instructions for specific registers. If the micro-controller’s accumulator which serves as a register for another micro-controller is processed independently the register value is probably changed. On the other hand, this register value can be used for communication also, e.g. a micro-controller simply could run in an infinite loop and looking for specific accumulator values in foreign micro-controllers. Complex communication networks can be realized with this setup.
2.3.8 Conditional execution, communication and sensing the environment

The conditional-part (C), see figure 2, is currently two bits wide. This coding was chosen because conditional execution is powerful and avoids the need of several instructions to code for conditional expressions. Two types of instructions exist: instructions are always executed and instructions are subjected to a conditional evaluation, see table 3.

High-level communication between micro-controllers is not only realized using other micro-controllers as registers, see section 2.3.7, but also utilizing the PF1-flag, see table 4. Additionally, indirect communication can be realized by setting other bits of the partner machine's status register. The value of the accumulator is mapped into the status-register of the attached micro-controller. Affected is the micro-controller which is available after a read-instruction (Load), see section 2.3.7. Please note, this read-instruction has as a side-effect: the increment of the read-counter.

The PF1-flag is the only flag which can be sensed directly by the executed program, see table 3. Providing the system from the outside with additional information is done by issuing the PF1-flag of specific micro-controllers, for example with a pixel-value of a black-and-white image. Output of the system can be realized in the same way, simply look at the PF1-flags of other specific micro-controllers or their accumulator values. With this procedure arbitrary communication networks can be evolved and trained. Furthermore, this output could be utilized to specify the hardware of micro-controllers, important in questions of hardware-evolution, see section 4.2.

2.3.9 Codon mapping

Reading and writing to other machines can be subjected to a codon-mapping – resembling the Watson-Crick pairing in biology. A special option of the simulation program allows to specify the mapping used. The identity-mapping has the value $c = \text{0xe4} (11100100)$ assuming a quaternary coding scheme. Thus a value $i = \text{0x2}$ is mapped to $a = \text{0x2}$ with $a = ((c \gg (i \times 2)) \land \text{0x3})$. With a mapping table $c = \text{0x53}$ the value $i = \text{0x2}$ is mapped to $a = \text{0x1}$. If the mapping table creates a complementary mapping (e.g. $c = \text{0xb1}$) a second application of the mapping brings back the old starting value. The cycle length is exactly one. According to the maximum length sequence (MLS) theory ($21; 9$), the maximum cycle-length of a quaternary mapping can be $2^2 - 1 = 3$. For example
with \( c = 0x4b \) \((0, 1, 2, 3) \rightarrow (3, 2, 0, 1) \rightarrow (1, 0, 3, 2) \rightarrow (2, 3, 1, 0) \rightarrow (0, 1, 2, 3)\) a cycle of length three is realized.

With these mapping-tables even unnatural translations can be tested for their evolutionary properties. The experiments reported in this work all have been done with the identity-mapping \( c = 0xe4 \). With the condition that there exists always a mapping to each possible codon two classes of mappings, apart from the identity-mapping, remain: Class A is of the Watson-Crick pair type \( c = 0xb1 \) - after two steps the starting point is back again and Class B cycles through the full space of possible codons.

Class A has three members with \( c = \{0x1b, 0x4e, 0xb1\} \) and a Class B comprises six members with \( c = \{0x1e, 0x39, 0x4b, 0x72, 0x8d, 0x93\} \). With the identity-mapping \( c = 0xe4 \) three possible evolutionary mapping-types are possible. A hypothesis is that the evolutionary behavior in between the classes will not change, though there might be peculiar interactions between the mappings in a class and the particular instruction-set. All other possible mappings restrict the codon-space and allow only a subset of codons to be executed in a usual way. Of course, it might be that through mutations non-reachable codons by chance still come into play.

The identity-mapping for a binary-base (only 0 and 1 as base) is \( c = 0x2 \) and an eighth-base-system (3 bits per base) is \( c = 0xfac688 \). Preliminary experiments with Watson-Crick-type mappings so far did not reveal viable replication systems.

### 2.3.10 Energy management in the system

The distribution of energy must be organized such that at least a very small number of microcontrollers can survive without doing something special. Additionally, energy should be distributed according to given external objectives. The natural choice would be the PF1-flag as the sensor for external input, see section 2.3.8. A set of dedicated accumulators can serve as output-channels. A first simple task to be performed might be the counting of set PF1-flags and write the result into an output-register. The amount of energy distributed per container would then be proportional to the correctness of the output. In the experiments reported energy was not subjected to some external tasks.
Six parameters for the simulations with energy-processing are defined:

_-OPT_TOT AL_ENER GY_- (-te #i) The total energy provided by the environment per generation and container. This energy is delivered to each living or newly instantiated micro-controller.

_-OPT_DEA TH_ENER GY_- (-de #i) A micro-controller which energy-resources drop below this threshold is interpreted as dead. Dead means that the micro-controller does not get CPU-time for execution. The program of the micro-controller instead remains intact as a type of crystal, meaning the information in the program will be conserved (as long as not an external mutation or rewriting event changed this information). Not only the information of the micro-controller remains intact, also the content of the accumulator is still accessible via others. The rational is the argument that the energy to access the accumulator of the dead micro-controller is already provided by the acting micro-controller.

_-OPT_EARN_ENER GY_- (-ee #i) The amount of energy which can be earned when doing a successful operation. An operation is defined as an action which somehow connects the micro-controller to the outside world. Thus, success is defined from the outside of the simulation and either part of the physics in the system or some external task to be solved. In the experiments reported this value remained constant throughout the simulation.

_-OPT_SPENT_ENER GY_- (-se #i) Energy to be spent for each executed instruction. It is possible to separate the real energy costs per specific instruction by coefficients. E.g., a register-access might be much more expensive than a SetFA-operation, and calling another micro-controller as a subroutine is energetically certainly different from attaching a different micro-controller.

_-OPT_PR OG_ENER GY_- (-pe #i) Energy to be spent for each program execution. The reason is the freedom of micro-controllers to do nothing and never going to die because simply surviving by the background-energy. With this cost high enough these parasitic micro-controllers will die and be replaced by new micro-controllers.
This is the minimum number of micro-controllers living at the beginning of a new generation. If less than the minimum number of living micro-controllers are in a container, dead micro-controllers are picked at random and reinitialized with the energy reservoir set to the total amount of energy. With reinitialization the accumulator value is set to the initial value and the program is preset with default values, e.g. random values. No former state of the micro-controller is maintained. The only exception is in the beginning of the simulation: then each micro-controller gets a starting energy. The reason is that starting with a totally randomized population will yield most of the micro-controllers dead after the first generation.

Several different energy coefficients have been tested, see also 2.6, but none of them systematically. It turned out that more or less neutral coefficients are most suited for evolutionary purposes.

2.3.11 Introducing variable length programs

Variable length programs are even more akin to physico-chemical systems found in nature. They immediately require operators utilizing length dynamics of the informational sequences. The first two most obvious operators are Ligation and Splitting, see table 7. But there are subtle problems or features as well: e.g. when does copying stop? With fixed-length programs copying was stopped after encountering the End-instruction. It simply did not make sense to waste CPU-time and also copy the rest of the program. In the same way it was feasible to overwrite the program of the prey without taking into account what was written in the prey’s program before. With variable sequence-lengths instead the situation changes. Sequences are now seen as physical molecules which are to be copied. Copying itself becomes a construction process, not a replacement process, as in the former case, and thus the length of the prey’s program is changed with the copying process. For example, this requires the adjustment of the instruction-pointer of this machine because after copying the prey might have shrunk and the pointer is addressing illegal memory locations. The dynamics with variable length sequences alter, data not shown, even providing replicative sequences at instantiation time only yield extremely bad proliferation and high sensitivity to parasites.

Variable sequence lengths also revealed a further problem for the emergence of replicative systems.
Table 6: Instructions available when variable program lengths are used. This is the right part of table 1. The *End/Cycle* instruction has been replaced by the instruction *Action*. Depending on the bits provided by the cargo-part certain actions are executed. All defined actions are specified in table 7. The instruction *SetAccu* took the place of the former *End/Cycle*-instruction. Please look at the symmetry of the instructions. *SetAccu* and *Site* are accessible with a machine-mask of 0x1, see section 2.3.12. Machine-mask 0x3 adds the instructions *Load* and *Store* which empower the machine to copy other programs. Machine-mask 0x7 (0xb when three bits are used for the special command (SP)) further adds two procedural commands *Action* and *Call* and two property changing commands *SetFA* and *SetFB*.

It turned out that very short, typically one instruction long sequences with a single *Site*-instruction occurred as targets for the replicating machines. Because of being extremely short these sequences easily could outperform the machines with a valid replication program. With a simple and biologically plausible trick this problem could be mitigated. Only sequences with a length of at least three instructions can partake at binding. Sequences shorter than this are not considered in specific binding though they still can be bound unspecifically as preys of the replication process.

An additional problem occurred with the emergence of replicators. Looking at the evolving programs revealed that many codes with a length of only one instruction occurred utilizing most of the CPU-time, see above on the minimum template length. Especially when these programs consisted of one *Store*-instruction neighboring machines could be overwritten with a single value independent of what useful functionality had been evolved before. This sort of denial of service or vandalism in the system made the emergence of replicative systems impossible. With a minimum program length of two instructions this problem is no longer existent. It is biologically plausible to require a certain minimum complexity or length of sequences before emergent replication can occur. This gives a lower bound of the length of nucleotides considered in biochemical origin-of-life experiments.
<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Ligate (00)</strong></td>
<td>If there are two machines located, one in the reading and another in the writing-slot, the program of the machine in the writing slot is appended to the program in the reading-slot. The machine in the writing slot is invalidated after that and will be replaced by a newly created machine at the next simulation round.</td>
</tr>
<tr>
<td><strong>Split (01)</strong></td>
<td>If there are two machines located in the reading and writing-slot the program of the machine in the writing slot is replaced by the program right after the current remote instruction pointer (typically after the recognition site). The program in the reading slot is shortened accordingly.</td>
</tr>
<tr>
<td><strong>CopyMask (10)</strong></td>
<td>Copy the machine-mask, see section 2.3.12, from the reading-slot-machine into the machine’s machine-mask attached at the writing-slot.</td>
</tr>
<tr>
<td><strong>CopyLen (11)</strong></td>
<td>Copy the length of the machine in the reading-slot into the machine of the writing-slot, which resembles a truncation or an elongation.</td>
</tr>
<tr>
<td><strong>Release reading-slot (100)</strong></td>
<td>If a machine is attached at the reading-slot it will be released via this action.</td>
</tr>
<tr>
<td><strong>Release writing-slot (101)</strong></td>
<td>If a machine is attached at the writing-slot it will be released via this action.</td>
</tr>
</tbody>
</table>

Table 7: Possible action-commands. Depending on the number of bits in the cargo-part of the instruction, see Figure 2, the shown actions can take place.

<table>
<thead>
<tr>
<th>Execute</th>
<th>S0</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
<th>S4</th>
<th>S5</th>
<th>S6</th>
<th>S7</th>
</tr>
</thead>
<tbody>
<tr>
<td>J0</td>
<td>SetAccu</td>
<td>Site</td>
<td>Load</td>
<td>Store</td>
<td>Nand</td>
<td>SL</td>
<td>SR</td>
<td>Goto</td>
</tr>
<tr>
<td>J1</td>
<td>Action</td>
<td>Call</td>
<td>SetFA</td>
<td>SetFB</td>
<td>GetFA</td>
<td>GetFB</td>
<td>Decr</td>
<td>RND</td>
</tr>
</tbody>
</table>

Table 8: Instruction set used with three bits in the special-commands part (SP). The possible flags are described in table 4. The default instruction for (J0:S0) is Cycle. This instruction lets the instruction pointer jump to address zero. The instruction (J0:S0) can be changed by clearing the _CYCLE_ -flag with the SetFB instruction.
A cron ym

Table 9: Eight further commands can be coded with a third bit available for the special command section (SP). See table 2 for the other possible instructions.

2.3.12 Different types of micro-controllers

Before delving into the area of hardware-evolution, see section 4.2, an intermediate step can be investigated: allowing different types of machines to participate in the evolutionary course. The special-command part (SP) in figure 2 was extended from two to three bits. This means an instruction with a cargo-width of two bits is then seven bits wide. See tables 2 and 9 for the additional instructions available.

To simulate now a machine with only six-bits of instruction-width the one added bit in the special-part of the instruction is simply mask ed out. A special property of a machine, the machine-mask, was introduced. This masking procedure in addition allows to simulate a three-bits instruction-width machine, which of course with two possible instructions is not able to evolve something (\textit{End} and \textit{Site} in the fixed program length scenario, or \textit{SetAccu} and \textit{Site} in the variable length scenario, see table 1). But with four bits width \textit{Load} and \textit{Store} become part of the instruction set and a replicating systems can evolve. With these machines (machine-mask is 0x3) 12 bits are needed for the minimal replication program, see 5 (right part). Experiments could show that the emergence of replicating systems was considerably easier in this case. Even with a machine-mask of 0x7 evolution starts more or less right from the beginning. The only drawback is that these machines are no longer universal. The interesting question arises is there an evolutionary pathway from simple machines to
; An even simpler replicating program.

sreg work 0x0
sreg my_label 0x3

; Search for my own kinship
site my_label
   do
      ld work ; Load instruction from neighbor
      st work ; and write it into the neighbor program
      cla ; clear accumulator
      while -- 0 ; jump back if zero-flag set
   halt

Figure 5: Listing of the simple replicator program. The Goto-instruction is not shown in table 1. The instruction Cla be represented by a SetAccu instruction with the cargo-part set to zero. With this Goto-instruction 28-bits are needed for a successful minimum-replication system to emerge.

complex machines possible when the machine-mask is subjected to evolution.

2.4 Simple assembler program

A simple assembler program, see figure 5, was written using the instruction set shown in table 1, left part. Reading from a neighboring program (Load) initiates an attachment procedure if and only if not a neighbor is attached already. Successfully attached, the machine is stored in the reading-slot. Writing into a neighboring program (Store) again initiates an attachment procedure, when needed, and stores this pointer into the writing-slot. The remote instruction-counters are reset (value 0) when realizing the attachment.

With assembling the simple replicator program, figure 5, a machine-code of 6 lines of code was generated (Site, Load, Store, SetAccu, Goto, End). With 6 bits per instruction the total number of bits needed is 36. Carefully analyzing this code reveals that only 28 bits are essential for the replicator to work. $2^{28}$ is already a search-space size which can be handled by current-day computers. Of course, two such replicator sequences have to emerge and they have to be in the same ambiance to be able to copy each other. Unfortunately, no conditions could be found to let these replicators emerge from scratch.

Only after simplifying the instruction-set even further an autonomous emergence of a replicator system could be observed, see figure 9. The trick was to reformulate the End-instruction as a jump-to-the-beginning (Cycle) instruction. This further simplification resulted in a genome '303002010000' = \{Site, Load, Store, Cycle\} with now 22 relevant bits, see table 5. And indeed, looking at the evolved
replicators, this was what evolution typically started with, see section 3.2.

2.5 Universality through simulation of a Turing machine

Universality can be shown if the machine in inspection is able to realize a Turing machine. Of course, with the parameters used here and a finite program length universality can be shown only in principle. With the system presented the tape of the Turing machine will be the program of an attached machine either in the reading or in the writing-slot. Without loss of generality it is assumed that the tape (or foreign program) only contains 0, 1 and blanks (which can be symbolized by the values 2 and 3). Each state of the Turing-machine is realized via a dedicated further machine which is called via the instructions Call or Call_ZF (Call_ZF is equal to Call but only executed if the zero-flag is set). The following instructions are needed: Load, Store, Call, Call_ZF, SetAccu and SetFB, see the assembler-code in figure 6.

2.6 Principles of the simulation experiments

After an initialization phase a container (one out of \( N = 4096 \), section 3.2), is picked at random. If the software (e.g. EvoCpu_i686) runs in a multicore-environment threads are created with a set of associated containers and each thread executes the processors in the associated containers. Per container a processor (machine) is picked at random and executed. This is done \( n \)-times with all \( m \)-numbers of processors. For the experiments shown here \( n = 400 \) and \( m = 8 \) (section 3.2). The particular values are chosen conveniently and are essentially arbitrary.

Each processed machine has a time-slice budget of e.g. \( s = 800 \) executed instructions. Whether the machines exploits the full time-slice depends on either the meaning of the End-instruction, see section 2.3.6, or the successful execution of a subroutine-call (Call), in which case the calling machine is suspended and the subroutine processed instead.

Together a maximum of \( N \times n \times m \times s \approx 10^{10} \) executed instructions per generation, see section 3.2, could have been possible in the experiments. Of course, the factual number is much lower, because most of the machines die before they can utilize the full time-slice.

A note to the term generation: generation in this work is an arbitrary notation, defined
; Main-assembler program
setfb for_step  ; Set the head-movement-pointer to forward-direction
ld reading_slot  ; Load a value from the machine attached to the reading-slot
st A             ; Store the value from the tape in intermediate register A
setfb back_step  ; Set the head-movement-pointer to backward-direction
ld reading_slot  ; Move tape-pointer one step back, because it was autoincremented.
ld A             ; Reread the value from the tape
call zf zero_state_0_routine  ; if accumulate is zero this function is called
call one_state_0_routine     ; else this function is called.
; this line is never reached because control goes to the aforementioned function.

; This function is called if the value on the tape was zero.
func zero_state_0_routine
    set 1
    setfb back_step  ; Direction pointer of head-movement is set to backward.
st reading_slot  ; Write it onto the tape (auto-increment will move the head)
call state_4_routine  ; This will be the next state of the Turing machine

; This function is called if the value on the tape was one.
func one_state_0_routine
    set 1
    setfb for_step  ; Direction pointer of head-movement is set to forward.
st reading_slot  ; Write it onto the tape (auto-increment will move the head)
call state_1_routine  ; This will be the next state of the Turing machine

func state_1_routine
    setfb for_step  ; Set the head-movement-pointer to forward-direction
    ld reading_slot  ; Load a value from the machine attached to the reading-slot
    st A             ; Store the value from the tape in intermediate register A
    setfb back_step  ; Set the head-movement-pointer to backward-direction
    ld reading_slot  ; Move tape-pointer one step back, because it was autoincremented.
    ld A             ; Reread the value from the tape
call zf zero_state_1_routine  ; if accumulate is zero this function is called
call one_state_1_routine     ; else this function is called.

Figure 6: Sketch of a simulated Turing machine by a concerted action of programs of different machines. Shown is the main-program which calls either \texttt{zero\_state\_0\_routine} if the value read from the tape was zero or \texttt{one\_state\_0\_routine} if the value was one. In these two routines the next action and the next written symbol is specified. In the example, the first routine calls that state-function \texttt{state\_4\_routine} and the second \texttt{state\_1\_routine}. These state-routines behave like the main-program, read the current values from the tape and call the according zero-, or one-state routines. As was developed in (48), each \texttt{Call}-instruction hands control over to a new machine with the according sub-program. Pointers to the reading- and writing-slot are forwarded to the machine called. Arbitrary many states can be realized with this system, only a sufficient number of bits in the cargo-part of the instruction have to be available, see figure 2. A halting state simply does not call further sub-routines. The \textit{Site} instruction necessary to provide a sufficient specificity of the function- and register-accesses is not shown here.
exactly as written above. This is in a certain sense counter-intuitive because usually a

generation is defined as the average time a self-replicator needs to replicate. Of course,
in our context, the time of replication strongly depends on the length and efficiency of
the evolving program and thus is not an external parameter of the system. A generation

here loosely is the time to process all containers on average once.

Each executed instruction needs a certain amount of energy and if the according energy coefficient

is non-zero the execution might earn or would cost energy, see section 2.3.10. Several instructions

are provided with energy coefficients which allow a fine-tuning of the physico-chemical parameters.

Preliminary experiments showed sometimes strange behavior with setting certain parameter values.

E.g. a high positive energy amount for a copy-operation is not necessarily beneficial for evolution

because the need to evolve many copy-operations is drastically limited if already a few raise enough

energy for surviving. On the other hand, a too low amount of energy per copy-operation, meaning

high costs, only favors artifacts which simply do only copying without trying to replicate a foreign

genome. Similar considerations hold for the other energy-coefficients as well. The energy coefficients

taken here are more or less neutral and chosen such that evolution, at least in the experiments done,
had best properties. See the software-source and published experiments for technical details (49).

A maximum age of micro-controllers (machines) was build in as well. The age-parameter was

chosen suitably to allow a smooth evolution of replicators. Age in this sense is defined as the

maximum number of executions of the according micro-controller. The age-boundary has two parts:
a deterministic part plus a random term which is a number between zero and one multiplied by the
deterministic part. This setting was necessary to avoid grid-effects because at the beginning of a
simulation all micro-controllers have the same age and of course aging in real life is not deterministic
as well.

3 Computational results

Only very few examples of evolution will be shown because the amount of data collected is consider-
erable. The examples shown here are available on-line (49).
Figure 7: Shown is the fingerprint-view of a replication system after seeding with a replicator in the upper left corner. The colors represent the returned value of a cyclic redundancy check of the evolved program. The absolute value of the color has no meaning, different colors symbolize different programs. Several different programs can have the same color by accident. The experiment’s time is 4000 generations, with 64 containers and 16 micro-controllers per container. The defects visible are about after 2000 generations. No external perturbations are subjected to the experiment. Time goes from top to bottom, all containers are shown from the left (container 0) to the right (container 63).

3.1 Simple instruction set seeded with a set of replicators

Seeding with replicators is easily done. Shown in figure 7, is a fingerprint of the evolving replicators after seeding the first four containers (in the upper left corner in the figure) with the program shown in figure 5.

More or less immediately the replicator takes over the whole system and only some minor defects occur in the middle of the experiment visible as the faint yellow and black stripes in figure 7. No further mutation-events or other perturbations have been subjected to the system. The fine-structure of the evolving dynamics show that aside from the replicators other program sequences are necessary as well (otherwise we would have one single color in the fingerprint). It remains to be investigated, what exactly the detailed fine-structure of the evolved programs are.

3.2 Simple instruction set seeded with random sequences

The experiment shown in figures 8, 9 and 14 has been conducted in a space of 4096 containers with 8 micro-controllers a container. The full evolution space is shown highly distorted in figure 8. On-line a movie can be seen which shows the full evolving area (49). Shown here are only to sections of this space, with figure 9 beginning at micro-controller 1216 and ending with micro-controller 1716 and with figure 14 beginning at micro-controller 22816 and ending with micro-controller 23316.
These experiments were done with the simple instruction set shown in table 1, left part. What turned out during experimentation was the superior emergence probability of self-organized structures when not every micro-controller was randomized at creation but the majority of micro-controllers initialized with the \textit{End}-instruction. The reason for this beneficial behavior is thought to be the combination of random-sequences and mostly non-random sequences which had no ability to perturb the organizing structures. This means, if replicators emerged then they needed empty micro-controllers as proliferation source and a low probability to not disturb each other.

From these findings the random initialization of the micro-controllers was realized at only two micro-controllers or even less per container. This is the reason why the faint vertical stripes occur in figures 8, 9 and 14.

Actually looking carefully at the full fingerprints of the 32768 different micro-controllers two origins of replication can be identified. After about 400 generations the first replicator emerged, see figure 10 in container 1364, location (B) in figure 8. The functional instructions of the emerged program are written in upper-case letters: \texttt{STORE}, call, \texttt{LOAD}, \texttt{zf_setfa}, \texttt{zf_setaccu}, \texttt{pfl_setfa}, \texttt{LOAD}, \texttt{store}, \texttt{CYCLE}. Interestingly this replicator replicates every program it finds and thus is extremely vulnerable to parasites. Later replicators become specific, a means to be robust, see section 2.3.1. Independently, 240 generations further-on the second replicator emerged at a different
Figure 9: Shown are different views of the evolution at a specific region of the experiment with 32768 micro-controllers, see figure 8. Five different properties are plotted, (A) the fingerprint-view, (B) the energy view, (C) the copy-operator-view, the brighter the color the more copy-operations per time (D) the write-into-other-program view and (E) the read-from-other-program view. View (A) clearly demonstrates the many different species which evolve and take over space at certain time-instances. That this evolution is not only neutral is emphasized by view (C), (D) and (E) where different copying strategies show changed functionalities over time. The flat slopes of the proliferation events resemble the long time evolutionary trades are stable and then replaced by other species.
Figure 10: The first replicator emerged at generation 400 in container 1364 (region F in figure 9). The instruction sequence is Store, Call, Load, ZF_setAccu, PF1_setAccu, Load, Store, Cycle. The instruction ZF_setAccu presets the accumulator with a certain value, in that case value 0, if it was zero before. Please note that the last instruction _SYM_END_ means jump-to-start of the program, see section 2.1.

Figure 11: The second replicator emerged independently at generation 640. The instruction sequence is PF1_setFA, PF1_Call, ZF_setFA, Store, Site, Load, Store, Cycle. Please note that the last instruction _SYM_END_ means jump-to-the-start-of-the program.
Figure 12: The emerged replicator A in container 2871 which populates the empty space. It is nearly minimal, with the instructions PF1_SetAccu, Site, PF1_SetAccu, Load, Store and Cycle. The instruction PF1_SetAccu is useless because even if executed the accumulator is not changed. The region where this replicator dominates is marked as ‘J’ in figure 14. Please note that the last instruction _SYM_END_ means jump-to-the-start-of-the-program.

location, see figure 11 at generation 640 in container 268, location at (A) in figure 8. This second emerging replicator has the active instructions (upper-case-letters) pf1_setfa, pf1_call, zf_setfa, store, SITE, LOAD, STORE, CYCLE, which is apart from the non-functional instructions at the beginning of the program a minimal replicator. This program later evolves to the program shown in figure 12 from container 2871 and proliferates into the right space, shown in blue of figure 8.

4 Discussion

Two achievements have been shown: the stability of non-self-replicator systems over very long timescales and the emergence of non-self-replicator systems without specific seeding. Both results are extremely important. Firstly, molecular-replication systems can be hardly envisaged as being constituted of self-replicators because self-replicators need to solve the compartmentalization problem, with all the problems described in section 2.3.2. Secondly, non-self-replicator systems are extremely vulnerable to parasitism and especially the beginning of the non-self-replication system makes it unlikely that no other machine or sequence in the vicinity is disturbing the process. Only after many replicated sequences a sufficient number of replicas are available which can take the duty to proliferate the species and in addition change the environment such that disturbing events cannot happen anymore, e.g. in the simplest case by overwriting all the other competitors with the species own code. Exactly this problem is thought to be the reason why hitherto the emergence from scratch of non-self-replication systems has not been possible.
Figure 13: Evolved replicator B in container 170, which is shown in figure 14 at the location where 'F' points to. The blue-brown hashed structure in the fingerprint section 'A' in figure 14 indicates two different replicators oscillating or at least being spatially organized. Looking at the instruction sequence, Site, Load, Store, SetFA, Site and Cycle, reveals two interesting features: when first executed only machines are searched for with recognition sites of length 2-Bit. After the first execution, only machines are looked at with a length of the recognition-site of 4-Bit, which are then copied. If during the first execution a machine was found, then this is copied successfully. If only a machine, after the first round, with recognition site of length 4-Bit is found this machine is copied. This means the replicator is specific to two species. Furthermore, the SetFA-instruction can have an impact on the machine which acts as a template, depending on the content of the template. Please note that the last instruction _SYM_END_ means jump-to-the-start-of-the-program.
Figure 14: At a different spatial location. Here two different species collide. From left a species which program is shown in figure 13 and from the right a nearly minimum replicator, program is shown in figure 12. Though it is different from the fingerprint (A) point of view and the ability to copy (C) the number of read- (E) and write-instructions (D) from and into the foreign genome are essentially the same – this is what also makes the energy consumption (B) indistinguishable for the two species.
A third indirect result of this work, when studying the emerging replication systems, is the unlikely development of replication from scratch purely through network interactions. This already could have been seen in (50) where no modular structures evolved when solving the multiplier evolving problem, even after long evolutionary times evolving robustness did not yield modules. On the contrary, every function-generator (4x16 function-generators with 4-bits input each evolve, they yield together a 4-bit-multiplier) of the multiplier evolved its special solution and the mathematical structure was not recognized or learned explicitly by the system. This is not a contradiction to the astonishingly simple solution when introducing self-assembly (16) because this self-assembly procedure only allows modules to be evolved. A consequence of this third finding is that no miraculous network-topology will help to create a replication system but only the sheer size of the search-space for finding a sufficiently small non-self-replication system, which in the current work and the currently available computer-power is around 22 unknown bits to be found. Transferring this result into biochemical systems require a systems setup such that physics and chemical properties have to provide the vast majority of information for getting a replication running and only a tiny amount of flexibility in the information carrying modules can be tolerated.

4.1 Reflecting literature in light of current results

The following list is a loose coupling of comments and remarks reflecting literature related to this work:

- Almost all micro-controller based software evolution studies assumed the existence of cellular environments, see e.g. (42) and follow-ups (2). The question of the transition from the abiotic to the biotic world has hardly been tackled with evolving software, apart of von Neumann’s and Holland’s work. The second usually taken approach was to search for self-replicators. But as (14) pointed out, self-replication has a strong tendency towards simplified replication phenomena and as has been mentioned in the introductory section 1, self-replicators are not feasible when asking the question of molecular replication without cellular environments.

- Actually the first work which tried to develop a computational model and provided formal proofs of emerging replication systems is Holland’s α-universe. Though he did not give evidence
that he indeed implemented this model and checked his proofs only one work of (34) tried to realize his model. McMullin was not able to validate Holland’s findings and argued that especially interactions with other not yet ready replication systems would hinder the emergence of stable replicators. Knowing now how to create non-self-replicator-systems from scratch a revisiting of Holland’s $\alpha$-universe, perhaps with the additional introduction of space, would be worthwhile.

- Pargellis (36) was the first to show that self-replicating software in micro-controllers could emerge. He streamlined the Tierra (42) instruction set such that one in about 100,000 random-sequences five-instructions sequences resulted in a self-replicator. The following work (37) extended the instruction-set to 32 instructions (1 in 20 million random sequences yielded self-replicators) also showing the emergence of self-replicators which are Turing-universal. As with Tierra the allocation of memory and the division of copies from the parent are difficult to map on biological systems.

- In the presented model two questions were not yet answered: whether the software evolving in the micro-controllers is indeed able to evolve arbitrarily complex features and secondly whether externally given tasks can be solved by these evolving micro-controllers. Currently, successful solving externally given tasks require very special fitness-landscapes introduced from the outside to allow complex features to be evolved, (29). Especially the intermediate steps had to be rewarded with software evolution in micro-controllers to evolve some simple Boolean functions, (30). This problematic situation might be relieved if self-assembly and thus structure-learning processes could be utilized (16).

4.2 From different types of machines towards evolvable hardware

With the different types of micro-controllers at hand, see section 2.3.12, and allowing evolution to switch between these types the natural extension is to let the machines being constructed directly by the dynamics or evolutionary processes in the system. Of course search space dramatically increases in size and it is a question whether a pathway from simple machines towards more complex ones still exists. It is obvious and was the recurring result of research in this area that machines with
even moderate complexity failed to spontaneously emerge replication systems. For example, only six additional bits are needed to modify the replicating program of the simplest machine to run successfully on the machine where the End-instruction really means End and the instruction SetFB is being replaced with a Goto-instruction, see table 1, left part. It should be possible for evolution to jump over this gap of six bits. Also straight forward is the concurrent existence of differing widths of the cargo-part in the system, see figure 2. Whether these jumps are possible during evolution has to be tested thoroughly. The more plasticity is build into the hardware, meaning the more the hardware as such is evolvable, the more it becomes a target of evolution. The pathway from one machine-type to another or from one hardware to the next should be as smooth as possible. How to realize such a smooth “hardware-landscape” is part of future research. The big advantage of replicating systems though is the high abundance of copies of successful replicators. This gives hopefully enough robustness to test and play with many types of machines in one system. Of course, extremely interesting is also the question how evolution behaves if only certain types of machines are possible in different parts of the simulation space, for example, containers 0-9 would allow only machines with 2 special-bits (SP) to be used and containers 10-19 only machines with 3 special-bits.

4.3 Connecting to molecular dynamics (MD)

Now with the spontaneous emergence of replicators at hand the artificial restriction to simple spatial topologies can be relaxed. Actually, it is easy and straight forward to incorporate the evolving software into a molecular dynamics code like for example LAMMPS http://lammps.sandia.gov/. Then information processing in the world of simulated molecules becomes feasible. Using the mesoscale simulation facility DPD of LAMMPS or the extension multipolar reactive DPD (18) we will be able to combine physically valid system dynamics with evolving soft- and hardware.

4.4 Transferring the results to biochemical experiments

Certain physical assumptions had to be made to allow for a successful spontaneous emergence of replicator-systems. Most of these assumptions have been reported in section 2.3 and are summarized here:
• tri-molecular reactions had to be assumed which translates into biochemistry that the catalyst
must slide along the template, see section 2.3.4. This could for example be a stochastic ratchet-
like process. If this is not possible then a physical structure has to provide the same effect, e.g.
the template being pushed and pulled via hydrodynamic pressure through an eye of a needle
or cavity with the catalyst connected to the opening.

• only very few bits can be encoded in the sequences everything else has to be provided by physics
and chemistry. It is no hope that in nature the available parallelism is gigantic compared to
what we have available in the computer: firstly, the explosion of the search-space outnumbers
the available resources right from the beginning and secondly, the physical non-determinism,
fuzziness and Brownian motions consume many of the parallelism-resources. This is a very
important finding. Even though proponents of the RNA-world hypothesis,(20) (26), believe
that ribozymes can in principle solve the replicator-emergence problem, still a gap between
the required fidelity of replication and the capabilities of ribozymes exist. This upper bound
of perhaps 20 to 30 bits of the exploitable informational search space requires that ribozymes
needing more then a few nucleotides will probably not be able to emerge spontaneously.

• magic network behavior probably doesn’t help. Before, it was not clear whether some com-
binations of building blocks of sequences which accidentally happens to be in the same area
could mimic a replicator. The consequence of not yet finding any hints of such a behavior in
the simulations makes it unlikely that networks of cooperating molecules would emerge into a
replication system in nature. If several components were working together then the connections
between these components have to be very profound and reliable and thus working as one entity
and not as a network of loosely coupled operators. This is not in contradiction to intriguing
catalytic properties of e.g. cleaving deoxyribozymes (31) building an auto-catalytic cleavage
process. Auto-catalytic replication of information is a much more complex process then simply
letting ribozymes cleaving circular rings of RNA which then become active ribozymes.

• already known: spatial resolution is important. That spatial resolution is an important ingre-
dient is known since many years (8; 47; 33; 17), it helps overcoming parasites and generates
diversity due to time-delays in communication. Translated to biochemical scenarios this means that well stirred and turbulent fluidic ensembles are not very suitable environments for the first emergence of life-like processes.

• sequence specificity is sufficient, compartments in a physical sense are not needed. That physical compartments in a strict sense are not important is good news because containers always pose the question on how they are created and maintained in the course of evolution. Though protocell research made considerable progress (39), the problem of linking informational molecules to the creation of vesicles with amphiphiles created by the chemistry itself and the necessary dividing of vesicles is still poorly understood. Furthermore, getting resources in and waste out is a fundamental problem of containers.

• circular plasmid like templates and thus perhaps circular catalysts have the big advantage that the catalysts can stay at the template and produce many copies of the same template. This gives a dynamically completely different robustness. The only problem, the common problem, how the copy will be released from the ternary complex. In the experiments an End-of-Sequence recognition was used which could be realized as a sort of ‘STOP-codon’ in the biochemical realization. Another possibility could be a sequence-replacement system which successfully has been realized in the DNA-computing realm (56). Also refolding of the secondary structure due to different salt-conditions or temperature gradients could change the enzymatic functions of the ribozymes (27).

• low perturbation by random sequences. It turned out that too many random sequences in the vicinity of a replication system are a problem for spontaneous emergence. In contrast to the protein-world of (25) labile replication systems are being subjected to perturbations by other not related sequences. Trinks (51), proposed ice-cavities as a possible space of the origin of life which seems to be a plausible location because of the huge parallelism, low energy intake and reliable environmental conditions, this view is supported by (52) who argues that the phosphodiester backbone of RNA can be stabilized in ice.
4.5 Conclusions

A long standing problem has been solved. The de novo emergence of enzyme-like replicating systems could be achieved. Key to success was the steady simplification of the micro-controllers and the addition of physically plausible constraints which again allowed further simplifications of the system. With the new knowledge gained, further origin-of-life-models can be changed accordingly and it is expected that many of them will be able to show emergent replication systems. The even more interesting question is whether we now have a guideline to realize emergent replication systems in real physical and chemical environments. It is the hope of this work that it can really produce hints on how chemical experiments have to be setup. A bridge between computer-science and experimental origin-of-life research is now visible and future work certainly will strengthen this tiny pathway.

Acknowledgments

The careful reading of the manuscript and the year-long support of John S. McCaskill is greatly acknowledged. Additionally, sincere thanks are given to the people, especially Brigitte and Merle, who being patient and cheerfully when confronted with my odd questions and uncertainties. This work was partly financed by the European Center of Living Technology in Venice, Italy and the Programmable Artificial Cell Evolution project PACE EU-IST-FP6-FET-002035 of the European Union.

References


