Classification of Dataflow Actors with Satisfiability and Abstract Interpretation

Matthieu Wipliez, Mickaël Raulet

Classification of Dataflow Actors with Satisfiability and Abstract Interpretation

Matthieu Wipliez, Mickaël Raulet
IETR/INSA
UMR CNRS 6164
F-35043 Rennes, France
Email: mwipliez@insa-rennes.fr
Email: mraulet@insa-rennes.fr

ABSTRACT

Dataflow programming has been used to describe signal processing applications for many years, traditionally with cyclo-static dataflow (CSDF) or synchronous dataflow (SDF) models that restrict expressive power in favor of compile-time analysis and predictability. More recently, dynamic dataflow is being used for the description of multimedia video standards as promoted by the RVC standard (ISO/IEC 23001:4). Dynamic dataflow is not restricted with respect to expressive power, but it does require runtime scheduling in the general case, which may be costly to perform on software. We presented in a previous paper a method to automatically classify actors of a dynamic dataflow program within more restrictive dataflow models when possible, along with a method to transform the actors classified as static to improve execution speed by reducing the number of FIFO accesses (Wipliez and Raulet (2010)). This paper presents an extension of our classification method using satisfiability solving, and details the precise semantics used for the abstract interpretation of actors. Our extended classification is able to classify more actors than what could previously be achieved.

Keywords: dataflow program, classification, satisfiability, abstract interpretation.

INTRODUCTION

The arrival of multi-core in the desktop computing market has renewed the interest in multi-processor programming. There are many different techniques to write multiprocessing programs, depending on memory architecture (shared or distributed), processor architecture (uniprocessor or multiprocessor), number of cores (single core, multi-core, many-core), etc. Most of these techniques constrain program design in a way that makes it difficult to use a different technique, should the program be ported to a different architecture than initially planned. Dataflow programming is a portable platform-agnostic alternative that allows an algorithm to be described so that parallelism is made explicit.

A dataflow description is a directed graph where vertices (or actors) process data and edges carry data, with the requirement that vertices cannot share data. Actors can only communicate with other actors through ports connected to edges. The semantics of a dataflow program are defined by a Model of Computation (MoC) that dictates conditions for existence of a valid schedule, bounded memory consumption, proof of termination, and other properties. MoCs go

from Synchronous Dataflow (SDF) with total compile-time predictability with respect to scheduling, memory consumption, termination, to dynamic dataflow where those properties are not predictable in the general case, with increasing expressiveness, for instance see (E. Lee and Messerschmitt (1987); Bilsen, Engels, Lauwereins, and Peperstraete (1996); Bhattacharya and Bhattacharyya (2001); Buck and Lee (1993); Buck (1994); E. A. Lee and Parks (1995)).

The Reconfigurable Video Coding (RVC) standard defines existing MPEG video standards as dynamic dataflow programs in which actors are written in a language called RVC-CAL (Mattavelli, Amer, and Raulet (2010)). These dataflow programs can be automatically translated and ported to a wide range of platforms and languages, from hardware to multi-core software (Mattavelli et al. (2010)). Contrary to SDF and a few other MoCs, RVC-CAL has a much greater expressive power, but this also means that runtime scheduling is mandatory in the general case, which impacts execution speed.

Fortunately, most signal processing applications are far from being entirely dynamic, and parts with static behavior need not be dynamically scheduled. The problem is to detect actors that behave statically or quasi-statically, since dynamic dataflow has an expressive power equivalent to a Turing machine (Buck and Lee (1993)), which means it is not possible to prove the termination of a dynamic dataflow program in general. We presented in a previous paper a method to automatically classify actors of a dynamic dataflow program within more restrictive dataflow models when possible, along with a method to transform the blocks classified as static to improve execution speed by reducing the number of FIFO accesses (Wipliez and Raulet (2010)).

This paper makes the following contributions:

- We describe an Intermediate Representation (IR) of dataflow actors that is more suitable for classification than the Abstract Syntax Tree (AST) of actors, and how to translate the AST of an actor to its IR.
- We describe the precise semantics behind the abstract interpretation of the IR of an actor, which is the basis for our classification method that is able to discover facts about an actor without the need for actual data. We show an example of how abstract interpretation can be used to find cyclo-static behavior.
- We present the extension of our classification method that is able to determine more precisely actors that present a time-dependent behavior. Time-dependent behavior can be used for low-level optimizations, but it makes the actor behave in a non-deterministic way (it is possible, however, for an application with some nondeterministic actors to behave in a deterministic way). Because the RVC standard specifies the behavior of video standards with RVC-CAL dataflow actors, and the specification is expected to be deterministic, it is vital to be able to prove that a given actor is deterministic.
- We show how the IR of a dataflow actor is translated to the standard format used for satisfiability solving called SMT-LIB. SMT (Satisfiability Modulo Theories) allows us to determine properties of the source code that are required by our classification method.

BACKGROUND

A Taxonomy of Dataflow Models of Computation

A Model of Computation (MoC) defines the behavior of a program described as a dataflow graph. A dataflow graph is a directed graph whose vertices are *actors* and edges are unidirectional FIFO channels with unbounded capacity, connected between ports of actors. Dataflow graphs respect the semantics of Dataflow Process Networks (DPNs) (E. A. Lee and Parks (1995)), which are related to Kahn Process Networks (KPNs) (Kahn (1974)) in the following ways:

- Those models contain blocks (processes in a KPN, actors in a DPN) that communicate with each other through unidirectional, unlimited FIFO channels.
- Writing to a FIFO is *non-blocking*, i.e. a write returns immediately.
- Programs that respect one model or the other must be scheduled dynamically in the general case (E. A. Lee and Parks (1995); Parks (1995); W. Haid et al. (2009)).

The main difference between the two models is that DPNs adds expressiveness to the KPN model in the form of non-determinism at the network level, without requiring the actor to be non-determinate, by allowing actors to test an input port for the absence or presence of data (E. A. Lee and Parks (1995)). Indeed, in a KPN process, reading from a FIFO is *blocking*: if a process attempts to read data from a FIFO and no data is available, it must wait. Conversely, a DPN actor will only read data from a FIFO if enough data is available, and a read returns immediately. As a consequence, an actor need not be suspended when it cannot read, which in turn means that scheduling a DPN does not require context-switching nor concurrent processes.

Dataflow MoCs are defined as subsets the more general DPN model. The taxonomy shown on Figure 1 reflects the fact that MoCs are progressively restricted from DPN towards SDF with respect to expressiveness, but at the same time they become more amenable to analysis. We first study the rules of DPN, and then present the models that can be used to model static, cyclo-static, quasi-static, and dynamic actors.

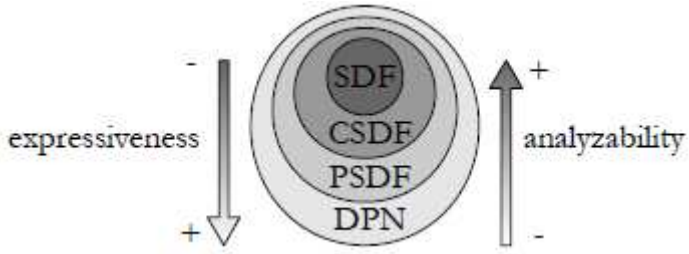

Figure 1: Dataflow Models of Computation. (c) 2010, Wipliez. Used with permission.

An actor of a DPN executes (or *fires*) when at least one of its *firing rules* is satisfied. Each firing may consume and produce tokens. An actor can have *N* firing rules, where each one represents an acceptable sequence of tokens. Additionally an actor has a firing function that takes a sequence of tokens and produces a sequence of tokens.

Synchronous Dataflow (SDF) (E. Lee and Messerschmitt (1987)) is the least expressive DPN model, but it is also the model that can be analyzed more easily. Schedulability and memory consumption of SDF graphs can be determined at compile-time, and algorithms exist that can map and schedule SDF graphs onto multi-processors in linear time with respect to the number of

vertices and processors (Pelcat et al. (2009)). Any two firing rules of an SDF actor must consume the same amount of tokens, and all firings must produce the same amount of tokens on the output ports. This definition is actually included in Lee's denotational semantics for SDF (E. A. Lee and Parks (1995)), which states that SDF actors have a single firing rule. Our definition simply allows SDF actors to have several firing rules as long as they have the same production/consumption rate, which in practice makes it easier to describe SDF actors that have data-dependent computations.

Cyclo-static Dataflow (CSDF) (Bilsen et al. (1996)) extends SDF with the notion of state while retaining the same compile-time properties concerning scheduling and memory consumption. State can be represented as an additional argument to the firing rules and firing function, in other words it is modeled as a self-loop.

Synchronous and cyclo-static dataflow allow signal processing algorithms to be modeled as graphs with fixed production/consumption rates. On the other hand, so-called "quasi-static" graphs can be used to describe data-dependent token production and consumption. Quasi-static dataflow differs from dynamic dataflow in that there are techniques that statically schedule as many operations as possible so that only data-dependent operations are scheduled at runtime (Buck and Lee (1993); Bhattacharya and Bhattacharyya (2001); Boutellier et al. (2009)). We chose to use the PSDF (Bhattacharya and Bhattacharyya (2001)) model as a target for our classification because it can be used to model static, cyclo-static and quasi-static behavior as a dataflow graph.

Dataflow Programming with RVC-CAL

In 2009, MPEG published the ISO/IEC 23001:4 standard, also known as Reconfigurable Video Coding (RVC), to describe existing and future video standards with dataflow programming. Within the RVC framework, a video decoder is described as a dataflow program, which is a hierarchical DPN whose actors are written with a Domain-Specific Language (DSL) called RVC-CAL. This language is a restricted version of CAL (Cal Actor Language), which was invented by Eker and Janneck and is described in their technical report (Eker and Janneck (2003)). Although our classification method is not limited to RVC-CAL (or CAL for that matters), this paper focuses on this language to explain our method. Indeed, the dataflow programs written by the RVC working group (and by others) form a basis of real-world applications for experimentations. Moreover, many concepts presented in the paper are more easily expressed with the means of a language in our opinion.

An RVC-CAL actor is an entity that is conceptually separated into a header and a body. The header describes the name, parameters, and port signature of the actor. For instance, the header of the actor shown on Figure 2 defines an actor called GzipParser. This actor takes two parameters, one boolean and one integer, whose values are specified at runtime, when the actor is instantiated, i.e. when it is initialized by the network that references it. The port signature of GzipParser is an input port I and two output ports HDATA and BDATA. The body of the actor may be empty, or may contain state variables declarations, functions, procedures, actions, priorities, and at most one Finite State Machine.

```
actor GzipParser(bool checkHeaderCRC, int acceptedMethods)
  int I ==> int HDATA, int BDATA :

// body

end
```

Figure 2: Header of an RVC-CAL actor.

RVC-CAL, like hardware description languages, has integers with an arbitrary bit-width. Integers can be signed (declared with the int keyword) or unsigned (declared with uint keyword). The bit width may be omitted, in which case the type has a default bit width, or it can be specified by an arbitrary expression. The other types supported by RVC-CAL are booleans (**bool**), floating-point real numbers (**float**), strings (**String**) and lists (**List**). The list type is generally used like an array type, in other words with a fixed type and a fixed size.

The language also features side-effect free expressions: an expression cannot modify variables or write to memory, as opposed to imperative languages such as C where an expression can increment a pointer or call a procedure that changes a state variable. The language of expressions includes references to variables (possibly with indexes when referring to a list), binary and unary operations, as well as calls to side-effect free functions. Expressions also borrow constructions from functional languages, like **if**/**then**/**else** conditional expressions, and list generators. A list generator is similar to the *map* function found in many functional programming languages, and is a kind of inline **for** loop that creates a list whose members are described by an expression. Figure 3 shows an example of an RVC-CAL expression that describes a list whose each element is the sum of x[i] and o[i] right-shifted by 0 or 3 depending on the value of the ROW variable, for each value of i between 0 and 7 inclusive.

```
[ (x[i] + o[i]) >> if ROW then 0 else 3 end :
    for uint(size=3) i in 0 .. 7 ]
```

Figure 3: Example of an RVC-CAL Expression

State variables can be used to define constants and to store the state of the actor they are contained in. Figure 4 shows the three different ways of declaring a state variable. The first variable called MAGIC_NUMBER is a 16-bit unsigned integer constant whose value is the number that identifies a GZIP stream (RFC 1951, IETF (1996)). The bits variable is a 16-bit unsigned integer variable without an initial value. The need_bits variable is a boolean that is initialized to true (note the difference between the = used to initialize a constant and the := used to initialize a variable). The initial value of a variable is an expression.

```
uint(size=16) MAGIC_NUMBER = 0x1F8B;

// 8 bits read, possibly spanning two bytes
uint(size=16) bits;

// if we need to read more bits
bool need_bits := true;
```

Figure 4: Declaration of State Variables.

Other features of the language include side-effect free functions, and imperative procedures. Both can have parameters and local variables, but differ by their contents: a function can only contain an expression, whereas a procedure contains a sequence of imperative statements that have side-effects. The kind of statements that can be used are (1) assignment of an expression to a local variable or a state variable, possibly with indexes when the target is a list, (2) call to a procedure or a function; the result of a function call can be assigned to a local variable or a state variable, (3) execution of statements a finite number of times with a **foreach** loop that resembles the generator expression, except its body is a sequence of statements: it defines an index variable and executes the statements it contains for each value of the index within defined bounds, (4) conditional execution of statements with an **if**/**then**/**else** construct, and (5) execution of statements an unknown number of times with a **while** loop.

The only entry points of an actor are its actions; functions and procedures can only be called by an action. An action may:

- Be identified by a **tag**, which is a list of identifiers separated by colons, such as "a.b.c". It is possible to select several actions with a given tag, for instance the tag "a" refers to all actions whose tag's first identifier is "a", such as "a.x" and "a.b.c".
- Read tokens from input ports, as described by an input pattern. It is possible to read several distinct tokens from a port, or to read a sequence of tokens with a **repeat** clause.
- Have firing conditions, called guards, which depend on the values of input tokens or the current state. Guards must be satisfied for the action to execute.
- Perform computations, described in the body of the action, which is the same as the body of a procedure.
- Write tokens to output ports, as described by an output pattern.

An actor is executed (or fired) by selecting a *fireable* action and firing it. An action is fireable if it has enough tokens, and its guards (if any) are true. Action selection may be further constrained using a Finite State Machine (FSM), to select actions according to the current state, and priority inequalities, to impose a partial order among action tags. Finally, it is possible to have tagged actions and untagged actions even in the presence of an FSM, in which case untagged actions have the highest priority.

```
an.example: action C:[c], I:[x] ==> O:[L] repeat 2
guard
  not c, count > 0
do
  uint(size=8) L[2] := [x & 0xFF, (x >> 8) & 0xFF];
  count := count - 1;
end
```

Figure 5: An example of an RVC-CAL action.

Figure 5 shows an example of an action that reads one token on the "C" port, one token on the "I" port, has two guards "c = false" and "count > 0". When there is at least one token on the FIFO connected to "C", and one token on the FIFO connected to "I", and "c" is false, and "count" is greater than zero, the action can fire. When it does, it computes the list L, decrements the "count" state variable, and writes two elements of L (as indicated with the repeat clause) to the "O" output port.

INTERMEDIATE REPRESENTATION OF DATAFLOW ACTORS

This section presents an Intermediate Representation (IR) of dataflow actors that is more suitable for classification than the Abstract Syntax Tree (AST) of actors, and how to translate the AST of an actor to its IR.

Need for an Intermediate Representation (IR) of Actors

Classification is the process that analyzes the behavior of an RVC-CAL actor in terms of number of tokens it receives and sends, patterns that may govern token exchanges, and possibly acceptable token values. In the simplest case, structural information of an actor is enough to classify it, for instance the rules for an actor to be considered SDF only depend on the input and output patterns of actions. In more complicated cases, it is necessary to gather information from an actual execution of the actor.

The structural information necessary for classification is not directly expressed in the AST of an RVC-CAL actor, and the AST must be annotated with pre-computed information first. For instance, token production/consumption rates for an action must be computed from the rules of input/output patterns, which depend on the number of tokens and repeat clause, or the type of tokens and repeat clause. Likewise, priorities only express a partial order on action tags, so one must compute the topological sort of the priority graph whose vertices are actions from each set used in priorities. The FSM uses action tags too, so a transition from one state to another may in fact become several possible transitions if the tag associated with the transition refers to several actions.

In cases where the structural information is not enough, the actor needs to be interpreted so its behavior can be properly analyzed. It is possible to execute the actor by interpreting the AST directly, but this is cumbersome. For example, RVC-CAL has a generator expression, a **foreach** construct, and a **while** construct. Writing an interpreter for the RVC-CAL AST means implementing these three separately, but they can all be transformed to **while** loops. Using an Intermediate Representation (IR) makes both structural and semantic information explicit and easier to manipulate and transform.

Description of the IR of an Actor

An IR actor has the same structural elements as the original actor with the notable exception of priorities. Contrary to the original actor, in which the order of actions may be under-specified, when priorities are absent or only specify a partial order, the IR of an actor contains a total order of actions. While not always the case, having only a partial order of actions can lead to non-deterministic behavior of the actor itself, because the order in which the scheduler will test the schedulability of actions is not fully determined.

Although this non-determinism was a design choice (as specified in the CAL Language Report (CLR)), it complicates analysis while being at most marginally useful. Further, non-determinism in the choice of actions is not compatible with the sequential ordering of firing rules specified in the DPN model. We also need to have reproducible and similar results on multiple executions of an actor. Finally, the CLR specifies that any resolution of the non-determinism is acceptable. All this explains why priorities are absent from the IR.

The Finite State Machine (FSM) of an RVC-CAL actor is transformed in our IR to an FSM in a form that is easier to manipulate and to generate code from while keeping the same information (initial state and list of transitions). The FSM in the IR contains a list of states to facilitate code generation. Rather than having several transitions departing from a single state, transitions are grouped by starting state.

The IR of an action reflects the semantic difference between its scheduling information (input patterns, output patterns, guards) and its body (local variable declarations, statements, expressions computed in the output pattern). The IR of an action is thus made:

- Tag of the action
- Input pattern, output pattern, peek pattern
- Scheduling procedure
- Body procedure

The scheduling procedure contains the IR of the guards, and the body procedure contains the IR of the body of the action. A pattern is an association between a port, the number of tokens read/written on that port by the action, and the variable in which these tokens are to be stored. The peek pattern is used by the scheduling procedure, because it represents the number of tokens that are peeked from the port (i.e. read but not removed from the FIFO connected to the port). Input/output patterns are used by the body procedure. The IR inside procedures is expressed with a simple language with statements that have side-effects and expressions that are side-effect free, and is represented as a Control Flow Graph.

```
a.b: action S:[s], I:[x] repeat 3 ==> O:[s, x[0] * x[1]]
guard
  s > 0
end
```

```
Tag: [a, b]
Peek pattern: [S=1]
Input pattern: [S=1, I=3]
Output pattern: [O=2]
Scheduler:
  function isSchedulable_a_b() --> bool :
    int s = S[0]; // load from peek pattern
    s > 0         // translation of guard
  end

Body:
  procedure a_b() begin
    int x_0 = I[0]; int x_1 = I[1]; int x_2 = I[2];
    O[0] = s; O[1] = x_0 * x_1;
  end
```

Figure 6: Example of the IR of an Action.

Figure 6 shows a simple RVC-CAL action and the corresponding IR, with scheduler and body procedures written in a pseudo-language. The tag is a list of identifiers, here "a" and "b". The peek pattern is a subset of the input pattern that contains only the number of tokens peeked on the S port because the guards only reference this port. Note that the scheduler function and the body procedure do not read/write from/to FIFOs, and only represent the semantics of the guards and action body respectively: reads/writes of FIFOs are performed solely based on the patterns.

Creation of the IR of an RVC-CAL Actor

This section presents an overview of the translation from RVC-CAL to the IR of an actor, as shown on Figure 7: Transformation from RVC-CAL to IR. A more complete description is available in (Wipliez (2010)).

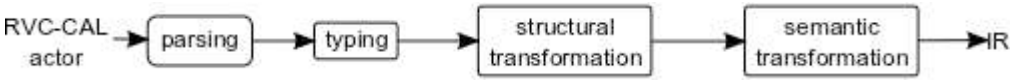

Figure 7: Transformation from RVC-CAL to IR

The first step parses the RVC-CAL to an Abstract Syntax Tree (AST), which will be manipulated in the subsequent stages. This AST is then fully typed, which means that the type of all expressions is computed, before checking that the code does not contain type errors. The next stage creates the structures of the IR of the actor: ports, state variables, FSM, and creates a sequence of actions sorted by descending priority. At this stage, the actions only have the tag and input/output/peek patterns filled in. Finally, the semantic transformation transforms expressions in functions, and expressions and statements in procedures and actions to IR expressions and statements.

ABSTRACT INTERPRETATION OF ACTORS

This section describes our abstract interpretation of an actor, and gives a complete example of an actor that is classified as CSDF.

Definition of the Abstract Interpretation of an Actor

Classifying an actor within a MoC is based on checking that a certain number of MoC-dependent rules hold true for any execution of this actor. Some of these rules are verified solely from the structural information of the actor, for instance the rules for a static actor only depends on the input and output patterns of actions. In more complicated cases, we need to be able to obtain information from an actual execution. The actor must be executed so that the information obtained is valid for *any* execution of the actor, whatever its environment (the values of the tokens and the manner in which they are available). As a consequence it is not possible to simply execute the actor with a particular environment supplied by the programmer. To circumvent this problem we use *abstract interpretation* (Cousot and Cousot (1977)).

Abstract interpretation evaluates the computations performed by a program in an abstract universe of objects rather than on concrete objects. Our abstract interpretation of an actor has the following properties:

- The set of values that can be assigned to a variable is

$$Values = \mathbb{Z} \cup \{\texttt{true}, \texttt{false}\} \cup \{\bot\}$$

The value $\bot$ is used for variables whose value is unknown, e.g. for uninitialized variables.

- The environment is defined as an association of variables and their values:

$$Env : Idents \rightarrow Values$$

Env initially contains the state variables of the actor associated with their initial value if they have one, otherwise with $\bot$.

- When the interpreter enters an action, the environment is augmented with bindings between the name of the tokens in the input pattern and $\bot$. In other words, a token read has an unknown value by default.

The abstract interpreter interprets an actor by firing it repeatedly until either one of the conditions is met:

1. The interpreter is told to stop because analysis is complete as determined by the classification algorithm.
2. The interpreter cannot compute if an action may be fired because this information depends on a variable whose value is $\bot$.

To fire the actor, the interpreter starts by selecting one fireable action, which is an action that meets the criteria defined in RVC-CAL (see Background section). As far as the quantity of tokens is concerned, the abstract interpretation models infinite FIFOs, which means an action always has enough tokens to fire. Other differences between concrete interpretation of an actor, and its abstract interpretation include the following. Any expression that references a variable *v* where *Env(v)* = $\bot$ has the value $\bot$. Conditional statements and loops that test an expression whose value is $\bot$ are not executed. However, guards evaluated as $\bot$ cause the abstract interpreter to stop as per condition 2.

Example of Abstract Interpretation

We show in this section an example of abstract interpretation on an actor called `Algo_Interpolation_halfpel`, which is a low-level description of half-pixel interpolation. This actor has an FSM presented on Figure 8.

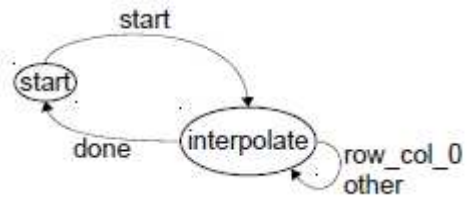

Figure 8: The Finite State Machine of Algo Interpolation halfpel.

Figure 9 shows the variables of the actor, two of which, `x` and `y`, act as loop counters, and other variables are used by computations in the actor.

```
int(size=5) x;
int(size=5) y;

int(size=3) flags;
int(size=2) round;

int(size=9) d0;
...
int(size=9) d9;
```

Figure 9: Variables of Algo Interpolation halfpel.

The first action that can be fired is the **start** action. This action assigns zero to `x` and `y`, and initializes the values of `flags` and `round`, as shown on Figure 10. After the action **start** is fired, the actor is in the *interpolate* state. In this state, any of the actions **done**, **row_col_0**, **other** (Figure 11) can be executed until the actor goes back to the *start* state. The actions are tested for schedulability in this very order as constrained by the priority statement of the actor.

```
start: action halfpel:[ f ] ==>
do
        x := 0;
        y := 0;
        flags := f >> 1;
        round := f & 1;
end
```

Figure 10: Action start of Algo Interpolation halfpel.

The **done** action contains the loop termination condition and after it is fired the actor is back in the *start* state. The two other actions compute data and increment the `x` and `y` loop indexes. Figure 12 contains the listing of the `loop_body` procedure.

```
done: action ==>
guard
        y = 9
end

row_col_0: action RD:[ d ] ==>
guard
        x = 0 or y = 0
do
        loop_body();
end

other: action RD:[ d ] ==> MOT:[ p ]
guard
        x != 0, y != 0, y != 9
do
        // computation of p omitted
        loop_body();
end
```

Figure 11: Actions fireable in the interpolate state.

```
procedure loop_body(int(size=9) d)
begin
        // dn := dn-1; d0 := d;
        x := x + 1;
        if x >= 9 then
                x := 0;
                y := y + 1;
        end
end
```

Figure 12: loop_body procedure.

The abstract interpretation of the actor starts with an initial environment that contains the variables { `x`, `y`, `flags`, `round`, `d0`, ..., `d9` }, all associated with $\perp$. For the sake of brevity we will not represent variables nor tokens valued as $\perp$ in the environment. Table 1 sums up the abstract interpretation of the actor.

Table 1: Abstract interpretation of Algo_Interpolation_halfpel.

State	Action fired	Environment
start	n/a	Ø
interpolate	start	{ x = 0, y = 0 }
interpolate	row_col_0	{ x = 1, y = 0 }
…	…	…
interpolate	row_col_0	{ x = 8, y = 0 }
interpolate	row_col_0	{ x = 0, y = 1 }
interpolate	row_col_0	{ x = 1, y = 1 }
interpolate	other	{ x = 2, y = 1 }
…	…	…
interpolate	other	{ x = 8, y = 1 }
interpolate	other	{ x = 0, y = 2 }
start	done	{ x = 0, y = 9 }

Like the concrete interpretation, the abstract interpretation starts by firing the only fireable action in the initial state, the **start** action. Since the token `f` read on the *halfpel* port has by definition the value $\perp$, the variables `flags` and `round` are set respectively to $\perp \div 2$ and $\perp \bmod 2$, in other words they are both set to $\perp$. The variables `x` and `y` both take the value 0. After the action is fired, the interpreter changes the state to *interpolate*. This is shown on the table as the first row.

In the *interpolate* state, there are three possible actions that can be executed. The interpreter schedules the first one whose input patterns are satisfied and whose guard is true, in this case it is the **row_col_0** action because we consider that tokens are always available, and the condition `x = 0 or y = 0` is true. When the action fires, the abstract interpreter executes $d_n := d_{n-1}$ for n in the interval [9..1], so variables `d9` to `d1` take the value $\perp$. Then it executes `d0 := d`, which assigns `d0` the value $\perp$ because the `d` token is $\perp$ too. The subsequent assignments to `x` and `y` are executed as per concrete interpretation rules since both variables have concrete values, so `x` takes the value 1, and `y` is unchanged. The **row_col_0** action is fired as long as either `x = 0 or y = 0` is true (second part of Table 1).

When `y` becomes greater than zero, the actor has a different behavior as can be seen on the third part of Table 1. In this case, **row_col_0** is executed once, and it is followed by 8 firings of **other**. Then, **row_col_0** can be fired again, followed by 8 firings of **other**, and so on. Finally, as soon as `y` equals 9, **done** fires and takes back the actor to its initial state.

DETECTION OF TIME-DEPENDENT ACTORS

This section presents an algorithm for the detection of actors that have a time-dependent behavior. The algorithm presented is an extension of the algorithm presented in (Wipliez and Raulet (2010)).

Characteristics of Time-Dependent Behavior

Time-dependent behavior occurs when the behavior of an actor depends on the time at which tokens are available. The description of time-dependent behavior (also called time-dependency) is only possible in the Dataflow Process Network model because an actor may test the presence of data on its input ports, something which is not allowed with more restrictive models (including Kahn Process Networks). Time-dependency is characterized in RVC-CAL when a given action reads fewer tokens from input ports than a higher-priority action, and these actions have guards that are not mutually exclusive.

Figure 13 shows an excerpt of an actor that computes the length of a network packet as long as there are bytes in the payload, and then sends the length computed so far. These two actions create a time-dependent behavior because the “done” action has a lower priority than the “transmit” action, and can only fire in the absence of tokens on the input port. Time-dependent behavior is interesting to use in cases similar to this one, namely when an actor needs to react to an absence of data.

```
transmit: action Payload:[byte] ==> Packet:[byte]
do
  length := length + 1;
end

done: action ==> Length:[length]
end

priority
  transmit > done;
end
```

Figure 13: Example of Time-Dependent Behavior.

The problem of time-dependent actors is that they may be impossible to classify with our abstract interpretation. Indeed, our abstract interpretation of an actor is that the FIFOs connected to it are infinite, in other words there will never be a lack of tokens on a port. This means that the abstract interpretation cannot capture the behavior defined in lower-priority actions enabled when there are not enough tokens. As such, it may not find a stop criterion and run forever (on the example above it would only fire the "transmit" action). On the other hand, if we wanted to create an abstract interpretation that modeled time-dependent behavior, it would have to explore the space of all possible sequences of tokens, whose size is exponential: there are N! different orderings for a sequence of N tokens. Therefore our classification method must detect and discard time-dependent actors.

Detection of Time-Dependent Behavior

An actor with time-dependent behavior can be automatically identified by an algorithm based on the following considerations. Time-dependent behavior occurs as soon as the choice of which action to fire depends on the time at which tokens arrive on input ports. If the actor has a Finite State Machine, the actions that can be fired at a given time are only those referenced by the current state, in addition to anonymous actions; if the actor does not have an FSM, the actions that are tested for fireability are all the actions of the actor. Time-dependent behavior is characterized by the fact that a fireable action reads fewer tokens from input ports than a higher-priority fireable action, and these actions are not mutually exclusive.

The algorithm starts by building all the lists of actions that can be fired in each state of the actor. Each list is sorted by descending priority, and anonymous actions, if present, are at the beginning of each list. If the actor does not have an FSM, there is only one list of actions to consider. For each non-empty list, Algorithm 1 is executed.

Algorithm 1: Returns true if a list of actions exhibits time-dependent behavior

```
higherPriorityActions := [ a1 ];
higherPriorityPattern := pattern of a1;
for each action ai (I >= 2) do
        if (pattern of ai is not a superset of higherPriorityPattern) then
                for each action aj in higherPriorityActions do
                        if (pattern of ai is not a superset of pattern of aj) then
                                if guards of ai and aj are compatible, return true
                        end
        end

        add ai to the higherPriorityActions list
        include pattern into higherPriorityPattern
end

return false
```

The algorithm maintains a list of higher-priority actions, and the associated higher-priority pattern, which at a given iteration contains the union of the patterns of all higher-priority actions visited so far. An action is considered to potentially induce time-dependent behavior if its input pattern is not a superset or equal to the higher-priority pattern. At this point, the algorithm checks that the guard of the action being examined is mutually exclusive with the guards of all higher-priority actions that have a pattern that is a subset of its pattern. This is handled by translating the guards to SMT-LIB as described in the next section, and using a solver. If the solver concludes that all guards are not mutually exclusive, it means the actor has a time-dependent behavior, otherwise the algorithm continues and examines the next actions.

This algorithm is an extension of the algorithm we presented in (Wipliez and Raulet (2010)). The main difference is that in our previous work we only considered the immediately preceding pattern, which as it turns out is not sufficient in some cases. Additionally, we previously checked the mutual exclusion of the guards of the current action and the guards of all higher-priority actions indiscriminately. However, this is too strict, because the lower-priority action only needs to have guards that are mutually exclusive with higher-priority actions that have a pattern that is a superset of its pattern to guarantee time-independent behavior.

CODE ANALYSIS OF AN ACTOR WITH SMT

This section explains the needs of our classification method in terms of code analysis and why SMT (Satisfiability Modulo Theories) fits these needs. We then give an overview of SMT and the SMT-LIB standard before showing how the IR of an actor is translated to SMT-LIB.

Code Analysis for Classification of Actors

Our classification method requires code analysis in two cases. First, to prove that an actor does not have a time-dependent behavior, the classifier must check if guards of actions are mutually exclusive. The second case is when trying to classify an actor as quasi-static. A quasi-static actor has a given set of N *configurations*. A configuration is a non-empty sequence of actions that behave according to the SDF or CSDF model. When the actor executes, one configuration is chosen with an action whose guards depend on the values of tokens read from *control ports*. The classifier needs to find the values of control tokens that allow each configuration i in [1..N] to be executed.

In our previous work, we successfully used constraint solving, more specifically Integer Linear Programming (ILP), to perform code analysis on a given number of actors (Wipliez and Raulet (2010)). However, ILP has a number of limitations that prevents it to be used in certain cases. For instance, bitwise operations cannot be modeled with ILP because they are not linear operations: let "&" be the "bitwise and" operator, (1 + 7) & 3 = 8 & 3 = 0 is different from (1 & 3) + (7 & 3) = 1 + 3 = 4. Additionally, ILP does not support constructs such as arrays or functions, which prevents it to be used with certain actors. Finally, the solver cannot handle mutually exclusive constraints because it tries to find the best value that satisfies them, which in this case does not exist, and leads to timeouts.

Satisfiability Modulo Theories (SMT)

Satisfiability Modulo Theories (SMT) is an extension of SAT, the Boolean satisfiability problem, which given a Boolean formula aims to find the values of this formula's variables so that it is true (*satisfiable*), or prove that it is always false (*unsatisfiable*) (Barrett et al. (2009)). SMT extends SAT with higher-level constructs expressed with first-order logic predicates, i.e. predicates that operate on variables, but not on other predicates. SMT supports formulas with integers (and associated arithmetic operations), bit-vectors (and associated bitwise operations), arrays, and functions.

The SMT-LIB initiative started in 2003 with the aim to create a common format for an SMT Library, including a standard language and a standard set of *theories* and *logics* (Ranise and Tinelli (2003)). A theory defines a set of types and functions over these types. Theories defined by SMT-LIB are ArraysEx (functional arrays with extensionality), Fixed_Size_BitVectors (bit vectors with arbitrary size), Core (core theory, defining the basic Boolean operators), Ints (integer numbers), reals (real numbers), reals_Ints (real and integer numbers). A logic defines the semantics of formulas that are supported based on existing theories. For instance, the QF_BV logic allows closed quantifier-free (QF) formulas based on the Core and Fixed_Size_BitVectors theories.

The SMT-LIB standard version 2.0 (SMT-LIB v2) defines a language based on Lisp's s-expressions (Steele (1990)) with which theories, logics, and formulas can be expressed. The language includes *commands* that allow communication with an SMT-LIB v2 compliant solver, which can be invoked interactively or with a *script* that contains a series of commands: definition

of the logic to use for solving, definition of functions and assertions, check satisfiability, retrieve values of variables, etc.

Translation from IR to SMT-LIB

This section describes the translation from parts of the IR of an actor to an SMT-LIB v2 script. In the context of classification, the only parts of the IR that require analysis are the guards of actions, either in the case of analysis of time-dependent behavior or to determine values of control tokens.

An SMT-LIB v2 script starts with the logic the solver will use for solving. In our case, we use the QF_AUFBV logic, which means quantifier-free formulas with arrays, bit-vectors, and functions. This is the minimal logic that can handle the formulas we generate from the IR: many variables, as well as tokens, are arrays, and guards may use bitwise operations, and call functions. The script follows by variable and function definitions: one SMT function is defined for the IR of each scheduler function that requires analysis, as shown on Figure 14. State variables and functions are defined on-demand before the function being translated when they are referenced. All operands of SMT bit-vector operations must have the same type, and at the moment we do not perform type coercion; instead, we use 32-bit vectors for all operations.

Depending on the kind of analysis that is needed, the script ends with the following commands:

- Asserts all scheduler functions must be simultaneously satisfiable, and check satisfiability.
- Asserts scheduler functions 1..N-1 must not be satisfiable, and scheduler function N must be satisfiable; check satisfiability, and retrieve values of control tokens.

Figure 14 presents the translation of an action found in an actor to SMT-LIB. The BTYPE port becomes an array from which tokens will be loaded. Because tokens may be loaded from BTYPE by several actions, we allocate one local array per port read by each action. The “assert” on “cmd_newVop_BTYPE” adds a constraint that the first element of the array equals the first element of the BTYPE array. The script then declares the SMT-LIB equivalent of the NEWVOP variable: it becomes a function with arity (number of arguments) 0, and is constrained to be equal to the 32-bit vector that represents the integer 2048. After that, the “isSchedulable_cmd_newVop” is defined, which is the scheduler function of the “cmd.newVop” action. This function is asserted to be satisfiable, and finally we see the commands that prompt the solver to check satisfiability and retrieve the value of the first element of the BTYPE array.

```
int NEWVOP = 2048;

cmd.newVop: action BTYPE:[ cmd ] ==>
guard
  (cmd & NEWVOP) != 0
end
```

```
(set-logic QF_AUFBV)
(declare-fun BTYPE () (Array (_ BitVec 32) (_ BitVec 32)))

(declare-fun cmd_newVop_BTYPE ()
  (Array (_ BitVec 32) (_ BitVec 32)))

(assert (=
  (select cmd_newVop_BTYPE (_ bv0 32)) (select BTYPE (_ bv0 32))))

(declare-fun NEWVOP () (_ BitVec 32))
(assert (= NEWVOP (_ bv2048 32)))

(define-fun isSchedulable_cmd_newVop () Bool
  (let (cmd (select cmd_newVop_BTYPE (_ bv0 32)))
    (let (local_NEWVOP NEWVOP)
      (let (result
        (not (= (bvand cmd local_NEWVOP) (_ bv0 32)))) result))))

(assert isSchedulable_cmd_newVop)
(check-sat)
(get-value ((select BTYPE (_ bv0 32)) ))
```

Figure 14: Translation of an Action to SMT-LIB.

RESULTS AND DISCUSSION

We have implemented the classification method in the compilation infrastructure supported by the Open RVC-CAL Compiler (Orcc) available at http://orcc.sf.net. Orcc compiles RVC-CAL actors to the Intermediate Representation (IR) described in this paper. The IR of actors can then be analyzed and transformed to source code in any of the target languages supported by back-ends, such as C, C++, Java, LLVM, and VHDL.

We have tested the SMT-LIB scripts produced by our method with the only two major SMT solvers that we have found to (1) implement a sufficient part of the SMT-LIB v2 standard, and (2) be able to handle the logic we require: CVC3 (Barrett and Tinelli (2007)), and Z3 (De Moura and Bjørner (2008)).

We had previously tested our classification method on 50 actors used by two dataflow descriptions of an MPEG-4 part 2 decoder sharing some actors. Table 2 shows the classification results with actors classified as static, cyclo-static, quasi-static, dynamic, time-dependent.

Table 2: Classification Results on 50 Actors.

Number of actors	Classification
6	static
14	cyclo-static
11	quasi-static
13	dynamic
6	time-dependent

The updated results on this same set of actors are slightly different as shown on Table 3. First, two actors have been modified and are now time-dependent (but they would have been detected as such with our previous algorithm too). Thanks to improvements in the detection of time-dependent actors and code analysis with SMT, two other actors were previously *assumed* to be time-dependent, but are now *proven* to be time-dependent. Another actor was assumed to be time-dependent, when in fact classification can now prove it is *not* time-dependent. Moreover, increased precision of the abstract interpretation leads to more dynamic actors being found than previously.

Table 3: Updated Classification Results on 50 Actors.

Number of actors	Classification
6	static
12	cyclo-static
10	quasi-static
15	dynamic
7	time-dependent

Additionally, our improved method can now handle more diverse actors, and we have tested it on an RVC-CAL description of an MPEG-4 part 10 decoder (Table 4). The results are quite different from the actors we classified before: a vast majority of actors have a dynamic or time-dependent behavior. We believe there are several reasons for this. First, the application is several times more complex than an MPEG-4 part 2 decoder, and is a lot more control-oriented, which

makes it difficult to have actors that are SDF or CSDF. However, quasi-static actors could be used advantageously here, which seems not to be the case.

Table 4: Classification Results on 68 Actors of an MPEG-4 part 10 Decoder.

Number of actors	Classification
12	static
3	cyclo-static
1	quasi-static
41	dynamic
11	time-dependent

Actors of the MPEG-4 part 10 application have been written as part of the RVC standardization effort. The granularity of actors seems to have been somewhat arbitrarily fixed: the static and cyclo-static actors are generally less than 50 lines of code, and most of dynamic actors are a lot larger, frequently between 500 and 1000 lines. Actors have been written by many different persons, and as such they are written in a heterogeneous way. All this makes it unlikely to find actors that behave according to more restricted MoCs, and leads us to believe that our classification method will yield the best results on applications described with fine-grain, small actors.

RELATED WORK

Zebelein, Falk, Haubelt, and Teich (2008) present a classification algorithm for dynamic dataflow models. In their model, actors are defined as SystemC modules that receive and send data via SystemC FIFOs. Their classification method is based on the analysis of read and write patterns and FSMs of the different modules. Compared to ours, their approach is limited by the fact that they ignore any C++ code that does not contain a read or a write, and that they do not classify quasi-static actors.

A different approach that is based on an analysis of the Control-Flow Graph is presented by Ârzén, Nilsson, and Platen (2010). On one hand, their approach is capable of finding actors that are static and cyclo-static, but not those that are quasi-static. These actors represent a non-negligible proportion of actors in our test application. Moreover, no distinction is made between dynamic and time-dependent actors, and one actor that our method finds to be cyclo-static is classified by their method as time-dependent. On the other hand, Ârzén et al. show an interesting constraint-based system to find the optimal scheduling of a set of static and cyclo-static actors, as well as an actor merging system. However, no experiments are shown with respect to the expected performance increase that should result from these techniques. Finally, the report does not present an equivalent of our loop rerolling transformation, which can dramatically reduce runtime scheduling (Wipliez and Raulet (2010)).

Boutellier et al. (2009) show how to express quasi-static RVC-CAL actors as PSDF graphs and how to derive a multiprocessor schedule from these graphs. However, they do not address the issues of automated classification and transformation: Quasi-static behavior is specified with parameters defined *manually*, and they do not explain how low-level Homogeneous SDF (HSDF) graphs created from quasi-static branches can be automatically transformed to high-level PSDF graphs. As a consequence, we believe that our work can serve as a preprocessing

step for their approach by automatically classifying actors as quasi-static and transforming them to high-level PSDF graphs.

Gu et al. (2009) present a technique to recognize a set of Statically Schedulable Regions (SSRs) within a dynamic dataflow program. SSRs are sets of ports that are *statically coupled*, which essentially means that the production of an output port matches the consumption of the input port(s) it is connected to (additional criteria are developed in Gu et al. (2009)). On the one hand, SSR classification has potentially more knowledge about static behavior because it looks at connected actors rather than just inside actors. On the other hand, by considering an actor as a whole our classification can discover its behavior (cyclo-static and quasistatic) and transform it into a high-level SDF or PSDF graph that will make merging easier. Using SSRs to obtain additional information as an input to our classification algorithm is a possible direction for future work.

CONCLUSION

This paper presented a method to automatically classify dynamic dataflow actors into more restricted dataflow MoCs, based on satisfiability and abstract interpretation. The paper detailed an Intermediate Representation (IR) of dataflow actors, and described an abstract interpretation of the IR of an actor, which is the basis for our classification method. We then showed how to find actors that present a time-dependent behavior, and finally explained how the IR of a dataflow actor could be translated to the standard format used for satisfiability solving called SMT-LIB, which allows us to perform code analysis required by our classification method. The method presented in the paper is an extension of our previous work on classification. We presented results and compared them with results we previously obtained. We experimented our method on a total of 118 actors, and discussed the results.

Future work could include classification of actors at the network level, in order to find sets of actors that have a static, cyclo-static or quasi-static behavior. Another direction for future work is to extend the MoCs that can be recognized, for instance the “quasi-static” model is only a subset of Parameterized SDF, which also allows modeling of dynamic loops and complex conditionals. Classification is an important contribution towards efficient implementation of dynamic dataflow programs, and is scheduled to be used in combination with other approaches (Boutellier et al. (2011); Gorin et al. (2011)). However, no publications have demonstrated yet that a large speedup could be achieved with clustering or static scheduling of actors classified with our approach; we aim to investigate the use of such techniques as future work.

REFERENCES

Årzén, K., Nilsson, A., & Platen, C. von. (2010). *Model Compiler* [Computer software manual].

Barrett, C. and Sebastiani, R. and Seshia, S.A. and Tinelli, C. (2009). Handbook of Satisfiability.

Barrett, C. and Tinelli, C. (2007). CVC3. *Proceedings of the 19th international conference on Computer aided verification*, Springer-Verlag.

Bhattacharya, B., & Bhattacharyya, S. S. (2001). Parameterized Dataflow Modeling for DSP Systems. *IEEE Transactions on Signal Processing*, 49, 2408–2421.

Bilsen, G., Engels, M., Lauwereins, R., & Peperstraete, J. (1996). Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, 44(2), 397–408.

Boutellier, J., Lucarz, C., Lafond, S., Gomez, V., & Mattavelli, M. (2009). Quasi-static scheduling of CAL actor networks for Reconfigurable Video Coding. *Journal of Signal Processing Systems*, 1–12.

Boutellier, J., Silven, O. & Raulet, M. (2011). Scheduling of CAL actor networks based on dynamic code analysis. *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*.

Buck, J. (1994). Static scheduling and code generation from dynamic dataflow graphs with integer-valued control streams. Presented at 28th Asilomar conference on signals.

Buck, J., & Lee, E. (1993). Scheduling dynamic dataflow graphs with bounded memory using the token flow model. *IEEE International Conference on Acoustics, Speech, and Signal Processing*, 429-432.

Cousot, P., & Cousot, R. (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Sigact-Sigplan Symposium on Principles Of Programming Languages* (pp. 238–252).

De Moura, L., & Bjørner, N. (2008). Z3: An efficient SMT solver. *In Proceedings of the theory and practice of software, 14th international conference on tools and algorithms for the construction and analysis of systems* (pp. 337–340).

Eker, J. & Janneck, J.. CAL Language Report. *Technical Report ERL Technical Memo UCB/ERL M03/48*, University of California at Berkeley, December 2003.

Gorin, J., Wipliez, M., Prêteux, F. & Raulet, M. (2011). LLVM-based and scalable MPEG-RVC decoder. *Journal of Real-Time Image Processing, Springer*.

Gu, R., Janneck, J., Bhattacharyya, S., Raulet, M., Wipliez, M., & Plishker, W. (2009). Exploring the concurrency of an MPEG RVC decoder based on dataflow program analysis. *IEEE Transactions on Circuits and Systems for Video Technology*.

W. Haid, L. Schor, K. Huang, I. Bacivarov, & L. Thiele. Efficient execution of Kahn process networks on multi-processor systems using protothreads and windowed FIFOs. In Proc. *IEEE Workshop on Embedded Systems for Real-Time Multimedia* (ESTIMedia), pages 35–44, 2009.

IETF. (1996, May). RFC 1951: DEFLATE Compressed Data Format Specification version 1.3 [Computer software manual].

Kahn, G. (1974, August). The semantics of a simple language for parallel programming. In *Proceedings of IFIP'74* (p. 471-475).

Lee, E., & Messerschmitt, D. (1987). Synchronous data flow. *Proceedings of the IEEE*, 75(9), 1235–1245.

Lee, E. A., & Parks, T. M. (1995, May). Dataflow Process Networks. *Proceedings of the IEEE*, 83(5), 773–801.

Mattavelli, M., Amer, I., & Raulet, M. (2010, May). The Reconfigurable Video Coding Standard [Standards in a Nutshell]. IEEE *Signal Processing Magazine*, 27(3), 159 -167.

Nevill-Manning, C., & Witten, I. (1997). Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7(1), 67–82.

Parks, T. M. (1995). Bounded Scheduling of Process Networks. *Doctoral dissertation*, Berkeley, CA, USA.

Pelcat, M., Piat, J., Wipliez, M., Aridhi, S., & Nezan, J. (2009). An open framework for rapid prototyping of signal processing applications. *EURASIP Journal on Embedded Systems*, 2009, 3.

Ranise, S. & Tinelli, C. (2003). The SMT-LIB format: An initial proposal. *Proceedings of the 1st Workshop on Pragmatics of Decision Procedures in Automated Reasoning.*

Steele, G.L. (1990). *Common LISP: the language*.

Stitt, G., & Vahid, F. (2005). New decompilation techniques for binary-level co-processor generation. In *IEEE/ACM international conference on computer-aided design*, 2005. iccad-2005 (pp. 547–554).

Wipliez, M., *Compilation Infrastructure for Dataflow Programs*, Ph.D. dissertation, National Institute of Applied Sciences (INSA) - Rennes, 2010.

Wipliez, M., & Raulet, M. (2010). Classification and Transformation of Dynamic Dataflow Programs. In *Design and Architectures for Signal and Image Processing (DASIP)*.

Zebelein, C., Falk, J., Haubelt, C., & Teich, J. (2008). Classification of General Data Flow Actors into Known Models of Computation. Proc. *MEMOCODE*, Anaheim, CA, USA, 119–128.