Fraglets – a Metabolistic Execution Model for Communication Protocols

Christian F. Tschudin
Computer Science Department, University of Basel, Bernoullistrasse 16
CH – 4056 Basel, Switzerland. Email: christian.tschudin@unibas.ch

***Abstract*— In this paper we introduce a molecular biology inspired execution model for computer communications. The goal is to lay the ground for automatic network adaption and optimization processes as well as the synthesis and evolution of protocol implementations. Our execution model is based on the unification of code and data, featuring a single unit called "fraglets" that are operands as well as operators. We have built a simulator and started to program classical communication tasks with fraglets that show metabolistic pathways patterns like the ones found in biological cells. We give an example of a fraglet implementation for a non-trivial flow–control–with–reordering protocol and briefly discuss how to search the fraglet program space with genetic algorithms.**

Keywords: **Fraglets, active networking, metabolistic computation, automatic protocol synthesis.**

I. Introduction

The hypothesis underlying our investigation is that the current process of protocol development – design, implementation, standardization and deployment – has hit a complexity wall that can only be overcome by automating the search for new or better algorithmic solutions. Network configuration is another field where the management of very large networks has to be automated and where optimization efforts should not require continuous human intervention. This applies for example to scenarios where each household hosts hundreds of network nodes as well as the current Internet where a large amount of manual management effort is needed to keep this infrastructure running. Our interest is in automating as much as possible of the network's operation, including the ability to adapt to changing border conditions and to evolve the network's functionality.

In this paper we introduce an execution model for communication protocols that resembles the chemical reactions in living organisms. The underlying theme is that nature has developed a mechanism both for continuous operations as well as evolution using the processing of macromolecules. We introduce "fraglets" as the equivalent of such molecules. Fraglets are small elements that represent *fragments* of a distributed computation. These computations are carried out by the processing and the exchange of fraglets. At the end there are no data or code packets: Because fraglets operate on fraglets – similar to molecules that operate on molecules – it is irrelevant to differentiate between "code" and "data". Although the old distinction might provide some guidance in the implementation process (by humans), these categories become void at the level of fraglet execution as well as the long term prospect of automated software evolution.

In the next section we will briefly review the main contexts underlying fraglets, namely "active packets" in computer networking and "metabolistic pathways" in molecular biology. In Section III a formal definition of fraglets will be given. We also introduce a small set of tag matching rules that becomes an instruction set well suited for the efficient implementation of protocol software. Section IV gives simple "fraglet processing" examples and shows a complex case of a flow control protocol implementation with fraglets. We discuss automated protocol synthesis in section V before concluding with section VI.

II. Related Work

A. Active Networking

The goal of active networking is to make a computer network "retargetable". This enables to customize a network and to decide at run time which function should be performed where [6]. Mobile code is used to inject new functionality at different levels of granularity: The "Programmable network" approach permits to remotely install complete software modules whereas "active packets" become self-contained capsule that control their own fate.

Active networking is in a first place a mechanism for handling mobile code in a data exchange network: Special languages and runtime environments are used that permit the conversion of program code into data packets as well as the implicit or explicit installation of code at the remote side. The conversion process from code to transferable unit is in many systems inaccessible or tightly controlled (e.g. Java based systems) in order to preserve the language's safety properties. In case of functional languages it is conceivable to use closures as active packets and to encode data in form of program instructions. As we will see, fraglets resemble this approach. However, their flat syntactic structure, the common storage space and the fraglet's operations for direct manipulation at the representation level is different and is directly geared toward mobile computations.

B. Cell Metabolism

The word metabolism stands for the chemical processes that take place in a living system. Typically, the cell's functioning is described as a continuous process where complex substances are broken down into simpler ones (catabolism) or energetically richer ones are assembled from simple units

(anabolism). Chemical pathways are a description of a cell's metabolism: They describe the chain of chemical reactions that implement catabolism or anabolism. Pathways are not simple "state transition diagrams" where one substance is transformed into another one: Instead, each segment of a pathway has an equilibrium as chemical reactions are usually reversible. Overall, however, the flow of activity has a direction that is driven by external energy wherefore metabolism is sometimes also defined as the capture, processing and transfer of (chemical) energy. One of the most known pathways is the "Tricarboxylic Acid Cycle" (TCA) where carbon molecules are converted to CO_2 and water. Figure 1 shows the stages of the TCA cycle, see also [1] for an online resource on metabolic pathways.

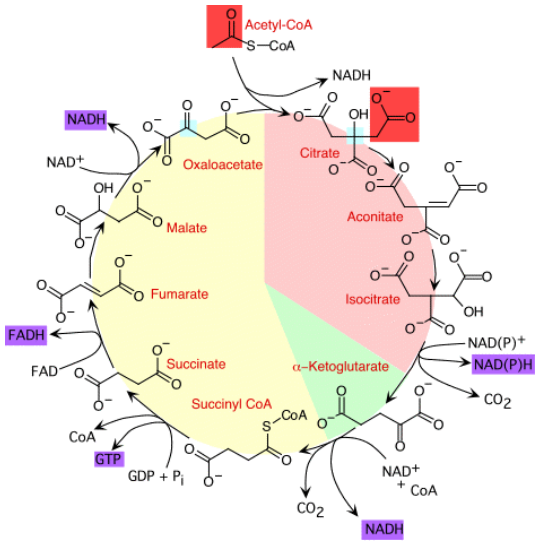

Fig. 1. Chemical pathway showing the TCA cycle [3].

Fraglets inherit the concept of chemical pathways but differ insofar as the fraglet model does not attempt to model chemical reactions. Fraglets can react only in very restricted ways and the operations are – in the current form of the model – irreversible.

C. Automatic Protocol Synthesis

There exist first results on the feasibility of synthesizing protocols and their implementations by automatic means. Perrig and Song describe in [2] their search for protocol specifications that satisfy a set of given security properties. Using an iterative deepening approach, a generator walks through the specification space and feeds the found candidates to a protocol screener. In [5] this work is extended by a code generation phase that transforms the found protocol specification into a Java implementation. Sharples and Wakeman show in [4] how protocol implementations can be synthesized from scratch. Using a genetic algorithm approach they evolve solutions with increasingly better fitness. The example protocol reported on is a reliable transfer protocol.

III. Fraglet Model

In this section we introduce the fraglet processing model and show its operations with a few basic examples. Fraglets have surprising strong ties to formal methods as well as molecular biology. At the theory level, fraglets belong to string rewriting systems and reach back to the work of Emil Post published in 1943. More recently, string rewriting (and splicing) has seen a renaissance in the context of DNA computing where it provides an adequate formal basis. For a more complete treatment we refer to [7].

A. Fraglet Packets and Tag Matching

Fraglets are symbol strings $[s_1 : s_2 \ldots : s_n]$ that represent data and/or protocol logic. Most naturally, fraglets are encoded as packets where the packet header fields contain the symbol values.

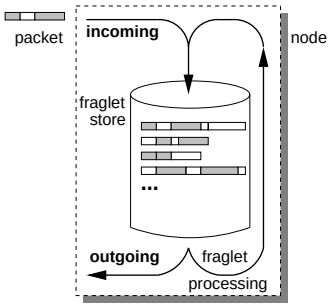

Fig. 2. Schematic model of a node storing and processing fraglets.

Each node in the communication network has a fraglet store to which incoming fraglets (packets) are added (Figure 2). The store implements a multiset i.e., if several instances of the same fraglet are received the system will keep track of its multiplicity. The node continuously examines the fraglet store and identifies which fraglets should be processed. To this end, a simple tag matching operation is used: Based on the front symbol of the fraglets the node can decide which action should be applied to one or more fraglets.

Simple actions lead to the transformation of a single fraglet, including its transfer to another node. More complex actions combine two fraglets as if they were involved in a chemical reaction. Both types of actions provide the basic steps through which a fraglet system makes progress. If several actions are possible at some point in time, the system randomly picks one action, atomically removes the involved fraglets from the store, processes them and puts potential results back into the store. It is also possible to execute fraglets in parallel as there are no side effects except for the atomic extraction from the store.

One important aspect of the tag matching is that the effort to decide on and execute an action should be bound. For example, we do not want to introduce complex pattern matching on fraglets. By limiting the tag matching to a few head symbols as well as restricting ourself to simple actions we can keep the time per fraglet operation constant. Also, assuming simple

enough operations we can imagine a hardware based implementation that processes fraglets "at wire speed".

B. Instruction Set for Fraglet Processing

We have defined a simple prefix programming language for fraglets. The instruction set is fixed and has six "transformation" and three "reaction" rules. A reaction means that two fraglets are jointly processed, while a transformation applies to a single fraglet only. In the following we denote fraglets as $[s_1 : s_2 : \ldots\ tail]$ where s_i is a symbol and $tail$ is a (possibly empty) sequence of symbols.

Transformation rules:

Op	*Input*	*Output*
nul	[$nul : tail$]	– (fraglet is removed)
dup	[$dup : t : u : tail$]	[$t : u : u : tail$]
exch	[$exch : t : u : v : tail$]	[$t : v : u : tail$]
new	[$new : t : tail$]	[$t : n_{i+1} : tail$]
split	[$split : t : \ldots : * : tail$]	[$t : \ldots$], [$tail$]
send	${}_A[\, send : B : tail\ ]$	${}_B[\, tail\ ]$ (unreliably)

The nul head, for example, means that a fraglet shall be removed from the execution context. dup duplifies the u symbol at the third position while the preceding field t will become the new fraglet's head symbol. $exch$ swaps the two symbols at the third and forth position, new creates a new and unique symbol for this context. The $split$ operator breaks the fraglet in two at the first marker position ($*$). Finally, the $send$ head symbol is responsible for (unreliably) transferring a fraglet to another context whose name is given by the second symbol. We use a subscript prefix ${}_X[\ldots]$ to specify the place where a fraglet is stored.

Reaction rules:

Op	*Input*	*Output*
match (merge)	[match : s : tail_1], [s : tail_2]	[tail_1 : tail_2]
matchP (persist)	[matchP : s : tail_1], [s : tail_2]	[tail_1 : tail_2] [matchP : s : tail_1]
matchS (sustain)	[matchS : s : t : tail_1], [s : t : tail_2]	[tail_1 : tail_2] [s : t : tail_2]

Each $match$ rule in this set of instructions combines two fraglets and produces one or two fraglets as a result. For example, the "merge" instruction concatenates two fraglets with matching tags. The "persist" variant ($matchP$) moreover puts back a copy of the initial $[matchP : \ldots]$ fraglet to the store, thus acts like a catalyst.

IV. Fraglet Programming Examples

A. Header Rewriting

The simplest useful operation is the rewriting of the header field. Given a fraglet $[in : tail]$ we would like to obtain a fraglet $[out : tail]$. The "program"

$$[matchP : in : out]$$

does the job (See Figure 3): This fraglet will bond with the input fraglet and produce the desired output by appending the input's *tail* to the new out tag.

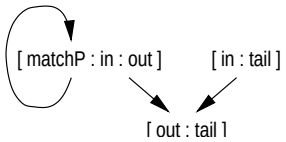

Fig. 3. Confirmed Delivery Protocol

B. Simple header tag rewriting.s

As a next example consider a confirmed-delivery protocol (CDP) that transfers some *data* to node B and returns an ack fraglet to the originating node A. Assuming that the channels are reliable, we can use the following program:

$${}_A[matchP : cdp : send : B : deliver]$$
$${}_B[matchP : deliver : split : send : A : ack : *]$$

Figure 4 shows the states through which the system is stepping. This program will transfer any incoming $[cdp : data]$ fraglet's payload to node B and will return an $[ack]$ fraglet.

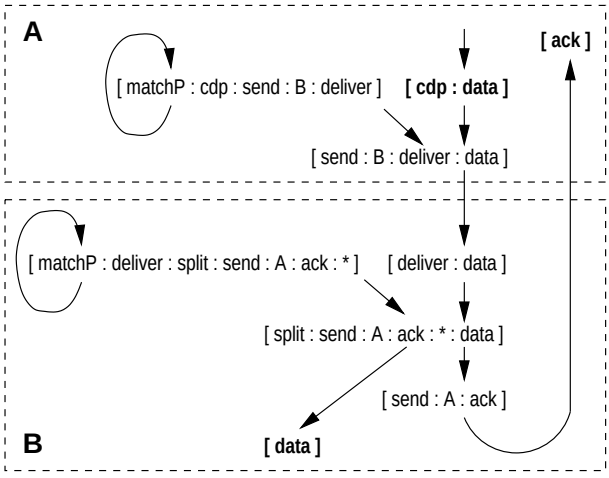

Fig. 4. Confirmed Delivery Protocol.

The previous implementation of the confirmed delivery protocol is expressed in the classical static protocol style which requires portions of the protocol logic to be preinstalled at the remote node B. The same protocol can be implemented in an "active networking" style where the node B does not know about the protocol in place and all activities are controlled by the sender node A. The active fraglet version of CDP is:

$${}_A[matchP : cdp : send : B : split : send : A : ack : *]$$

The following execution trace shows that again the $data$ string has been transferred and an ack has been returned. The initial catalytic program is shown only in the first entry where it is "activated" by the $[cdp : data]$ request:

$$
\begin{array}{ll}
 & {}_A[cdp : data] \\
 & {}_A[matchP : cdp : send : B : split : send : A : ack : *] \\
\Rightarrow & {}_A[send : B : split : send : A : ack : * : data] \\
\Rightarrow & {}_B[split : send : A : ack : * : data] \\
\Rightarrow & {}_B[send : A : ack] \\
 & {}_B[data] \\
\Rightarrow & {}_A[ack] \\
 & {}_B[data]
\end{array}
$$

C. Credit–Based Flow Control with Reordering

The previous example of confirmed-delivery provides a basic flow-control functionality if the sender waits for the $[ack]$ fraglet (by picking it up with a $[match : ack : \ldots]$). For a more complex example we implemented a flow control protocol where the sender can send up to N packets before blocking on the ack for the first message. An additional complexity is that we permit the channel to reorder (but not lose) messages, so the protocol has to re-establish the correct sequence of packets before delivering them. In the following we describe in words the protocol's functioning and discuss its execution graph.

The driving element of the system is the producer loop which injects a new data item to ship, then waits for a local 'OK' signal, after which it starts producing the next item. For each new datum we acquire a token which will identify the datum and as soon as there is such a token available, the 'OK' signal is dispatched. Thus, the producer can generate new items as long as there is a token available.

The token together with its corresponding data item is then sent as a combined fraglet to the remote node. Because of the reordering channel the incoming fraglets have to wait until it is their turn to proceed. This is achieved by having a waiting fraglet that matches exactly the next expected token's name. Once the next valid token has arrived and been identified we split the thread of control. One side continues towards the consumer which consumes the datum and signals that it is OK to release the token. The other thread of control is the token that waits for the consumer's signal. Once both have joined the token returns to the originating host. There it queues in the right order and waits for its turn to start a new journey.

The full system comprises less than 40 fraglets (see the Appendix). An interesting view is revealed by the execution graph of this system (figure 5). However, without further annotations the graph looks like a complex web of chemical pathways.

An interpretation of this web can be done that associates loops with the abstract entities of classical protocol implementations. For example, the producer and consumer loops (magenta) are easily detectable. Other control loops are the sender credit book keeping at the sender side and the reordering logic at the receiver side (red). Finally, we have the credit token loops (green) and the payload deliver path that crosses the protocol's metabolism (brown). The uncolored edges are either synchronization events between control loops or parameter passing operations during computations on other arguments.

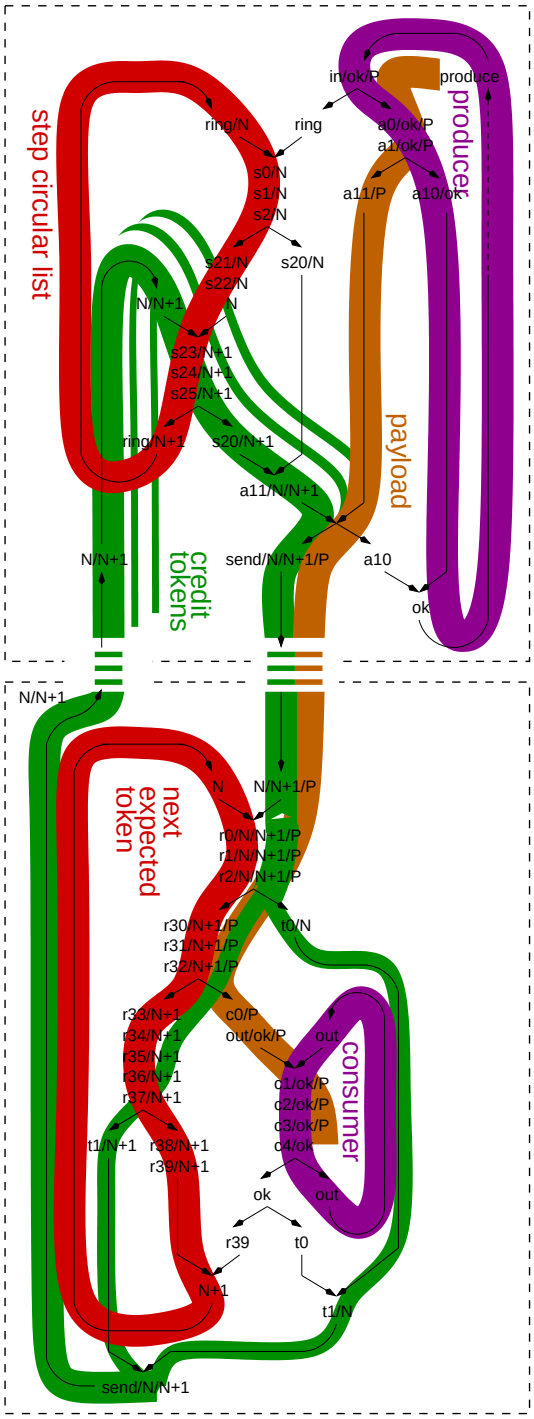

Fig. 5. Flow control with credit and reordering: "control loops"

V. Evolving Protocol Implementations

We have started work on the automatic fraglet synthesis using genetic algorithms. The fraglet environment is well suited for such a code breeding approach for two reasons. First, unlike other programming languages, fraglets have (almost) no syntactic constraints: Any string of symbols is a valid fraglet. Second, fraglets tend to be small (see the examples) and are

therefore easy to search for: The challenge of evolutionary protocol synthesis lies in the trial of different re-combinations of simple base fraglets, not the synthesis of the one–big–program.

For the time being our system works by doing off-line evolution of complete communication systems. That is, we generate initial configurations by placing a variable number of random fraglets in the nodes. Each system configuration is then run for a limited number of steps and its score evaluated. The fitness function is dependent of the target behavior we would like this system to have. For example, it can include the number of successfully acknowledged data packets, the "activeness" of the solution found (less pre-installed code on a remote node is better) or its effectiveness (number of steps to achieve a transfer/ack cycle). After ranking the systems we select the best individuals and add new combination of systems (generated via crossover and mutation) to the pool of configurations before starting a next tournament round.

Our first experiences indicate that optimizing protocol implementations is well feasible. More difficult is the creation of working protocols from scratch which is an effect of an all-or-nothing threshold where either the protocol is working or not. The definition of good fitness functions is also crucial as the system is excellent at finding "cheats".

In the long run we expect new "programming styles" to emerge from evolving protocol software. Instead of having manually defined functional "modules" inside communication layers we will see a dense mesh of intersecting "pathways". The next step will be to move from off-line evolution to an on-line system where the communication software continues executing its task while optimization and adaption processes are running in parallel.

VI. Conclusions

The *tag matching system* introduced in this paper shows a packet processing model where packets (fraglets) represent code and data at the same time. This enables to express mobile code based protocols in a very natural way. Unifying code and data is an important step towards the automatic synthesis of communication software as it enables the evolution of parameter data as well as the generation of functional code and code shipping logic with a single formalism. The resulting system will resemble complex metabolistic pathways and depart considerably from the layered engineering approach to communication software.

References

[1] Kyoto Encyclopedia of Genes and Genomes, Metabolic Pathways. `http://www.genome.ad.jp/kegg/metabolism.html`

[2] A. Perrig and D. Song. A First Step towards the Automatic Generation of Security Protocols. In Proc. Network and Distributed System Security NDSS 2000, Feb 2000.

[3] T. Paustian. Microbiology Webbed Out. University of Wisconsin-Madison, `http://www.bact.wisc.edu/microtextbook/`, 2003.

[4] N. Sharples and I. Wakeman. Protocol construction using genetic search techniques. In Real-World Applications of Evolutionary Computing – EvoWorkshops 2000, LNCS 1803, Apr 2000.

[5] D. Song, A. Perrig and D. Phan. AGVI – Automatic Generation, Verification, and Implementation of Security Protocols. In Proc 13th Conference on Computer Aided Verification CAV 2001, Jul 2001.

[6] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall and G. J. Minden. A Survey of Active Network Research. IEEE Communications Magazine 35(1), 1997.

[7] C. Tschudin. A Metabolistic Execution Model for Active and Passive Protocols. Uppsala University Technical Report 2003-30, May 2002.

APPENDIX
FRAGLET IMPLEMENTATION OF A FLOW-CONTROL PROTOCOL WITH CREDITS AND REORDERING

Producer side:

```
# Producer loop: emit a [in : ...] fraglet
# with a unique name as payload,
# wait for the 'ok' fraglet before continuing
_A[ matchP : p_produce : new : p_request ]
_A[ matchP : p_request : in : ok ]
_A[ matchP : ok : split : p_produce : * : nul ]
_A[ p_produce ]

# ring of identifiers (currently 3 credits)
# enlarge the ring for more credits.
_A[ ring : n0 ]
_A[ n0 : n1 ]
_A[ n1 : n2 ]
_A[ n2 : n0 ]

# catch a new 'in' request, store the params:
# a10: callback name, a11 : payload
_A[ matchP : in : split : match : ring : dup : s0 :
    * : exch : a0 : a11 ]
_A[ matchP : a0 : exch : a1 : * ]
_A[ matchP : a1 : split : match : a10 ]
_A[ matchP : s0 : exch : s1 : s21 ]
_A[ matchP : s1 : exch : s2 : * ]
_A[ matchP : s2 : split : s20 ]

# Let the "ring" pointer advance by one element,
# leave a copy of next pointer value at tag 's26'
_A[ matchP : s21 : matchS : s22 : match ]
_A[ s22 : dup : s23 ]
_A[ matchP : s23 : exch : s24 : s26 ]
_A[ matchP : s24 : exch : s25 : * ]
_A[ matchP : s25 : split : ring ]

# The forwarding logic - currently done with passive
# version: send a [ n_i : n_{i+1} : Data ] packet
# and a pre-installed fraglet at the receiver side
# will continue processing
_A[ matchP : s20 : match : s26 : match : a11 : split :
    a10 : * : send : B ]
```

Consumer side:

```
# The "passive" receiver waits at the next
# expected n_i tag. This receiver fraglet
# is rewritten after each successful delivery
_B[ match : n0 : r0 : n0 ]

# Isolate fields: n_i, n_{i+1} and payload.
# Call the 'out' delivery
_B[ matchP : r0 : exch : r1 : r30 ]
_B[ matchP : r1 : exch : r2 : * ]
_B[ matchP : r2 : split : t0 ]
_B[ matchP : r30 : exch : r31 : c0 ]
_B[ matchP : r31 : exch : r32 : * ]
_B[ matchP : r32 : split : r33 ]
_B[ matchP : c0 : out : ok ]

# build a new resequence start code
# (the passive entry point), store it at tag 'r39'
_B[ matchP : r33 : dup : r34 ]
_B[ matchP : r34 : exch : r35 ]
_B[ matchP : r35 : exch : r36 : t1 ]
_B[ matchP : r36 : exch : r37 : * ]
_B[ matchP : r37 : split : dup : r38 ]
_B[ matchP : r38 : exch : r39 : r0 ]

# consumer loop, waiting for a fraglet with
# form [ out : _handshake_tag_ : payload ]
_B[ matchP : out : exch : c1 : nul ]
_B[ matchP : c1 : exch : c2 : * ]
_B[ matchP : c2 : split ]

# Receiver side callback (tag 'ok'), will be called
# by consumer loop. Callback does:
# - enable new receiver side entry (at 'r39')
# - send back the credit token fraglet
_B[ matchP : ok : split : match : r39 : match : * :
    match : t0 : match : t1 : send : A ]
```