FROM DESIGN-TIME CONCURRENCY TO EFFECTIVE IMPLEMENTATION PARALLELISM: THE MULTI-CLOCK REACTIVE CASE

V. Papailiopoulou, D. Potop-Butucaru, and Y. Sorel[1], *R. de Simone*[2], *L. Besnard and J.-P. Talpin* [3]

[1]INRIA Rocquencourt, Domaine de Voluceau, BP105, 78153 Le Chesnay Cedex, France
[2]INRIA Sophia Antipolis, 2004, route des Lucioles, 06902 Sophia Antipolis Cedex, France
[3]INRIA/IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France

ABSTRACT

We have defined a full design flow starting from high-level domain specific languages (Simulink, SCADE, AADL, SysML, MARTE, SystemC) and going all the way to the generation of deterministic concurrent (multi-threaded) executable code for (distributed) simulation or implementation. Based on the theory of weakly endochronous systems, our flow allows the automatic detection of potential parallelism in the functional specification, which is then used to allow the generation of concurrent (multi-thread) code for parallel, possibly distributed implementations.

***Index Terms*—** multi-clock synchronous, weak endochrony, concurrent, multi-thread, distributed

1. INTRODUCTION

Synchronous reactive formalisms [1] are modeling and programming languages based on concepts borrowed from synchronous circuit design: synchronous concurrency and the Mealy machine paradigm. These concepts are generalized, and then applied at different design levels, such as the system level of embedded control applications.

Existing results show that multi-clock synchronous (polychronous) formalisms such as Signal/Polychrony [2] allow the natural representation of comprehensive sub-sets of common domain-specific languages (DSLs) such as Simulink, SCADE, MARTE, AADL, SysML, SystemC, etc. [3]. When the development of a complex embedded system involves the use of several DSLs, the partial specifications can be translated in Signal/Polychrony, and then assembled into a global specification, as pictured in Fig. 1. When first assembled, a global specification can be functionally incomplete and non-deterministic, its behavior depending on unknown implementation details (*e.g.* computation and communication speeds), or on implementation decisions that have yet to be made, such as the choice of scheduling policies, resource allocation, or synchronization between various components.

This initial specification is then converted into a deterministic running implementation by progressively determining the functionality and defining the architecture mapping.

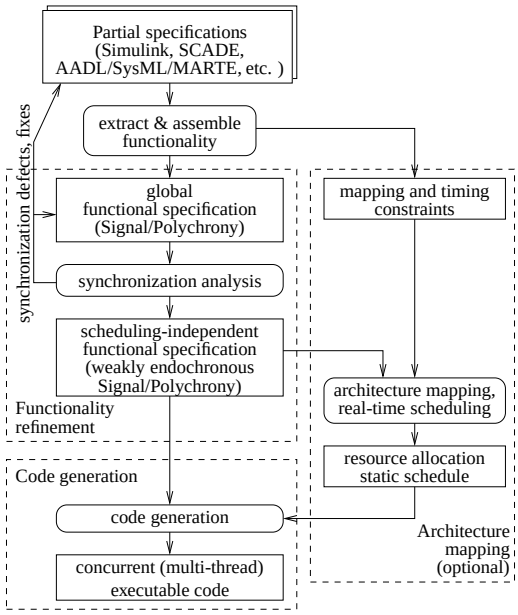


Fig. 1. Proposed design flow.

The implementation process is governed by two objectives: correction and efficiency. While maintaining the formal correctness imperative specific to the synchronous paradigm [1], we focus here on a particular efficiency question: *The mapping of the potential parallelism of the specification into effective parallelism on a parallel, possibly distributed execution platform*. This mapping is obviously limited by the capabilities of the platform. Also, the specification is often very concurrent in form, but potential parallelism is limited by the user-defined interactions and by the synchronous model itself, whose cyclic execution behavior involves an implicit system-wide synchronization at each execution cycle.

We structure the implementation process as pictured in Fig. 1. The *functionality refinement* phase completes the initial specification into a deterministic one, independent of mapping or scheduling choices. It also identifies potential parallelism by determining that, under the original synchronous form, some of the behaviors were actually independent, so that their computations can safely be kept asynchronous, without recreation of a global execution cy-

cle and the associated system-wide sequencing. The output of functionality refinement can be directly used to generate concurrent executable code for (distributed) simulation or implementation, which is what we present in this paper. It can also serve as input to architecture mapping, which is based on heuristic architecture exploration and static real-time scheduling [4].

The keypoint of the approach, defining its capabilities, is the functionality refinement phase. To allow the *automatic* detection of potential parallelism and the generation of *concurrent* implementations, our functionality refinement and code generation phases are based on the theory of weakly endochronous systems [5]. The current paper builds up on a series of articles by us and fellow co-authors, from the early motivations of [6] to extensive developments on the theory of weakly endochronous systems, including abstract algorithms solving part of the considered implementation problem [7]. The contribution of the current paper is to take these abstract notions and algorithms and make them effective, investigating code generation, distribution, language connection, and weak endochrony analysis issues with an efficiency-oriented viewpoint. We have implemented our symbolic data structures and algorithms in a prototype version, linked to the Signal/Polychrony-SME design environment as a back-end tool. Thus, programs are written in Signal, and various other analyses available for this language are thus possibly combined with our distributed code generation. Benchmarking on existing large Signal programs is under way.

The remainder of the paper is structured as follows: We first review related work. Then, we briefly introduce weak endochrony and the notion of *atomic behavior* which stands at the basis of our design flow. Then, we define the new code generation scheme (the main technical contribution of the paper, which completes the design flow). We conclude with some experimental results.

2. RELATED WORK

The brute force compilation schemes of synchronous languages, together with several optimizations taking active modes and dynamic signal sensitivity into consideration, are accounted for in [1, 8] and further previous articles are mentioned in them. The optimized executions (here event-driven) go one (small) step further in the direction of asynchronous execution of synchronous programs. N-synchronous processes [16] also allow some freedom in the original timings.

The concern for distributed execution of synchronous programs goes back to Girault and Caspi [9]. While it looks at first presumably simple (mapping logical parallelism onto physical one), the reaction to absence mandates that void absent-signal messages are physically sent to whichever process/task contains the signal as input. This heavily penalizes the efficiency of the distributed code.

Although it deals with the signal values (which may be used to decide which further signals are to be present next causally in the reaction), (weak) endochrony is in essence strongly related with the notion of *conflict-freeness*, which simply states that once enabled, an event cannot be disabled unless it has been fired, and which was first introduced in the context of Petri Nets. Various conflict-free variants of data-flow declarative formalisms form the area of *process networks* (such as Kahn Process Networks [10]), or various so-called domains of the Ptolemy environment such as SDF Process Networks [11]. Conflict-freeness is also called *confluence* ("diamond property") in process algebra theory [12], and monotony in Kahn Process Networks. Conflict-freeness criteria similar to endochrony are alreay used in the *latency-insensitive design* os systems-on-chips [13, 14, 15].

3. WEAK ENDOCHRONY

The theory of weakly endochronous (WE) systems [5] defines the sub-class of synchronous systems whose behavior does not depend on the timing of the various computations and communications, or on the relative arrival order of two events. Weak endochrony is compositional and provides the necessary and sufficient condition ensuring a deterministic asynchronous execution of the specification. In particular, it determines that compound program reactions that are apparently synchronous can be split into independent smaller reactions that can be executed in any order, even at the same time, taking advantage of the potential concurrency of the specification. Most importantly, any possible program reaction can be generated by the union of such compound independent reactions.

Given a synchronous program P, the set of its variables (also called *signals*) is denoted by $\mathcal{S}_P$. All signals are typed; the data type of a signal S is denoted by $\mathcal{D}_S$. A behavior (or *reaction*) of P is a tuple assigning one value to each of its signals. The absence of a signal is represented by the special value $\perp$. Thus, a reaction r of P assigns to each $S \in \mathcal{S}_P$ a value $r(S) \in \mathcal{D}_S^{\perp} = \mathcal{D}_S \cup \{\perp\}$.

Given a set of signals $\mathcal{S}$, the set of all valuations of the signals into their domains is expressed as $\mathcal{R}(\mathcal{S})$. Similarly, $\mathcal{R}(P)$ denotes the set of all reactions of a program P. Obviously, $\mathcal{R}(P) \subseteq \mathcal{R}(\mathcal{S}_P)$.

The partial order $\leq$ is introduced on each $\mathcal{D}_S^{\perp}$, defined by $\perp \leq v$ for all $v \in \mathcal{D}_S$. This relation can be extended component-wise to a partial order on sets of reactions. The least upper bound and greatest lower bound operators induced by $\leq$ on the sets $\mathcal{R}(\mathcal{S})$ are represented by $\vee$ (union) and $\wedge$ (intersection) respectively. Since $\leq$ defines a lower semilattice, the $\vee$ operator is not always defined.

If two reactions r and r' share no common present signals, their union $r \vee r'$ can be defined and we shall say that r and r' are non-contradictory and write $r \bowtie r'$. In case that r and r' can be distinguished by testing the present value of a common signal, they are contradictory ($r \not\bowtie r'$). This can be done even when observing the execution of P in an asynchronous environment where absence cannot be sensed. When $r \bowtie r'$,

their *difference* $r \setminus r'$ is also defined as the reaction exclusive with r' that satisfies $(r \setminus r') \vee (r \wedge r') = r$.

Definition 1 (Weak endochrony) *Under the previous notations, a synchronous program P is weakly endochronous whenever $\mathcal{R}(P)$ is:*

- *Closed under $\vee$ (recall that $\vee$ is defined on $\mathcal{R}(\mathcal{S}_P)$).*
- *Closed under $\wedge$ and $\setminus$ applied on non-contradictory reactions.*

and it also includes the stuttering *reaction $\perp$ which assigns $\perp$ to all signals.*

From our code generation point of view, the most important consequence of this definition is that for any weakly endochronous program there exists a subset of reactions $Atoms(P) \subseteq \mathcal{R}(P) \setminus \{\perp\}$, called *atomic reactions* or *atoms*, with two key properties:

Generation: Any reaction $r \in \mathcal{R}(P)$ is uniquely defined as a union of zero or more atoms.

Independence: Any two different atoms that are non-contradictory share no common present signal, so that they can be freely united to form composed reactions.

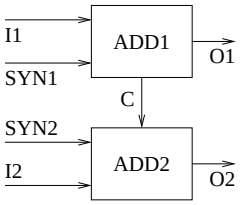
In other words, atoms are the elementary reactions of P, and two atoms are either contradictory (they can be distinguished in an asynchronous environment), or independent (they can be executed without any synchronization).

The theory of WE systems is defined in a non-causal framework where no difference is made between input and output signals. To allow for the synthesis of systems that are deterministic, we need to ensure that the behavior only depends on input values. This property is easily expressed at the level of atoms, by requiring that any two conflicting atoms have one conflicting input:

Determinism: When two atoms a, a' are contradictory, there exists an input signal I that is present in both a and a' with $a(I) \neq a'(I)$.

We illustrate weak endochrony on the small program of Fig. 2, which models a simple reconfigurable adder, where two independent single-word ALUs (`ADD1` and `ADD2`) can be used either independently, or synchronized to form a double-word ALU. The inputs of each adder are modeled by a single signal (`I1` and `I2`). This poses no problem because from the synchronization perspective of this paper the two integer inputs of an adder have the same properties.

At each execution cycle where `I1` is present, `ADD1` computes `O1` as the modulo sum of the components of `I1`. Similarly, `ADD2` computes `O2` whenever `I2` is present. The choice between synchronized and non-synchronized mode is done using the Boolean signals `SYN1` and `SYN2`. When they are both present and true, both `I1` and `I2` are present and a carry value is propagated from `ADD1` to `ADD2` through signal `C`. A value of false on either `SYN1` or `SYN2` means that the corresponding adder functions independently.




```
process WE_ADDER =
(? int_pair I1,I2;
   boolean SYN1,SYN2;
 ! event O1,O2;)
(| (O1,C) = ADD1(SYN1,I1)
 | O2 = ADD2(SYN2,I2,C)
 | when SYN1^=when SYN2^=C
 | I1^=SYN1 | I2^=SYN2 |)
where event C; end
```

Fig. 2. A weakly endochronous version of a configurable 2-word adder.

The atom set of `WE_ADDER` consists of 3 atoms (absent signals are not represented in the tuples):

$$\begin{aligned} a_1^{i1} &= (SYN1 = f, I1 = i1, O1 = o1) \\ a_2^{i2} &= (SYN2 = f, I2 = i2, O2 = o2) \\ a_3^{i1,i2} &= (SYN1 = t, I1 = i1, O1 = o1, \\ &\quad SYN2 = t, I2 = i2, O2 = o3, C = c) \end{aligned}$$

Here, t and f stand for *true* and *false*, $o1$ and c are the values produced by function `compute1` from $i1$, $o2 =$ `compute2`$(i2, \textit{false})$, and $o3 =$ `compute2`$(i2, c)$. Note here that a_1^{i1} and a_2^{i2} are independent for all $i1$, $i2$, but they are both contradictory with $a_3^{i1,i2}$.

4. MULTI-THREADED CODE GENERATION

The simplest possible multi-threaded implementation of a synchronous program has one thread for each atomic reaction of the program, as pictured in Fig. 3. The thread `thread_a` corresponding to atom a cyclically performs the following sequence of operations:

1. Wait for the input configuration where the signals used as input by a have all arrived with the required values.

2. Place a lock on these inputs. These locks inform other threads waiting for the same values on the same signals that the current values will be consumed by `thread_a`. The other threads will have to wait for new values.

```
void thread_a() {/* thread pseudocode */
  for(;;){
    await_inputs(I_1 = a(I_1),...,I_n = a(I_n));
    for(i=1;i<=n;i++)lock(I_i);
    compute(); send_outputs();
    for(i=1;i<=n;i++)
      consume_and_unlock(I_i);}}
void main(){/* driver pseudocode */
  forall(a ∈ Atoms(P))
    start_thread{thread_a();}}
```

Fig. 3. Simple generic implementation of a weakly endochronous program. In `thread_a` the signals $\mathtt{I}_1$,...,$\mathtt{I}_n$ are the inputs used by a.

3. Perform the actual computation and send the outputs.
4. Consume the current value, and then unlock each of the input signals. Consuming means removing the old value, which allows new ones to arrive.

The late consumption of the inputs, as opposed to creating a local copy in step 2 and allowing new inputs to arrive, ensures that fast computing atoms cannot overtake slow atoms, so that the outputs are well ordered and there is no conflict in writing the outputs. The lock mechanism is part of the late consumption mechanism, forbidding a thread to use input values that are currently used and will be consumed by some other thread. Together, late consumption and locks ensure the atomicity of atom computation.

This implementation has two advantages (generality and simplicity), and three drawbacks: The possible explosion in the number of threads, the inefficiency of the late consumption and lock mechanism, and distribution problems related to the fact that threads are built on a semantic, rather than resource locality arguments. In the following, we provide solutions to overcome these drawbacks.

4.1. Reducing the number of threads

We define in this paper techniques for the compact representation of atom sets, based on symbolic representation and hierarchization. Given the simple mapping between atoms and threads, the compact representation of atoms is directly mapped into compact executable code. This approach is also beneficial on the analysis side, where compact representations of the atom sets support better (more efficient) analysis algorithms for constructing the atom set. In addition, reducing the number of threads leads also to more efficient late consumption and lock mechanisms.

4.1.1. Symbolic atom representation

Recall that atoms are reactions, which are valuations of the various signals. For instance, the atom definition a_1^{i1} of the previous section represents as many atoms as there are values in the domain of `I1`. When signals range over large or infinite domains, like the numeric types, generating one thread per atom is impossible. Even for small finite types, such as booleans, this translation is inefficient, as it artificially creates concurrency between threads representing exclusive computations, and thus raises the cost of synchronization.

The basic solution to this problem is provided by the very notation used to represent these atoms, which lets the value of input signals range over entire domains, like in the definitions of a_1^{i1}, where $i1$ ranges over all integer pairs that a single-precision adder can receive.

But replacing the inputs with variables ranging over domains also means that the outputs must be replaced with data expressions, which leads to complex (possibly undecidable) analysis.

For instance, in a_1^{i1} the value of `O1` is computed as one of the outputs of `compute1`$(i1, \&o1, \&c)$. In our example, this poses no problem. However, when the computed value is later used to make decisions (tests) that affect the synchronization (and therefore the number and form of the atoms), the formalism used to represent the atom sets must allow variables to range over inverse images of Boolean values through compositions of such data expressions. This also poses computability/complexity problems at analysis time, as the analysis algorithms must compute these inverse images and then test their disjointness.

For these reasons, an exact analysis of weak endochrony is not feasible in the general case (any data type/function).

In its current status, our prototype tool can analyze finite data types (`event`, `bool`, and enumerated types). All large and infinite types (including numeric ones) and all data functions on them are treated as uninterpreted types and function symbols. The only functions that accept the symbolic representation presented above are identity and Boolean negation. The resulting representation of an atom set is a set of symbolic atoms of the form:

$$< (S_1 \in D_1, \ldots, S_n \in D_n), Eq, Neg > \qquad (1)$$

where a set of possible non-absent values D_i is specified for each signal S_i, and Eq and Neg are sets of pairs (S_i, S_j). If $(S_i, S_j) \in Eq$, then their values are equal. If $(S_i, S_j) \in Neg$, then $S_i = not S_j$ (both signals must be Boolean).

The expressivity is satisfactory, as it allows the representation and analysis of all combinational (non-sequential) clock relations used in the compilation of synchronous languages, like the clock tree of Signal/Polychrony [17] or the selection tree of Esterel [8]. The abstraction is conservative: Some weakly endochronous programs will be rejected for code generation, but all accepted programs are weakly endochronous.

Starting from this symbolic representation, code generation is done using a simple variant of the generic technique defined above. For each symbolic atom a of the form given in Equation 1 one thread `thread_a` is produced. The only difference between this thread and the one of Fig. 3 is that the `await_inputs` statement is replaced with a more complex one capable of detecting input configurations belonging to a symbolic atom:

`await_inputs(`$\mathtt{I}_1 \in \mathtt{D}_1, \ldots, \mathtt{I}_n \in \mathtt{D}_n, Eq, Neg$`) ;`

4.1.2. Hierarchic atom set representation

The generic code generation technique defined above produces code containing no conditional (`if`) statement. Thus, even when two atoms correspond to the branches of a test we still have to use two threads to encode them, with the accompanying synchronization overhead. For instance, this is the case for atoms a_1^{i1} and $a_3^{i1,i2}$ of `WE_ADDER`, which correspond to the branches of a test on the Boolean value of `SYN1`.

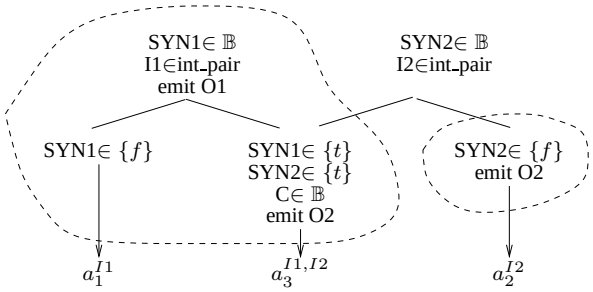


Fig. 4. Hierarchic atom set representation and partition into threads

This is clearly suboptimal. Our objective in this section is to allow the grouping of atoms into sets of mutually contradictory atoms so that each set can be encoded by one single sequential thread. By sequential thread we understand here code formed using `if` tests and the primitives already used by the generic implementation and its symbolic extension. Not all sets of mutually contradictory atoms can be transformed into a single thread. The simplest example where this is impossible is the following system having 3 Boolean inputs (A, B, and C) and the atom set $\{(A = t, B = t), (B = f, C = t), (A = f, C = f)\}$. Encoding into sequential code is impossible here because no static order of wait and test statements allows the choice between the 3 atoms.

However, when such a static order exists, we represent it using a data structure similar to the decision trees used in the compilation of Signal/Polychrony [17, 6] and Esterel [8]. Such are the trees surrounded by dashed lines in Fig. 4. In the tree rooted in $SYN1 \in \mathbb{B}$, each node contains a symbolic input configuration and a set of actions to be realized (signal emissions and/or function computations). For the tree to be a decision tree, the input configurations of two nodes having the same parent node must be contradictory, and the union of the configurations of the children nodes must cover the parent node. Each leaf of the tree corresponds to exactly one atom. To ensure coherence with the late consumption mechanism, all the atoms whose input configuration is smaller than the input configuration of a node n must be represented by leaves of the sub-tree rooted in n. In our example, the top node specifies reactions where `SYN1` and `I1` are available and `O1` is emitted. Its two children represent the symbolic atoms a_1^{i1} and $a_3^{i1,i2}$ which correspond to the branches of a test on `SYN1`.

Generating code from such a tree consists in transforming the hierarchy of the tree into a hierarchy of data tests. When an input signal is only used by one thread, no lock is needed for it, because the cyclic execution of the thread ensures the needed exclusiveness property. The code generated for the left thread of Fig. 4, provided below, needs no lock on signals `SYN1` and `I1`.

```
void thread_1(){/* thread pseudocode */
  for(;;){
    await_inputs(SYN1,I1);
    compute1(I1_val,&c,&o1);
    send_output_O1(o1);
    if(SYN1_val){
      await_inputs(SYN2∈ {t},I2);
      lock(SYN2,I2);
      send_output_O2(compute2(I2_val,c));
      consume_and_unlock(SYN2,I2);}}}
```

The data structure defined above not only allows the representation of decision trees associated with threads, but also the representation of full atom sets, as illustrated in Fig. 4 (the full picture). Instead of a tree, the representation is a forest with possibly multiple toplevel nodes. To generate code from such a forest, we first determine a subset of nodes of the forest, such as the sub-trees rooted in these nodes, which define a partition of the leaves (symbolic atoms). In our example, two nodes/trees are necessary. Then, code generation is performed separately for each tree.

4.2. Distributed code generation

In the previous sections we focused on generating multi-threaded code, but without considering distribution problems arising from resource locality. Assume, for instance, that the code of the `WE_ADDER` example must be distributed over two processors, one responsible for receiving `I1`, emitting `O1`, and performing the computation of `ADD1`, and the other responsible for receiving `I2`, emitting `O2`, and performing the computation of `ADD2`. Then, the code produced by the technique of the previous section is not good, because `thread_1` involves signals and computations of both `ADD1` and `ADD2`.

As each thread corresponds to an endochronous decoding process, the first idea would be to rely on previous work by Girault and Caspi [9], applied separately on each thread. However, this approach does not work due to the particular semantics of the `await_inputs` statement, which atomically waits for multiple input configurations instead of the simple arrival of individual signals, and considers the inputs read only when the configuration is complete. In single-processor implementations this primitive can easily be implemented by input polling. However, in a multi-processor framework where each input arrives to exactly one processor, direct polling is impossible. Some data must be explicitly exchanged between processors to allow computation to advance, which amounts to replacing the polling-based protocol with one based on classical single-input blocking `wait` statements which do not involve domain tests.

This transformation amounts to providing an implementation of the atomic `await_inputs` over blocking `wait`s. This transformation is necessarily global, because a signal can be argument to several `await_inputs` statements. Since one blocking wait is generated for each `await_inputs` statement, the input must be broadcast to all active wait points. Once this expansion has been realized, we rely on the previous results of Girault and Caspi.

5. EXPERIMENTAL RESULTS

We have implemented our symbolic data structures and algorithms in a prototype version, linked to the Signal/Polychrony-SME design environment as a new back-end tool.

We ran our tool over 3 representative Signal/Polychrony examples of average size (an oscilloscope model, a mouse driver, and an equation solving algorithm).

Example	Signals (In/Out/Loc)	Statements	Running time	Threads
oscillo	9/9/45	78	17.7s	89
mouseDrv	4/9/35	61	3.4s	34
eqSolve	8/5/34	53	10.7s	48

The running times of our tool (of the order of seconds on an Intel Core2 Duo CPU running at 2.8GHz) are encouraging, given that we mostly focused on the data structures, so that much remains to be done in the algorithm optimization field. Memory was not an issue, even for larger examples.

The number of threads corresponds to the number of generators in the symbolic representation of Section 4.1.1 which uses signal domains, but no interpreted function, and no hierarchic compaction. The figures are still large. However, preliminary tests on the hierarchic representation of generator sets drastically reduce these figures (2 threads for eqSolve, 5 for oscillo). Moreover, full expansion is not without interest, as for small and medium size programs it may allow better optimization of the code for each thread.

6. CONCLUSION

We have defined a full design flow starting from high-level DSLs and going all the way to the generation of deterministic concurrent (multi-threaded) executable code for simulation or (possibly distributed) implementation. Based on the theory of weakly endochronous systems, our flow allows the automatic detection of potential parallelism for use in the generation of concurrent code. From a tool perspective, the flow uses the front-end of the Signal/Polychrony-SME environment, and our own prototype back-end tool. We provide initial benchmarks for the back-end.

We currently focus on the extension of the data structures and analysis algorithms to use hierarchical generator representations. One possible lead in this direction is given by previous work of Ouy, Talpin *et al.*[18], which produces decision forests for exploiting weakly endochronous programs (but only considering top-level concurrency).

7. REFERENCES

[1] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, "The synchronous languages 12 years later," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 64–83, Jan. 2003.

[2] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann, "Polychrony for system design," *Journal for Circuits, Systems and Computers*, April 2003.

[3] H. Yu, Y. Ma, Y. Glouche, J.-P. Talpin, L. Besnard, T. Gautier, P. LeGuernic, A. Toorn, and O. Laurent, "System-level co-simulation of integrated avionics using polychrony," in *Proceedings SAC'11*, TaiChung, Taiwan, March 2011.

[4] T. Grandpierre and Y. Sorel, "From algorithm and architecture specification to automatic generation of distributed real-time executives: a seamless flow of graphs transformations," in *Proceedings MEMOCODE*, Mont Saint-Michel, France, June 2003.

[5] D. Potop-Butucaru, B. Caillaud, and A. Benveniste, "Concurrency in synchronous systems," *Formal Methods in System Design*, vol. 28, no. 2, pp. 111–130, March 2006.

[6] A. Benveniste, B. Caillaud, and P. Le Guernic, "Compositionality in dataflow synchronous languages: Specification and distributed code generation," *Information and Computation*, vol. 163, pp. 125 – 171, 2000.

[7] D. Potop-Butucaru, R. de Simone, Y. Sorel, and J.-P. Talpin, "From concurrent multiclock programs to deterministic asynchronous implementations," in *Proceedings ACSD*, Augsburg, Germany, July 2009.

[8] D. Potop-Butucaru, S. Edwards, and G. Berry, *Compiling Esterel*, Springer, 2007.

[9] P. Caspi, A. Girault, and D. Pilaud, "Automatic distribution of reactive systems for asynchronous networks of processors," *IEEE Transactions on Software Engineering*, vol. 25, no. 3, pp. 416–427, May/June 1999.

[10] G. Kahn, "The semantics of a simple language for parallel programming," in *Information Processing '74*, J.L. Rosenfeld, Ed. 1974, pp. 471–475, North Holland.

[11] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogenous systems," *International Journal in Computer Simulation*, vol. 4, no. 2, 1994.

[12] R. Milner, *Communication and Concurrency*, Prentice Hall, 1989.

[13] L. Carloni, K. McMillan, and A. Sangiovanni-Vincentelli, "Theory of latency-insensitive design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 9, pp. 18, Sep 2001.

[14] J. Boucaron, R. de Simone, and J.-V. Millo, "Latency-insensitive design and central repetitive scheduling," in *Proceedings MEMOCODE'06*, Napa, CA, USA, 2006.

[15] J. Cortadella, A. Kondratyev, L. Lavagno, and C. Sotiriou, "Desynchronization: Synthesis of asynchronous circuits from synchronous specifications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 10, pp. 1904–1921, October 2006.

[16] A. Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, and M. Pouzet, "N-synchronous kahn networks: a relaxed model of synchrony for real-time systems," in *Proceedings POPL'06*. 2006, pp. 180–193, ACM Press.

[17] P. Amagbégnon, L. Besnard, and P. Le Guernic, "Implementation of the data-flow synchronous language signal," in *Proceedings PLDI'95*, La Jolla, CA, USA, June 1995.

[18] J.-P. Talpin, J. Ouy, T. Gautier, L. Besnard, and P. Le Guernic, "Compositional design of isochronous systems," *Science of Computer Programming*, 2010.