Fault-tolerant Coordination of S-Net
Stream-processing Networks

A Feasibility Study

Raimund Kirner, Vicent Sanz-Marco, Michael Zolda, Frank Penczek
University of Hertfordshire, School of Computer Science, Hatfield, United Kingdom
{r.kirner, v.san-marco, m.zolda, f.penczek}@herts.ac.uk

Abstract

Fault tolerance and robustness are system properties of increasing importance, both in the domain of embedded computing as well as in the domain of high-performance computing.

In this paper we study the applicability of fault-tolerance mechanisms in the context of stream-processing networks, in particular based on the coordination-language S-Net. We identify three basic fault tolerance mechanisms and discuss the technical solutions for them within S-Net. The applicability of these mechanisms depends on the requirement of the concrete application domain. The contribution of this paper is a feasibility study of tool-supported fault tolerance mechanisms in a flexible coordination language allowing for asynchronous execution. As part of this feasibility study we discuss potential extensions of the S-Net language and runtime system in order to implement the identified solutions.

General Terms concurrent computing methodologies, parallel computing methodologies, reliability

Keywords dependability, fault-tolerance, robustness, software engineering, coordination languages, stream processing, data-flow, S-Net

1. Introduction

Computing systems in the embedded domain as well as in the high-performance domain share some common challenges. Arguably the most important one of them is fault tolerance, either to protect human lives, or to improve the performability of a system. At the same time, stream-processing computing is a popular pattern in both domains for its benefits in multi-core programming as streams can be used as a form of implicit synchronisation. Implicit synchronisation is a powerful tool to cope with the inherent complexity of concurrent programming.

Fault tolerance has been studied thoroughly in the field of dependable real-time systems [12]. A classical way to achieve fault tolerance is the use of triple modular redundancy, which is based on voting over redundant computations. With the increasing complexity and fault likelihood of computer systems, additional fault-tolerance techniques are useful to extend the performability of the system. For example, reconfiguration can be used to increase the dependability of core services, which is especially useful in mixed-criticality systems [10].

Fault tolerance in stream-processing systems is an important issue, as stream-processing applications is a frequent application pattern that is well-suited for programming parallel systems. In stream-processing systems with an explicit coordination layer the compiler can use structural information about the application to generate code to support fault tolerance.

For example, Balazinska et al. provide fault tolerance for distributed stream processing, using checkpoint/restore [5] approaches [2]. This approach is based on redundant computation for fault-tolerance with support for reintegration after a fault, but not reconfiguration.

In this paper we study how different fault-tolerance strategies are applicable to coordination-based stream processing in a flexible execution model, namely that of S-Net [8], which allows for asynchronous message processing. The language S-Net is introduced in Section 2. In Section 3 we describe how to realise different fault-tolerance techniques based on three basic mechanisms. Section 4 discusses related work. Section 5 concludes this document.

2. Key-Concepts of S-Net

The approach taken by S-Net is targeted at stream processing. S-Net is a very compact, powerful and declarative coordination language for describing streaming networks of asynchronous components. It reflects the modern notions of subtyping, encapsulation and inheritance, while completely separating all communication and concurrency concerns from the application code.

Streaming networks are defined using an expression language featuring four network combinators as operators: serial composition, parallel composition, serial replication and parallel replication. With the exception of serial composition, the combinators come in two flavours each: the deterministic versions preserve the order of data on streams, whereas non-deterministic variants trade this property for improved throughput. Two primitive components serve housekeeping and synchronisation purposes. Streams are associated with record types: collections of data where each item is uniquely identified by its name. Structural subtyping on records directs the flow of data through the streaming network.

Instead, the language is built upon data-flow principles: an application is a collection of computational components, also called boxes, that are put in relation to each other only by their data dependencies. To express such dependencies, the language provides means to construct a hierarchical data-flow graph. In this graph the
boxes are forming the set of nodes, the data dependencies between boxes are captured by the set of edges. A data dependency between two boxes is expressed by placing an edge between them. An intuitive interpretation of the data-flow graph is that of a communication infrastructure between computational components: The edges of the graph are directed channels through which components communicate with each other when they need to exchange data. The program defines a streaming network in which data items are received and processed by a box before the result is sent on to the next adjacent box. In an S-Net program, the implementation of a box is not accessible, i.e., boxes are opaque components that may be implemented in a range of conventional programming languages, as for example C.

Within an S-Net program we have no means to inspect the data that a box receives and produces; this is an important consequence of the freedom of choice of a computational language for the implementation of boxes. The data types of the computational language are never exposed to the outside. Yet, we need a handle on data from the box domain in order to receive results from boxes, route them through the data flow network and supply them as arguments to other boxes. For this, S-Net employs its own, record-based notion of types. Data that is received from a box is handled as a set of label-value pairs, i.e., a record, and it is assigned a record type based on the type signature that is supplied with a box. The organisation of data as records allows us to refer to data symbolically by a label while keeping the value, i.e., the data of the box domain, opaque. Even more importantly, types play a fundamental role in the way records are routed through the network. At split points the type of a record determines the choice for the branch the record takes, i.e., at runtime the communication between different parts of the network is established through type-directed routing.

As an illustration, consider a generic fork-join pattern that applies a computation in parallel to an input. An S-Net program for this example may look similar to what is shown in Figure 1. In this representation, boxes that require results from other boxes are located to the right of the component they depend on. Boxes that can execute independently of each other are vertically aligned. In this program, a split box that decomposes an input into several smaller chunks is the first component to execute. This is followed by several compute boxes that may execute in parallel, implementing the actual computations that we are interested in. After the execution of the computation finishes and the result is output, the results are collected by a sub-network fold, to collect and output the overall result.

The core language of S-Net that allows us to express coordination programs like the one above is very compact. The atomic building blocks of programs, as mentioned earlier, are boxes. These represent computations without revealing their actual implementations. Only an abstract type signature, similar to a function prototype in a C header file, is exposed to the S-Net program that describes the data flow network within which the boxes are instantiated. As S-Net programs are based on data-flow graph modelling, the essential ingredients of the language are combinators for graph construction. Only four combinators together with one primitive for explicit synchronisation suffice to express elaborate coordination programs. Two combinators allow for static sequential and parallel composition. Two further combinators define dynamically unfolding sequences and dynamically expanding parallel compositions. By allowing hierarchical graphs, i.e., graphs in which nodes are graphs again, even complex coordination programs can be expressed in a structured and intelligible way. The design of the language does not rely on a specific execution model, but is built on the assumption that all boxes within a graph may execute asynchronously as soon as input becomes available. Boxes strictly consume a single data item per invocation, but are allowed to output any number of data items as long as boxes return from their computations eventually, i.e., boxes may not implement infinite producers. Furthermore, boxes are not allowed to maintain an internal state across invocations. The latter requirement enables us to very cheaply re-instantiate and duplicate boxes, i.e., computational tasks, on different resources: S-Net allows for explicit task-to-resource mappings on the level of boxes and networks using a notation where $\texttt{R}$ is a resource identifier. Depending on the underlying executing machinery, the resource identifier may, for example, refer to a specific node in a cluster, a processor in a multi-processor system or a specific core in a SoC.

From a software-engineering perspective it is worth nothing that S-Net fulfills the six principles of modular software construction [4]:

- Information Hiding: The user doesn’t need to know the internal mechanism of the boxes to use it. In the example of Figure 2, the user doesn’t know the procedure inside the boxes. Remember, the implementation of a box is not accessible.
- Invariant Behaviour: The same box maybe use at different locations in the network. These boxes have the same behaviour regardless of their location in the network.
- Data Generality: The interface to a box is capable of passing any messages an application may require.
- Secure Arguments: The boxes do not exert side-effects on their inputs.
- Recursive Construction: Any S-Net network constructed with boxes, can be used as a subnetwork within an S-Net application.
- System Resource Management: Resource allocation is performed by the runtime system and not by boxes.

Figure 2 shows an example of an S-Net program [7]. In the example, we can find four different kinds of boxes: $\texttt{leq}$, $\texttt{if}$, $\texttt{dec}$ and $\texttt{mult}$. They reflect the basic building blocks of the functional implementation of factorials: The box $\texttt{leq}$ computes the termination condition; the result is stored in field $\texttt{p}$. The box $\texttt{if}$ makes the boolean value of the field $\texttt{p}$ visible to S-Net by choosing it either into a tag $\texttt{<T>}$ or a tag $\texttt{<F>}$ Last but not least, the boxes $\texttt{dec}$ and $\texttt{mult}$ do the required arithmetic. Also, in the example, there are two more boxes $\texttt{filter}$ and $\texttt{sync}$.

The primitive $\texttt{filter}$ box is devoted to all kinds of housekeeping operation. Effectively, any operation that does not require knowledge of field values can be expressed by this versatile primitive box in a simpler and more elegant way that using an atomic box and a box language implementation.

The synchronisation cell, or synchrocell for short, is the only stateful box in S-Net. Syntactically, a pattern merely is a record type. The principle idea behind the synchrocell is that it keeps incoming records which match one of the patterns until all patterns have been matched. Only then the records are merged into a single
one that is released to the output stream. Matching here means that type of the record is a subtype of the pattern. The pattern acts as an input type specification of the synchroncell: a synchroncell only accepts records that match at least one of the patterns.

This example shows how concepts of functional programming (e.g., nested function definitions, function applications) can be expressed in the framework of S-Net in a systematic way. The example in particular serves as blueprint for expressing linear recursive functions in S-Net. In the factorial example the box language code is extremely simple, one atomic instruction each. However, without changing the principles of the S-Net we could replace the box inscriptions by complex computations with record fields referring to large data structures. As long as the algorithmic pattern remains the same, we can easily turn a toy example like factorial into a real application. Leaving the concrete example behind, our example sketches out a methodology to convert functional programs into S-Net in order to express and to exploit concurrency.

3. Fault-Tolerance of Stream-Processing Networks

It is the property that enables a system to continue operating adequately to a hardware or software failure. The fault tolerance is very important in systems that must work all the time. Given a failure, another component or a special procedure can take control to remedy or mitigate the effects of the error.

A fault-tolerant system may be able to tolerate one or more fault-types including:

1. hardware faults,
2. software errors,
3. externally induced upsets or physical damage.

An extensive methodology has been developed in this field over the past thirty years, and a number of fault-tolerant machines have been developed. A large amount of supporting research has been reported. Fault tolerance covers a wide spectrum of applications, for example, ranging across embedded real-time systems, commercial transaction systems, transportation systems, and military/space systems. These areas often involve widely diverse core expertise ranging from formal logic, mathematics of stochastic modeling, graph theory, hardware design and software engineering.

It’s important don’t confuse fault tolerance with system maintenance. Because the system maintenance requires an external agent to work when there is any problem. On the other hand, we speak of fault tolerance when an error occurs in the system and the system can handle that error without any exterior help.

Fault tolerance for stream-processing networks in principle is not different from generic fault-tolerance precautions. The challenge is to provide an acceptable level of service even in the presence of faults. However, in our aim to cover embedded computing as well as high-performance computing domains, we identify the following properties that influence the design criteria of adequate fault tolerance mechanisms:

1. Fault tolerance is transparent to the box level, i.e., fault tolerance is dealt with at the level of the coordination language S-Net and its runtime system
2. Mixed Criticality: The fault tolerance precautions do not necessarily have to be applied to the whole S-Net program, i.e., the developer has fine-grained control over the operators for which fault tolerance is needed
3. We aim to handle both, permanent and transient faults [1]
4. We provide multiple fault-tolerance precautions to provide adequate techniques for the high-performance domain, i.e., average execution time is the main metric, as well as for the embedded domain, i.e., worst-case execution time is the main metric for the real-time constraints
5. Fault tolerance should be available for faults in the time domain (includes fail silent faults) and in the value domain (for the computation of messages as well as for their communication)

In previous work we already investigated how a compiler can simplify the programming of safety-critical applications by keeping the fault tolerance mechanisms orthogonal to the application programming [11]. However, the analysis in [11] was focusing on a compiler for a functional programming language. While functional programming languages already provide benefits, like the ability to achieve a significant reduction of state size for checkpointing [3], we see further potential when applying them to a coordination language like S-Net. First of all, the boxes in S-Net are free from persistent state, like a functional program would be. But further, a streaming connection between boxes provides a natural choice for fault-tolerance mechanisms like checkpointing, as the state size is relatively small. In addition to the fault tolerance requirements listed above, we aim to support the following software behavior imposed by the S-Net language:

6. S-Net boxes are functional, i.e., they always produce the same output for the same input. This feature simplifies fault tolerance as re-computation for an input will still produce the same value as it would have in the first attempt.
7. S-Net networks and subnetworks can be constructed in such a way, that they have the MIMO property (multiple input stream, multiple output stream). Based on this property we can use a logical time for messages ordering instead of having to use real-time with the overhead of clock synchronisation.
8. An S-Net box may have a multiplicity larger than one, i.e., it may produce more than one output message from one input message. This raises consistency issues in the context of fault tolerance, as boxes may get faulty after only part of the output has been written.
9. Asynchrony: Except for the special case of where order is enforced, messages in S-Net may be processed out of order.
10. Non-determinism: Routing of messages can be non-deterministic, i.e., the routing is not defined at S-Net language level. This behaviour is meant for fairness-based routing, but causes additional consistency challenges for fault-tolerance mechanisms.

There are several fault-tolerance techniques we aim to support. However, these techniques will be based on one or more basic mechanisms. We have identified the following basic mechanisms to be added to S-Net to support fault-tolerance:

- Checkpointing/restoring of program state
- Dynamic Reconfiguration
- Redundant Computation

These mechanisms are building blocks for the realisation of different fault tolerance techniques as listed by Avizienis et al. [1]. The meaning of these mechanisms and their contribution to fault tolerance is discussed in the following subsections.

3.1 Checkpointing/Restoring of Program State

A very basic mechanism for fault tolerance is to checkpoint the system state regularly and in case a fault has been detected just reload the state of the latest checkpoint and retry it again. The main overhead in checkpointing in modern systems stems from storing the associated state [5]. Techniques like uncoordinated checkpointing help to reduce this overhead [6]. In case of a fault, doing a restore
it is assured that the input pass messages have been passed through. When this has happened, the passed-through flags of the passes will store the unique id of messages, which can be discarded.

The indication that all output messages $o_{k,i}$ have been passed will require the knowledge of the multiplicity $m_{k,i}$, which can be calculated at runtime through control records.

The passed-through flag buffer will store the unique id of messages, where the computation of the id of an output message solely depends on the id of the input message. This two-dimensional buffer is typically implemented as a vector of linked lists, as the multiplicity for each box can be different, resulting in a sparse occupation of the buffer.

3.2 Dynamic Reconfiguration

Dynamic reconfiguration in general allows to change the configuration of a stream-processing subnetwork or even change the subnetwork itself.

The reasons to reconfigure a network may be manifold: A box has to be replaced, because an improved version of an algorithm a box is currently running became available. A network suffers from poor throughput, because of imbalanced execution times and bottlenecks, or a box has crashed and does not process any records anymore. Reconfiguration is key to maintain, improve or restore system stability and performance, as well as it is a means to implement fault-tolerance mechanisms in general.

Providing separate mechanisms to address any of such issues in particular is possible, but in the context of S-Net we are developing a more general approach that is directly integrated into the language. Following the design-principles of the language, a combinator-based approach is proposed. In fact, a total of just two combinators prove to be versatile enough to natively support the implementation of various fault-tolerance mechanisms. This two combinators are replacement combinator and feed-back combinator.

Replacement combinator

As with the other combinators, the replacement combinator may be applied to any network. By applying it to a network, the network is made replaceable. The combinator may be applied to any network. By applying it to a network, the network, after that, can be replaced.

Or network as part of a larger context becomes, for example, to the location of a box which crashes.

Figure 3. Two-dimensional message passing buffer for checkpoint/restore.
dard data processing. For this to be integrated seamlessly, we promote networks to first-class citizens of the language: A network may be part of a message, just like other data. This way the deployment of new networks follows the same routing principles as plain data. Messages that contain new network operands are routed on the same basis as before, i.e., routing is determined by the name and the type of the network operand, again, just as is the case with other data. When a message arrives at a replacement combinator, the message may be treated in two different ways. If the messages contains a network, i.e. a potential operand for the combinator, this network will be inspected. If it is compatible to the currently deployed operand, the replacement combinator reads and removes the network from the message. The combinator replaces its current operand by the new network and discards its old operand. If the message contains an incompatible network or no network at all, the message is sent to the operand of the replacement combinator. In this case, the message is processed as usual.

**Feed-back combinator.**

The replacement combinator allows for reconfiguration to be triggered by a network component that lies upstream of it or from the global input of the network, i.e., triggered from the outside. However, it is often desirable to monitor the results of a network and then make a decision on if or which kind of reconfiguration is required. In order to allow for the back-propagation of collected information, a feed-back combinator complements the replacement mechanism. The feed-back combinator introduces a back-edge in the streaming network and allows for a communication of a component with another component that lies further upstream.

The combination of these two combinators, i.e., a reconfiguration combinator as the operand to a feed-back combinator, is very versatile as it allows for the implementation of self-adaptation: By collecting runtime information of the operand of a replacement combinator and then acting on this information once a result exits the combinator’s operand network, we may make a decision on how the network should be reconfigured. A reconfigured operand may then be sent back through the feed-back edge to the replacement combinator that is under observation.

Dynamic Reconfiguration can be used handle different kinds of faults:

**Hardware Faults:** When some resource doesn’t work correctly, then it needs to be changed. One possible strategy is to use resource reallocation, and change the bad resource for another good resource. To do this, Dynamic Reconfiguration knows about all the resource in the system and must choose a spare resource to replace a broken one. Its possible to see one example in Figure 4. In the figure, such as dynamic reconfiguration observed the network. When a problem occurs, inform the relevant software (runtime monitor) and the software acts accordingly. In the example, resource R1 is broken. Then dynamic reconfiguration does not catch the error. The error is recognized by the runtime monitor and dynamic reconfiguration is used to remove the fault.

**Software Faults:** The system is affected by a software fault. The author of this software can find the problem and change this implementation. The basic building blocks of an S-Net program are boxes, and we only need to replace the box with the error with another box with alternative implementation. Dynamic Reconfiguration is a method for replacing a bad box with another box. (Figure 5)

**Timing Faults:** When there is a significant delay in the execution time and the program needs to finish before a determinate time 

(DeadLine). Dynamic Reconfiguration considers all the available resources to chooses the best one at any moment to do the less execution time. In order to illustrate how these mechanisms may be put to use, we consider the case of dynamic reconfiguration of a system with three resources (R1, R2, R3). The application consists of three boxes called B1, B2 and B3. Initially, resource R1 will execute the box B1; resource R2 will execute the boxes B2 and B3; resource R3 will be free (Figure 6). But there is a problem with the delay, because resource R2 is busy when box B3 arrives. Then Dynamic Reconfiguration will perform Resource Reallocation and it will put the one available resource (in this case R3) for execute the box B3 (Figure 7). There isn’t any further delay and the program’s deadline can be met.

**Channel Faults:** There is a problem with the connection between boxes. Different problems can occur, for example, a message can have become corrupted before it arrives at the receiving box or the message can be lost (connection broken). To know if the messages are wrong, it is possible to use checksums for messages. To repair a faulty channel, Dynamic Reconfiguration can create an alternative channel between boxes to repair this problem (Figure 8). Then the message will travel to the alternative channel. If the fault persists, maybe it is not a channel fault, it will be a hardware fault.

We can do this by sending a restructured network with a new resource annotation over the same input stream that we use to feed all other data items into the application. If the requirements change over time this reconfiguration may be repeated in order to maintain high system efficiency under given constraints.

Where manual intervention is not desired or unpractical we may also implement an automated reallocation. Figure 9 an example of this that is possible to use for solve all the several problems. The boxes showing a clock are time-stamp generators that attach a time-stamp to the messages that flow past them. The network operates as follows. A data item that enters the network is extended by a timestamp to the messages that flow past them. The network operates as follows. A data item that enters the network is extended by a time-stamp. A result that is emitted by the network is again time-stamped, i.e., a message carries two time-stamps in total. The Trigger box inspects these two timestamps and may then
Figure 5. Example software fault.

Figure 6. T3 cannot complete within deadline because resource is busy.

Figure 7. Dynamic resource allocation with timing fault.

Figure 8. Example channel fault.

Figure 9. Timestamps on data messages can be used to trigger automatic reconfiguration of a system.

decide if the processing time lies within acceptable boundaries. If this isn’t the case, the box outputs a new resource mapping that is sent back to the beginning of the processing network through a feedback stream (the back-edge that is seen on top of Figure 9; this is achieved by applying the feedback combinator to the processing network).

If the measured processing time lies within acceptable bounds the trigger box strips the time-stamps from the result before forwarding it, without producing a new resource mapping.

Using this and similar techniques, other reconfiguration and adaptation strategies may be implemented such as automatic restarting of services using heartbeats, dynamic re compilation and optimisations of box code and re-balancing load on computational clusters, and also fault tolerance by remapping resources for critical tasks in a mixed-criticality system in case of partial loss of resources due to a fault.

3.3 Redundant Computation

Besides re-computation, another fundamental pattern of fault tolerance is redundant computation. In many applications, redundancy is a extensively used strategy to avoid misunderstandings or decoding errors. Descriptively, redundancy is strategic communication factor consists in intensifying and repeat the information contained in the result, even without a fault occurring. Redundancy can be used in several different ways for fault tolerance:

Fault detection uses redundancy in order to obtain inconsistencies of the produced results in case of a fault.
Fault correction means fault tolerance without the need of re-computation after a fault has been detected. Fault correction has to be used in real-time systems if re-computation would not be possible without risking a deadline violation.

Assuming a deterministic system, to tolerate the case where a subsystem shows one fault failure (incomplete but no wrong output produced) [1], redundant computation with only two instantiations of the computation is sufficient to correct this silent failure of the subsystem.

In case the subsystem fails with erratic failures [1], a majority voting over several instantiations of the computation has to be used to tolerate this fault without the need for restore and re-computation. For example, to tolerate one failure either in the communication system or in the computational units, triple-modular redundancy (TMR) can be used [12].

Redundancy in S-Net

Regardless of using redundancy for fault detection or fault correction in S-Net, the results of the different outputs within the redundancy region have to be held back and buffered until a conclusion can be drawn.

Figure 10 shows an S-Net subnetwork that is surrounded by a redundancy region FT-RED(n) meaning that n instances of the subnetwork will be created. Again, the input stream can be seen as a simple FIFO (first in, first out) queue. But in contrast to the buffer of checkpoint/restore as described in Section 3.1, the output messages of the covered subnetwork have to be temporarily stored and held back in a three-dimensional buffer. For each input message \( i_k \) each of the n redundant S-Net subnetworks may produce \( m_k \) output messages: \( o_0 \ldots o_{m_k-1} \), stored separately for each of the \( n \) redundancy instances. The FT-RED(n) region will output a message only when enough of redundant outputs have been produced. "Enough" in this context is dependent on the fault-tolerance technique to be realised with the redundancy, e.g., a) for fault detection with two instances of the subnetwork, all two messages have to arrive before it can be forwarded, and b) for TAMR at least two of the redundant outputs have to be produced (in the absence of faults) before it can be forwarded. Note that the holding back and storing of output is only for redundancy purposes, i.e., the \( m_k \) individual messages resulting from multiplicity if the subnetwork are not needed to be held together. The multiplicity is either ignored (e.g., for fault-masking with TAMR) or is handled in the same way as described in Section 3.1 for checkpoint/restore (e.g., when combining fault-detection based on redundancy with checkpoint/restore).

We finally have to note that above discussion is slightly simplified, as we only want to show principal feasibility of fault-tolerance with S-Net based on redundancy. In practical realisations it is slightly more complicated. For example, when using TMR regions, the output of each redundant subnetwork instance of a FT-RED(n) region has to be forwarded to each of \( n \) subnetwork instances of the succeeding FT-RED(n) region [12]. Without such an individual connection between the subnetworks of two adjacent FT-RED(n) regions the voting would be a single point of failure.

4. Related Work

Software-controlled fault tolerance is a broad field of research. For example, Elnozahy et al. give a good overview of different rollback-recovery protocols in message-passing systems [5]. The authors also conclude that the main overhead of checkpointing in modern systems stems from storing the associated state. Choi et al. discuss compiler support for automatic checkpointing in general [3]. The frequency of checkpointing with general programming languages is limited due to this high overhead.

Therefore it is beneficial to minimise the state that is to be stored. Rich type systems as they are commonly found in functional programming languages can be of great value in this context. By exploiting structural information on heap objects derived from their types, the size of the checkpoint data can easily be reduced by up to 70% [3]. Functional programming languages here have the key advantage that due to there call-by-value semantics. The call-by-value semantics ensures the arguments to functions are immutable. Further, side effects would need to be explicitly modelled. This allows an effective containment of function state in a functional setting. Only the arguments to a function are required to restart its computation and the function itself does not modify any global state. Therefore, support for reversing and repeating a computation in a compiler for functional languages is simple to implement compared to the imperative setting [9]. Kirner et al. discussed several techniques for software-controlled fault tolerance in a functional setting [11].

S-Net is not a strictly functional language, as there is state on the network, e.g., in buffers of streams or in case of explicit synchronisation with so-called Synchro-cells. But boxes in S-Net have a functional setting, i.e., there is no persistent state inside the boxes. Thus, it is expected to be able to take advantage of this property for S-Net, similar to what has been shown for functional languages.

5. Conclusion

Development of fault-tolerant systems has its strong need in the domain of embedded computing as well as high-performance computing, although these two domains have apparently different application requirements. Within this paper we have studied similarities of fault tolerance for these two application domains.

The study has been based on S-Net, a coordination language for stream-processing systems. As a result we have shown that fault-tolerance mechanisms from the safety-critical embedded domain can be supported by a tool chain for S-Net. This is remarkable, as S-Net, which has been originally developed for the high-performance domain, has a rather flexible execution model that supports asynchronous computation and processing of partial output of its components.

As future work we will develop the control protocol needed to trigger the fault-tolerance actions and implement the approach.

Acknowledgement

The research leading to these results has received funding from the IST FP7 research project “Asynchronous and Dynamic Virtualization through performance ANalysis to support Concurrency Engineering (ADVANCE)”, and the ARTEMIS-JU research project “ConstRaint and Application driven Framework for Tailoring Embedded Real-time Systems (CRAFTERS)”. 
References


