

Feasible Reactivity for Synchronous Cooperative Threads

Roberto M. Amadio ¹

Université de Paris 7

Frédéric Dabrowski ²

INRIA Sophia-Antipolis

Abstract

We are concerned with programs composed of cooperative threads whose execution proceeds in synchronous rounds called *instants*. We develop static analysis methods to guarantee that each instant terminates in time polynomial in the size of the parameters of the program at the beginning of the computation.

Key words: Synchronous and cooperative programming, Resource bounds, Quasi-interpretations, Termination, Polynomial time.

1 Introduction

In [8], Boussinot and De Simone introduced the *Synchronous Language* (SL). A program in SL is a set of cooperative threads interacting through shared signals whose execution proceeds in synchronous rounds called *instants*. A fundamental hypothesis of the model is that the reaction to the absence of a signal within an instant can only happen in the following instant. Reactivity is the essential property that one should guarantee of an SL program. This means that at each instant the program fed with an input will ‘react’ producing an output.

The SL language has gradually evolved into a general purpose programming language for concurrent applications and has been implemented in various programming environments such as C, JAVA, SCHEME, and CAML. Typical applications effectively developed in these languages include event-driven controllers, data flow architectures, graphical user interfaces, simulations, web services, and multi-player games (see, *e.g.*, [13,5]).

¹ Email: amadio@cmi.univ-mrs.fr

² Email: frederic.dabrowski@inria.fr

All the extensions of the SL language mentioned above introduce data types such as integers, lists, trees. What does it mean to ensure reactivity in this context? One may consider three increasingly ambitious goals. The first one, is to ensure that every instant terminates. The second one, is to guarantee that the computation of an instant terminates within feasible bounds which depend on the size of the parameters at the beginning of the instant. The third one, is to guarantee that the parameters of the program stay within certain bounds, and thus the resources needed for the execution of the program are controlled for arbitrarily many instants.

In this note we introduce a basic version of the SL model enriched *with data types* and develop static analysis methods to guarantee that each instant terminates in time polynomial in the size of the parameters of the program at the beginning of the computation. Following previous work by one of the authors [1], the method is based on a combination of standard termination techniques for term rewriting systems and an analysis of the size of the computed values based on the notion of quasi-interpretation. With respect to [1], the main novelties are: (1) A more general and abstract formalisation of the model. (2) A method to generate inequalities whose satisfaction in suitable structures entails a *polynomial bound on the size* of the parameters of the program for arbitrarily many instants (theorem 4.2). (3) A new method to ensure *polynomial time termination* which can be regarded as a specialisation of size change termination (theorem 5.1).

2 Model

A *program* is a multi-set of *threads* described by a list of mutually recursive type, function, and behaviour definitions. Threads interact through shared signals which may carry general values (including signals). The language should be regarded as an *intermediate* code where complex control structures have been compiled into a simple tail-recursive form.

Types and Constructors. We assume a list of *types* t, t', \dots and a list of *constructors* c, c', \dots . For constructors of particular ‘signal’ types we may use the notation r, r', \dots and refer to them as *reference values*. A *value* v is a first order term built out of constructors. The *size* $|v|$ of a value v is defined by $|c| = 0$ and $|c(v_1, \dots, v_n)| = 1 + |v_1| + \dots + |v_n|$.

We will use the notation \mathbf{a} to denote a vector a_1, \dots, a_n of elements and denote with σ, σ', \dots a *substitution* $[\mathbf{v}/\mathbf{x}]$ mapping variables to values. Types and constructors are declared by a system of equations having one of the following shapes:

$$(1) \quad t = \dots \mid c \text{ of } t_1, \dots, t_n \mid \dots \quad (2) \quad t = \text{Sig}(t') \text{ with } \dots \mid r := v \mid \dots$$

In equation (1), we declare a type t and a constructor c with functional type $(t_1, \dots, t_n) \rightarrow t$. In equation (2), we declare a type of *evaluated signals*

including a signal r whose value at the beginning of each instant is v . It is intended that the value v has type t' (see below) and for the sake of simplicity we assume $|v| = 0$. Signals can be read and written and their values are always defined.

The system of equations is subject to the usual convention that types and constructors occurring in it are declared exactly once. This means that we can assign to every constructor a unique type $(t_1, \dots, t_n) \rightarrow t$ where $n \geq 0$. With respect to this assignment, values are typed according to the rule: the value $c(v_1, \dots, v_n)$ has type t if c is assigned the type $(t_1, \dots, t_n) \rightarrow t$ and the values v_i have type t_i for $i = 1, \dots, n$. Finally, we have a special *behaviour type* beh : elements of this type do not return a value but produce side effects. In the following we will manipulate various syntactic concepts: variables, values, patterns, expressions, expression bodies, substitutions, behaviours, programs, ... and we will always assume that they are *well typed*. The typing rules are standard and are omitted.

Expressions. Let x, y, \dots denote *variables* ranging over values. A *pattern* p is a well-typed term built out of constructors and variables. In particular, a *linear pattern* p is a pattern whose variables are all distinct. In the following all the patterns are supposed to be linear.

An *expression* e has the shape $h(e_1, \dots, e_n)$ where $n \geq 0$ and h can be either a variable x , or a constructor c , or a function symbol f . A *function symbol* f of type $(t_1, \dots, t_n) \rightarrow t$ is specified by an equation $f(\mathbf{x}) = eb$. Here eb is an *expression body* defined by the grammar: $eb ::= e \mid \text{match } x \text{ with } \dots p \Rightarrow eb \dots$. To simplify the presentation, we assume that a (well-typed) value matches exactly one pattern.

A closed expression body eb evaluates to a value v , written $eb \Downarrow v$, according to the following standard rules where σ denotes a matching substitution and $\mathbf{e} \Downarrow \mathbf{v}$ stands for $e_1 \Downarrow v_1, \dots, e_n \Downarrow v_n$.

$$\begin{array}{c}
 (e_1) \frac{\mathbf{e} \Downarrow \mathbf{v}}{c(\mathbf{e}) \Downarrow c(\mathbf{v})} \quad (e_2) \frac{\mathbf{e} \Downarrow \mathbf{v}, \quad f(\mathbf{x}) = eb, \quad [\mathbf{v}/\mathbf{x}]eb \Downarrow v}{f(\mathbf{e}) \Downarrow v} \\
 (e_3) \frac{\sigma p = v' \quad \sigma eb \Downarrow v}{\text{match } v' \text{ with } \dots p \Rightarrow eb \dots \Downarrow v}
 \end{array}$$

Thread behaviours. In the following we let ϱ, ϱ', \dots range over both variables and reference values. We denote with b, b', \dots behaviours defined as follows:

$$\begin{array}{l}
 b ::= \text{stop} \mid f(\mathbf{e}) \mid \text{yield}.b \mid \text{next}.f(\mathbf{e}) \mid \varrho := e.b \mid \\
 \text{match } x \text{ with } \dots p \Rightarrow b \dots \mid \text{read } \varrho \text{ with } \dots p \Rightarrow b \dots [x] \Rightarrow f(\mathbf{e})
 \end{array}$$

where f is a functional symbol of type $(t_1, \dots, t_n) \rightarrow beh$ and defined by an equation $f(\mathbf{x}) = b$. In the examples, we may omit the branches $p \Rightarrow b$ or the

where the program $\lfloor P \rfloor_s$ representing the result of the execution of the threads at the end of the instant is defined as follows:

$$\begin{aligned} \lfloor P \rfloor_s &= \{\lfloor b \rfloor_s \mid b \in P\} & \lfloor stop \rfloor_s &= stop \\ \lfloor next.f(\mathbf{e}) \rfloor_s &= f(\mathbf{e}) & \lfloor read\ r \ \dots \ [x] \Rightarrow f(\mathbf{e}) \rfloor_s &= [s(r)/x]f(\mathbf{e}) \end{aligned}$$

Remark 2.1 [fairness] The execution of a program within an instant consists of a serialisation of the execution of the behaviours that compose it until all behaviours are suspended. Rule (p_2) allows a completely non-deterministic scheduling of the behaviours. We say that an execution (within an instant) is *unfair* if there is a behaviour which is run (at least) twice and between the two runs there is a distinct behaviour which is continuously enabled but never run. Programs including the *yield* instruction may rely on the hypothesis that all executions are fair (cf. example 3.2).

3 Constraints generation

We introduce a suitable control flow analysis associating with a program a set of inequalities over first order terms.

Read once condition. We require that threads perform any given read instruction at most once in an instant. This can be checked by a simple control flow analysis rejecting programs that may traverse several times within an instant the same read instruction. Following this check, we assign to every read instruction in a program a distinct fresh label, y , and we collect all these labels in an ordered sequence, y_1, \dots, y_m . In the following, we will sometimes use the notation $read_{\langle y \rangle} \rho$ with \dots in the code of a behaviour to make visible the label of a *read* instruction. Then with every function symbol f defining a behaviour we associate a list \mathbf{y}_f composed of the labels of the read instructions we may execute within an instant starting from f . The important point is that the computation of f within an instant can be regarded as a *function* of the parameters and the values read within the instant.

Control points. A *control point* is a triple $(f(\mathbf{p}), be, i)$ where, intuitively, f is the currently called function, \mathbf{p} represents the patterns crossed so far in the function definition plus possibly the labels of the read instructions that still have to be executed, be is the continuation, and i is an integer flag in $\{0, 1, 2\}$ that will be used to associate with the control point various kinds of conditions. If the function f returns a value and is defined by the equation $f(\mathbf{x}) = eb$, then we associate with f the set $\mathcal{C}(f, \mathbf{x}, eb)$ defined as follows:

$$\begin{aligned} \mathcal{C}(f, \mathbf{p}, eb) &= \text{case } eb \text{ of} \\ e &: \{(f(\mathbf{p}), e, 0)\} \\ \text{match } x \text{ with } \dots p \Rightarrow eb' \dots : \dots \cup \mathcal{C}(f, [p/x]\mathbf{p}, eb') \cup \dots \end{aligned}$$

On the other hand, suppose the function f defines a behaviour by the equation $f(\mathbf{x}) = b$. Then we generate a fresh function symbol f^+ whose arity is that of f plus the number of variables in \mathbf{y}_f (the ordered sequence of labels corresponding to read instructions that may be performed by f within an instant). When unfolding the definition of \mathcal{C} the parameters \mathbf{x} and the labels \mathbf{y}_f of the function f^+ may be replaced by patterns.

When going from one instant to the following, we need to control the size of the parameters. The basic idea is that if f may call g in the current instant or in the following then the parameters of f should control the size of the parameters of g . This idea has to be implemented with some care because some values may depend on read instructions and some parameters may be discarded before the following instant begins. Therefore, we identify first the function symbols that may start a behaviour. They include all function symbols with which we can start the computation of a thread and all function symbols that follow a *next* instruction or a $[y] \Rightarrow \dots$ branch. We call these function symbols *initial*. Next we need some notation. Let $\mathbf{0}$ be a fresh constant. If h is a function of arity n and $I \subseteq \{1, \dots, n\}$ then $h(e_1, \dots, e_n)_I$ is defined as $h(e'_1, \dots, e'_n)$ where $e'_i = e_i$ if $i \in I$ and $e'_i = \mathbf{0}$ otherwise. Intuitively, in $h(e_1, \dots, e_n)_I$ we set to 0 all arguments that are not in I . For each function symbol f defining a behaviour of arity n with a related function symbol f^+ of arity $n + m$ we define a set $I_f \subseteq \{1, \dots, n\}$ with the condition that $I_f = \{1, \dots, n\}$ if f is initial. In particular this means that we neglect all arguments that correspond to the read instructions.

With this convention, the set of control points associated with f^+ is the set $\mathcal{C}(f^+, (\mathbf{x}, \mathbf{y}_f), b)$ defined as follows:

$$\begin{aligned}
 \mathcal{C}(f^+, \mathbf{p}, b) &= \text{case } b \text{ of} \\
 (\mathcal{C}_1) \text{ stop} &: \emptyset \\
 (\mathcal{C}_2) g(\mathbf{e}) &: \{(f^+(\mathbf{p}), g^+(\mathbf{e}, \mathbf{y}_g), 0), (f^+(\mathbf{p})_{I_f}, g^+(\mathbf{e}, \mathbf{y}_g)_{I_g}, 2)\} \\
 (\mathcal{C}_3) \text{ yield}.b' &: \mathcal{C}(f^+, \mathbf{p}, b') \\
 (\mathcal{C}_4) \text{ next}.g(\mathbf{e}) &: \{(f^+(\mathbf{p})_{I_f}, g^+(\mathbf{e}, \mathbf{y}_g)_{I_g}, 2)\} \\
 (\mathcal{C}_5) \varrho := e.b' &: \{(f^+(\mathbf{p}), e, 1)\} \cup \mathcal{C}(f^+, \mathbf{p}, b') \\
 (\mathcal{C}_6) \text{ match } x \text{ with } \dots p \Rightarrow b' \dots &: \dots \cup \mathcal{C}(f^+, ([p/x]\mathbf{p}), b') \cup \dots \\
 (\mathcal{C}_7) \left(\begin{array}{l} \text{read}_{\langle y \rangle} \varrho \text{ with } \dots \\ p \Rightarrow b' \dots [y] \Rightarrow g(\mathbf{e}) \end{array} \right) &: \{(f^+(\mathbf{p})_{I_f}, g^+(\mathbf{e}, \mathbf{y}_g)_{I_g}, 2)\} \cup \\
 &\dots \mathcal{C}(f^+, [p/y]\mathbf{p}, b') \dots
 \end{aligned}$$

Note that in the clause (\mathcal{C}_2) , the read once condition guarantees that the labels \mathbf{y}_g occur in the patterns \mathbf{p} . An *instance* of a control point $(f(\mathbf{p}), be, i)$ is an expression body or a behaviour $be' = \sigma(be)$, where σ is a substitution mapping the free variables in be to values. In order to carry on the proofs, it is convenient to reformulate expression body evaluation and behaviour evaluation on instances of control points. A hint on how to do this is given in the

proof of theorem 4.2 and a complete treatment is available in [1].

We associate with a control point $(f(\mathbf{p}), be, i)$ a constraint $f^+(\mathbf{p}) \succeq_i be$ for $i = 0, 1, 2$, and say that the constraint has *index* i . Intuitively, we rely on the constraints of index 0 to enforce termination of the instant, on those of index 0, 1 to enforce a bound on the size of the computed values within an instant, and on those of index 0, 1, 2 to guarantee a bound on the size of the computed values for arbitrarily many instants. Note that the constraints are on pure first order terms, a property that allows us to reuse techniques developed in the standard term rewriting framework.

Example 3.1 As a running example, we consider the case of a server $f(s, x)$ that at every instant, yields the control, reads a list of requests on the signal s , and serves the requests:

$$\begin{aligned} f(s, x) &= \text{yield.read } s \text{ with } l \Rightarrow f'(s, x, l) \\ f'(s, x, l) &= \text{match } l \text{ with nil} \Rightarrow \text{next.f}(s, x) \mid \\ &\quad \text{cons}(\text{req}(r, y), l') \Rightarrow r := h_1(y, x).f'(s, h_2(y, x), l') \end{aligned}$$

The server maintains a state x . A request contains a data y and a return signal r . We leave the functions h_1 and h_2 unspecified; the first is used to reply to the request and the second to compute the following state of the server. A client $g(s, r, y)$ that wishes to use the server could be defined as follows:

$$\begin{aligned} g(s, r, y) &= \text{read } s \text{ with } l \Rightarrow s := \text{cons}(\text{req}(r, y), l). \\ &\quad \text{yield.read } r \text{ with } z \dots \end{aligned}$$

Notice that the operation of inserting a message in a list requires a read operation and therefore the read once condition forbids to iterate this kind of operation within an instant. However, arbitrarily many behaviours may perform the operation within an instant. We compute the constraints of index 0, 1, 2 assuming that f is initial, f' is not initial, and $I_f = \{1, 2\} = I_{f'}$.

$$\begin{aligned} f'^+(s, x, \text{cons}(\text{req}(r, y), l')) &\succeq_0 f'^+(s, h_2(y, x), l) & f^+(s, x, l) &\succeq_0 f'^+(s, x, l) \\ f'^+(s, x, \text{cons}(\text{req}(r, y), l')) &\succeq_1 h_1(y, x) & f^+(s, x, 0) &\succeq_2 f'^+(s, x, 0) \\ f'^+(s, x, 0) &\succeq_2 f^+(s, x, 0) & f'^+(s, x, 0) &\succeq_2 f'^+(s, h_2(y, x), 0) \end{aligned}$$

Example 3.2 [registers] In our framework, registers can be regarded as signals that preserve their values from one instant to the following. We can simulate a register r with a signal (with the same name) and a thread whose behaviour $f(r)$ is described by:

$$f(x) = \text{read } x \text{ with } [y] \Rightarrow g(x, y), g(x, y) = x := y.f(x)$$

The behaviour $f(r)$ waits the end of the instant to read the value y of r and in the following instant it writes y again in r . Since in the simulation r is a signal, at the beginning of the instant its value is reset. Then we have to make sure that the behaviour $f(r)$ runs before any other behaviour tries to read r .

For this purpose, we rely on the fairness hypothesis (cf. remark 2.1), and transform all other behaviour definitions so that they start with a *yield*. We can extract from this simulation conditions to control the size of the values in the registers. In particular, we note that the constraint $f^+(x, \mathbf{0}) \succeq_2 g^+(x, y)$ can only be satisfied if the value y contained in the register has bounded size. This is an important restriction we have to impose on the programming language.

4 Size bounds

In order to bound the size of the values computed by a program we rely on the notion of *quasi-interpretation* (see [6,2,3,7]). In a nutshell, a quasi-interpretation q_f of a function symbol f is a numerical function ensuring that there is a constant k such that the size of the largest value computed by f when called with arguments v_1, \dots, v_n is bounded by $q_f(k|v_1|, \dots, k|v_n|)$.

The *synthesis* of quasi-interpretations can be mechanised to some extent. The existence of a quasi-interpretation *does not* entail termination but it *does allow* to control the complexity of the computed function following a well-known result of S. Cook [9] who showed that a *polynomially bounded* auxiliary push down automaton can be simulated by a Turing Machine in exponential time using a ‘table’ to store intermediate results. We refer to [3] for an extended discussion of these issues.

Assignments and quasi-interpretations. Suppose given a program. An *assignment* q associates with each constructor and function symbol h , a function q_h over the natural numbers \mathbf{N} such that:

- (1) If c is a constructor with arity n then $q_c = 0$ if $n = 0$ and $q_c(x_1, \dots, x_n) = d + \sum_{i \in 1..n} x_i$ if $n > 0$ where $d \in \mathbf{N}$ and $d \geq 1$ (this guarantees that the quasi-interpretation of a value is proportional to its size). In particular, the quasi-interpretation of the special constant $\mathbf{0}$ introduced in the constraints is the natural number 0.
- (2) If f is a function symbol with arity n then $q_f : (\mathbf{N})^n \rightarrow \mathbf{N}$ is a monotonic function over the natural numbers.

We say that a function $U : \mathbf{N} \rightarrow \mathbf{N}$ bounds the assignment q if $\forall x \in \mathbf{N} \ q_h(x, \dots, x) \leq U(x)$. In particular, we say that q is *polynomially bounded* if the function U can be a polynomial. We associate with an expression e without variables a natural number q_e as follows:

$$q_{h(e_1, \dots, e_n)} = q_h(q_{e_1}, \dots, q_{e_n}) . \tag{1}$$

We write $q \models e_1 \succeq e_2$ if the assignment q satisfies a constraint $e_1 \succeq e_2$ where e_1, e_2 are expressions (possibly with variables). This is defined as:

$$q \models e_1 \succeq e_2 \text{ if } \forall \sigma \ q_{\sigma e_1} \geq q_{\sigma e_2} \tag{2}$$

where σ is a substitution associating values with variables. We also write: $q \models e_1 \succ e_2$ if $\forall \sigma \ q_{\sigma e_1} > q_{\sigma e_2}$.

An assignment q is a *quasi-interpretation* if it satisfies the constraints of index 0, 1, 2 generated by the program and moreover if it satisfies $f(x_1, \dots, x_n) \succeq x_i$ for all $i = 1, \dots, n$ and all function symbols f . This last condition allows to control the size of the values computed by a function and not just the size of its result. Thus if $f(v_1, \dots, v_n) \Downarrow v$ then we know that $q_{f(v_1, \dots, v_n)} \geq q_v$ and *moreover* that for any value u computed by the function $q_{f(v_1, \dots, v_n)} \geq q_u$.

Example 4.1 We define a quasi-interpretation for the running example 3.1. We suppose the functions h_1 and h_2 operate over values of bounded size. Then we can just take $q_{h_1} = q_{h_2} = \lambda(x, y).k$ for some suitable constant $k \in \mathbf{N}$. We can also set $q_{\text{cons}}(x, l) = x + l + 1$ and $q_{\text{req}}(r, y) = r + y + 1$. Then we can satisfy all the constraints by assuming $q_{f^+}(s, x, l) = q_{f^+}(s, x, l) = \max(l, k)$.

Theorem 4.2 *If a program P has a polynomially bounded quasi-interpretation q then the size of the largest value computed by P is polynomial in the size of the parameters of the program at the beginning of the computation.*

In order to prove theorem 4.2, we define a *small step* reduction of behaviours on instances of control points. The reduction makes abstraction of the memory and the scheduler while depending on an assignment δ associating values with the labels of the read instructions. The assignment δ is a kind of *oracle* that provides the thread with the values it may read within the current instant. A fresh assignment is generated whenever we move from one instant to the following one (rules \mathbf{b}'_3 and \mathbf{b}'_6).

$$\begin{array}{ll}
 (\mathbf{b}'_2) & (f^+(\mathbf{p}), \text{yield}.b, \sigma, \delta) \quad \rightarrow \quad (f^+(\mathbf{p}), b, \sigma, \delta) \\
 (\mathbf{b}'_3) & (f^+(\mathbf{p}), \text{next}.g(\mathbf{e}), \sigma, \delta) \quad \rightarrow \quad (f^+(\mathbf{p}), g(\mathbf{e}), \sigma, \delta') \\
 (\mathbf{b}'_4) & (f^+(\mathbf{p}), \text{match } x \text{ with } \dots p \Rightarrow b \dots, \sigma, \delta) \rightarrow (f^+([p/x]\mathbf{p}), b, \sigma_1 \circ \sigma, \delta) \text{ if (1)} \\
 (\mathbf{b}'_5) & (f^+(\mathbf{p}), \varrho := e.b, \sigma, \delta) \quad \rightarrow \quad (f^+(\mathbf{p}), b, \sigma, \delta) \text{ if } \sigma e \Downarrow v \\
 (\mathbf{b}'_6) & (f^+(\mathbf{p}), \text{read}_{\langle y \rangle} \varrho \dots [y] \Rightarrow g(\mathbf{e}), \sigma, \delta) \quad \rightarrow \quad (f^+(\mathbf{p}), g(\mathbf{e}), [\delta(y)/y] \circ \sigma, \delta') \text{ if (2)} \\
 (\mathbf{b}'_7) & (f^+(\mathbf{p}), \text{read}_{\langle y \rangle} \varrho \dots p \Rightarrow b \dots, \sigma, \delta) \quad \rightarrow \quad (f^+(\mathbf{p}), b, \sigma_1 \circ \sigma, \delta) \text{ if (3)} \\
 (\mathbf{b}'_8) & (f^+(\mathbf{p}), g(\mathbf{e}), \sigma, \delta) \quad \rightarrow \quad (g^+(\mathbf{x}, \mathbf{y}_g), b, \sigma', \delta) \text{ if (4)}
 \end{array}$$

where: (1) $\equiv \sigma_1 p = \sigma x$, (2) \equiv (no pattern matches $\delta(y)$), (3) $\equiv \sigma_1(p) = \delta(y)$, and (4) \equiv ($\sigma e \Downarrow \mathbf{v}, g(\mathbf{x}) = b, \sigma' = [\mathbf{v}/\mathbf{x}]$). Relying on the small step semantics we then prove the following lemma from which theorem 4.2 follows directly.

Lemma 4.3 *Suppose q is a polynomially bounded quasi-interpretation for a program P with m distinct read instructions and n threads. Let i denote a thread of the program and $P(i)$ the associated behaviour.*

(1) *Suppose that at the beginning of the computation $P(i) = f(\mathbf{v})$ and that $P(i) = g(\mathbf{e})$ at the beginning of a following instant. Then $q_{f(\mathbf{v})} \geq q_{g(\mathbf{e})}$.*

(2) Suppose that at the beginning of an instant $P(i) = f(\mathbf{v})$. Then the size of the values computed by the thread i during that instant is bounded by $q_{f+(\mathbf{v}, \mathbf{u})}$ where \mathbf{u} are the values contained in the signals at the time they are read by the thread (or their default value, otherwise).

(3) Suppose c is a bound on the quasi-interpretation of the parameters of the behaviours at the beginning of an instant and that U is a polynomial bound on the quasi-interpretation. Then the size of the values computed by the program P during an instant is (polynomially) bounded by $U^{n \cdot m + 1}(c)$.

5 Polynomial time reactivity

Marion *et al.* [11,6,7] have shown how to ensure termination in polynomial space or time by combining the existence of a polynomially bounded quasi-interpretation with termination by suitable restrictions of the *recursive path ordering* (see, *e.g.*, [4]). In this section we propose a more flexible approach where we compare the arguments of the functions using the quasi-interpretation rather than the restricted recursive path order.

The constraints of index 0 have one of the following shapes: (A) $f(\mathbf{p}) \succeq_0 e$ or (B) $f^+(\mathbf{p}) \succeq_0 g^+(\mathbf{e})$. We start by building the least pre-order (reflexive and transitive) \geq_F on the function symbols such that $f \geq_F g$ if f appears on the left hand side and g on the right hand side of a constraint of index 0. We write $f =_F g$ if $f \geq_F g$ and $g \geq_F f$. We note that a function symbol that returns a value can never call a function symbol that generates a behaviour, so we have that the latter is always larger than the former.

As in recursive path orderings, we associate a *status* with each function symbol which determines how to compare the arguments of the function. In our case, we consider either a *lexicographic* or a *multi-set* status. We assume that function symbols that are equivalent with respect to the pre-order \geq_F have the same arity and the same status.

Following [7], we say that a constraint is *linear* if there is at most one function symbol on the right hand side which is equivalent to the function symbol on the left hand side. Henceforth, we assume that all the constraints of index 0 are linear (note that the constraints of shape (B) are always linear).

Suppose given a polynomially bounded quasi-interpretation q for the program. We rely on the quasi-interpretation to compare the arguments of equivalent function symbols. Therefore, we depart from the standard RPO method and rely on a size change analysis as in [10].

For *lexicographic comparison*, we write: $q \models (p_1, \dots, p_n) >_{lex} (e_1, \dots, e_n)$ if there is an $i < n$ such that for all σ , $q_{\sigma p_1} \geq q_{\sigma e_1}, \dots, q_{\sigma p_{i-1}} \geq q_{\sigma e_{i-1}}$, and $q_{\sigma p_i} > q_{\sigma e_i}$. For *multi-set comparison*, we write: $q \models (p_1, \dots, p_n) >_{mset} (e_1, \dots, e_n)$ if for all σ , $\{q_{\sigma p_1}, \dots, q_{\sigma p_n}\} >_{mset}^{\mathbf{N}} \{q_{\sigma e_1}, \dots, q_{\sigma e_n}\}$, where $\{\dots\}$ is our notation for multi-sets and $>_{mset}^{\mathbf{N}}$ is the multi-set order over the natural numbers. We say that the quasi-interpretation is *compatible* with the order if in all con-

straints of index 0 of the shape:

$$f(p_1, \dots, p_n) \succeq_0 C[g(e_1, \dots, e_n)]$$

where C is a one hole context and $f =_F g$ with status $st \in \{lex, mset\}$ we have that $q \models (p_1, \dots, p_n) >_{st} (e_1, \dots, e_n)$.

Theorem 5.1 *Suppose the program has a compatible and polynomially bounded quasi-interpretation. Then the computation of an instant terminates in time polynomial in the size of the parameters at the beginning of the instant.*

PROOF HINT. In our language all function definitions are *first-order*. Then the state of each thread can be represented by a stack of *frames*. A frame is a triple $(f, pc, v_1 \cdots v_n)$ where f is the name of a function, pc points to the instruction of the function to be executed, and $v_1 \cdots v_n$ is a stack of values. The maximum number of values that can be on the stack can be statically determined. Thus the size of a frame is determined by the size of the values that can be found on the stack and the quasi-interpretation provides a polynomial bound for that. Our task is to bound the *number of frames* that a thread can allocate within an instant. The *rank* of a function symbol f is the length of the longest chain of functions such that $f >_F f_1 >_F \cdots >_F f_n$ with respect to the pre-order on function symbols. We estimate the number $ncall(o)$ of frames that a function f of rank o can generate when called with arguments of size at most B using the linearity of the constraints of index 0 and the hypothesis that the quasi-interpretation is over the natural numbers. \square

Example 5.2 With reference to the running example 3.1, 4.1, we assume $f^+ >_F f'^+$ and a lexicographic status (from right to left) for the function f'^+ .

Remark 5.3 The method in [7] is based on a recursive path ordering that coincides with the homeomorphic embedding \triangleright_{emb} on constructors. We observe that if $p \triangleright_{emb} p'$ and q is an assignment then $q \models p \succ p'$. It follows that the method presented here succeeds whenever the one presented in [7] does (and therefore all PTIME functions can be represented). The converse fails since we can never have $p \triangleright_{emb} e$ when e contains a function symbol.

6 Conclusion

We have presented a static analysis that guarantees that a program composed of synchronous cooperative threads reacts *every instant* in time polynomial in the size of the parameters at the beginning of the computation. The conditions we have imposed refine and extend those proposed in [1] to control the resources within an instant. As it could be expected, the possibility of controlling the resources for arbitrarily many instants comes at a price. First, the parameters of each thread at the beginning of each instant have to be essentially non-size increasing. To satisfy this requirement it is often necessary to rely on the fair yield hypothesis and to reprogram the application

so that an instant is sufficiently large; typically, an instant corresponds to a protocol transaction or to a logical simulation step. Second, the registers (or persistent signals) have to carry values of bounded size while unbounded data structures can still be allocated on signals (which are reset at the beginning of each instant).

Acknowledgement The authors were partially supported by *ACI CRISS*. This work was started while the first author was at the *Université de Provence*.

References

- [1] R. Amadio and S. Dal-Zilio. Resource control for synchronous cooperative threads. In *Proc. CONCUR*, SLNCS 3170, 2004. Extended version to appear in *Theoretical Computer Science*.
- [2] R. Amadio. Max-plus quasi-interpretations. In *Proc. of TLCA*, Springer LNCS 2701, 2003.
- [3] R. Amadio. Synthesis of max-plus quasi-interpretations. In *Fundamenta Informaticae*, 65(1-2):29–60, 2005.
- [4] F. Baader and T. Nipkow. *Term rewriting and all that*. Cambridge University Press, 1998.
- [5] L. Mandel and M. Pouzet. *Reactive ML, a reactive extension to ML*. In *Proc. ACM PPDL*, 2005.
- [6] G. Bonfante, J.-Y. Marion, and J.-Y. Moyen. On termination methods with space bound certifications. In *Proc. PSI*, SLNCS 2244, 2001.
- [7] G. Bonfante, J.-Y. Marion, and J.-Y. Moyen. Quasi-interpretations. Draft November 2004. Available from the authors.
- [8] F. Boussinot and R. De Simone, The SL Synchronous Language. *IEEE Trans. on Software Engineering*, 22(4):256–266, 1996.
- [9] S. Cook. Characterizations of pushdown machines in terms of time-bounded computers. *Journal of the ACM*, 18(1):4–18, 1971.
- [10] C. Lee, N. Jones, and A. Ben-Amram. The size-change principle for program termination. In *Proc. ACM POPL*, 2001.
- [11] J.-Y. Marion. *Complexité implicite des calculs, de la théorie à la pratique*. Université de Nancy. Habilitation à diriger des recherches, 2000.
- [12] J. Ousterhout. Why threads are a bad idea (for most purposes). Invited talk at the USENIX Technical Conference, 1996.
- [13] Reactive Programming, INRIA Sophia-Antipolis, Mimosa Project. <http://www-sop.inria.fr/mimosa/rp>.