

From evolving software towards models of dynamically self-assembling processing systems

Uwe Tangen

7th April 2006

BioMIP Ruhr-University Bochum c/o BMZ Otto-Hahnstr. 15, 44227 Dortmund

Abstract

Several types of micro-controllers are put into a primordial soup abstracted as a ring. The special feature of this setup is that self-replication is not possible. Micro-controllers only have access to foreign program-code not to their own. It turns out that the seeding programs vanish long before being able to proliferate when each program is automatically granted access to the neighboring code. An active attachment-procedure with a lock-and-key scheme (i.e. specific binding) is required to allow an evolutionary start. As expected, the error-threshold [1] applies here as well. As a second step, these fully fledged assembler programs are then partitioned into dynamical assembling pieces of simple linear program pieces essentially without any further control-structures but exhibiting the same functionality. Two different micro-controllers are studied under evolutionary conditions and longterm evolutionary behavior is investigated. A pathway towards the direct simulation of self-assembling nanoscale chemistry is opened.

1 Introduction

The wish to understand the origin of complexity in life is as old as science and mankind. With the advent of electronic computing machines, experimental studies [2, 3] have been backed up by modeling on computers. The first serious scientific modeling of a self-reproducing system was devised by von Neumann [4] in the attempt to deal with the problem of an automaton capable of self-replication. Via a series of intermediate simplifications, Life as a game [5] marked as a general endpoint in developing simple deterministic cellular automata as model-platforms. In the seventies, Holland, the inventor of Genetic Algorithms and Classifier Systems [6, 7] created also the α -universe [8]. With this model he tried to mimic the genetic replication apparatus in a one-dimensional simple string-processing system. Unfortunately, McMullin [9] could show that side-reactions in this model destroyed the self-reproducing capabilities of entities.

More or less parallel to this strand of research, the first observed computer-viruses sparked research in evolving software, which became prominent with the game CoreWars [10], a collection of instruction-pointers of small von Neumann-micro-controllers working again on a one-dimensional circular string as was the case with Holland's α -universe. CoreWorld [11] and Tierra [12] followed this avenue and could at least show phenotypical behaviors found in the Cambrian explosion. Again unfortunately, Tierra has subtle problems severely limiting the explanatory power of origin-of-life scenarios [13].

From a biological point of view, experimental studies in the sixties and seventies [14, 15, 16] led to the development of the quasi-species theory by Eigen [1], showing that replication errors drastically limit the amount of information which can be stabilized in replicating systems (error threshold). Further, Artificial Chemistry took a more operator-based view from chemistry on the self-replicating computer entities [17, 18], see [19] for a review. Trying to unite quasi-species theory and evolving software in micro-controllers gave rise

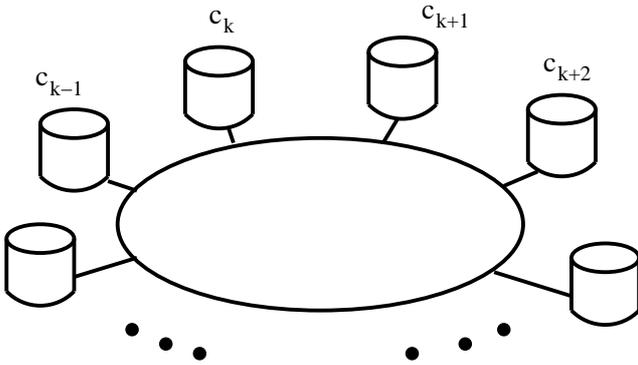


Figure 1: Spatial setup of the evolving software system. N containers with m individual entities are connected like beads on a ring. Rare exchange events between neighboring containers mimic a slow diffusion of particles along the ring.

to a combined model of stack-automata [20] which could demonstrate, again in a one-dimensional ring of evolving entities, that spatial properties strongly affect evolutionary properties. These spatial properties have been investigated in detail by McCaskill and co-workers [21, 22] accompanying a series of experiments [23, 24, 25, 26] investigating evolutionary properties of *in vitro* replicating systems. Combining spatial organization with self-assembly then could show enormous evolutionary power in evolving multipliers and other tillable optimization problems [27].

All these experimental and theoretical studies on replication and evolution culminated in the endeavor to really build artificial cells, see protocell <http://protocell.org> for an overview on some of these projects. Especially the EU-granted PACE project asks the question on how computer-science can take advantage of possible information processing in primitive artificial cellular organisms. One avenue in getting artificial cells realized is described in Rasmussen et al.[28]. The central question in artificial cell research inevitably will become how hereditary information in these cells can be stabilized and make an essential contribution to the survival and robustness of these primitive entities. Though quasi-species theory, spatial organization and artificial chemistries have provided important insights into these types of problems a dynamical model showing the possibility of robust algorithmic self-replication is still missing. The current paper tries to dig a little bit further into this missing link between computer-science and biochemical experiments.

2 The model investigated here

The model presented has its roots in the attempt to extend the quasi-species theory with functional properties, i.e. adding operators and simple stack-automata [20] and using real massively parallel reconfigurable hardware [29, 30, 31, 32] investigating electronic hardware evolution. Thus, the model is inspired by being realizable in real hardware and in the context of the PACE project showing a route towards the realization in wet chemical systems. Essentially the model is a population of processors and their programs interacting pairwise in containers which are distributed in a one-dimensional topology.

2.1 Code-density-optimized micro-controller

Computing machines can be as simple as a Turing-machine and as complex as a modern XEON-processor from Intel. All realized computing machines always resemble a certain compromise between computing power, code-density, power-consumption and ease of compiler-development. Especially, micro-controllers with very few resources suffer from this compromise: code-density and power-consumption are central optimization criteria, because most memory and thus silicon-area is consumed for storing the program in the

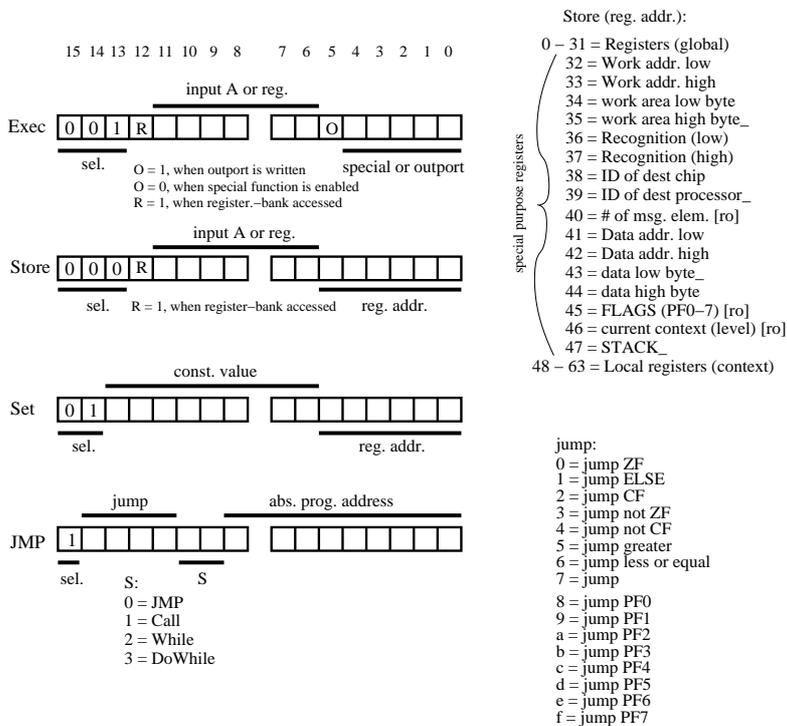
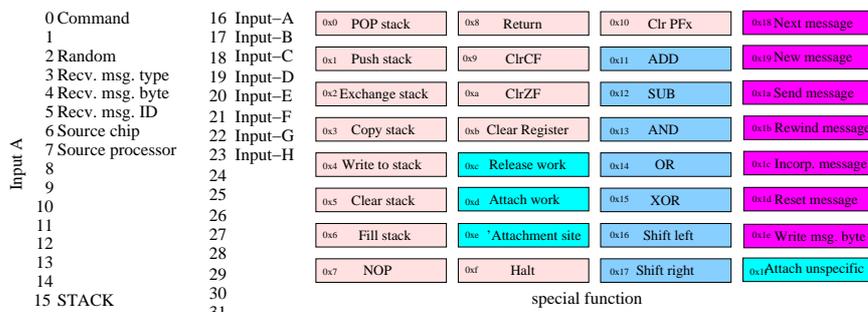


Figure 2: The code-density optimized micro-controller is a 16 bit micro-controller with Harvard architecture and 8-bit wide registers. The additional data-memory area is 16-bit wide and big enough to store a whole program of length 1024 instructions.

micro-controller. The micro-controller shown in Figure 2 is a late derivative of the simple micro-controller [33] provided by Xilinx, the commercial leader in reconfigurable logic worldwide. A simple macro-assembler has been developed allowing the use of structured programming constructs like if-then-else, for-next, while and do-while. A predecessor of the micro-controller in Figure 2 has been realized and used in the machine MereGen [34]. Essentially, it is a 16-bit micro-controller in Harvard architecture with 8-bit wide registers and a 16-bit wide data-area of size 2048 words. Standard arithmetic and Boolean operators are available with a small stack and a very primitive interrupt capability. Programs can become as long as 1024 instructions and several input- and output-ports allow for the communication with other processors. In addition, a message-passing-interface is provided but the simulations reported here are not using messages at all.

2.2 Topology and processor interactions

Currently, the model is embedded in the more or less standard ring-topology already employed in the very many investigations exemplified before, see Figure 1. In the simulations reported, the number N of containers is 32 and the number m of processors per container is 16 in the standard case and 256 in the self-assembling case.

A container is a well-stirred reaction vessel with no further spatial organization. Each processor is able to issue a command *SITE*, see Figures 2 and 5 which action is to identify this processor in the container with a marker. This recognition site is stored in a binary tree to allow for a rapid attachment procedure. Two different types of attachment are available (in the simulations used so far): specific and unspecific. Specific, means the attaching processor tells the environment exactly of what type the foreign processor has to be. A molecular equivalent would be to change the molecule's conformation and presenting an attachment site to the environment but these functions also may be realized via collections of molecules. In the case of the density optimized micro-controller in Figure 2, 64 different recognition sites are possible. In the case of the more simplified processor, Figure 5, even 255 different recognition sites can be addressed. The second type of attachment is fully unspecific, meaning that a processor takes whatever processor is randomly selected. Both types of attachment procedures assure that the processor cannot attach itself. Currently each processor can have only one other processor attached at the same time -> only bimolecular reactions are possible, this of course, can be easily extended in the future. Each processor actively can detach (special command *Release Work*) its workload. This certainly is a major problem for macromolecules (dissociation problems always were important in origin-of-life experiments) and has to be investigated further.

After a certain time, depending on the simulation parameters, two neighboring containers are randomly selected and from each a processor is picked at random. Both processors are exchanged – essentially the program code is exchanged. In addition, depending on the simulation parameters, processors in the containers are randomly subjected to mutations – instructions are picked randomly and replaced by new random instructions. The mutation probability specified defines the probability of replacing a single instruction in a program. Because this is no optimization study, genetic crossover or other types of higher-level variation are not considered.

2.3 Enzyme-like replication assembler program

In the attempt to model chemical replication systems enzyme like replication seems to be better suited than self-replication which usually are found in cellular organisms. The seeding program being used in the simulations is shown in Figure 9 in the appendix 6. The general control-flow of the program is as follows: Try to establish contact to a neighboring different processor of a certain type (special command *ATTACH*, see recognition-sites in section 2.2) which program-area is then mapped into the reach of this program (see registers *Work addr. low*, *Work addr. high*, *work area low byte* and *work area high byte*).

In case of success of this attachment procedure a flag is set and the program then copies at least a part of the foreign code into its own data-area. When executed the next time the program recognizes the filled data area, tries to attach another processor, now unspecific (see special command ATTACH UNSPECIFIC) and starts copying the data into the then mapped code area of the foreign processor. Of course, each processor only is given a certain slice of instructions executions and if the copying process is not finished in time, another processor starts execution and might destroy the already copied contents. Because each active processor has full access to the code-area of the attached one, it can do all the calculations necessary to improve robustness and/or to edit the contents at will. The active processor does not have access to the inner-states of the attached one. No registers or flags or the memory in the data area are available from outside. On the other hand, a processor just being mapped by a foreign processor has no knowledge about this fact. In addition, no processor has access to its own code-area, making true self-replication impossible. Neither is a processor able to get any information about its environment other than inspecting the code-area of other processors. Of course, as has been pointed out by [9] and others, this type of organization is extremely sensitive to perturbations from parasites and other side-reactions in the system. A further problem is brittleness [11], making evolution of about 130 instructions in this case extremely difficult, because almost all mutations not only lead to non-functional programs but to a considerable extent of programs acting as side-reactions destroying other programs.

2.4 From structured towards dynamical assembled programming

Usually assembler-programs are like spaghetti-code. Macro-assemblers alleviate these highly-optimized cryptic pieces of software. As has been outlined in section 2.1 essentially two (typically four) different types of control-structures are available for allowing universal computation – if-then-else and while-do (the others switch-case and for-next can be easily replaced by the former).

The central idea of self-assembling programs is to replace each sub-block of these control-structures by a separate subroutine which is no longer called as in the usual case but connected via an attachment procedure (like co-factors in typical enzymatic reactions or runtime loading of methods in object-oriented programming).

Note: If there are several subroutines with the same recognition site, a subroutine is picked randomly.

The **if-then-else** construct becomes a special subroutine-call, e.g.

```
call_if == 0 free_working_area copy_working_area
```

and the **while-do-loop** is transformed into, e.g.

```
call_while > w_start_addr dau_to_work_loop,
```

meaning that if the *accu* has value zero then the subroutine `free_working_area` is executed, or else the subroutine `copy_working_area` is executed. Or in the second example, as long as the *accu* has a value greater than the value in register `w_start_addr` the function `dau_to_work_loop` is called repeatedly. When control has been switched to a subroutine the then active micro-controller can read the inner states and registers of the calling micro-controller. Its own inner-states and registers are hidden then!

This simple shift in semantics has severe consequences. A program no longer is a big continuous chunk of instructions but distributed onto several processors each able to execute independently the subroutine. From the current 130 instructions 14 programs are

formed with a typical program length of only a few instructions. Not only the possibilities of exploitation increase, an inherent robustness can be observed. Several copies of a subroutine might exist, giving the calling program the chance to catch a subroutine which is still functional. Of course, the opposite is also true: subroutines with the correct recognition site but wrong program-code might also easily occur. It is to be expected that the evolutionary behavior of a dynamically self-assembling system is basically different to classical software evolution.

2.5 Complexity measures

In the case of an absent optimization criterium, it is extremely difficult to find out what really is going on in the system. Tedious analysis of evolved programs is in many cases impossible when dynamical self-assembly is in operation. The static view of the programs does not tell which instance of a subroutine really had been taken. One possible way out of this observation dilemma is to look at pattern repeats. The idea is that all sequences which are evolutionary successful should be abundant because otherwise they might be wiped out by accident. To look at the whole evolved system of programs, a special algorithm has been developed which finds all repeats of arbitrary length, see [35] for further details on the algorithm.

All program-codes of the whole system are concatenated to a single long string with special markers denoting the begin and end of each program. Instead of taking the original instructions as letters of the alphabet (in the first case this would give 65K different letters) a coarse graining procedure is done before the pattern analysis: this coarse graining results in 41 different symbols per instruction in the first case and 24 different symbols in the much simplified case. This coarse-graining reflects the control-structure in the programs. Constants or attachment sites are not considered specifically here.

The complexity measures are:

- AVERAGE is just the average length of patterns, $average = \frac{1}{n} \sum patlen$, found having a minimal given frequency in the system of all program codes,
- NEGENTROPY tells something about the regularity of the found pattern lengths, $negentropy = \sum patlen * \log(patlen)$. The term entropy here is not quite correct because $patlen$ is not between 0 and 1,
- TREEAREA is a normed area of all patterns found times the maximum length of a single pattern, $treearea = \frac{maxpatlen * nrpat}{seqlen}$, accounted in the whole system and
- EFFORT counts the number of processing steps needed in per element to find all repeats in the system, $effort = \frac{noperations}{seqlen}$.

2.6 Preliminary evolutionary results I

Typical simulation runs are done ten times and the average is plotted with error-bars attached. To get an idea of the interaction consequences, always two runs per parameter set are undertaken, one with attachment effectively switched off and the other with normal interactions. Initially all programs are preset with random instructions but a few (two programs out of 16 possible in each container) seeded with the replicator programs shown in the Appendix 6. In case of dynamical assembling two times 14 programs out of 256 available are seeded in each container. The registers and data-areas of the processor are set to value null each.

2.6.1 Standard evolving programs

It was not expected to suddenly see unbounded growth of complexity. Indeed, with a low mutation rate the system pretty quickly gets trapped in a trivial configuration and is from then on only optimizing robustness with even simpler systems emerging.

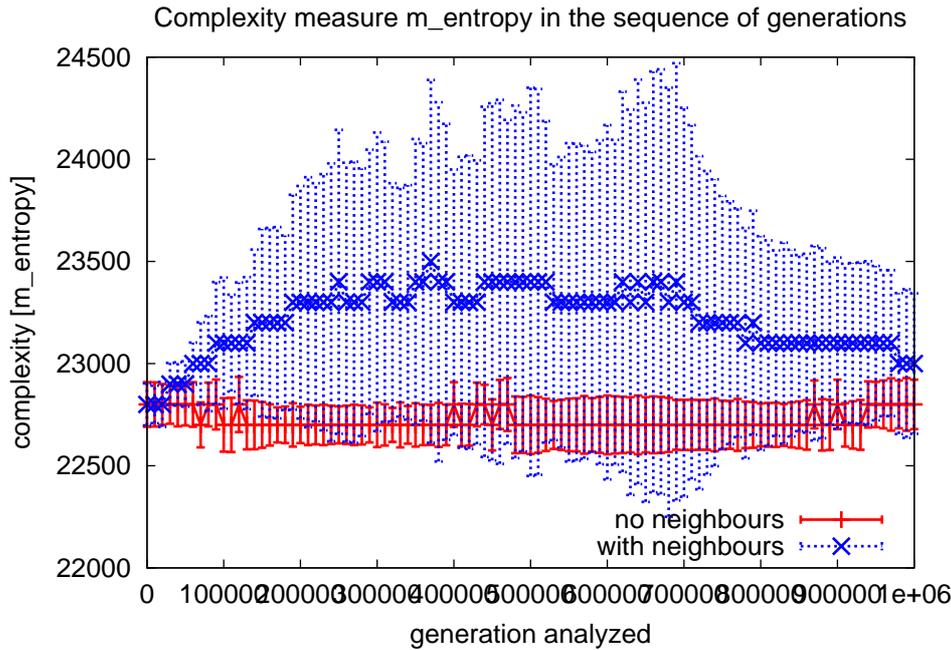


Figure 3: One million generations with all repeats occurring at least four times and the NEGENTROPY-measure used.

Thus the change in measured complexity, see Figures 3 and 4 is mainly due to side-reactions destroying randomness and creating program-codes with all instructions the same value or very very simple patterns.

3 A second micro-controller – strongly simplified

Analyzing the 130 instructions of the normal seed program reveals that large parts of the code are spent by creating counters and tracking how many instructions already be copied. This overhead in program complexity can – with only a negligible loss of generality – be avoided when shifting the physics of the system towards string processing, see the α -universe [8] already using it. With string processing not only counters can be avoided, in addition, the whole arithmetic machinery is superfluous too – under an evolutionary viewpoint arithmetic is very costly and brittle. The introduction of strings yield that each string is stopped via a null-element (hex 0x0) and each memory access, being it data or work-load, followed by an increment of the address.

In principle, even the Boolean logic processing capability is no longer needed because the knowledge of an attached processor can also be signaled via one of the four flags available. But removing the Boolean operators from the system makes universal computation of these types of processors extremely difficult (see the discussion on universality with the game Life) and evolutionary intractable and thus effectively throwing out all higher level computations possible. Only taking the minimum necessary Boolean logic of course (only NOR or NAND) is feasible but doesn't change the picture too much. See Figure 5 for the specification of the such reduced micro-controller. Be aware, aside from the missing arithmetic, the programming access and the dynamical assembling remains untouched. It can be seen that now only three instructions are needed. The whole micro-controller shrunk from 16-bit to 8-bit and thus all data and work-load areas also became 8-bit wide. Only the EXEC-instruction need an additional byte to specify recognition sites and absolute values being written into the registers. The number of special commands was halved and the number of jump-bits could also be reduced because of the missing arithmetic. Questions like greater, or less-or-equal can no longer be posed, a carry flag no longer is needed.

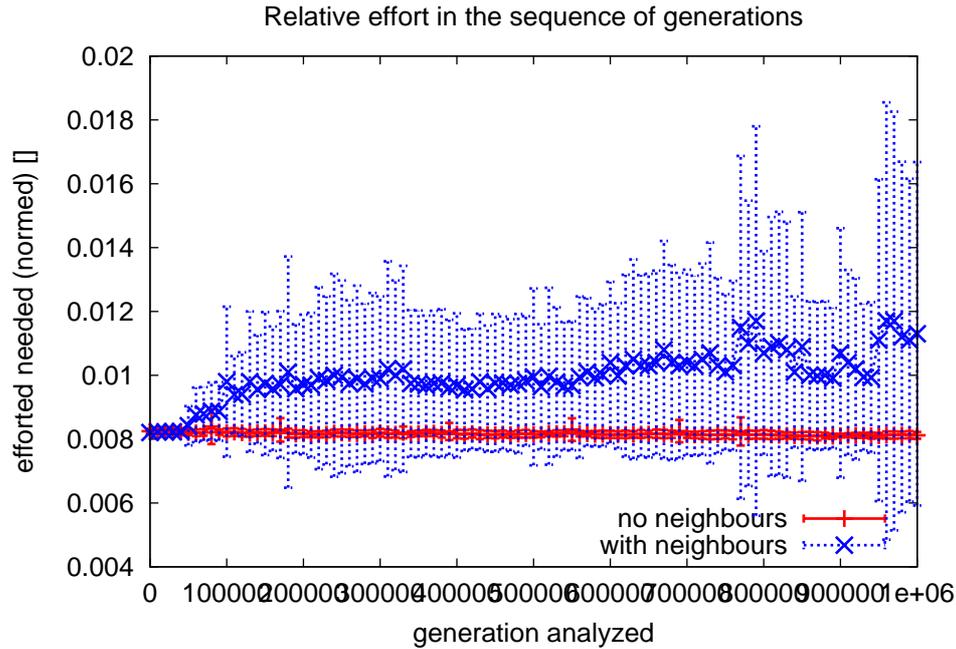


Figure 4: The same simulation as in Figure 3 but now with complexity measure EFFORT. The slight increase in finding all repeating patterns stems from the increasing ambivalence when analyzing sequence with many simple concatenated repeats.

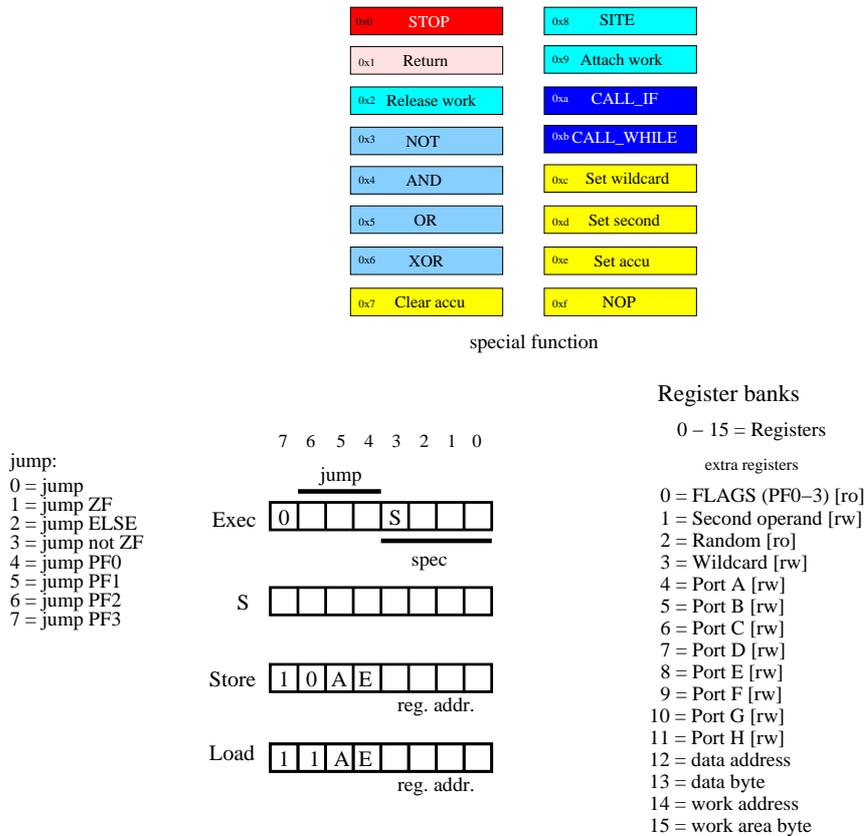


Figure 5: A much simplified micro-controller. When bit **A** is set the data is stored in the *second*-register else in the *accu*. A bit set in the *wildcard*-register means masking of the attachment-bit. The second byte of the EXEC-instruction is used when the bit **S** is set and every null (0x0) means end of sequence.

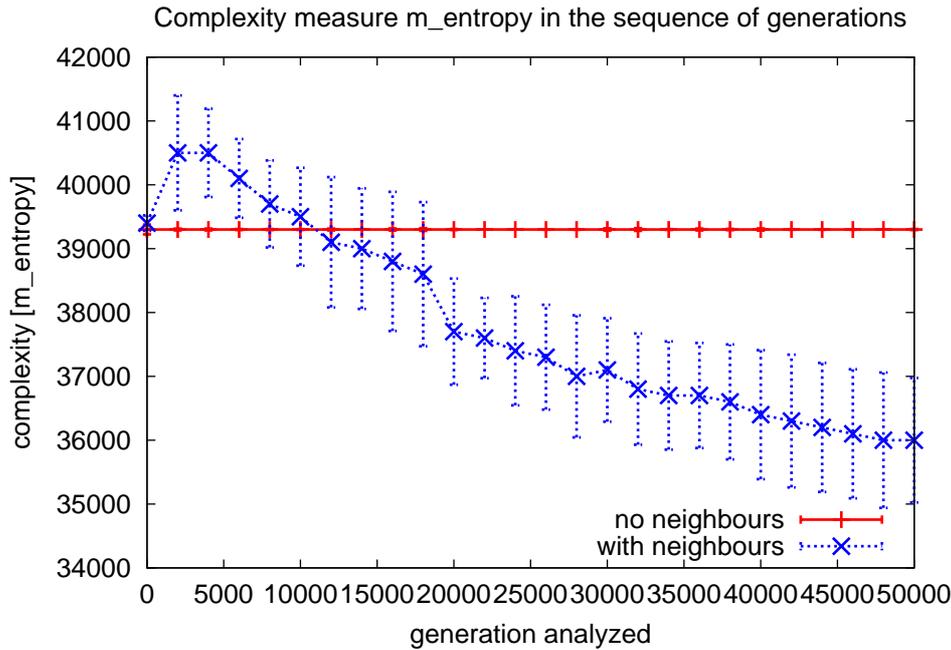


Figure 6: The NEGENTROPY with dynamical assembling the subroutines.

The Zero-flag and the four special flags are the only ones which can be used in conditional expressions.

A further important aspect turns out with this type of encoding: all special commands are now under the control of conditions. For example, the AND-operation can be done when the Zero-flag is set and other the accumulator is not changed. This conditionality of all special commands can lead to very powerful programming and it would be of interest in itself if evolution is able to utilize these features.

3.1 The self-assembled program evolving

Because of the considerable shorter program-length (the number of bits having to be stabilized in an evolutionary context is now about a third compared to the micro-controller in Figure 2) in the dynamical assembly case all time-spans are different compared with the standard case described in section 2.6. Because dynamical assembly is consuming quite a lot of CPU-time the number of generations simulated is considerably less than before. Nevertheless, some interesting aspects can be noticed. In Figure 6 the NEGENTROPY measure even declines with a minimum number of four instances per repeat and time. This behavior is reversed when a minimum of eight instances is required (data not shown). The reason is that the pattern length in the system becomes more uniform due to the exploitation via very short subroutines essential presetting other programs with simple values, very often creating strings of maximum program length (in that case 128 instructions). This uniformity is no longer so obvious when more instances per repeat are required.

Though not very pronounced, there is a slight increase in complexity concerning the effort to find all repeating patterns, see Figure 7. This slight increase stems from the increasing ambivalence of repeats of repeats when sequences become more homogeneous. Looking at the other complexity measures AVERAGE and TREEAREA (data not shown) one clearly can see that AVERAGE considerably increase whereby TREEAREA is declining, explaining that the number of long sequences with only one single instruction populate the whole system.

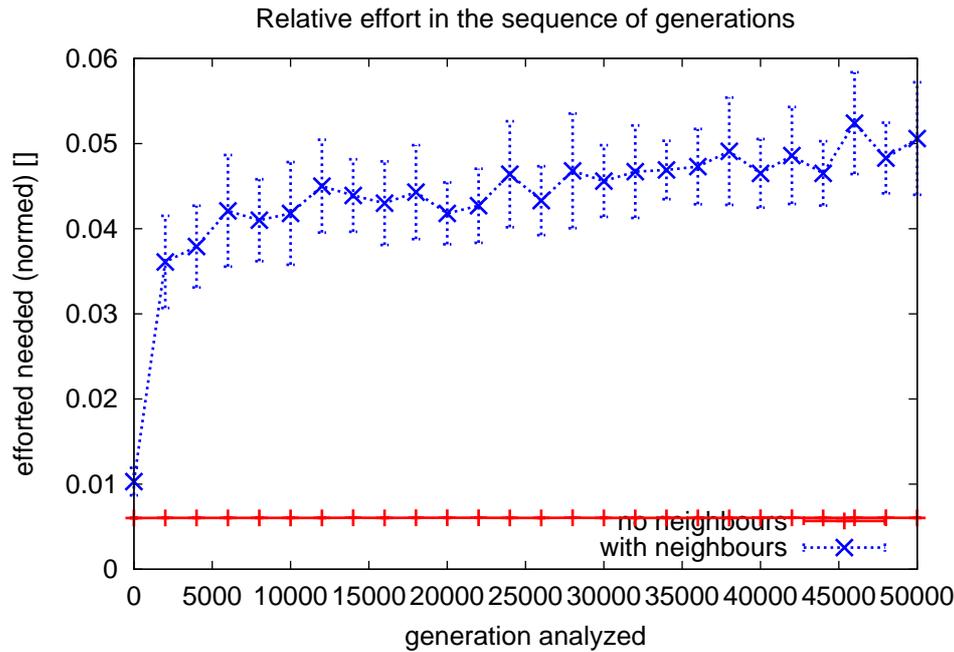


Figure 7: The EFFORT measure when using dynamically assembly. The slight increase is due to the increasing ambivalence when analyzing homogeneous sequences.

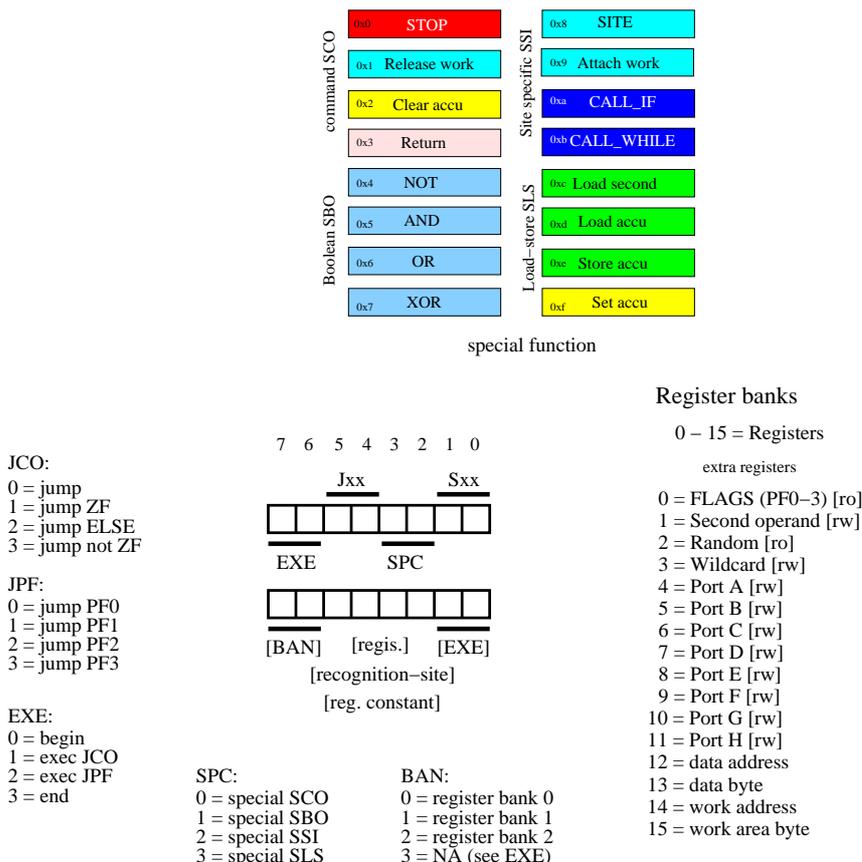


Figure 8: Simplifying the micro-controller even further, in this case without losing functionality, abandons the concept of instructions altogether, only one instruction is left and this one need no longer a special notation. Essentially, this micro-controller, though written as an 8-bit type, is a two-bit type micro-controller. Of course, registers might be 8-bit wide or of any other size, because register addressing is now also done via a string and potentially as a dynamical assembled register.

3.2 An even more simplified micro-controller

We can simplify the micro-controller even further, see Figure 8. Without loosing functionality instructions are abandoned altogether, only one instruction (EXEC) is left and it no longer needs a special notation. Essentially, this micro-controller, though written as an 8-bit type, is a two-bit type micro-controller. Of course, registers might be 8-bit wide or of any other size, because register addressing is now also done with a string. Additionally the size of recognition sites can now be adjusted at will. The consequence of this machine is that it becomes more and more reminiscent of an automaton and a Turing-machine. Another feature appears: the special commands are no longer context-free. Depending in which state the micro-controller just is, the meaning of these bits change. Thus different reading frames and all the other features known from viruses become meaningful now. Last but not least, because of only two bits remaining the whole program-code can be written easily in quaternary notation with probably optimal evolutionary traits, see Fontana [36].

4 Discussion

The key distinguished feature of dynamically self-assembling software is that there is a canonical partition of structured programming: namely, put every block (think of Nassi Shneiderman diagrams) into a different processor and realize the jump to the correct entry-point via an attachment procedure. This seemingly small shift in semantics has dramatic consequences in the evolutionary system:

- The number of processors increases considerably, whereby the average single program length decreases considerably.
- A main-program can no longer be sure that the subroutine it called is from its own offspring. It might even have a completely unknown functionality, but it can still be a valid subroutine though the original one has been destroyed -> the systems robustness increases.
- Most importantly: the big problem of needing long sequences of instructions being copied before a new replicating program can be established is now reduced to **copying many but small programs**. In view of considerable error-rates and side-reactions this can be the way out off this so far unsolved hen-and-egg problem.
- From a computer-science point of view the control-structure of a sequential program is mapped onto a network of communicating processors each doing a much simpler task than before – but, the overall complexity has not been reduced! Meaning that the evolutionary task as such is NOT simpler, on the contrary, side reactions are much more pronounced now.
- This framework allows a more or less continuous pathway from classical concepts of von Neumann machines towards Holland’s classifier systems, the α -universe, functional programming languages, Lindenmayer systems or even cellular automata. These several entirely different computing paradigms can now be viewed from a common basis and be programmed by a structured assembler code.
- There is a semantic shift with context-dependent codons possible. This semantic shift capability is structurally different from questions on evolvability in genetic and evolutionary algorithms.
- It also becomes clear that the question on evolvability in this context is an entirely different one compared to classical optimization studies in genetic or evolutionary algorithms, see [37], because evolvability here means using semantic shifts as the major source of new inventions. Nature is extremely brilliant in using these shifts.

5 Conclusions

The most fascinating aspect of this work to me is the common framework now established. The vast majority of computing paradigms can now be looked at from a single viewpoint and thus be questioned as to how they can contribute to questions of evolvability and modeling of macromolecular dynamical regimes. The complexity of the micro-controllers used can be changed from standard von Neumann type to string- or list-processing machines, even to cellular automata or enhanced Turing-machines. It is easy to exchange them and a common assembler-code interface allows for a quick testing of different instruction sets. Especially, when changing instructions to improve evolvability and/or to ease the mapping of micro-controller instructions towards realization of these computations in systems with real macromolecules, the biological consequences can be clearly seen and motivated.

Another central advantage of this self-assembly ansatz is the simplicity of the now linear sub-routines. The mapping of the functionality of these code-patches onto real molecules should now be much easier. In addition, the way of thinking towards attachment sites and the then inherent parallelism leads to new insights and a more biological way of thinking.

One aspect which already turned out to be feasible is to bypass the criticism of [9], which certainly applies to many other origin-of-life-models, by the experimental observations made in [25] that side-reactions might be not only be harmless but even the cause of evolutionary robustness, if they can be incorporated by the evolving system. Of course, it can not be expected to design such a system by hand and the framework presented here might offer valuable help.

Acknowledgments This work is partly supported by the PACE-project EU-IST-FP6-FET-002035. The idea to employ self-assembly arose in a discussion with J. S. McCaskill following earlier work co-pioneered by T. Maeke and R. M. Füchslin [27], with whom discussions were also appreciated. The fruitful discussions B. McMullin are appreciated too.

6 Appendix

References

- [1] M. Eigen. Selforganization of matter and the evolution of biological macromolecules. 58:465–523, 1971.
- [2] L. E. Orgel. Molecular replication. 358:203–209, 1992.
- [3] A. Lazcano and S. L. Miller. How long did it take for life to begin and evolve to cyanobacteria. 39:546–554, 1994.
- [4] J. von Neumann. *Theory of Self-Reproducing Automata*. Burks, A. W. University of Illinois Press, Urbana, 1966.
- [5] E. R. Berlekamp, J. H. Conway, and R. K. Guy. *Winning Ways for Your Mathematical Plays*. Academic Press, New York, 1982.
- [6] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, 1975.
- [7] J. H. Holland. A mathematical framework for studying learning in classifier systems. 22:307–317, 1986.
- [8] J. H. Holland. Studies of the spontaneous emergence of self-replicating systems using cellular automata and formal grammars. In A. Lindenmayer and G. Rozenberg, editors, *Automata, Languages, Development*, pages 385–404. North Holland Publishing Company, Amsterdam, 1976.

```

... declarations here ...
site_a          ; Declare this program to belong to site A
out 0x0 enable  ; Tell the environment that we are able to copy
st 0x0 address_high ; where the length of the code in
                  ; the daughter is stored.
st 0x0 work_addr_high ; Reset the address in accessing work area
ld flags
and work_avail   ; We mask out the other flags
if == 0          ; With a working area available continue
attach s_b      ; Search for a program which contains
                  ; SITE_B as splicing site
endif

st 0x0 work_addr_low ; Reset the address in accessing work area

st anz_ele_addr address_low ; Write 0 into register address
ld dau_low_byte
if == 0          ; If zero then the daughter area is free
                  ; and we can start copying the other
                  ; working area part into the daughter
st neigh_n0_addr address_low
st work_area_low_byte dau_low_byte ; Remember to others ID
st work_area_high_byte dau_high_byte

st anz_ele_addr address_low ; Write 0 into register address
st 0xc0 dau_low_byte ; Write the number of bytes to be copied
st 0x0 dau_high_byte

call cp_work_to_dau ; Copy working area into daughter
release            ; Release the working copy
at_unspec         ; Search for another program in which the
                  ; copied daughter can be copied.

else              ; We do have bytes in the daughter
st 0x0 known_neigh
st neigh_n0_addr address_low
ld dau_low_byte
sub work_area_low_byte
if == 0
ld dau_high_byte
sub work_area_high_byte
if == 0
st 0x1 known_neigh
endif
endif

ld known_neigh
if == 0          ; The copied daughter program is not from
                  ; the current neighbour, thus we might
                  ; copy now.
call cp_dau_to_work ; Copy daughter into the working area

st anz_ele_addr address_low ; Write register address
st 0x0 dau_low_byte
release            ; Release the working copy
endif
st anz_ele_addr address_low ; Write register address and
st 0x0 address_high         ; Also reset the upper byte
st 0x0 dau_low_byte         ; and reset counter for new run

endif
halt

```

Figure 9: A simple enzyme like replication assembler program executed by the micro-controller in Figure 2.

```

; Hardwareevolution with MERLE (evocpu_d_psm)
;
; A simple replicating program. General idea is to copy
; the code of a neighboring program into the daughter
; memory and to wait until another neighboring program is
; available, in which then the stored code is copied.
; Program and data is interpreted as string with auto-increment
; of addresses.
;-----
; Subroutines in this file:
;-----
;
;
; INI_REG
;
;
;
; IN: Non
; OUT: several registers
;
;-----
; A site 0x0 would yield a premature stop of copying process!
def s_a      0x2 ; Identifier for splicing site A
def s_b      0x4 ; Identifier for splicing site B

def work_avail 0x1 ; Flag is set if a working area is available

reg flags      0x10
reg second     0x11
reg random     0x12
reg wildcard   0x13
reg port_a     0x14
reg port_b     0x15
reg port_c     0x16
reg port_d     0x17
reg port_e     0x18
reg port_f     0x19
reg port_g     0x1a
reg port_h     0x1b
reg address_low 0x1c
reg dau_low_byte 0x1d
reg work_addr_low 0x1e
reg work_area_low_byte 0x1f

;-----
;
;
; MAIN
;-----
site s_a          ; Declare this program to belong to site A
ld flags
and work_avail   ; We mask out the other flags
call_if == 0 attach_dau ; With a working area available continue

cla
st address_low   ; Write 0 into register address
ld dau_low_byte

; If zero then the daughter area is free
; and we can start copying the other
; working area part into the daughter
call_if == 0 fill_data_area copy_data_area
halt

;-----
; If working area is empty fill it now.
;-----
func fill_data_area
; If zero then the daughter area is free
; and we can start copying the other
; working area part into the daughter

```

```

cla
st address_low ; Write 0 into register address
st work_addr_low ; Reset the address in accessing work area
ld l
call_while != 0 work_to_dau_loop ; As long as there are lines copy
release ; Release the working copy
at_unspec ; Search for another program in which the
          ; copied daughter can be copied.

return ; fill_data_area is finished!!

;-----
; Copy program from work to data area
; The number of elements to be copied is stored at
; the daughter address 0. The copy region starts
; at the address daughter_start (must be below 256)
;-----
func work_to_dau_loop
ld work_area_low_byte
st dau_low_byte ; Copy lower byte

return ; work_to_dau_loop is finished!!

;-----
; Copy data area into working area
;-----
func copy_data_area
cla
st address_low ; Write 0 into register address
st work_addr_low ; Reset the address in accessing work area
ld l
call_while != 0 dau_to_work_loop ; copy lines
cla
st address_low ; Write 0 into register address
st dau_low_byte ; Reset the daughter again
release ; Release the working copy
call attach_dau ; Attach the next neighbour
return ; copy_data_area is finished!!

;-----
; Attach another program
;-----
func attach_dau
attach s_b ; Search for a program which contains
          ; SITE_B as splicing site
return ; attach_dau is finished!!

;-----
; Copy program from daughter to work
; The number of elements to be copied is stored at
; the daughter address 0. The copy region starts
; at the address daughter_start (must be below 256)
;-----
func dau_to_work_loop
ld dau_low_byte ; Load lower byte
st work_area_low_byte ; Copy lower byte
return ; dau_to_work_loop is finished!!

```

Figure 10: The dynamical assembled program of the micro-controller in Figure 5

- [9] B. McMullin. The holland α -universes revisited. pages 317–326. "", 1992.
- [10] A. K. Dewdney. Computer-kurzweil. 10:8–11, 1988.
- [11] S. Rasmussen, C. Knudsen, R. Feldberg, and M. Hinsholm. The coreworld: Emergence and evolution of cooperative structures in a computational chemistry. 42:111–134, 1990.
- [12] T. S. Ray. An approach to the synthesis of life. In C. G. Langton, C. Taylor, J. D. Farmer, and S. Rasmussen, editors, *Artificial Life II*, pages 371–408. Addison-Wesley, New York, 1991.
- [13] R. K. Standish. Open-ended artificial evolution. 3:167, 2003.
- [14] S. Spiegelman, I. Haruna, I. B. Holland, G. Beaudreau, and D. R. Mills. 54:919, 1965.
- [15] C. K. Biebricher, M. Eigen, and W. C. Gardiner. Quantitative analysis of selection and mutation in self-replicating rna. pages 317–337, 1991.
- [16] C. K. Biebricher, M. Eigen, and J. S. McCaskill. Template-directed and template-free rna synthesis by $q\beta$ replicase. 231:175–179, 1993.
- [17] J. S. McCaskill. *Polymer Chemistry on Tape: A Computational Model for Emergent Genetics*. Max-Planck-Society, Göttingen, Germany, 1988.
- [18] W. Fontana. Algorithmic chemistry: A model for functional self-organization. In C. G. Langton, editor, *Artificial Life II*, pages 159–202. Addison-Wesley, Reading, Massachusetts, 1991.
- [19] P. Dittrich, J. Ziegler, and W. Banzhaf. Artificial chemistries - a review. 7:225–275, 2001.
- [20] U. Tangen. *The Extension of the Quasi-Species to Functional Evolution*. PhD Thesis, Jena, 1994.
- [21] S. Altmeyer and J. S. McCaskill. Error threshold for spatially resolved evolution in the quasispecies model. 86:5819–5822, 2001.
- [22] J. S. McCaskill, S. Altmeyer, and R. M. Fuchsli. The stochastic evolution of catalysts in spatially resolved molecular systems. 382:1343–1363, 2001.
- [23] J. Yin and J. S. McCaskill. Replication of viruses in a growing plaque: A reaction-diffusion model. 61:1540–1549, 1992.
- [24] G. J. Bauer, J. S. McCaskill, and H. Otten. Traveling waves of *in vitro* evolving RNA. 86:7937–7941, 1989.
- [25] R. Ehricht, T. Ellinger, and J. S. McCaskill. Cooperative amplification of templates by cross-hybridisation (CATCH). 243:356–364, 1997.
- [26] B. Wlotzka and J. S. McCaskill. A molecular predator and its prey: Coupled isothermal amplification of nucleic acids. 4:25–33, 1997.
- [27] R. Fuchsli, T. Maeke, U. Tangen, and J. S. McCaskill. Evolving inductive generalization via genetic self-assembly. *Adv. in Compl. Systems*, 2005.
- [28] S. Rasmussen, L. Chen, M. Nilsson, and S. Abe. Bridging nonliving and living matter. 9:269–316, 2004.
- [29] J. S. McCaskill, T. Maeke, U. Gemm, L. Schulte, and U. Tangen. NGEN a massively parallel reconfigurable computer for biological simulation: towards a self-organizing computer. 1259:260–276, 1997.

- [30] U. Tangen, L. Schulte, and J. S. M^cCaskill. A parallel hardware evolvable computer POLYP: Extended abstract. *Proceedings of the FCCM'97 in IEEE Symposium*, 1:238–239, 1997.
- [31] U. Tangen and J. S. M^cCaskill. Hardware evolution with a massively parallel dynamically reconfigurable computer: POLYP. In M. Sipper, D. Mange, and A. Pérez-Urbe, editors, *ICES '98 Evolvable Systems: From Biology to Hardware*, volume 1478, pages 364–371. Springer, Heidelberg, 1998.
- [32] U. Tangen. Self-organisation in micro-configurable hardware. In M. Bedau, J. S. M^cCaskill, N. H. Packard, and S. Rasmussen, editors, *Artificial Life VII*, pages 31–38. The MIT Press, Cambridge, Massachusetts, 2000.
- [33] K. Chapman. Dynamic microcontroller in an XC4000 FPGA. *Xilinx Application Note*, 1994.
- [34] U. Tangen, T. Maeke, and J. S. M^cCaskill. Advanced simulation in the configurable massively parallel hardware MereGen. In K. H. Hoffmann, editor, *Caesarium 2000, LNCS*, pages 107–118. Springer, 2001.
- [35] H. D. Lohrer and U. Tangen. Investigation into the molecular effects of single nucleotide polymorphism. *Pathobiology*, 68:283–290, 2001.
- [36] W. Fontana, D. A. M. Konings, P. F. Stadler, and P. Schuster. Statistics of rna secondary structures. 33:1389–1404, 1993.
- [37] H. Suzuki. Evolvability analysis: Distribution of hyperblobs in a variable-length protein genotype space. In M. A. Bedau, J. S. M^cCaskill, N. H. Packard, and S. Rasmussen, editors, *Alife 7*, pages 206–220. MIT Academic Press, Cambridge, Massachusetts, 2000.