2018 20th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)

Towards Performance Evaluation Programming

Eneia Nicolae Todoran
Department of Computer Science
Technical University of Cluj-Napoca
Cluj-Napoca, Romania
Email: Eneia.Todoran@cs.utcluj.ro

***Abstract*—In recent work we have introduced an experimental concurrent programming language which supports a systematic approach to performance analysis and formal verification correlated with a programming style called performance evaluation programming [19]. For the purpose of formal verification, the ranges of variables must be bounded and concurrent programs are translated into corresponding (finite state) Continuous Time Markov Chains (CTMCs) which are analyzed by using the PRISM tool. Activities in a CTMC model are abstracted by their rates. In the language introduced in [19] an activity is the evaluation of a function expressed in a functional sub-language. The solution presented in [19] supports formal verification in a systematic manner, but not automatically, requiring the programmer to generate certain data for the performance evaluation experiments. In this paper we refine the design of the functional sub-language introduced in [19] by using concepts of functional programming with dependent types. We use dependent types to control the ranges of variables. The solution presented in this paper is devised to support automatic performance evaluation and formal verification of (bounded versions of) concurrent programs.**

***Keywords*-concurrent programming language; performance evaluation; dependent types; continuous time Markov chains;**

I. Introduction

In most semantic investigations of programming languages it is assumed a computing model providing infinite memory resources. Values of basic types are taken from infinite sets, such as the set of integer or natural numbers. Lists are assumed to be of finite or infinite length. This general assumption is also reflected in the design of real programming languages, which treat the various memory overflow events as "exceptions". In fact, the memory of any real computer is always finite, and the size of any (basic or structured) value stored in the memory of a computer is bounded. Thus, any real computer is in fact a finite state system (although the number of states can be very large).

Model checking is an automatic technique which can be used for verifying models expressed as finite state machines [3]. It is a successful formal verification technique. However, it faces the so-called "state explosion problem", hence, in practice, it may fail to handle models of realistic systems. In order to perform automatic verification of computer programs by using model checking techniques we should control somehow the size of programs. The solution investigated in this paper relies on notions and techniques introduced in the context of model checking and functional programming with dependent types [21].

Performance Evaluation Programming or *Quantitative Programming* [19] is a programming paradigm which supports using model checking techniques to accomplish performance analysis and formal verification of bounded versions of (concurrent) programs in a systematic manner, or automatically. In [19] we presented an experimental concurrent programming language - that we named $\mathcal{L}_{PEP}$ - which supports performance evaluation programming by translating $\mathcal{L}_{PEP}$ programs into corresponding Continuous Time Markov Chains (CTMCs) [8], which are analyzed using the PRISM probabilistic model checker [9].

Intuitively, $\mathcal{L}_{PEP}$ is a process algebra language where elementary actions are function evaluations. $\mathcal{L}_{PEP}$ functions are specified in a functional sub-language λ_{PEP}. The structure of an $\mathcal{L}_{PEP}$ program is similar to the structure of a CTMC model expressed in the modeling language of the PRISM model checker [9], comprising a collection of modules executed concurrently. However, $\mathcal{L}_{PEP}$ is not a model checker but a general purpose programming language. Activities are abstracted in a PRISM CTMC model by their rates. In $\mathcal{L}_{PEP}$ an activity is the evaluation of a function. Thus, in $\mathcal{L}_{PEP}$ we can distinguish a control flow kernel, which is a concurrent state-based language based on PRISM, and a functional sub-language λ_{PEP}.

$\mathcal{L}_{PEP}$ supports variables of both primitive types (booleans and integers) and non-primitive types, e.g., lists. For the performance evaluation, the programmer has to bind the variables' ranges occurring in an $\mathcal{L}_{PEP}$ program. Next, the CTMC model corresponding to an $\mathcal{L}_{PEP}$ program must be constructed. The main difficulty is to determine the activity rates of the target CTMC model. The solution presented in [19] relies on organizing the data space of an $\mathcal{L}_{PEP}$ program according to the principle of mathematical induction, which we see as a strong point of [19]. However, the solution presented in [19] is not fully supported by the constructions of $\mathcal{L}_{PEP}$, requiring the programmer to generate lists of argument tuples for all λ_{PEP} functions of an $\mathcal{L}_{PEP}$ program. This means that $\mathcal{L}_{PEP}$ supports formal verification in a systematic, but not automatic manner.

In this paper, we refine the design of the functional sub-language introduced in [19] by using concepts of functional

978-1-7281-0625-0/18/$31.00 ©2018 IEEE
DOI 10.1109/SYNASC.2018.00054

programming with dependent types. We use dependent types to control the ranges of variables for the purpose of automatic performance evaluation and formal verification of (bounded versions of) concurrent programs. We describe an experimental concurrent programming language that we name QL0. [1] The control flow kernel of QL0 is based on the control flow kernel of $\mathcal{L}_{PEP}$ (see [19], Section II.A); however, the functional sub-language is different.

The functional sub-language of QL0 is called λ_{QL0}. λ_{QL0} extends λ_{PEP} with the restricted notion of dependent types implemented in Dependent ML (DML) (where type checking can be reduced to a constraint satisfaction problem, which is decidable) [21], [13]. In DML, type dependency on terms is only allowed for certain *index sorts*, which is sufficient for our present purposes.

The paper reports work in progress. Various possible refinements and improvements are considered in Section V as options of future research. We are currently developing a type checker and an executable interpreter for QL0 together with a component designed to automatically translate a QL0 program into a corresponding PRISM CTMC model. This way, we could obtain a fully automatic version of the software formal verification approach introduced in [19].

We describe the language QL0 (including its sub-language λ_{QL0}) in Section II. In Section III we present a QL0 example program. In Section IV we study the performance of the QL0 program given in Section III.

A. Discussion and Related Work

Model checking is a successful formal verification technique. Yet, it is not used to verify an actual implementation, but a system model, therefore results are as good as the system model [2]. Moreover, model checkers tend to support low-level languages [23]. In the performance evaluation programming approach [19] the same general purpose high-level language (QL0 or $\mathcal{L}_{PEP}$) is used both in the programming phase and in the formal verification phase; this can be deemed as a strong point of our approach [19]. The approach presented in this paper presumes a final step carried out by a model checking tool. Hence, we face the state explosion problem like in traditional model checking. An analysis of the time and space complexity of the algorithms involved in our approach to formal verification is provided in [19].

Let us mention some recent works focusing on combining advanced topics in types and programming languages with techniques developed for the purpose of software formal verification. The K framework [16] supports formal specification and verification of complex systems by using a flexible first order theory called matching logic. CARMA [10] is a recently developed stochastic process algebra language which can describe the behavior of collective adaptive systems evolving according to the CTMC model. An approach to formal verification based on combining type checking and model checking techniques is presented in [14].

[1] The abbreviation QL stands for *Quantitative programming Language*.

II. The Language QL0

The language QL0 has the same control flow kernel as the language $\mathcal{L}_{PEP}$ [19]; only the functional sub-language is different. The control flow kernel of QL0 is a concurrent language, based on the modeling language of the PRISM model checker, which in turn is based on the Reactive Modules formalism [1]. QL0 also provides a functional sub-language λ_{QL0}, which extends λ_{PEP} (the functional sub-language of $\mathcal{L}_{PEP}$) with the restricted notion of dependent types incorporated in DML [21].

The structure of a QL0 program is similar to that of a CTMC model expressed in the PRISM modeling language [9]. However, QL0 is not a model checker but a programming language. Activities are abstracted in a PRISM CTMC model by their rates. In the language QL0 an activity is the evaluation of a function expressed in the functional sub-language λ_{QL0}. A QL0 program is a collection of concurrently executed modules which can synchronize using a rendezvous mechanism described in Section II-A. Like in PRISM, a QL0 module can read the variables of any other module, but it can only modify the values of its own variables. QL0 supports variables of both primitive types, e.g., booleans and (bounded ranges of) natural numbers, and non-primitive types, e.g., lists (of bounded length).

The syntax of language $\mathcal{L}_{PEP}$ is described formally in [19]. We define the syntax of language QL0 almost exactly like the syntax of $\mathcal{L}_{PEP}$. The language QL0 provides constructions for specifying *modules* and *guarded commands*, and each module may declare a number of *functions* and *variables*. Only the functional sub-language λ_{QL0} is specific to QL0. The functional sub-language λ_{QL0} extends λ_{PEP} with DML-like dependent types, and refines definitions of basic sorts with associated operations. The language λ_{QL0} is described in Section II-B.

In this paper, we present the formal syntax of QL0 and we explain the behavior of QL0 programs informally. The development of a formal semantics for QL0 remains a subject of future research. Most QL0 syntactic constructs behave the same as the corresponding $\mathcal{L}_{PEP}$ syntactic constructs; only the constructs denoting operations on basic sorts and data structures (lists) are specific to QL0. The evaluation mechanism of QL0 programs is described in Section II-C.

In the rest of Section II, we present the syntactic constructions of QL0 (including the sub-language λ_{QL0}) and we explain their meaning informally. To illustrate how these constructions are used, a QL0 example program is presented in Section III.

A. Modules, commands, concurency and synchronization

We let Prg be the class of QL0 *programs* ρ and Md be the class of *module declarations* μ defined by the following

syntax:

$$\rho ::= \mu^+$$

$$\mu ::= \ \mathsf{module}\ M\,\{\,(\,\iota;\,)^*\ (\,\kappa;\,)^+\,\}$$

A QL0 program $\rho \in Prg$ is a non-empty list of module declarations. A module declaration $\mu \in Md$ is a construction of the form $\mu = \ \mathsf{module}\ M\,\{\,\iota_1; \ldots \iota_n; \kappa_1; \ldots \kappa_m;\,\}$, comprising a (possibly empty) list of *initialization statements* $(\iota_1; \ldots \iota_n;)$ followed by a non-empty list of *commands* $(\kappa_1; \ldots \kappa_m;)$; M is the name of the module, which must be unique.

The class $Init$ of initialization statements ι is given by:

$$\iota ::= f : T = t \mid w : T = t \mid v : \tau = e$$

Each variable in QL0 has a specific type. An initialization statement $\iota \in Init$ settles the type associated to a function or variable name for the entire duration of a program's execution. Function and variable definitions are *global* and can be used in any module. Above, f is a *function name* in a set $Fvar$, and w is an *(ordinary) variable name* in a set $Wvar$. v is an element of another set of variables $Vvar$, called *performance variables* in [19]. The sets $Fvar, Wvar$ and $Vvar$ are assumed to be (finite or countable and) pairwise disjoint. t is an λ_{QL0} term, and T is an λ_{QL0} type. We let Exp with typical element e be the class of *simple expressions*. τ is an λ_{QL0} base sort. The class of base sorts includes $\mathbf{Bool}$ and $\mathbf{N}$, together with bounded numeric ranges of the form $[i..j] = \{n : \mathbf{N} \mid i \leq n \wedge n \leq j\}$, where $i, j \in \mathbf{N}$, $i \leq j \leq \omega$ are natural numbers bounded by some fixed $\omega \in \mathbf{N}$. The classes of λ_{QL0} terms, types and base sorts are described in Section II-B.

A statement $f : T = t$ declares a function with name $f \in Fvar$ and type T; t is an λ_{QL0} term, usually a λ abstraction or a recursive function definition. A variable initialization statement $w : T = t$ assigns the value of the λ_{QL0} term t to w $(\in Wvar)$. In an initialization statement $f : T = t$ or $w : T = t$, t is an λ_{QL0} term, T is an λ_{QL0} type, and the type of term t must be T. An initialization statement $v : \tau = e$ assigns the value of expression e to v $(\in Vvar)$; the type of e must be τ. After initialization, the value of a variable w (performance variable v) can be modified by an assignment statement $w := t$ (an update $v := e$, see below). Like in PRISM, a module can read (access) the variables of any other module, but it can only modify the values of its own variables. However, the type of a variable cannot change.

Note that a (performance) variable $v \in Vvar$ can only store values of primitive types. A variable $w \in Wvar$ can also store values of non-primitive types, e.g., lists.

An expression $e \in Exp$ evaluates to a value belonging to some base sort τ. In $\mathcal{L}_{PEP}$ the class Exp of simple expressions is given by a grammar (providing arithmetic, relational and boolean operators) of the form: $e ::= \alpha \mid v \mid e + e \mid \cdots \mid e = e \mid e < e \mid e \ \mathsf{and}\ e \mid \cdots$, where $v \in Vvar$ and $\alpha ::= n \mid b$, with $n \in \mathbf{N}$, $b \in \mathbf{Bool}$. The class of simple expressions is extended in Section II-B with operators working on base sorts of the form $[i..j]$.

As in [19], the class Exp of expressions e is a subclass of the class of λ_{QL0} terms, and the class of base sorts τ is a subclass of the class of λ_{QL0} types.

The class Cmd of *commands* κ is given by:

$$\kappa ::= [a]\, g \to \gamma : u$$

An $\mathcal{L}_{PEP}$ command $\kappa \in Cmd$ is a construction of the form $[a]\, g \to \gamma : u$, where $a \in Act_\epsilon$, g is called a *guard*, γ is called an *activity specification*, or simply an *activity*, and u is a list of *updates*. $Act_\epsilon = Act \cup \{\epsilon\}$ is a set of *action names*, Act is a given (finite or countable) set and $\epsilon \notin Act$ is a distinct element.

Instead of $[\epsilon]\, g \to \gamma : u$ we write $[]\, g \to \gamma : u$. A command of the form $[]\, g \to \gamma : u$ is executed independently (without synchronization) by a module. On the other hand, concurrent modules synchronize over their common actions using the rendezvous mechanism described below.

The set of guards is given by the little grammar $g ::= e$; here $e \in Exp$ and the type of e must be $\mathbf{Bool}$, i.e., a guard is a boolean valued expression.

The set of updates is given by:

$$u ::= \mathsf{nop} \mid v := e\ \ (\,\&\ v := e\,)^*$$

u is a (possibly empty) list of updates. u may be the inoperative statement nop or a non-empty list of updates $(v_1 := e_1) \& \cdots \& (v_n := e_n)$. An assignment statement $v := e$ is called an *update*.

An *activity* γ may be skip, an λ_{QL0} term t, an assignment statement $w := t$, a send statement $!\,t$ or a receive statement $?\,\psi$, where ψ may be a variable $v \in Vvar$ or the symbol $\mathsf{passive}$. The precise syntax of activities is:

$$\gamma ::= \mathsf{skip} \mid t \mid w := t \mid !\,t \mid ?\,\psi$$

$$\psi ::= v \mid \ \mathsf{passive}$$

Remark 2.1: An expression $e \in Exp$ (occurring in a guard or an update) describes a simple, non-recursive computation. On the other hand, the execution of an activity may involve evaluating an λ_{QL0} term t, which can represent a recursively defined function. In the formal verification approach presented in this paper it is assumed that each λ_{QL0} term occurring in a QL0 program denotes a computation that terminates after a finite number of steps. Non-terminating computations (e.g., infinite loops) can be expressed using the control flow kernel of QL0.

Executing an activity t, $w := t$ or $!\,t$ involves evaluating the λ_{QL0} term t to an irreducible value (which requires a finite number of computation steps, as explained in Remark 2.1). An activity $w := t$ assigns the value of λ_{QL0} term t to

the global variable w. An activity of the form t is executed only for its side effects. In the example given in Section III this construction is only used to call the primitive $\mathsf{delay}(n)$, which delays the execution of the command in which it occurs for n time units, $n > 0$. skip is the fastest QL0 activity. skip behaves exactly like $\mathsf{delay}(1)$, delaying the execution of the current command with just one time unit, and producing no other side effect and no interesting value.

Remark 2.2: As in [19], we use the notion of a *time unit* assuming an execution model in which each elementary evaluation step takes one unit of time. For example, performing an arithmetic operation (like $+$ or $-$) or performing one (λ calculus) beta reduction step is assumed to take exactly one unit of time. The primitive $\mathsf{delay}(n)$ delays the execution of the current QL0 command by exactly n time units.

Concurrent QL0 modules can synchronize using a mechanism inspired by the ADA rendezvous [4]. Two commands $[a]\, g \to !\, t : u$ and $[a]\, g' \to ?\, \psi : u'$, occurring in two different modules (and labeled with the same action name $a \in Act$, $a \neq \epsilon$), can be executed synchronously in any state where both guards g and g' are satisfied (i.e., in any state where both g and g' evaluate to true).

A synchronous interaction involving two commands $\kappa = [a]\, g \to !\, t : u$ and $\kappa' = [a]\, g' \to ?\, \psi : u'$ entails evaluating the term t to an (irreducible) value. If $(\psi = v)$ is a performance variable, $v \in Vvar$, then the term t must reduce to a value of a base (primitive) type τ and the value of t is assigned to v.[2] If $(\psi = \mathsf{passive})$ then the term t - occurring in the send statement $!\, t$ - could have any type and its value is ignored; in particular, if $(\psi = \mathsf{passive})$ then t may be delay(n) whose type is $\mathbf{Unit}$ [19].

Remark 2.3: During a rendezvous involving two commands $\kappa = [a]\, g \to !\, t : u$ and $\kappa' = [a]\, g' \to ?\, \psi : u'$, command κ' is suspended while command κ evaluates term t. The command κ executing the send statement $!\, t$ imposes the duration of the rendezvous; the command κ' executing the receive statement $?\, \psi$ is passive in this time interval. After the synchronized execution of $!\, t$ and $?\, \psi$ completes, the corresponding updates u and u' are performed concurrently, and the rendezvous terminates.

B. The Functional Sub-Language λ_{QL0}

The functional sub-language λ_{QL0} extends λ_{PEP} with the (restricted) notion of dependent types incorporated in Dependent ML (DML) [21]. Like λ_{PEP}, λ_{QL0} is based on the simply typed λ calculus extended with simple features, such as lists and stores, and the primitive $\mathsf{delay}(n)$. The primitive $\mathsf{delay}(n)$ can be used to delay the execution of a QL0 process by n time units; see Remark 2.2. The formal syntax and semantics of λ_{PEP} are presented in [19]. In this section we focus on the differences between λ_{QL0} and λ_{PEP}.

[2]It is reasonable to consider that values of non-primitive types (e.g., lists) cannot be transmitted between concurrent modules in a single step.

Notation 2.4: λ_{QL0} extends λ_{PEP} with a notation for expressing dependent product types and dependent sum (existential) types. The notation that we use for dependent product types $\Pi x : A.B$ and dependent sum (existential) types $\Sigma x : A.B$ (including the notation for index sorts and propositions) is mainly based on [13] (chapter 2) and [21]. The dependent sum type $\Sigma x : A.B$ is the type of pairs (a, b) where $a : A$ and $b : B(a)$. For selecting the components of a pair t we use the dot notation [13]: we write $t.1$ for the first projection from t and $t.2$ for the second projection from t.

We use $\mathbf{Bool}$ for the sort of boolean values and $\mathbf{N}$ for the sort of natural numbers. Let $\omega \in \mathbf{N}$ be a fixed natural number. For any $i, j \in \mathbf{N}$, $0 \leq i \leq j \leq \omega$, we use the notation: $[i..j] = \{n : \mathbf{N} \mid i \leq n \wedge n \leq j\}$. We write $[i..i]$ as $[i]$. We define the class $(\tau \in)\mathbf{TyA}$ of *base sorts* by:

$$\tau ::= \mathbf{Bool} \mid \mathbf{N} \mid \cdots \mid [i..j] \mid \cdots$$

We assume that $[i..j]$ is a base sort for any $0 \leq i \leq j \leq \omega$. The sort $\mathbf{N}$ can be used to write programs with arbitrarily large natural numbers. However, no real computer supports such numbers. If we take ω sufficiently large, we can always work with numbers taken from a sort $[i..j]$, for some $0 \leq i \leq j \leq \omega$. For any $i, j \in \mathbf{N}$, $i \leq j \leq \omega$, we define a bounded summation operation $+_{[i..j]}$ over $[i..j]$ as follows:

$$n_1 +_{[i..j]} n_2 = \begin{cases} n_1 + n_2 & \mathsf{if} \quad n_1 + n_2 \leq j \\ j & \mathsf{otherwise} \end{cases}$$

Other operations can be defined over base sorts $[i..j]$ in a similar manner.

Starting from the class $\mathbf{TyA}$ we can define a class of *index sorts* with typical element I as explained in [13], [21]. Apart from the basic constructions for types (base type families, function space), the class of λ_{QL0} *types*, with typical element T, provides constructions for dependent product types, dependent sum (existential) types and type family application [13], specific to DML style programming [21] with dependent types:

$$T ::= \cdots \mid \Pi x : I.T \mid \Sigma x : I.T \mid T[i]$$

where I is an index sort and i is an index term.

Let $\mathbf{List}\langle T\rangle[n]$ denote the type family of lists of length n with elements of type T. For building up lists with elements of type T we use constructors $\mathbf{nil}\langle T\rangle$ and $\mathbf{cons}\langle T\rangle$, whose types are $\mathbf{nil}\langle T\rangle : \mathbf{List}\langle T\rangle[0]$ and

$$\mathbf{cons}\langle T\rangle : \Pi x_m : \mathbf{N}.\Pi x : [0..x_m]. \\ T \to \mathbf{List}\langle T\rangle[x] \to \mathbf{List}\langle T\rangle[x +_{[0..x_m]} 1]$$

Remark 2.5: We write application for Π types using square brackets. Note that, if $t_h : T$ and $t_t : \mathbf{List}\langle T\rangle[n]$ is a list of length $n < max$ then $\mathbf{cons}\langle T\rangle[max][n](t_h)(t_t) : \mathbf{List}\langle T\rangle[n+1]$, as expected. However, if $n = max$ then $\mathbf{cons}\langle T\rangle[max][n](t_h)(t_t) : \mathbf{List}\langle T\rangle[n]$, which means that in this case one element is lost at execution time. For simplicity,

in the rest of the paper, we assume that when $n = max$ the element t_h is not added in the front of the list t_t, i.e., if $n = max$ then $\mathbf{cons}\langle T\rangle[max][n](t_h)(t_t) = t_t$ (the element t_h is simply lost). In such cases the assumption could be that the computer which evaluates the operation $\mathbf{cons}\langle T\rangle[max][n](t_h)(t_t)$ cannot store lists of length $n > max$. Similarly, $n_1 +_{[i..j]} n_2$ yields j whenever $n_1 + n_2 \geq j$.

In the QL0 example program presented in Section III, we only use the symbol for (unbounded) summation $+$. The bounds (i and j) of summation operations ($+_{[i..j]}$) can be easily inferred from the types of the variables involved. For example, the type of variable w_{file} is declared in module $Server$ as $\Sigma x_n : [0..3].\mathbf{List}\langle[1..4]\rangle[x_n]$ (see Notation 2.4). Variable w_{file} stores a list containing at most 3 natural numbers in the range $[1..4]$. Therefore, in the assignment ($w_{file} :=$ let $x_n = (w_{file}.1) + 1$ in $\cdots$) the operator $+$ should behave the same as $+_{[0..3]}$.

C. Evaluation Mechanism

The evaluation mechanism of $\mathcal{L}_{PEP}$ programs is described in [19]. The development of a type checker and an executable interpreter for the language QL0 is in progress. The language QL0 relies on the same evaluation mechanism as $\mathcal{L}_{PEP}$. Most QL0 constructs behave in the same way as the corresponding $\mathcal{L}_{PEP}$ constructs; only the constructs denoting operations on basic sorts (e.g., $+_{[i..j]}$) and the operations on lists of bounded lenght (e.g., $\mathbf{cons}\langle T\rangle[max][n]$) described in Section II-B are specific to (λ_{QL0} and) QL0.

The execution of QL0 programs starts with an initialization phase, in which all modules execute their initialization statements. Next, the execution proceeds in a loop where each iteration comprises a scheduling phase followed by an evaluation phase. In the scheduling phase the guards of all commands are evaluated in order to determine the commands that are enabled (i.e., the commands whose guards evaluates to true) in the current state. The commands are enabled either independently or in rendezvous pairs. Next, in the evaluation phase the activities of the enabled commands are evaluated. Upon the completion of an activity its corresponding updates are executed. When the synchronized activities of a pair of commands involved in a rendezvous are completed, their corresponding updates are performed concurrently and the rendezvous terminates.

A QL0 program is a collection of modules which are executed concurrently. Each module is deterministic [19], meaning that each module can execute at most one command in each possible state.[3] Following [19], the performance of QL0 programs is analyzed formally assuming a Continuous Time Markov Chain (CTMC) model of computation [8]. Note that, although each module is deterministic, commands from different modules may be active concurrently and may proceed at different execution rates. In the performance evaluation phase these execution rates are determined experimentally and are used to construct a CTMC model capturing the performance behavior of the QL0 program under consideration [19].

As stated above, commands from different modules may be active concurrently and may proceed at different execution rates; intuitively, there is a race between activities (occurring in comands) executed by concurrent modules. The execution of an update by a command κ can modify the state of the system, say from state σ to state σ'. The activity of any concurrent command κ' which is (enabled in state σ and) no longer enabled in state σ' will be preempted; in an actual implementation the command κ' could be aborted or merely interrupted. Note that in either case, it is not necessary to maintain information about the remaining lifetime of an activity. This is due to the memoryless property of the exponential distribution - which is specific to the CTMC model [5], [8]. How to construct a CTMC describing the behavior of a QL0 program is explained in Section IV.

[3]The conditions which ensure that a module is deterministic are verified at execution time in the $\mathcal{L}_{PEP}$ implementation available at [24]. The possibility of verifying these conditions statically is also considered in [19].

III. A QL0 Example Program

We present a QL0 version of the $\mathcal{L}_{PEP}$ program given in [19]. The two programs are very similar. Here we focus on aspects that are specific to the QL0 implementation. We note that (unlike the $\mathcal{L}_{PEP}$ version) the QL0 version includes annotations for specifying bounded domains for both primitive (base) types and non-primitive types (lists).

The QL0 program given below implements a cyclic server polling system, based on [6] (a PRISM CTMC model for this system is available at [25]). The system comprises 2 client stations handled by the polling server. The module $Server$ declares one λ_{QL0} function, namely f_{index}. It also declares one variable of non-primitive type w_{file} storing a small list of numbers, and several variables storing numeric values. Modules $Station1$ and $Station2$ declare only numeric variables. We note that the ranges of all variables are bounded. Thus, the number of states of a QL0 program is also bounded.

```
module Server {
  f_index : Πx_n : [0..3].[1..4] → List⟨[1..4]⟩[x_n] → [0..x_n] =
   λx_n : [0..3] . λx : [1..4] . λx_lst : List⟨[1..4]⟩[x_n] .
     match x_lst with
       nil⟨[1..4]⟩ → 0
       cons⟨[1..4]⟩[x_m][x_r](x_h)(x_t) →
           if (x_h = x) then 1
           else let x_i = f_index[x_n − 1](x)(x_t)
                in if (x_i = 0) then 0 else (x_i + 1);
  v_max : [3] = 3; v_s : [1..2] = 1; v_a : [0..2] = 0;
  v_val1cp : [1..4] = 1; v_val2cp : [1..4] = 1;
  v_idx1cp : [0..3] = 0; v_idx2cp : [0..3] = 0;
  w_file : Σx_n : [0..v_max].List⟨[1..4]⟩[x_n] =
       (1, cons⟨[1..4]⟩[v_max][0](3)(nil⟨[1..4]⟩));
```

```
  [a_loop1a] (v_s = 1) and (v_a = 0) →
      ! delay(5) : (v_s := v_s + 1);
  [a_loop1b] (v_s = 1) and (v_a = 0) →
      ! delay(5) : (v_a := 1);
  [a_serve1] (v_s = 1) and (v_a = 1) →
      ! (f_index [w_file.1] (v_val1) (w_file.2)) :
      (v_a := 2)&(v_val1cp := v_val1)&(v_idx1cp := v_idx1);
  [] (v_s = 1) and (v_a = 2) and (v_idx1cp = 0) →
      w_file :=
            let x_n = (w_file.1) + 1 in
            (x_n, cons⟨[1..4]⟩[v_max][x_n](v_val1cp)(w_file.2)) :
       (v_s := v_s + 1)&(v_a := 0);
  [] (v_s = 1) and (v_a = 2) and (v_idx1cp > 0) → skip :
      (v_s := v_s + 1)&(v_a := 0);
  [a_loop2a] (v_s = 2) and (v_a = 0) → ! delay(5) : (v_s := 1);
  [a_loop2b] (v_s = 2) and (v_a = 0) → ! delay(5) : (v_a := 1);
  [a_serve2] (v_s = 2) and (v_a = 1) →
      ! (f_index [w_file.1] (v_val2) (w_file.2)) :
      (v_a := 2)&(v_val2cp := v_val2)&(v_idx2cp := v_idx2);
  [] (v_s = 2) and (v_a = 2) and (v_idx2cp = 0) →
      w_file :=
            let x_n = (w_file.1) + 1 in
            (x_n, cons⟨[1..4]⟩[v_max][x_n](v_val2cp)(w_file.2)) :
       (v_s := 1)&(v_a := 0);
  [] (v_s = 1) and (v_a = 2) and (v_idx2cp > 0) → skip :
      (v_s := 1)&(v_a := 0);
}
module Station1 {
  v_s1 : [0..1] = 0; v_val1 : [1..4] = 1; v_idx1 : [0..3] = 0;
  [] (v_s1 = 0) → delay(100) : (v_s1 := 1);
  [a_serve1] (v_s1 = 1) and (v_val1 < 4) →
      ? v_idx1 : (v_s1 := 0)&(v_val1 := v_val1 + 1);
  [a_serve1] (v_s1 = 1) and (4 ≤ v_val1) →
      ? v_idx1 : (v_s1 := 0);
}
module Station1loop {
  [a_loop1a] (v_s1 = 0) → passive : nop;
  [a_loop1b] (v_s1 = 1) → passive : nop;
}
module Station2 {
  v_s2 : [0..1] = 0; v_val2 : [1..4] = 1; v_idx2 : [0..3] = 0;
  [] (v_s2 = 0) → delay(100) : (v_s2 := 1);
  [a_serve2] (v_s2 = 1) and (v_val2 < 4) →
          ? v_idx2 : (v_s2 := 0)&(v_val2 := v_val2 + 1);
  [a_serve2] (v_s2 = 1) and (4 ≤ v_val2) →
          ? v_idx2 : (v_s2 := 0);
}
module Station2loop {
  [a_loop2a] (v_s2 = 0) → passive : nop;
  [a_loop2b] (v_s2 = 1) → passive : nop;
}
```

A small list of numbers is stored on the server in variable w_{file}. The function f_{index} determines the position of an item in this list. The list may contain at the most 3 elements. Each client station can transmit a request to the server to find the position of a small natural number ≤ 4 in the list (the position may be $1, 2$ or 3). Each client station stores the searched item in variable v_{vali}, $i = 1, 2$. To serve such a request by client station i, the server engages in a rendezvous with station i. If the searched item is not found, then the server adds it to the list, unless the list already contains 3 elements. In this case the server returns constantly 0.

The program starts by executing the initialization statements. Next, the servers enters a polling loop (implemented based on the commands labelled with action names a_{loopia}, $i = 1, 2$), in which it checks the status of each client station. The value stored in variable v_s encodes the number of the station (1 or 2). Upon the arrival of a request on client station i, the server is notified (this interaction is implemented by the commands labelled with action names a_{loopib}, $i = 1, 2$) and it sets the value of variable v_a to 1. The value stored in variable v_a encodes an action: 0 for polling, 1 for serving (searching an item in the list stored in w_{file}), and 2 for list updating (adding an item to the list, in case it is not already there and the length of the list is < 3).

The server imposes the polling rate by executing the activity delay(5). The client stations simulate the arrival of requests in the system at a much lower rate by executing delay(100). To serve a request by station i, the server engages in a rendezvous with station i, which is accomplished by executing the commands labelled with action names a_{servei}, $i = 1, 2$. During a rendezvous with station i the server evaluates the function call $(f_{index}\ [w_{file}.1]\ (v_{vali})\ (w_{file}.2))$, which searches the item stored in variable v_{vali} in the list stored in variable w_{file}. The result is transmitted to client station i (which receives the result in variable v_{idxi}), and the server sets $v_a := 2$ to initiate the list update operation.[4]

If the searched item was not found, then the server adds it to the list (unless the list already contains 3 elements) by executing the assignment statement $w_{file} :=$ let $x_n = (w_{file}.1) + 1$ in $(x_n, \mathbf{cons}\langle[1..4]\rangle[v_{max}][x_n](v_{valicp})(w_{file}.2))$.[5] Next, the server (sets $v_a := 0$ and thus it) starts a new polling loop, and the interaction continues forever.

IV. Performance Evaluation

We study the performance of QL0 programs following the approach introduced in [19]. In this approach, the ranges of variables must be bounded and formal verification is achieved by translating QL0 programs into corresponding

[4]To avoid using values from a previous request, the server saves the values of v_{vali} and v_{idxi} into variables v_{valicp} and v_{idxicp}, respectively; see the commands labeled with action names a_{servei}, $i = 1, 2$.

[5]This statement is implemented in [19] using a second $\mathcal{L}_{PEP}$ function, named f_{insert}. In QL0 the job of f_{insert} is accomplished with the aid of primitive $\mathbf{cons}\langle\cdot\rangle$, which works as explained in Remark 2.5.

Continuous Time Markov Chains (CTMCs) [8] which are analyzed using the PRISM tool [9].

In ongoing work we develop a tool designed to translate automatically a QL0 program into a corresponding (finite state) PRISM CTMC model. Note that the translation is defined only for QL0 programs in which the ranges of all variables are bounded. Here we explain the main rules of such a translation. For further explanations we refer the reader to [19]. The translation can proceed as follows. Each QL0 module is translated into a corresponding PRISM module, and each QL0 command is translated into a corresponding PRISM command.[6]

Based on type annotations, all QL0 variables with bounded ranges can be translated automatically into corresponding PRISM variables. Consider the initialization statements $v_a : [0..2] = 0$ and $w_{file} : \Sigma x_n : [0..v_{max}].\mathbf{List}\langle[1..4]\rangle[x_n] = \ldots$, occurring in the QL0 program presented in Section III. Performance variables can be handled straightforwardly. For example, the QL0 statement $v_a : [0..2] = 0$ could be translated into the following PRISM declaration: `va : [0..2] init 0.`

QL0 values of non-primitive types can be encoded in the corresponding PRISM CTMC model as explained in [19]. For example, the type of variable w_{file} is $\Sigma x_n : [0..3].\mathbf{List}\langle[1..4]\rangle[x_n]$, which means that w_{file} can store a list of lenght $0, 1, 2$ or 3, containing natural numbers in the range $[1..4]$. The number of distinct m-permutations of n items is $n!/(n-m)!$. Hence, there are $4!/(4-4)! + 4!/(4-3)! + 4!/(4-2)! + 4!/(4-3)! = 1 + 4 + 12 + 24 = 41$ distinct lists of the type $\Sigma x_n : [0..3].\mathbf{List}\langle[1..4]\rangle[x_n]$. Using the Haskell notation [12] these lists are: $[], [1], [2], [3], [4], [1, 2], \ldots, [1, 2, 3], \ldots$. For the purpose of the QL0 to PRISM translation one can assign distinct natural numbers to each list in this sequence: $1, 2, 3, 4, 5, \ldots, 41$. In this codification 1 represents the empty list $[]$, 3 represents the list $[2]$, 5 represents the list $[4]$, etc. Using this encoding, the QL0 statement $w_{file} : \Sigma x_n : [0..v_{max}].\mathbf{List}\langle[1..4]\rangle[x_n] = \ldots$ could be translated into the following PRISM declaration: `wfile : [1..41] init ...`.

Function evaluations are abstracted as rates in the CTMC model corresponding to a QL0 program. For the purpose of formal verification we need to determine the CTMC rates accurately. The duration of an activity with rate r in a CTMC model is exponentially distributed with parameter r (mean $1/r$) [8], [5]. An activity can be performed several times, the duration can be measured for each execution and the rate can be determined as the inverse of the average duration. However, results may not be accurate when the probabilities of the various executions are unknown.

A more accurate approach is used in this study, which does not rely on the estimation of the activity average execution time. Rather, it generates a CTMC where a different rate is attached to each particular function execution. We count the number steps (elementary operations) performed during the execution and we compute accurately the activity rate as the inverse of the number of steps. Note that the number of steps is given by the number of elementary operations which does not depend on the particular execution environment. In the language QL0, an activity is the evaluation of a function expressed in the functional sub-language λ_{QL0}.

The rates of the CTMC corresponding to a QL0 program can be determined experimentally as explained in [19], by executing each λ_{QL0} function occurring in the QL0 program for all possible input arguments in the established ranges. For example, in the QL0 program presented in Section III the function f_{index} takes as arguments a natural number of the type $[1..4]$ and a list of the type $\Sigma x_n : [0..3].\mathbf{List}\langle[1..4]\rangle[x_n]$. As explained above, there are 41 distinct lists of the type $\Sigma x_n : [0..3].\mathbf{List}\langle[1..4]\rangle[x_n]$. Hence, the function f_{index} must be executed $164 = 4 * 41$ times.

In [19] we presented an approach where performance evaluation and formal verification of $\mathcal{L}_{PEP}$ programs can proceed in a systematic manner, but not automatically. More precisely, the solution presented in [19] requires the programmer to generate the lists of argument tuples for all λ_{PEP} functions of an $\mathcal{L}_{PEP}$ program. For example, the arguments for the 164 experiments needed to determine the rate for each possible execution of f_{index} must be generated by the programmer. This is not a difficult programming task, but clearly, we prefer a fully automatic solution.

The big advantage resulting from the use of dependent types in the language QL0 introduced in this paper is given by the fact that these lists of argument tuples can be generated automatically based on the type annotations which are used to describe the bounded ranges of variables and function arguments. For example, the argument tuples for the 164 experiments required to determine the rates corresponding to the function f_{index} could be determined automatically from its type specification $f_{index} : \Pi x_n : [0..3].[1..4] \rightarrow \mathbf{List}\langle[1..4]\rangle[x_n] \rightarrow [0..x_n]$. A Haskell [12] implementation of this solution is in progress.

For the purpose of formal verification the ranges of variables must be bounded: in the language QL0 this can be achieved by using type annotations. By varying the ranges of variables, various experiments can be performed (e.g., in PRISM style [9]). This way, the programmer can identify patterns, trends and anomalies in quantitative results expressing the behavior of QL0 programs.

V. Conclusion and Future Research

This paper investigates the possibility of using techniques specific to functional programming with dependent types [21], [13] in order to obtain a fully automatic version of the

[6] We assign a *passive rate* [7] to the PRISM command that corresponds to a QL0 command executing a receive statement $?\psi$; see Remark 2.3.

software verification approach introduced in [19]. Anyway, much work remains as subject of future research.

We are currently developing a type checker and an executable interpreter for the language QL0 together with a component designed to translate automatically a QL0 program into a corresponding PRISM CTMC model. Prototype versions of these tools will be implemented in Haskell, starting from the $\mathcal{L}_{PEP}$ implementation [24], which is designed with continuations [20]. We also intend to develop a formal specification of the (static and dynamic) semantics of language QL0.

The solution presented in this paper relies on explicit type annotations. In the future, we could investigate the possibility to use techniques for automatically inferring (dependent) type information [15], [18]. However, note that in our approach, type information used in the specification of the performance evaluation experiments should remain under the control of the programmer.

Next, various options for improving the performance of these tools will be considered. One option would be to use (approximate or) statistical model checking [22], which is supported by the PRISM tool [25]. This way, we could formally verify larger QL0 programs (at the expense of generating approximate results for the PRISM properties). We also aim to investigate the possibility of reducing the state space of the CTMC model corresponding to a QL0 program by dividing the set of states into equivalence classes based on the behavior of functions (a technique known as *equivalence partitioning* in software testing [17]).

In this paper we use Continuous Time Markov Chains. Investigating the possibility to adapt the approach presented in this paper to a discrete time model, e.g., by using Discrete Time Markov Chains [8] or Probabilistic Timed Automata [11], remains an objective of future research.

REFERENCES

[1] R. Alur and T. Henzinger, "Reactive Modules," *Formal Methods in System Design*, vol. 15(1), pp. 7–48, 1999.

[2] C. Baier and J.-P. Katoen, *Principles of Model Checking,* MIT Press, 2008.

[3] E.M. Clarke, T.A. Henzinger, H. Veith and R. Bloem, editors, *Handbook of Model Checking,* Springer, 2018.

[4] US Department of Defense, *Reference Manual for the ADA Programming Language,* ANSI-MIL-STD-1815A, 1983.

[5] J. Hillston, *A Compositional Approach to Performance Modelling,* Cambridge University Press, 1996.

[6] O. Ibe and K. Trivedi, "Stochastic Petri Net Models of Polling Systems," *IEEE Journal on Selected Areas in Communications*, vol. 8(9), pp. 1649–1657, 1990.

[7] M. Kwiatkowska, G. Norman, and D. Parker, "Quantitative Analysis with the Probabilistic Model Checker PRISM," *Electronic Notes in Theoretical Computer Science*, vol. 153(2), pp. 5-31, Elsevier, 2005.

[8] M. Kwiatkowska, G. Norman and D. Parker, "Stochastic Model Checking," *Lecture Notes in Computer Science*, vol. 4486, pp. 220–270, 2007.

[9] M. Kwiatkowska, G. Norman and D. Parker, "PRISM 4.0: verification of probabilistic real-time systems," *Lecture Notes in Computer Science*, vol. 6806, pp. 585–591, 2011.

[10] M. Loreti and J. Hillston, "Modeling and Analysis of Collective Adaptive Systems with CARMA and its Tools," *Lecture Notes in Computer Science*, vol. 9700, pp. 83-119, 2016.

[11] G. Norman, D. Parker and J. Sproston, "Model Checking for Probabilistic Timed Automata," *Formal Methods in System Design*, 43(2):164-190, Springer, 2013.

[12] S. Peyton Jones and J. Hughes, "Report on the Programming Language Haskell 98: a Non-Strict Purely Functional Language," 1999; available at `http://www.haskell.org`.

[13] B. Pierce, *Advanced Topics in Types and Programming Languages,* MIT Press, 2005.

[14] Z. Ren, "Combining Type Checking with Model Checking for System Verification," Ph.D. Thesis, Boston University, 2017.

[15] P.M. Rondon, M. Kawaguchi and R. Jhala, "Liquid Types," *Proc. PLDI 2008*, pp. 159-169, 2008.

[16] G. Roşu and T.F. Şerbănuţă, "An Overview of the K Semantic Framework," *Journal of Logic and Algebraic Programming*, vol. 79(6), pp. 397–434, 2010, `http://kframework.org`.

[17] I. Sommerville, *Software Engineering,* 10th Ed, Pearson, 2016.

[18] N. Swamy, C. Hritcu, et al., "Dependent Types and Multi-Monadic Effects in F*," *Proc. POPL 2016*, pp. 256–270, 2016.

[19] E.N. Todoran, "An Approach to Performance Evaluation Programming," *Proceedings of IEEE 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2017)*, pp. 320–329, IEEE Computer Press, 2017.

[20] E.N. Todoran and N. Papaspyrou, "Concurrency Semantics in Continuation-Passing Style," *Fundamenta Informaticae*, vol. 153(1–2), pp. 125-146, 2017.

[21] H. Xi and F. Pfenning, "Dependent Types in Practical Programming," *Proc. POPL 1999*, pp. 214–227, 1999.

[22] H. Younes, M. Kwiatkowska, G. Norman and D. Parker, "Numerical vs. Statistical Probabilistic Model Checking," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 8(3), pp. 216–228, 2006.

[23] F. Zervoudakis, D.S. Rosenblum, S.G. Elbaum and A. Finkelstein, "Cascading Verification: an Integrated Method for Domain-Specific Model Checking," *Proceedings of ESEC/ACM SIGSOFT FSE 2013*, pp. 400-410, 2013.

[24] `http://ftp.utcluj.ro/pub/users/gc/eneia/synasc17`

[25] PRISM website: `www.prismmodelchecker.org`