Concurrency Engineering

Chris Jesshope and Alex Shafarenko
Institute for Informatics, University of Amsterdam
and
Department of Computer Science, University of Hertfordshire
jesshope@science.uva.nl *and* a.shafarenko@herts.ac.uk

Abstract

This is a discussion paper on a very important topic that is about to become mainstream. It deals with the issues of software engineering in concurrent systems. It introduces this topic and illustrates the arguments for a change of perspective. It underlines these arguments with two examples, an asynchronous stream-based programming model and an asynchronous thread-based virtual machine model. Both support concurrency on very different abstractions but both capture similar support for concurrency engineering.

1. Introduction

The computer industry is currently in crisis. Despite some decades of research into programming concurrent systems, by which we mean systems that are both parallel and asynchronous. This is a difficult and error-prone activity. Evidence for this can be found in Microsoft and Intel's recent funding of a parallel software lab at Berkeley [1]. That laboratory seeks a model for programming the next-generation of multi-core CPUs. Why this should so different from programming the existing infrastructure of supercomputers, grids etc. is far from clear. The issues, problems and solutions are the same and it is only parametric changes that make a difference, i.e. the speed of communication and synchronisation relative to that of computation.

There are differences but not technical ones; whereas previously it was only in the domain of high performance computing that these problems needed to be recognised and solved, the recent and quite predictable power wall that industry faces means we can no longer rely on clock speed to improve a computer's performance. Future generations of commodity processors will be sold on "number of cores" rather than "GHz" but unless there is a

perceived benefit from this, there will be a significant slowdown in computer replacement. The difference then is that in high-performance computing, it is permissible to hire PhDs to engineer solutions to these problems, whereas for the run of the mill applications this is out of the question. The potential benefit of these new generations of CPUs must still be accessible to a broad spectrum of programmers.

2. Concurrent Software Engineering

Software engineering [2] is the application of the discipline of computer science and to a lesser extent, project management and other techniques, in order to develop software applications. The main purpose of this discipline is to improve the reliability and maintainability of software systems [3]. The achievement of these goals will suffer a severe blow in this shift to explicit concurrency in mainstream computer systems, as has been noted by Lee [4], based his group's experience with a well-engineered application when making this shift themselves. This paper attempts to map the impact these forces have on the software engineering discipline and to propose both generic and very specific solutions to those problems. We believe firmly that the process of concurrent software engineering must be partitioned into its constituent components, namely that:

Concurrent Software Engineering =
Concurrency Engineering + Algorithm Engineering.

However, we use of the term algorithm engineering in a different context to that described in [5], where the process is described as what is required to transform a pencil-and-paper algorithm into a robust, efficient, well tested, and usable implementation. Their definition encompasses a number of low-level issues, such as cache behaviour, and its main focus is experimentation. Our use of this term is more

abstract as we approach the algorithm from the perspective of capturing its specification, which may include concurrent operations but independently of any implementation issues, which make the algorithm non-portable. We believe that low-level solutions are required in order to eliminate, as far as is possible, the experimentation from algorithm engineering but this requires systems to be designed from the ground upwards, namely from the processor's ISA, which must abstract and embed explicit concurrency and manage this in a dynamic manner [6]. We will return to this in due course (see Section 5).

The key issue here however, is this separation of concerns. This in turn reflects a separation of expertise, where the engineers contracted to develop an application are partitioned into those with an application knowledge and/or expertise in algorithms and those with expertise in concurrent systems. It is also clear, that due to the complexity of both parts (in the former from the sheer size of the components in terms of lines of code and in the latter in terms of the explosion of states that concurrency exposes) both must retain or even strengthen the Software Engineering principle of reuse. Currently this does not seem to be a major issue in engineering concurrent code. In this paper we give examples of the embedding this principle into two related but quite separate developments.

3. Related work

As we are looking for general solutions to a problem that has been researched for decades, by implication we cannot cover all related work and restrict our references to those papers that have had a major influence on the work described.

The idea of S-Net was proposed, and the initial sketch of the language and its type system was made by Shafarenko. The first comprehensive solution for S-Net was by Grelck and Shafarenko (see [20], where the language definition and some relevant algorithms are presented). Further development of the type system was done by Cai et al [21] and some recent examples of the use of S-Net in applications are found in [18].

Stream processing as a discipline goes back to Kahn's seminal work [11] and the languages Lucid [12] and Esterel [13]. S-net network combinators resemble some structures in [14], but in fact go back much further to the pioneering work of Stefanescu [16] and Broy [17].

SVP has its roots in the microthread machine model originally proposed by Bolychevsky et al [26] and extended by Jesshope in [6,7]. It is a general

model of concurrency with implementations at the ISA [8] and language [9-10] levels.

Capturing concurrency in a computer's ISA has two influential precedents, the transputer concept [23], which captured the CSP model of concurrency and the pioneering work by Burton Smith on the Delencor HEP [24], the Horizon, and eventually the Tera architecture [25] (or Cray MTA). Both provided instructions to create/terminate processes and to communicate between these; in the transputer by mapping channels at link-time and in the HEP by synchronisation on shared memory locations. There are also examples, too numerous to mention, where these basic concepts have been extended and/or restricted in support of specific application areas, such as real-time applications or graphics processing.

4. S-Net

Encapsulation. Since the late 90s methodologies of software design have danced around the concepts of decomposition and encapsulation. Surprisingly, these were seen as vehicles of software reuse only, but not necessarily as central concepts of parallel computing. A problem decomposition results in a representation of an application as a set of black-box components, whose functionality is defined in terms of the interface description and some "glue" code that holds the components together in a way that ensures the expected system behavior is achieved. On the one hand, the components "hiding behind" their interfaces are highly reusable, since no code modification is required inside them when an alteration of *system* specifications occurs. Indeed, the altered functionality is achieved by "deriving" new components in an OOP fashion: orthogonal addition and redefinition of functions (i.e. methods or "ports").

Object state breaks encapsulation. When a component is a black box, this means that its interface description is enough to fully understand its behavior, with the exception of, perhaps, cost. That behavior, for a simple method interface, which includes the method name and some parameters, can only be one of two kinds: the effect of the method invocation on an object (i.e. an instance of the component) with internal state, and the production of a returnable result. It is the former that causes great difficulties in encapsulation. The problem is that the internal state is time sensitive, which means that it requires some time reference for accessing it, even in a distributed parallel system, which has no single clock. It is also place sensitive. Even when an object is quiescent, it cannot easily be moved from one processing place to another, since its state has an

implicit association with certain processes, which are specifically placed. If there are several processes using the same object, then even *where* it should be placed and by what discipline its simultaneous use may be governed are not clear. That information is not part of the interface, it is at best implicit in the object state, and at worst is only found in the client code. So in a sense, encapsulation fails: the behavior is no longer localized and abstracted between the input and output interfaces.

Solution. It is our contention that state transitions in the component world should be structured and managed in the same way as control flow is structured and managed in ordinary programming. We argue that the best way to achieve this is to strip user-defined components of all persistent state, so that they become pure functions that map a tuple of parameters onto a similar collection of results. As soon as the latter is produced, the internal state should effectively be destroyed[1]. Such components are easy to reason about and debug, they are inherently mobile, and usable as a black box in a parallel computing environment - but there is also a price to pay. The gluing environment would have to provide sufficient scaffolding to support an evolving state (or local states!) of the computation. In other words, it will need to hold the effective state of one or more component for them and present it back to the components' inputs in combination with any data to be processed. This is similar to thread-safe code where the intermediate state is held in the thread memory, except in this case it is not the intermediate, but, say, the end-of-iteration state that is being held and managed outside the component.

Language. To support the parallel component technology being discussed, a coordination language has been designed and implemented [20,18]. The language is called S-Net, which stands for Streaming Networks. Its purpose is to support writing coordination code that instantiates components as "boxes" and connects them with anonymous data streams so that an application is represented as a network between the standard input and output, which are two external streams connecting the whole application with its environment. We shall now briefly outline the main concepts of S-Net.

The box concept. Any S-Net component can be instantiated to a Single-Input, Single-Output (**SISO**) box. The box has a limited life cycle: it accepts one item from the input stream (these items are called "records", see below), does some processing and yields zero, one or more items to the output stream, after which it destroys its internal state (i.e. re-initialises) and waits for the next input item to arrive. There is one standard component, called a *synchrocell*, which has the ability to hold state, but which cannot perform computations of any kind; thus component encapsulation isn't violated. Components are written in a box language, using the S-Net communication API (which consists of a single entry point: *snet_out*, which allows a box to insert a new item in its output stream). At present C is supported as a box language and so is SaC [19]

The streaming data concept. All boxes accept records as units of their input. A record in S-Net is a set of fields and tags. Both fields and tags have *names* and *values*. Field values are unavailable to S-Net: they are only processed by the box language, while tag values are standardized as integers and are available to both the box language and S-Net itself. Records are nonrecursive in the sense that it is not possible to define an unlimited linked structure, such as a list. Every user-defined component contains a program unit (a function or similar) written in a box language, and a type signature written in S-Net that defines the type of records (in terms of their field/tag name sets) that the box accepts and the types of any output records that may be produced. Streams between boxes are sequences of records. Even though all boxes are SISO, the data relationships between them are not one-to-one, since streams can be split and merged using *combinators*.

Combinators. These are second-order functions that connect one or two boxes into a SISO network. First of all there are **series and parallel combinators**, A..B and A‖B, respectively. The series combinator connects the output of box[2] A to the input of box B, with the input of A and the output of B becoming those of the resulting network. The parallel combinator splits the single input stream into two streams according to the type match with the A and B interfaces, and merges the resulting two streams together. S-Net regards nondeterminism as an exploitable characteristic and provides two versions of the parallel combinator, a deterministic one A‖B, and a nondeterministic one: A|B. In the latter case the order in which the output streams are merged is arbitrary. This allows the recipient of the stream to reduce the latency of any response, provided that the algorithm allows it. Also we allow for nondeterminism at the input even when the

---

[1] N.b. we are not arguing for *functional programming* as our components can be written in any imperative language.

[2] All combinators are applicable to arbitrary combinator networks not just atomic boxes.

combinator is deterministic and has to merge the output streams in the input stream's order. The nondeterminism at the input occurs when a record matches A and B equally well, e.g. a record with the field-label set {x,y,z} when A expects {x,y} and B {y,z}. This allows for arrangements where two different routes are possible and the choice between them is on the basis of nonfunctional parameters, such as power or load. The type system of S-Net is powerful: it is based on set-theoretical subtyping with some extra controls in the form of binding tags, but we have no space here to expose it even briefly; suffice it to say that the exact destination (in the deterministic case) or set of destinations (in the nondeterministic case) is always statically known with only one exception, see below.

S-Net has two unary combinators for network replication: the **series** A**p and the **parallel** A!!<t> **replicators**. The former is equivalent to an infinite chain A..A..A.. ... in which any record that matches the pattern *p* is removed from the chain and sent to the output. The latter is equivalent to an infinite network A||A||A||... where each replica corresponds to a certain value of the tag <t> expected in the input record. This is the only situation in which the record destination is value-dependent but the destination is guaranteed to be one of the boxes with identical type signatures, and so it is type-safe. In implementation, these infinite data structures present no difficulty whatsoever since for the !! combinator only a finite variety of <t> values is expected at any given time and since any replicas of A that do not contain active synchrocells (see below) are garbage-collectable owing to the absence of state information (such replicas can be instantiated again if the same value of <t> is encountered later). As for the ** combinator, the network only unfolds as far as the point where no records that match the A input type are produced (which means that all records at this point, if any, match the pattern *p*). This is similar to ordinary while-loop termination, except the resources being used are both space and time. Again those replicas without active synchrocells anywhere on the chain can be fused with their predecessors and successors in implementation. Unique resources are not required for boxes as all replicas are stateless and identical. Finally, it should be noted that there are nondeterministic versions of the replicators, * and !.

When data comes from two different sources and has to be processed together, one needs some sort of synchronization facility. In ordinary distributed programming it is the computational code that is burdened with synchronisation, due to the multiplicity of communication channels and the state-transition nature of communication. In S-Net, user-coded boxes cannot be used as synchronizers even in principle, since they are stateless. Synchronisation is performed via a special box supplied by S-Net itself and only configured by the user: the **synchrocell**. The way it works is as follows: a cell [| {x,y}, {z,w} |] is initially *empty*. The first record that comes must match either {x,y} or {z,w} and it is stored in the synchrocell memory, the synchrocell now becomes *active*. Records of the same type from this point on are passed through and the first record of the other type causes the joining of the two records into an output record {x,y.z.w} after which the cell becomes *dead*. Dead synchrocells pass all records through. The reader can satisfy herself that, for example, [| {x,y}, {z,w} |]**{x,y,z,w} is an asynchronous version of the *zip* function, familiar from functional languages, and that [|{x},{z,<t>}|]]!<t> is analogous to the Explicit Token Store known from dataflow research (here a subtyping rule is used to get rid of the second copy of <t>). There are many more useful patterns that can be built using synchrocells.

Examples and design methodology. Readers are referred to the S-Net site on the Web for details of our S-Net implementation [22]. Due to the limited space we can only state here that a compiler is available, which translates an S-Net program into C with calls to an extensive run-time library that uses p-threads to achieve concurrent execution. Here is a tiny example, which exhibits asynchronous, parallel, streaming execution of an $n!=1×2×...×n$ producing network. The input stream supplies a sequence of *n*. (N.b. boxes in this example may implement variable-precision arithmetic and so may be non-trivial.)

```
net fac ({n} -> {n,m}) {
  net facit ({x,r} -> {r}) {
    box leq ((x) -> (x,p));
    box if ((p) -> (<T>) | (<F>));
    box dec ((xx) -> (xx));
    box mult ((x,r) -> (rr));
  }
  connect (leq..if..([{<T>}->{<stop>}]
              || [{<F>,x,r}->{x,r};{xx=x}]
              .. (dec||mult)
              .. [|{xx},{rr}|]*{xx,rr}
              .. [{xx,rr}->{x=xx,r=rr}]) ** {<stop>})
              ..[{<stop>,x}->{}];
  box one (() -> (one));
}
connect one .. [{n,one}->{n,x=n,r=one}]
            .. facit .. [{r}->{m=r}];
```

The main syntax construct of S-Net is

net *list-of-net-n-box-defs* **connect** *formula*

The *formula* in each net clause is a combinator expression defining the structure of the network. The user-defined boxes leq, if, dec and mult are self-explanatory thanks to their expressive type signatures.

The parenthesis there [...] signify the filer construct (not to be confused with a synchrocell [|...|]), which is a housekeeping box offered as syntactic sugar by S-Net, but one that can be written by the user for each specific case. Its role is to rearrange a record into one or more output records by renaming/copying or dropping fields/tags as indicated by the expression inside the brackets. This hopefully requires no explanation. Finally, to understand the working of this network one needs to be aware of *flow inheritance*, a stream specific form of inheritance whereby any unmatched fields/tags at the input are appended to each produced output record.

Concluding this section, we would like to comment on the design methodology using this language. S-Net promotes top-down design, which is known to be very effective but which in practice is hard to support by conventional programming languages. The way to do it in S-Net is as follows. First the whole application is given a name and a type signature, which details what data collections are being processed and what type of potential output they cause. Next the monolithic application is broken down into a small network of networks by identifying closed functionalities and the combinators needed to stream the data as appropriate. Those functionalities are then reified as further nets, type signatures are determined and then refinement continues until the items connected by the network are truly atomic and could be defined directly in the box language using nothing more than data-parallelism without loss of exploitable concurrency. At each stage, data streams can be reasoned about and animated and also at the final stage boxes, being stateless, fully-encapsulated entities, can be unit-tested, too.

5. The SVP model

Summary of the SVP Model. SVP stands for SANE Virtual Processor, where a SANE is a Self-Adaptive Network Entity. SVP also manages asynchronous concurrency but at the level of machine instructions. It is based on threads that are created dynamically with a context of scalar, synchronising variables, which is garbage collected on thread termination. Data-structuring is managed by thread replication within indexed families, which may be unbounded.

Input and output to these families of threads is via asynchronous shared memory.

SVP instructions capture data-, instruction- and task-level concurrency. Collectively they form an operating system kernel implemented in the processor's ISA that manages work creation, termination, pre-emption and all mapping and scheduling of threads. SVP is designed so that programs are free of deadlock under composition, deterministic where required and to enable the migration of data and/or code in a distributed computing environment to better manage an implementation's efficiency through self-adaptive control.

SVP is defined by five actions that dynamically create and asynchronously control the concurrent execution of families of threads. Those actions are {*create, sync, kill, squeeze, and break*}. Together, create and sync define a concurrent section between a creating thread and one or more identical created threads (the *family*) each of which is aware of its unique index value. Kill and squeeze terminate named families and break terminates a thread's own family. Squeeze differs from kill and break in that a squeezed family can be re-executed to completion from a breakpoint. Reflection on termination is provided by a return code received by the sync, which signals when all threads and their writes to asynchronous memory have completed. The return code also indicates how the family was terminated, i.e. whether normally or via one of the terminating actions. A return value may also be received on sync, which is either a thread index, determining the breakpoint in the family from a squeeze action or a value set by a thread when it succeeds in executing a break.

There are two further abstractions that complete the definition of the model. The first is that threads are blocking, i.e. they capture not only function but also synchronisation, which in turn supports data-driven instruction scheduling. Threads execute their operations strictly in-order and block if they do not have the data required to complete an operation. Instructions may always write data and hence a thread's context of synchronising variables are i-stores supporting dataflow synchronisation. Threads suspend on these i-stores awaiting a write from another thread or a hardware process (such as a read from asynchronous memory) and are scheduled only when that data has been written.

Communication between threads in the model is deliberately restricted in order to expose locality without reference to resource mapping. The parent may write to its first child thread and each created thread may write to its successor in the family created. This restriction means that a compiler can

perform static optimisations to achieve concurrency and locality. It has the additional advantage of offering concurrent composition in the model without inducing deadlock.

The second abstraction concerns the dynamic management of resources in SVP and is the concept of a *place*, an implementation-dependent definition of a processing resource. It is through this parameter to the create action that families of threads are bound to processors. The use of place must also be accompanied by a *place-server* in an implementation to define an available place on request.

Support for Software Enginering. The SVP model is uniform and hierarchical, and captures concurrency from fine-grain instruction or data concurrency up to the highest levels of task concurrency in a system. However, the introduction of places partitions the model in a way that correspond to the partition described in Section 2, namely between algorithm and concurrency engineering.

The first usage level is static. Here, code is resource naive and captures an algorithm and its concurrency. This may be compiled to an implementation, e.g. a microgrid of microthreaded processors [6], without knowledge of the number of processors to be used in executing it. This usage is deterministic and binary programs can be combined concurrently without inducing deadlock and distributed arbitrarily to clusters of processors at run time. The model's locality allows an efficient mapping of the computation onto any hardware even though the program's resources will be assigned dynamically to this compiled code.

The second usage level is completely dynamic and is defined only when a place is specified in SVP's create action. It is at this level that concurrency engineering is achieved and it requires the binding of a unit of work (a family of threads and any subordinate families) to a place that will execute the work. If a thread in family A creates a subordinate family, B say, at the default place, then family B will share the same processing resources used by A. If, however, the thread provides a named place in the create action, the execution of that work is delegated to the new resources defined by the implementation's definition of that place. Now family A and B will be distributed relative to each other and communication will be required. The implementation of the named place will provide the necessary address and protocol for creating the family remotely and also any authentication required for creating a family there. Thus the place provides both abstract networking and security issues. An implementation of create for a given place will also understand issues such as memory model (shared or distributed).

Place servers in SVP. The use of a place when creating a family of threads is the key abstraction that allows dynamic binding of resources to code. SVP implementations require a mechanism to capture this cycle of defining a place and using it in the create action. This concept of a place server is familiar to most people who have used dynamic memory but in this case there is a distribution in both time and space.
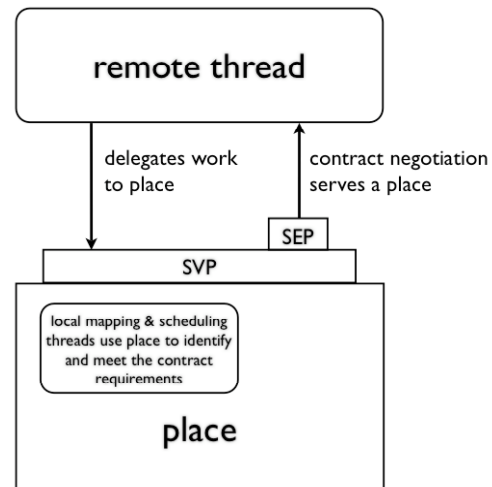

Figure 1. Cycle of serving and using a place

Every SANE processor, See Figure 1, provides an interface and protocols to define SVP actions and an interface and protocols to serve places. The latter is called the Systems Environment Place (SEP), which supports a standard API, e.g. *SEP_request*, and *SEP_release* to allocate and release processors. These API threads are created at the SEP using the SVP create protocol. Place provides control of non-functional properties in SVP but also introduces the non-determinism required to implement the place server itself and more generally, non-deterministic choice. The place server shares resources between concurrent activities. This is implemented by defining some places as being exclusive, i.e. they will serialise the execution of families of threads at such places. The SEP is obviously an example of an exclusive place.

6. S-Net on SVP

Streaming networks are generally implemented using static dataflow principles, i.e. boxes are assigned to resources and computation is triggered by input to those boxes. The two models described above, namely programming and machine models (S-net and SVP) uniquely lend themselves to implementations of streaming based on dynamic dataflow principles. This idea comes from perceiving

an S-Net from the perspective of a record being communicated between the standard input and output and processing this flow as a sequence of continuations (threads) in SVP. In a way, it is similar to the Lagrangian view of the fluid motion in physics, which describes what happens to a small volume of fluid as it travels rather than attempting to describe the evolution of the velocity field – that latter view is called Eulerian. The current implementation of S-Net on pthreads in that sense is Eulerian, as we define the behaviour of all boxes simultaneously and assume the existence of channels between them. This provides a rather static view on resources and does not map onto SVP that captures and abstracts resources dynamically at the level of the machine model.

Thus, in the Lagrangian view, an S-Net network is represented as a bulletin board on which extended data records may be posted, and an abstract (constant) graph which is available globally. There is no unfolding of the graph, and there are no processes associated with boxes, channels, or any other elements of the network. The only active agent in this view is the Graph Walker (GW) whose job is defined along the following lines:

1. collect a record posted on the bulletin board;
2. read from the record the target graph location;
3. determine what processing is required, invoke the appropriate family of threads that implements this and bind this to the most appropriate resources at this time;
4. this will in turn result in zero or more additional records being posted to the bulletin board.

At any given time during program execution, there can be any number of GWs operating in parallel. The Lagrange implementation does not have processes and need not implement FIFO queues to represent channels, hence the correct sequencing of records is also the GWs' responsibility. A number of techniques are being investigated to abstract a record's position in the input stream. These include adding a serial number to records as a tag and maintaining a cons list of records and their thread applications. All methods require some overhead in maintaining order, but the potential benefit of exploiting S-Net and SVP's rejection of encapsulated state brings major advantages in the exploitation of concurrency and adaptivity in its scheduling. This overhead is small compared to the execution time of a box, which is assumed to be a reasonably substantial component.

The S-Net * and ! combinators cause replication of a part of the network in the Eulerian view; the Lagrangian view, being devoid of material boxes and channels, uses additional indices that, together with the graph location, specify which replica is being used. To summarise, records emitted for execution from boxes are extended with their location in the stream and one or more indices that fix the replica numbers of the environments inside which the target location is found.

We do not have sufficient space in this paper to discuss this implementation in detail. However, in SVP the Graph Walker is a family comprising one thread that is created with parameters including the node number in the static S-Net graph structure, the type of the record, used for selecting the box the record is routed to and any sequencing information required for merging and synchronising records.

Such a family is created whenever a record is emitted by box code, which can be written in or compiled into SVP. The GW uses this information together with information about resources available, constraints on execution etc. to place an instance of the target box for that record. The latter is derived from the record's position in the network and its record type. If necessary a box can be replaced in a network by a serial combination of its cost function and its execution, in order to manage situations where cost is a dynamic function of input parameters. Execution of the box can either be immediate, if resources allow, or the GW may schedule the execution. For example, the GW may evaluate its data structures to see if it can aggregate the execution of this record with other similar records in order to amortise the configuration costs in an FPGA for the execution of this function.

Implementing the only state-full elements of S-Nets, i.e. the synchro-cells, may at first appear to be problematic in SVP but like resource management, the synchro-cells are each implemented using an exclusive place, so that concurrent updates to the cell's state are sequentialised. A simple partitioning and distribution of synchro-cells is the mechanism that enables control of contention at exclusive places if this is an issue.

7. Conclusions

This paper has explored some of the issues that will face the computer industry over the next few decades, as Moore's law provides more and more cores on silicon devices and as processing resources become more diverse (e.g. FPGA accelerators). It explores the issues in concurrent software engineering that allow software for this time frame to be made more reliable and to allow its reuse. The paper outlines from a high-level, both a programming model and a machine model that allow the separation of concerns in this endeavour, namely being able to separate the tasks of algorithm engineering and concurrency engineering, where it should be noted

that the former is nor devoid of concurrency yet must be removed from low-level issues such as mapping to resources, scheduling, communication and above all, synchronisation.

8. Acknowledgements

9. References

[1] R. Merritt (2008) Wintel will fund parallel software lab at Berkeley, http://www.eetimes.com/news/design/ showArticle.jhtml?articleID=206503988

[2] IEEE Standard Glossary of Software Engineering Terminology, IEEE std 610.12-1990, 1990, Chapter 1: Introduction to the guide Guide to the Software Engineering Body of Knowledge (February 6, 2004), retrieved on 2008-02-21.

[3] M. Pecht (1995) Product Reliability, Maintainability, and Supportability Handbook, CRC Press. ISBN 0-8493-9457-0.

[4] E lee (2006) The problem with threads, IEEE Computer, 36(5), pp. 33-42.

[5] D. A. Bader, B. M. E. Moret and P. Sanders (2002) Algorithm Engineering for Parallel Computation, Fleischer et al. (Eds.): Experimental Algorithmics, LNCS 2547, pp. 1–23.

[6] C. R. Jesshope (2008) Operating systems in silicon and the dynamic management of resources in many-core chips, to be published: *Parallel Processing Letters[3]*.

[7] C R Jesshope (2008) A model for the design and programming of multi-cores, In *Advances in Parallel Computing*, **16,** *High performance Computing and Grids in Action* (Ed. L. Grandinetti), IOS Press, ISBN 978-1-58603-839-7, pp37-55.

[8] C R Jesshope (2006) µTC – an intermediate language for programming chip multiprocessors, Proc. Pacific Computer Systems Architecture Conference 2006 - ACSAC06, ISBN 3-540-4005, LNCS 4186, pp147-160.

[9] T. Bernard, K. Bousias, L. Guang, C. R. Jesshope, M. Lankamp, M. W. van Tol and L. Zhang (2008) A general model of concurrency and its implementation as many-core dynamic RISC processors, to be published: *Proc. SAMOS 08*.

[10] M. W. van Tol, C. R. Jesshope, M. Lankamp and S. Polstra (2008) An implementation of the SANE Virtual Processor using POSIX threads, submitted to: *Journal of Systems Architecture*.

[11] Kahn, G.: The semantics of a simple language for parallel programming. In Rosenfeld, L., ed.: Information Processing 74, Proc. IFIP Congress 74. August 5-10, Stockholm, Sweden,

North-Holland (1974) 471–475

[12] Ashcroft, E.A., Wadge, W.W.: Lucid, a nonprocedural language with iteration. Communications of the ACM 20 (1977) 519–526

[13] Berry, G., Gonthier., G.: The esterel synchronous programming language: Design, semantics, implementation. Science of Computer Programming 19 (1992) 87–152

[14] Michael I. Gordon et al: A stream compiler for communication-exposed architectures. In: Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA. October 2002. (2002)

[15] Stephens, R.: A survey of stream processing. Acta Informatica 34 (1997) 491–541

[16] Stefanescu, G.: An algebraic theory of flowchart schemes. In Franchi-Zannettacci, P., ed.: Proceedings 11th Colloquium on Trees in Algebra and Programming, Nice, France, 1986.
Volume LNCS 214., Springer-Verlag (1986) 60–73

[17] Broy, M., Stefanescu, G.: The algebra of stream processing functions. Theoretical Computer Science (2001) 99–129

[18] Grelck, C., Scholz, S.B., Shafarenko, A.: Coordinating Data Parallel SAC Programs with
S-Net. In: Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS'07), Long Beach, California, USA, IEEE Computer Society Press, Los Alamitos, California, USA (2007)

[19] Scholz, S.B.: Single Assignment C — efficient support for high-level array operations in a functional setting. Journal of Functional Programming 13 (2003) 1005–1059

[20] C. Grelck, A. Shafarenko: Report on S-Net: A Typed Stream Processing Language Part I: Foundations, Record Types and Networks. Technical Report University of Hertfordshire Department of Computer Science Compiler Technology and Computer Architecture Group Hatfield, England, United Kingdom, 2006

[21] H. Cai, S. Eisenbach, C. Grelck , A. Shafarenko,: Extending the S-Net Type System. To be published

[22] snet-home.org

[23] D May and R Shepherd (1984) The transputer implementation of occam, Proc. *Intl Conf on Fifth-Generation Computer Systems*, Tokyo, pp533-541.

[24] J W Moore (1983) The HEP Parallel Processor, *Los Alamos Science*, Fall 1983, pp 72-75. http://library.lanl.gov/cgi-bin/getfile?09-04.pdf

[25] R Alverson, et al. (1990) The Tera Computer System, *Proc. of the 4th International Conference on Supercomputing*, Amsterdam, The Netherlands, 11-15 June, pp. 1-6. ACM Press, New York, NY, USA.

[26] A Bolychevsky, C R Jesshope and V B Muchnick (1996) Dynamic scheduling in RISC architectures, *IEE Trans. E, Computers and Digital Techniques*, **143**, pp 309-317.

---

[3] All papers to be published can be downloaded from: http://www.science.uva.nl/~jesshope/Papers/