Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
Electronic Notes in Theoretical Computer Science 238 (2009) 57–69
www.elsevier.com/locate/entcs

# Generating Multi-Threaded code from Polychronous Specifications

Bijoy A. Jose [a,1,2], Hiren D. Patel[b,3], Sandeep K. Shukla[a,1,4] and Jean-Pierre Talpin [c,5]

[a] *FERMAT Lab*
*Virginia Polytechnic Institute and State University*
*Blacksburg, VA, USA*

[b] *Ptolemy Group*
*University of California, Berkeley*
*Berkeley, CA, USA*

[c] *ESPRESSO Project*
*IRISA/INRIA*
*Rennes, France*

**Abstract**

SIGNAL, Lustre, Esterel, and a few other synchronous programming language compilers accomplish automated sequential code generation from synchronous specifications. In generating sequential code, the concurrency expressed in the synchronous programs is sequentialized mostly because such embedded software was designed to run on single-core processors. With the widespread advent of multi-core processors, it is time for model-driven generation of efficient concurrent multi-threaded code. Synchronous programming models capture concurrency in the computation quite naturally, especially in its data-flow multi-clock (polychronous) flavor. Therefore, it seems reasonable to attempt generating multi-threaded code from polychronous data-flow models. However, multi-threaded code generation from polychronous languages aimed at multi-core processors is still in its infancy. In the recent release of the Polychrony compiler, multi-threaded code generation uses micro-level threading which creates a large number of threads and equally large number of semaphores, leading to inefficiency. We propose a process-oriented and non-invasive multi-threaded code generation using the sequential code generators. By *noninvasive* we mean that instead of changing the compiler, we use the existing sequential code generator and separately synthesize some programming glue to generate efficient multi-threaded code. This paper describes the problem of multi-threaded code generation in general, and elaborates on how Polychrony compiler for sequential code generation is used to accomplish multi-threaded code generation.

*Keywords:* Synchronous Programming Model, Polychrony, SIGNAL, Multi-core, Multi-threading, Embedded software, Synthesis

[1] This work is supported by NSF Grant CCF-0702316

[2] Email: bijoy@vt.edu

[3] Email: hiren@eecs.berkeley.edu

[4] Email: shukla@vt.edu

[5] Email: jean-pierre.talpin@irisa.fr

doi:10.1016/j.entcs.2008.01.006

## 1 Introduction

In the last few years, parallel processing has claimed its niche in improving performance and power trade-off [12] for general purpose computing. So, it is no surprise that multi-core architectures will make inroads into the embedded processor markets as well. Currently, major processor vendors have already released products containing multiple cores on a single die [11], however, one must employ concurrent programming models when designing their programs to exploit the architecture with multiple cores in such products. One such programming model, and one that most designers are familiar with, is the multi-threaded programming model.

This multi-threaded programming model is commonly used in providing uniprocessor architectures with concurrency, and thus it is one candidate for true parallelism with parallel processing architectures. However, due to our long association with the von Neumann sequential programming models, it is often hard to write correct multi-threaded code [15]. Usually, writing such code involves expressing the computation as a collection of tasks, analyzing their dependencies, finding concurrency between the tasks, finding synchronization points, and then expressing all those with the programming idioms available in a language of one's choice.

We limit our discussion on multi-threaded programming for multi-core architectures with the C programming language, which to many, naturally implies the use of POSIX thread primitives or some other threading library APIs. Since writing multi-threaded C-code for a computation specified in sequential manner is hard, it would be easier if a concurrent model of computation was used instead, to specify the computation. For example, Petri nets may be used to specify the computation. With a highly concurrent Petri net model for a given computation, one could discover concurrency, synchronization points etc, much more readily when compared to standard C multi-threaded programming.

However, Petri nets have their own limitations as a specification formalism. First, one could unintentionally go very close to the implementation model in the Petri net itself, by sequentializing some transitions unnecessarily, and thereby eliminating possibility of concurrency. Figure 1 describes one such scenario where in Net 1, there is concurrency between tasks $T_1$ and $T_2$, but in Net 2, the concurrency is eliminated by ordering them. The application being modeled in this example may have had no reason to sequentialize the two tasks $T_1$ and $T_2$, except that the modeler had made a decision to do so. Also, since Petri nets are graphical formalism, often it is hard to manage for large scale programs.

An alternative to Petri nets is to use the dataflow models of computation, where the variables are considered as infinite sequences of data values, and the valuation from one step to the next is done by various operations on the data streams. The concurrency is usually difficult to sequentialize inadvertently in such specifications, because one has to explicitly impose special scheduling relations to sequentialize two operations. Polychronous languages and in particular in this paper, SIGNAL, are examples of such dataflow languages that have the notion of rate of data arrival, and multi-rate data value computation built into the language. These rates are

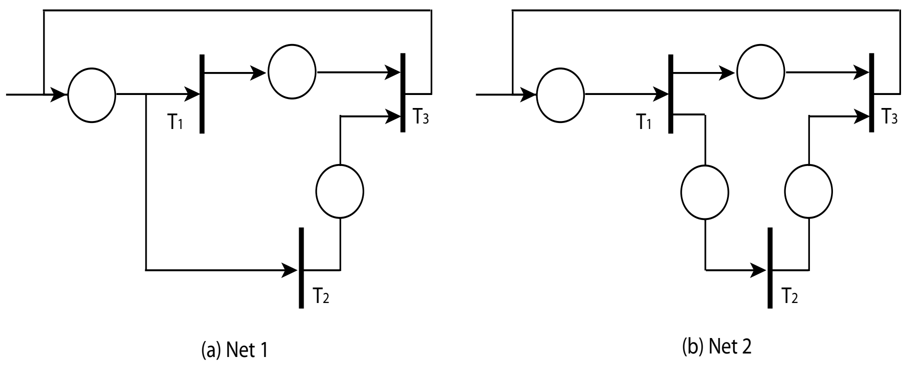

Fig. 1. Petri net representation of sequential and parallel processes

represented by a notion of *clocks*, which is not to be confused with the hardware clock. So the language SIGNAL is a multi-rate dataflow computation language, which captures concurrency of data valuations and enables one to sequentialize when necessary using 'clock relations' [14].

Specifications of computations as dataflow can be compiled into C-code using SIGNAL compiler. Not all dataflow computation can be correctly compiled into C-code, unless it has a property called *endochrony*[4]. Endochronous specifications are those for which the ordering of data value evaluations can be completely determined at the compile time, and therefore, *deterministic sequential* C-code can be generated. In order for generating distributed code however, the endochrony property is not enough. If two parts of the dataflow computation are synthesized into independent entities such as threads, one has to guarantee that their communication has special property of *isochrony* [4,3]. *Isochrony* is a condition that mainly applies on common variables of two synchronous components where they must agree on the values which are assigned to it at each time instant. If the isochrony property is violated, then the communication between the threads will need specialized protocols such as hand-shake or other synchronization.

Synchronous-Flow Dependence (SDF) graph is a convenient way of representing SIGNAL specifications. The nodes of the SDF graph would represent a *Signal* in the code, while the arcs will represent the dependencies between the clocks.

A recent enhancement of the SIGNAL compiler generates multi-threaded code with tight synchronization between atomically executing threads. We find this a valuable step towards managing concurrency, but multi-threading if not judiciously employed has severe performance drawbacks due to the usage of too many threads resulting in high context switching overheads. The current SIGNAL compiler, in our opinion, performs micro-threading that generates C programs with an unnecessary number of threads in relation to the parallel computation. We illustrate this with the following pseudo SIGNAL code example:

```
process P = (? integer x; !integer y,z;)
(| y := x + 1 | z := x-1|)
```

This simple SIGNAL code fragment provides an input argument `x` and two output arguments `y` and `z`. The behavior of `P` specifies parallel execution of computing and assigning $x + 1$ and $x - 1$ to `y` and `z` respectively. Figure 2 shows the SIGNAL implementation and its thread-call structure.

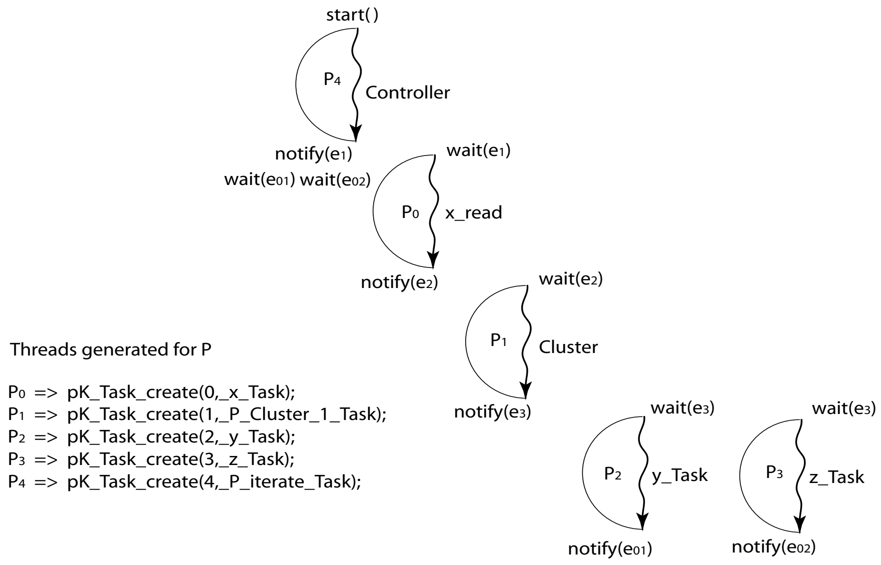

Fig. 2. Current Micro-threading Structure by SIGNAL compiler

For this simple code fragment, the current SIGNAL compiler generates 5 threads; one for reading values of `x`, two for outputting `y,z`, one for the computation of the values of `y,z` based on new values of `x`, and one controller thread that sequentializes the reading of `x`, followed by the computation, and finally followed by writing `y,z`. The use of 5 semaphores to synchronize the sequentialization is very inefficient. In case the main computation is multi-rate, this compiler will generate several threads by clustering the computations driven by the same clock in single threads, and then again controlling the sequence using semaphores. We feel that such micro-level threading may be counter productive in terms of efficiency. However, a bigger problem is that the SIGNAL compiler is not open-source yet. So, if one does not agree with such micro-threading strategy in their current threading implementation, any invasive change in the compiler is not possible for external users. Moreover, parallel architectures for embedded computing have a limited number of resources for multi-threading, which makes judicious selection of threads for parallelism extremely important.

In this paper, we investigate a *noninvasive* methodology for utilizing polychronous dataflow specifications, the only available compiler for such specification, and no change in the compiler, but still generate multi-threaded code, even when the endo-isochrony is not satisfied between the partitioned parts of the dataflow computation specified. In doing our investigations, we hope to provide a proposal for extending the SIGNAL compiler with better granularity in its thread-based code generation.

The main contribution of this paper is to show how multi-threaded C-code can

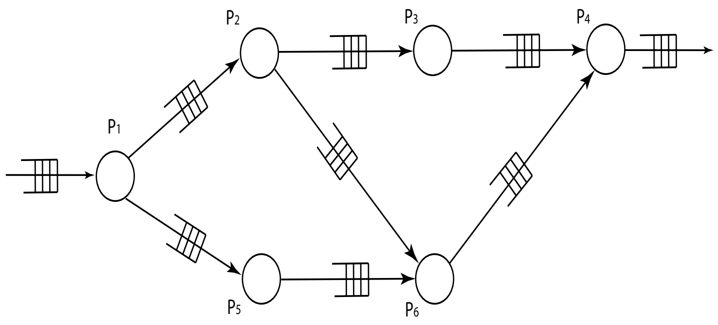

Fig. 3. A sample KPN model

be generated using polychronous specifications and corresponding compilers for sequential code generation with designer imposed threading granularity and designer specified threading boundaries. This alleviates the problem of (i) having to write multi-threaded code by hand from a sequential specification of the computation being implemented; and (ii) micro-level thread granularity causing unnecessary performance overhead. We illustrate our methodology with a running example from the STARMAC project [1].

## 2 Related Work on Concurrent Code Generation

In this section we outline a few relevant related works on concurrent code generation from concurrent specification of computation. We discuss some of the Models of Computation (MoC) used to express concurrency between processes. Other than that, we reemphasize some of the issues we have with existing synchronous programming languages based multi-threading implementations.

Kahn process networks (KPN) [13] is a primary specification formalism for dataflow between processes. KPN model has been proposed to capture concurrent processes in a system. In a KPN, the processes (nodes) communicate by unbounded unidirectional FIFO channels (arcs), with the property of non-blocking writes and blocking reads on channels. A KPN model when transformed to implementation level faces problems related to buffer sizing. Several scheduling algorithms have been proposed for estimating the buffer sizes for KPN models and for validating their correct behavior [9,18]. Another MoC which closely resembles KPN model is dataflow networks [16]. Here the nodes represent actors which when fired takes in a finite number of tokens and put out another finite number of tokens. In contrast with KPN, this MoC can calculate the buffer size with respect to the number of computations in the network.

A sample KPN model is shown in Figure 3 where 6 processes $(P_1, P_2, .., P_6)$ and their intermediate FIFOs are shown. In an ideal environment with infinite size FIFOs, each Kahn process is a concurrent task with asynchronous communication. Here, the processes $P_2$ and $P_5$ (also $P_3$ and $P_6$) are parallel to each other, since they do not have a dependency between them. With these concepts in mind, we can implement the given KPN model in multi-threaded programming model as follows:

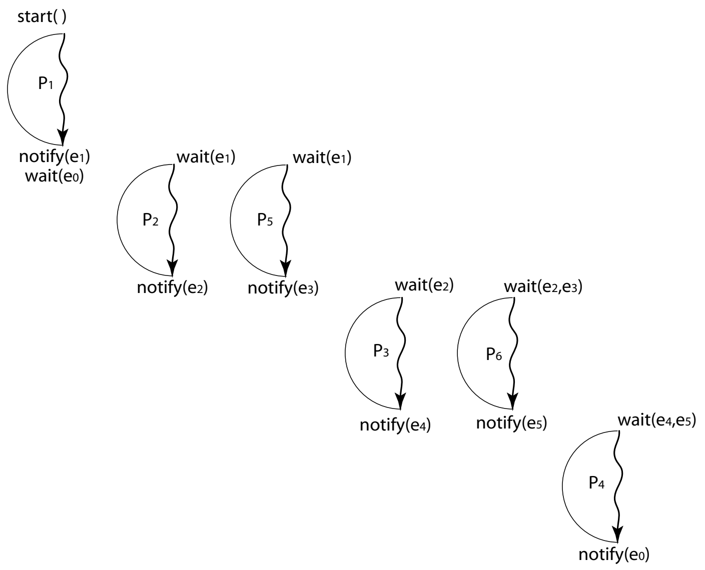

Fig. 4. Execution of KPN model in a parallel fashion

$$[P_1\ ;\ (P_2\ ||\ P_5)\ ;\ (P_3\ ||\ P_6)\ ;\ P_4]^*$$ [6]

This expression represents one of the possible implementations from the design specification as a KPN. The assumption here, is that each process needs to execute to completion to produce data for the subsequent processes. For implementation purposes, the given KPN model can be transformed into multiple threads with a deterministic order for their executions except for the processes that are completely independent (e.g., $P_2$ and $P_5$). In Figure 4 one multi-threaded implementation for the specification in Figure 3 is shown. As shown in Figure 4 there is a deterministic order of the thread executions except between $P_2$ and $P_5$ and $P_3$ and $P_6$. In case, sequential code is to be generated, two of the possible orders for execution are expressed as follows:

$$Order1 : [P_1\ ;\ P_2\ ;\ P_5\ ;\ P_3\ ;\ P_6\ ;\ P_4]^*$$

$$Order2 : [P_1\ ;\ P_5\ ;\ P_2\ ;\ P_6\ ;\ P_3\ ;\ P_4]^*$$

Many tools have been developed based on these various dataflow models. Simulink is an environment for multi-domain simulation and model-based design for dynamic and embedded systems [20]. Simulink has a Stateflow coder that generates highly readable code which can be easily traced back to the Stateflow chart. Recently, a method for multi-threaded code generation from Simulink models with reduction of thread communication overhead was proposed [6]. This work is similar to our current methodology in the sense that they also concentrate on *larger granularity* for threads to reduce overhead of thread synchronizations. Lustre [10] is a declarative synchronous dataflow language used for describing reactive systems. Esterel [5] is an imperative language for describing control. It consists of

[6] Here ';' represents the sequential composition, '||' represents the parallel composition and '$*$' represents repetition

nested tasks which communicate using signals. Both Esterel and Lustre are tools designed for embedded systems which generate sequential C code after an intermediate conversion to Object code. A recent work related to Esterel [7], presents a few code generation techniques. One method aims to perform aggressive scheduling operations after dividing the code into atomic tasks, while another creates a linked list to track pieces of code and their dependencies. The objective is to generate multi-threaded C code using their Esterel compiler. The parallelism implemented in their work concentrates on dividing the design requirement into very small tasks, whereas we would like to implement a process-based parallelism in our work with large granularity of tasks. Another work proposes a new compilation scheme for Esterel for distributed execution on multiprocessors [21]. This scheme based on Potop-Butucaru's Graph Code format [19] enables parallelism through actual concurrent execution on multiple processors or through emulated concurrent execution on single processor.

As we discussed earlier, our specification language is SIGNAL. SIGNAL is based on synchronized dataflow (flows + synchronization): a process is a set of equations on elementary flows describing both data and control [14]. Polychrony constitutes a development environment for critical systems, from abstract specification until deployment on distributed systems [8]. It relies on the application of formal methods, allowed by the representation of a system, at the different steps of its development, in the Signal polychronous semantic model. The synchronous distribution of SIGNAL programs [2] was investigated and the SIGNAL dependencies and equations were formally was proved to be kept true until the final stage of distributed code generation. Another work on SIGNAL motivates a scheduling strategy for distributed implementation [17]. As we discussed before, the recent implementation of [17,2] in the SIGNAL compiler leads to generation of multi-threaded code with too much overhead which we attempt to alleviate in the current work.

Besides the multi-threading code generation, the Polychrony tool mainly generates sequential C code from SIGNAL. Every variable coded in SIGNAL (termed as signal) is associated with its own *rate*, which decides its update frequency. Statements which assign a relation between signals, also define a relation between the *rates* of the respective signals. Groups of dependent signals will have a common *rate* being generated. A higher level logic is generated with a root *rate* combining the independent groups of dependent signals according their relative rates. Even though we separate the independent tasks which make the signals concurrent, the execution is not parallel, since a sequential C code is being generated.

## 3 The STARMAC Example

A group at Stanford [1] is currently working on a testbed for multi-agent control called STARMAC. It contains four rotors that are controlled by a supervisory control unit managing the flight control commands. We present the height supervisory control unit in Figure 5 with its design requirement and eventual implementation using Simulink. Looking first at the design requirements in Figure 5(a), we see the

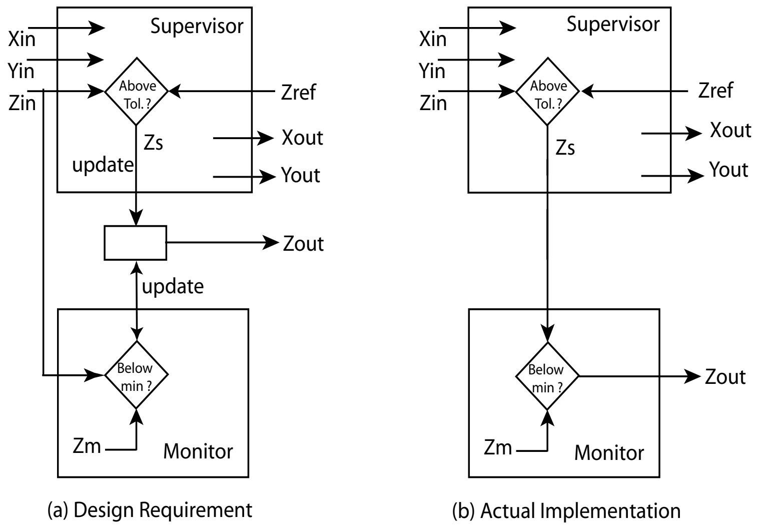

Fig. 5. STARMAC Design requirement

`Supervisor` and `Monitor` entities. The `Supervisor` is responsible for reacting to the major functions of flight control such as computing possible next coordinates to move to. The `Monitor` on the other hand only focuses on maintaining the elevation of the STARMAC testbed by proposing adjustments to the elevation when below a certain threshold. Labels $X$, $Y$, and $Z$ represent the x, y and z coordinates of the STARMAC and the *in* and *out* suffixes the respective inputs and outputs from this supervisory control unit. $Zm$ is the allowable minimum elevation for a particular rotor.

Ideally, these entities need to execute concurrently, which is a natural call for representing each of these entities as a separate thread. However, when looking at the actual implementation of this supervisory control unit in Figure 5(b), we see a sequentialization of the concurrency; in particular, the `Supervisor` is executed before the `Monitor`. Simulink toolbox Stateflow was used to implement the design requirement. The parallel *AND states* in Stateflow need a specific ordering to generate C code. This choice of sequentialization may have further ramifications on the implementation in our opinion is not faithful to the specification. On the flip side, modeling these two entities as threads means that the designer must explicitly handle the critical section for $Zout$. Note in Figure 5(a) that there is a shared box pointed to with the arrows labeled `update` that eventually sends the $Zout$. Writing the embedded program for this supervisory control unit mandates care in order to get the possible interleaving correct.

## 4 The SIGNAL Approach to Concurrent Software Specification and Generation

As discussed already, a new version of Polychrony has just been released with multi-threading in place [8]. The approach to multi-threading in the tool is based on signal flow. The number of threads created would be dependent on the number of input and output signals and the dependencies between them. Individual threads are created to read an input, to do the tasks involving that input signal and finally to write the output signal. Every action related to a combination of several inputs would require additional threads. Now these threads are ordered using semaphores which indicate a condition being satisfied for the next thread to start.

A scaled down version of the STARMAC requirement concerning only the elevation monitoring function was modeled in SIGNAL to highlight the concurrency issues. The `Supervisor` and `Monitor` entities are modeled as processes called from a `Control` main process. Both `Supervisor` and `Monitor` processes return their output values back to the `Control` process. There is a deterministic ordering in place to decide who should be updating the output value of *Zout*. Even though the two processes are having different rates, SIGNAL would not allow us to have a random order for update of the output. This would take us back to the original problem in implementation shown in Fig 5(b). So a complete parallel implementation in C is not possible with the current multi-threaded model. The SIGNAL code used to model the STARMAC design requirement is as follows:

```
process control = ( ? integer Zin, Zref;
    event tick1, tick2;
  !  integer Zout;  )
     ( | Zoutsupervisor ^= tick1
       | Zoutmonitor ^= tick2
       | Zoutsupervisor := supervisor (Zin,Zref)
       |  Zoutmonitor:= monitor {1} (Zin)
       | Zout := Zoutsupervisor default Zoutmonitor
       |  )
       where
       integer  Zoutsupervisor init 0, Zoutmonitor init 0;

       process supervisor =
       ( ? integer Zins, Zrefs;
       ! integer Zouts;
        )
           (| Zouts := Zins when (Zins < Zrefs) default Zrefs
            | );

       process monitor =
       { integer Zm}
```

```
        ( ? integer Zinm;
        ! integer Zoutm;
         )
            ( | Zoutm := Zinm when (Zinm > Zm)
              |);
end;
```

A thread model based on the threads generated from SIGNAL tool is shown in the Figure 6. Here the 3 process example is converted into 7 threads ordered using semaphores. Concurrent functions like read operation for $Zin$, $Zref$ and $tick1$ are executed in a parallel fashion and the operations on them are performed using multiple threads. Finally the $Zout$ signal is updated using an additional thread. This example clearly illustrates the micro-threading model in Polychrony tool.

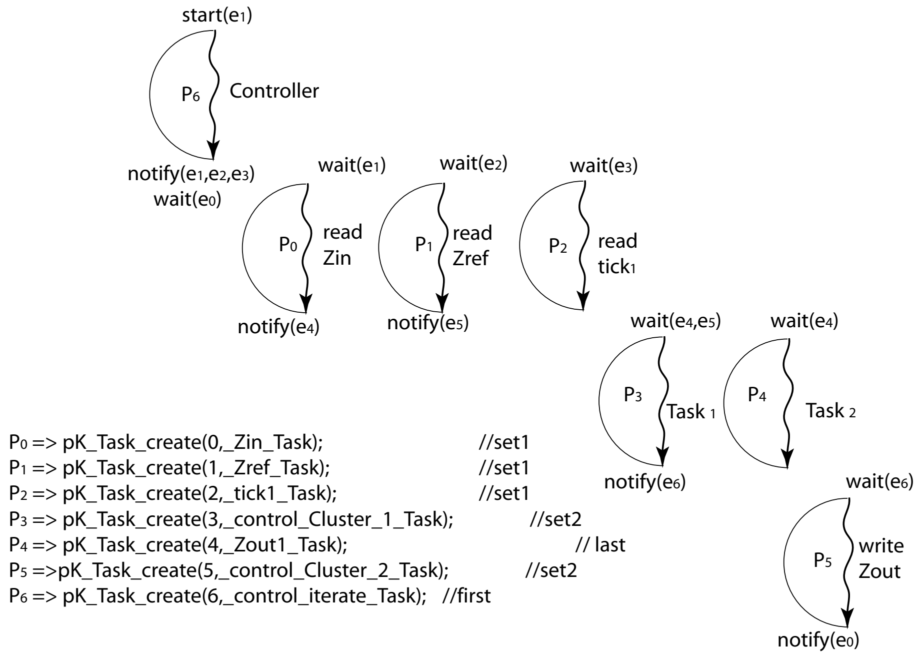
start(e1)
P6
Controller
notify(e1,e2,e3)
wait(e0)
wait(e1)
P0
read Zin
notify(e4)
wait(e2)
P1
read Zref
notify(e5)
wait(e3)
P2
read tick1
wait(e4,e5)
P3
Task 1
notify(e6)
wait(e4)
P4
Task 2
wait(e6)
P5
write Zout
notify(e0)
P0 => pK_Task_create(0,_Zin_Task); //set1
P1 => pK_Task_create(1,_Zref_Task); //set1
P2 => pK_Task_create(2,_tick1_Task); //set1
P3 => pK_Task_create(3,_control_Cluster_1_Task); //set2
P4 => pK_Task_create(4,_Zout1_Task); // last
P5 =>pK_Task_create(5,_control_Cluster_2_Task); //set2
P6 => pK_Task_create(6,_control_iterate_Task); //first

Fig. 6. Micro-threading Structure for STARMAC

In the current strategy, the programmer is kept away from decisions regarding the number of threads and how they should be grouped. In order to utilize multi-core architectures, it is important for the programmer to be aware of the implementation level details, so that he can tailor it according to the hardware resources available to him. Therefore, we suggest an alternative multi-threading solution that may be inferred from the SIGNAL specification, which uses only two threads for the STARMAC example.

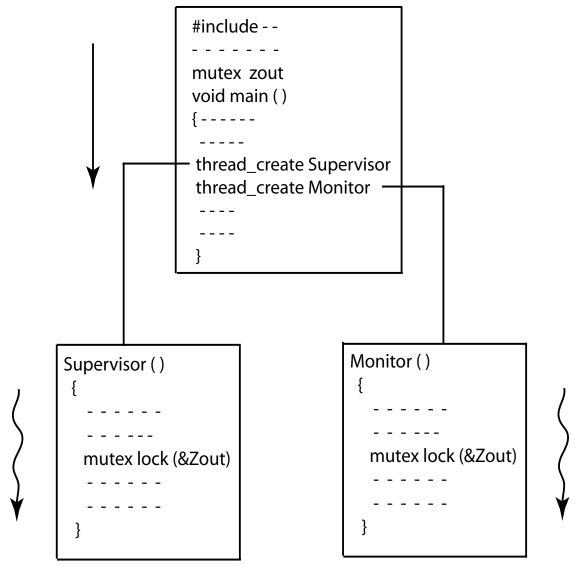

Fig. 7. Multi-thread model using SIGNAL

## 5 An Improved Threading Model for SIGNAL Programs

We use SIGNAL and attempt to faithfully model the specification of the STARMAC supervisory control unit shown in Figure 5(a). By using the multi-threaded code generation from the Polychrony compiler we suffer from the micro-level threading discussed earlier in this paper. In order to bypass the roadblock of not having access to the compiler, we take a non-invasive strategy in implementing our model. We use the sequential code generation option in the Polychrony tool and model both the two processes individually and operate on them with a minimal glue logic to realize the specification.

In order to generate a noninvasive threaded model using SIGNAL, we implemented concurrent processes as individual threads, with a main controller code. The Figure 7 shows the proposed thread based model, generated as processes using Polychrony tool and joined manually in C. The main controller process has to be executed sequentially and two threads are generated with the two different functions (Supervisor and Monitor) inside. Now these threads can work in parallel, switching between themselves.

During the process of implementing this new model, we first implemented the two functions Supervisor and Monitor as independent SIGNAL programs and observed their generated C code without using the threading option. With the sequential C code for each of the programs as reference, they were converted into a combined code with a controller main calling each as threaded functions. We can observe from the design requirement that the $Zout$ value will be updated by both processes and it needs to be defined in the controller part. The shared variables were all defined in the controller, with the variable needing an update ($Zout$) implemented with Mutex lock. Several inputs well below the minimum value were given as input to verify the functionality of the code. Later the Supervisor process was modified to add more delay into the process. This would mean Monitor process will be updating the *Zout* variable to ensure correct output. The manual threading inserted into the Polychrony model worked correctly for this particular example. In order to generate threads directly from Polychrony tool, we need to identify those functions which can be executed from separate threads. Any shared signal would require Mutex lock system associated with it. Perl scripts could be written to combine these threads and generate a Main process which co-ordinates interaction between them. The amount of rewrite required to generate multi-threaded C code from SIGNAL compiler is minimal. Most of it would be transferring pieces of code from the different files generated from the compiler into the main C file. Now they are addressed as functions and threads create and join are written to access each of them. Mutex lock arrangement is implemented on shared variables for consistency and for avoiding deadlocks.

The current implementation will be behaving in a similar manner as required in design requirement of Figure 5(a). There is no specific sequential order at the implementation level and multi-rate scenarios can be modeled faithfully. Now the amount of parallelism is dependent on the scheduler or the kernel which will have to act fast enough to extract each input correctly. Our original motivation for multi-threading was the vulnerabilities of the software design which cannot make full use of the multi-core architecture available to us. This would mean kernel being fast enough is not a constraint but a feature we are converting to an advantage in the model.

## 6 Summary and Future Work

Concurrency has been a hot topic in computation field from the very beginning. Concurrency has been implemented by software tools by including additional dependencies between processes due to sequential ordering at a lower abstraction level. This has removed any parallelism that was present in the original high-level design. Multi-core architectures provide us an opportunity to utilize the multi-threaded model of software design. A parallel implementation of concurrent processes has been performed successfully on a sample problem with promising results.

Even though multi-threading is a popular approach among software engineers, automatic code generation using this model has not been explored enough. SIGNAL language has been utilized for creating polychronous independent processes using the Polychrony tool. A multi-threaded version of the Polychrony compiler takes a micro-threaded approach towards parallelizing operations. We feel this amount of micro-threading is wasteful and a process-based approach is more suitable for current multi-core architectures. We also find that a sequential order is still present in the multi-threaded model, which is a deviation from our original objective. We demonstrate these issues with an example dealing with concurrency. Right now we have a non-invasive method for generating parallel multi-threaded code which is being merged manually. The automation of multi-threaded code generation with more research into grouping and scheduling of threads would be possible way forward to achieve sophisticated multi-core implementations.

## References

[1] STARMAC Project Group, *The Stanford Testbed of Autonomous Rotorcraft for Multi-Agent Control Overview*, http://hybrid.stanford.edu/starmac/overview.

[2] Aubry, P., P. Le Guernic and S. Machard, *Synchronous distribution of SIGNAL programs*, in: *Proceedings of HICSS-29* (1996), pp. 656–665.

[3] Benveniste, A., B. Caillaud and P. Le Guernic, *From synchrony to asynchrony*, in: *International Conference on Concurrency Theory, CONCUR'99*, LNCS **1664** (1999), pp. 162–177.

[4] Benveniste, A., B. Caillaud and P. Le Guernic, *Compositionality in Dataflow Synchronous Languages: Specification and Distributed Code Generation*, Information and Computation **163** (2000), pp. 125–171.

[5] Boussinot, F. and R. de Simone, *The ESTEREL language*, Proceedings of the IEEE **79** (1991), pp. 1293–1304.

[6] Brisolara, L., S. Han, X. Guerin, L. Carro, R. Reis, S. Chae and A. Jerraya, *Reducing fine-grain communication overhead in multithreaded code generation for heterogeneous MPSoC*, 10th International Workshop on Software and Compilers for Embedded Systems (2007), pp. 82–89.

[7] Edwards, S. and J. Zeng, *Code Generation in the Columbia Esterel Compiler*, EURASIP Journal on Embedded Systems **2007** (2007), pp. 1–31.

[8] ESPRESSO Project, IRISA, *The Polychrony Toolset*, http://www.irisa.fr/espresso/Polychrony.

[9] Geilen, M. and T. Basten, *Requirements on the Execution of Kahn Process Networks*, in: *Proceedings of European Symposium on Programming, ESOP'03*, LNCS **2618** (2003), pp. 7–11.

[10] Halbwachs, N., P. Caspi, P. Raymond and D. Pilaud, *The Synchronous Data-Flow Programming Language LUSTRE*, Proceedings of the IEEE **79** (1991), pp. 1305–1320.

[11] Intel Corporation, *Intel Multi-core technology*, http://www.intel.com/multi-core/.

[12] Intel Multicore Resource Center, *The route to multithreading, the drive towards parallelism*, http://www.developers.net/intelmcshowcase.

[13] Kahn, G., *The Semantics of a Simple Language for Parallel Programming*, Proceedings of Information Processing (1974), pp. 471–475.

[14] Le Guernic, P., M. Le Borgne, T. Gauthier and C. Lemaire, *Programming real time applications with SIGNAL*, Proceedings of the IEEE **79** (1991), pp. 1321–1335.

[15] Lee, E. A., *The problem with threads*, Computer **39** (2006), pp. 33–42.

[16] Lee, E. A. and T. M. Parks, *Dataflow Process Networks*, Proceedings of the IEEE **83** (1995), pp. 773–801.

[17] Maffeis, O. and P. Le Guernic, *Distributed Implementation of Signal: Scheduling & Graph Clustering*, in: *3rd International School and Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems* (1994), pp. 547–566.

[18] Parks, T. M., "Bounded Scheduling of Process Networks," Ph.D. thesis, University of California, Berkeley (1995).

[19] Potop-Butucaru, D., "Optimizations for faster simulation of Esterel programs," Ph.D. thesis, Ecole des Mines, Paris, France (2002).

[20] The Mathworks, Inc., *Simulink, Simulation and Model-Based Design*, http://www.mathworks.com/products/simulink/.

[21] Yoong, L.-H., P. Roop, Z. Salcic and F. Gruian, *Compiling Esterel for Distributed Execution*, in: *Proceedings of Synchronous Languages, Applications, and Programming, SLAP'06*, Vienna, Austria, 2006.