A Logic for Real–Time Systems Specification, Its Algebraic Semantics, and Equational Calculus

Gabriel A. Baum[1], Marcelo F. Frias[2], and Thomas S.E. Maibaum*[3]

[1] Universidad Nacional de La Plata, LIFIA, Departamento de Informática.
gbaum@sol.info.unlp.edu.ar
[2] Universidad de Buenos Aires, Departamento de Computación, and
Universidad Nacional de La Plata, LIFIA, Departamento de Informática.
mfrias@sol.info.unlp.edu.ar
[3] Imperial College, 180 Queen's Gate, London SW7 2BZ, U.K.
tsem@doc.ic.ac.uk

Abstract. We present a logic for real time systems specification which is an extension of first order dynamic logic by adding (a) arbitrary atomic actions rather than only assignments, (b) variables over actions which allow to specify systems partially, and (c) explicit time. The logic is algebraized using closure fork algebras and a representation theorem for this class is presented. This allows to define an equational (but infinitary) proof system for the algebraization.

1 Introduction

The motivation for this work is the need to describe *industrial processes* as part of a project for a Brazilian telecommunications company. We want to be able to give formal descriptions of such processes so as to be able to analyze such descriptions. For example, we want to be able to calculate critical paths for tasks in processes, throughput times of processes, etc. We also want to demonstrate correctness of process descriptions in relation to their specifications (where this is appropriate), derive implementations of process specifications in terms of the available concrete apparatus in the factory, validate (using formal techniques) an implementation against its abstract description, and so on. Available languages for describing processes are unsuitable for various reasons, most having to do with the nature of the formalization of such processes being used in the project.

The method used in the project for describing processes (*the method*) is based on the ideas presented in [11]. This method sees the world as being modeled in terms of two (and only two) kinds of entities: products and processes. A *product* is a description of an entity in the real world (*a referent*) in terms of measurable attributes. (Here, we use *measure* and *measurable* in the traditional sense of science and engineering. See [2][11][14].) A *product instance* is characterized by the

* The third author would like to thank the EPSRC(UK), CNPq(Brasil), Imperial College, LMF-DI/PUC-RJ and The Royal Academy of Engineering for their financial support during the conduct of this research.

A.M. Haeberer (Ed.): AMAST'98, LNCS 1548, pp. 91–105, 1998.

values (measures) associated with its attributes (and, implicitly, by the theory of the product, i.e., the defined relationships between the potential measured values of its attributes). Hence, such a product instance may be seen as a *model*, in the sense of logic, of the product. We may see products as being characterized by data types in first order logic, for example.

The distinguishing characteristic of products is that they exist 'independently' at an instant in time, where *time* is used here in its normal scientific sense. (Independence here means that a product is defined without recourse to any other referent or only in terms of other (sub)products. Products of the former kind are called *atomic* products.) In fact, all products have a time attribute whose value in a product instance indicates the time instant at which the values of the attributes were (co)determined, presumably by some appropriate measurement procedures. On the other hand, processes are distinguished entities which do not exist at a time instant, but which have time duration. Further, processes are not independently definable, but are defined in terms of their input and output products.

Processes also model entities of the real world and again are defined in terms of attributes. The method imposes a very restrictive notion of process, namely one in which *all processes have a single input and a single output.* (The reasons for this restriction need not detain us here, except to say that they are methodologically very well motivated. The restriction clearly will have a profound influence on the nature of the language we define below.) Distinguished attributes of a process include the transfer function(s) 'computed' by the process (i.e., how the input product is transformed into the output product), upper and lower bounds on the time taken for the process to execute, a flag indicating whether the process is 'enabled', and so on. The transfer function may be described in terms of an underlying state machine used to organize phases of the process being defined and to 'sense' important external state information required to control the execution of the process. Like products, processes may be defined in terms of 'sub-processes' and we now turn to this language of processes, as its formalization is the subject matter of the paper.

We will use an analogy with conventional sequential programming languages to motivate the nature of the formalization. Consider such a programming language. The programs in the language are constructed from basic commands (usually just assignment) and various control structures. The programming language data types are used to model the inputs and outputs of programs. Let us focus on a program that exhibits simple input/output behavior. We realize this behavior by executing the program on some machine (a real machine for a low level language and an 'abstract' machine for a high level language). Hence, we can see an analogy between inputs/outputs and products and between programs and processes. Both programs and processes are intended to model entities that define families of executions on the machine used to execute the program/process. This is exactly how we want to understand processes, i.e., as defining a class of potential executions over some (abstract) machine.

The following questions must be answered, amongst others, in order to make the analogy more exact: What is the nature of the abstract machine over which processes are defined? What 'data types' are allowed as inputs and outputs of processes? What 'control structures' may be used? What do we mean by 'execution' of a process? (Obvious further questions include: How do we specify required processes? What do we mean by 'refinement' and how do we derive refinements or prove their correctness?)

As to the first of these questions, we do not envisage a single abstract machine which will underpin all potential processes. Rather, we assume that our abstract machine is provided by an object, in the sense of object oriented programming. Such an object has

- a set of internal states;
- a set of methods, with appropriate input and output parameters, that it can execute and which change the internal state;
- a set of potential behaviors that it can exhibit, with the behavior being exhibited 'chosen' by the program being executed.

So, the purpose of a process, like a program, is to 'choose' a particular behavior allowed by the object (our abstract machine). Of course, the object itself is used, in our case, to model the basic capabilities of the organization whose industrial processes are being modeled or prescribed. These basic capabilities may be those of machines (computers, presses, conveyor belts, etc.), or people (programmers, hardware engineers, salesmen, managers, etc.), or even (sub)organizations. The aim of the exercise is to choose from all the potential behaviors (jointly) exhibitable by this abstract machine those which have the appropriate characteristics (i.e., manufacturing a product with appropriate quality and other characteristics and in a dependable manner).

We should add here two important comments about our underlying 'object'. Firstly, such a complex underlying object may itself be built in a structured manner from sub-objects by using standard object oriented structuring methods. See [3][4] for a formal account of this. Secondly, there are lacuna in object oriented programming methods to do with exactly what we are attempting here, i.e., defining a particular subclass of behaviors from those potentially exhibitable by the object. In object oriented programming, this problem of defining *threads of computation* over objects is usually overcome by defining a 'system object' that drives the choice of desirable behaviors. Different applications over the same object base then require different 'system objects' to choose the different behaviors.

The methods of this abstract machine represent the atomic 'machine executable' processes from which our industrial processes will have to be built. Consider the example below, presented using the diagrammatic notation of the method.

The two entities we have discussed, products and processes, are denoted, respectively, by arrows and boxes. (We use lower case letters for process names and upper case letters for products.) Each product is represented in terms of its attributes and each execution of the process will assign to each of these attributes

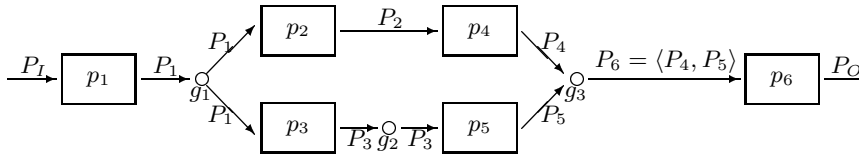

Fig. 1. A complex process example.

specific values from the appropriate domains. Each process is an atomic method from the underlying object. There is a third kind of entity in the diagram that we have not yet discussed. This is the *gate*, which is an artifact of the method used basically for two purposes: i) helping to enforce the single input, single output regime of processes, and ii) acting as guards on processes so as to control the computation. For example, g_1 copies the output product P_1 to create single inputs for each of p_2 and p_3. The gate g_2, on the other hand, is a guard which is intended to stop progress of P_3 until, for example, p_2 has terminated (or even forcing p_5 to wait for an external event, like turning on the machine). Gate g_3 is used to create the single input to process p_6 from the outputs P_4 and P_5 of processes p_4 and p_5, respectively. P_6 is simply a tuple of products synchronized in time. (The single input mechanism here is used to enforce a unique time for the process to be initiated with the required input. The method uses the unique initiation and termination times of processes to attain a specific notion of well definedness and the single input and single output regime is an aid to accomplishing this.) As we see, we need the following constructs in our language:

- sequential composition of processes;
- parallel composition of processes (with parallelism being interpreted as 'don't care' parallelism, in the sense that it is potential parallelism of which an implementer may take advantage);
- nondeterministic choice among processes;
- guards for processes (which may be combined with copying and 'restructuring' of products);
- a loop construct to allow us to define iterative processes with a guard to control the number of iterations.

The language is formalized by extending first order dynamic logic with a parallel combinator and the ability to express real time constraints. (The only construct above not used in dynamic logic is the parallel combinator.) The semantics of dynamic logic uses a notion of transition system that is used to represent the underlying abstract machine capable of executing the atomic processes. The logic is extended with variables over processes so that we can specify abstractly the processes we are interested in building. There is a notion of refinement associated with such specifications, allowing us to demonstrate that a process satisfies its specification. Finally, we demonstrate, using techniques

developed in [7][8], how to algebraize this logic and thus obtain an equational proof system for our process formalism. In order to algebraize the logic we will use *omega closure fork algebras* (ω-CFA). These algebras are extensions of relation algebras [13] with three new operators, a pairing operator called *fork* [6][5], a *choice* operator [12] and the Kleene star. A consequence preserving function mapping formulas of the logic to equations in the language of ω-CFA will be defined. We will also present a representation theorem which, together with the mapping, will allow to reason equationally about properties of the logic.

The paper is organized as follows: in Section 2 we will present a first order formalization of objects. In Section 3 will be presented the logic we propose for specifying and reasoning about the properties of processes. In Section 4 we introduce the class of omega closure fork algebras. In Section 5 we present the algebraization. Finally, in Section 6 we present our conclusions about this work.

2 Objects

The first problem we confront when trying to formalize these concepts is that of characterizing the 'abstract machine' over which our processes will be defined. These processes are meant to use the underlying capabilities of the organization, as represented by the behaviors displayed by individual components within the organization. (Such individual components may be people, groups, manufacturing machines, etc.) These behaviors are organized (at least in some abstract sense) into a joint behavior which IS our 'abstract machine'. In the last decade, we have learned to organize such behaviors in terms of concepts used in object oriented programming. Objects are characterized by the data structures that are maintained by the object (seen in terms of the different states of the object) and the methods (which we call actions below) that may be executed by the object and which may change its state.

Hence, we will assume as given some object (which may be very complex and built as a system from less complex components [3]), which represents the potential behaviors of the organization as an abstract machine. This object will represent the actions/methods, state variables, external events/actions and some prescription of allowed behaviors from which individual processes must be built. (We note again the analogy between computers and programs, on the one hand, and the object/abstract machine and processes, on the other. Our processes will be used to define specific classes of behaviors in which we are interested, our required processes, from the very large class allowed by the object.) The definitions below give a somewhat non standard account of objects in terms of the underlying transition system defining the object's allowed behaviors. However, the standard parts of such descriptions (i.e., methods, state variables, etc) are easily distinguishable.

Definition 1. An *object signature* is a pair $\langle A, \Sigma \rangle$ in which $\Sigma = \langle S, F, P \rangle$ is a many-sorted first-order signature with set of sorts S, set of function symbols F and set of predicate symbols P. Among the sorts, we will single out one sort

called the *time sort*, denoted by T. A is a set of *action symbols*. To each $a \in A$ is associated a pair $\langle s_1, s_2 \rangle \in (S^*)^2$ called its *arity*. We will denote the input arity of a by $ia(a)$ and the output arity of a by $oa(a)$.

Definition 2. Given an object signature $\mathcal{S} = \langle A, \langle S, F, P \rangle \rangle$, an *object structure for* $\mathcal{S}$ is a structure $\mathcal{A} = \langle \mathbf{S}, \mathbf{A}, \mathbf{F}, \mathbf{P} \rangle$ in which $\mathbf{S}$ is an S-indexed family of nonempty sets, where the set T is the T-th element in $\mathbf{S}$. In general, the set corresponding to sort s will be denoted by s. $\mathbf{A}$ is an A-indexed family of binary relations satisfying the typing constraints of symbols from A, i.e., if $ia(a) = s_1 \dots s_m \in S^*$ and $oa(a) = {s'}_1 \dots {s'}_n \in S^*$, then $a^{\mathcal{A}}$ (as we will denote the a-th element from $\mathbf{A}$) is contained in $(\mathsf{s}_1 \times \cdots \times \mathsf{s}_m) \times (\mathsf{s'}_1 \times \cdots \times \mathsf{s'}_n)$. To each $f : s_1 \dots s_k \to s$ in F is associated a function $f^{\mathcal{A}} : \mathsf{s}_1 \times \cdots \times \mathsf{s}_k \to \mathsf{s} \in \mathbf{F}$. To each p of arity $s_1 \dots s_k$ in P is associated a relation $p^{\mathcal{A}} \subseteq \mathsf{s}_1 \times \cdots \times \mathsf{s}_k \in \mathbf{P}$.

Regarding the domain T associated to the time sort T, we will not deepen on the different possibilities for modeling time, but will rather choose some adequate (with respect to the application we have in mind) representation, as for instance the fields of rational or real numbers, extended with a maximum element ∞. We will distinguish some constants, as 0, ϵ, etc.

3 The Logic, the Relational Variables, and the Time

We will extend a standard notation for specifying and reasoning about programs, namely dynamic logic. What we want to do is define processes/programs over our objects which reflect the intuitive model outlined in the introduction. Dynamic logic starts with basic actions and constructs programs by using certain combinators. The usual basic action is assignment, but we will replace this with the actions of the underlying object. The basic actions will be represented by binary relations. The input and output domains of such relations will be tuples of state variables or a choice of a set of state variables, thus reflecting the single input, single output idea of processes. The combinators are also extended with one to allow us to express (potential) parallelism of processes (defined via the intersection operator for binary relations).

Another important aspect of processes, as we wish to define them, is the real–time aspect. In defining processes, we often want to reason about time: throughput time, critical paths, optimization of processes. This requires that we are able to deal with reasoning about time within the formalism. We adapt a real–time logic developed in [1] which presents an extension of the logic presented in [3]. Each basic action is supplemented with a specification of lower and upper time bounds for occurrences of that action. These bounds may have various interpretations, amongst which we have the following: the lower bound is interpreted as the minimum time that must pass before which the action's effects are committed to happen and the upper bound gives a maximum time by which the action's effects are committed to happen. Specifications of processes will also have associated lower and upper bounds, and refinements will be expected to provably meet these bounds.

In this section we will present the *Product/Process Modeling Logic* (P/PML). Consider the formula $\varphi(x) := [x\,A\,x]\beta(x)$ where A is an action term (a binary or n-ary relation) and the notation $[x\,A\,x]\beta$ means that "all executions of action A establish the property β". According to our previous discussion about processes and products, we read φ as stating that β is a truth of the system A, then proving the truth of φ can be seen as the *verification* of the property β in the system described by A. Opposed to the previous view, is the notion of an *implicit specification* of a system, in which A is not a ground term, but rather may contain some relational variables that represent subsystems not yet fully determined. In what follows we will denote by *RelVar* the set of relational variables $\{ R, S, T, \dots \}$.

Definition 3. Given an object signature $\mathcal{S} = \langle A, \langle S, F, P \rangle \rangle$, the sets of *relational terms* and *formulas* on $\mathcal{S}$ are the smallest sets $RT(\mathcal{S})$ and $For(\mathcal{S})$ such that

1. $a \in RT(\mathcal{S})$ for all $a \in A \cup RelVar \cup \{ 1'_t : t \in S^* \}$.
2. If $r \in RT(\mathcal{S})$ and $ia(r) = oa(r)$, then $r^* \in RT(\mathcal{S})$. We define $ia(r^*) = oa(r^*) = ia(r)$.
3. If $r, s \in RT(\mathcal{S})$, $ia(r) = ia(s)$ and $oa(r) = oa(s)$, then $r+s \in RT(\mathcal{S})$ and $r \cdot s \in RT(\mathcal{S})$. We define $ia(r+s) = ia(r \cdot s) = ia(r)$ and $oa(r+s) = oa(r \cdot s) = oa(r)$.
4. If $r, s \in RT(\mathcal{S})$ and $oa(r) = ia(s)$, then $r;s \in RT(\mathcal{S})$. We define $ia(r;s) = ia(r)$ and $oa(r;s) = oa(s)$.
5. If $\alpha \in For(\mathcal{S})$ is quantifier free and has free variables $x_1, \dots, x_n$ with x_i of sort s_i, then $\alpha? \in RT(\mathcal{S})$ and $ia(\alpha?) = oa(\alpha?) = s_1 \dots s_n$.
6. The set of first-order atomic formulas on the signature Σ is contained in $For(\mathcal{S})$.
7. If $\alpha, \beta \in For(\mathcal{S})$, then $\neg\alpha \in For(\mathcal{S})$ and $\alpha \vee \beta \in For(\mathcal{S})$..
8. If $\alpha \in For(\mathcal{S})$ and x is an individual variable of sort s, then $(\exists x : s)\,\alpha \in For(\mathcal{S})$.
9. If $\alpha \in For(\mathcal{S})$, $t \in RT(\mathcal{S})$ with $ia(t) = s_1 \dots s_m$ and $oa(t) = {s'}_1 \dots {s'}_n$, $\vec{x} = x_1, \dots, x_m$ with x_i of sort s_i, $\vec{y} = y_1, \dots, y_n$ with y_i of sort ${s'}_i$ and l, u are variables of sort T, then $\left\langle \vec{x}\ {}_l t^u\ \vec{y} \right\rangle \alpha \in For(\mathcal{S})$.

Definition 4. Let $R \in RT(\mathcal{S})$ with $ia(R) = s_1 \dots s_m$ and $oa(R) = {s'}_1 \dots {s'}_n$, $\vec{x} = x_1, \dots, x_m$ with x_i of sort s_i, $\vec{y} = y_1, \dots, y_n$ with y_i of sort ${s'}_i$, and l, u variables of sort T. An expression of the form $\vec{x}\ {}_l R^u\ \vec{y}$ is called a *timed action term*.

We will assume that a lower and an upper bound are assigned to atomic actions, namely $l_a \in \mathsf{T}$ and $u_a \in \mathsf{T}$ for each action $a \in A$. From the bounds of the atomic actions it is possible to define bounds for complex actions in a quite natural way.

Definition 5. Let $\mathcal{S}$ be an object signature. The functions l and u from $RT(\mathcal{S}) \cup For(\mathcal{S})$ to T are defined as follows[1]:

1. If $a \in A$, then $\mathsf{l}(a) = l_a$ and $\mathsf{u}(a) = u_a$.
2. If $R = X \in RelVar$, then $\mathsf{l}(X) = 0$ and $\mathsf{u}(X) = \infty$.
3. If $R = 1'_t$, with $t \in S^*$, then $\mathsf{l}(R) = 0$ and $\mathsf{u}(R) = \epsilon$ (ϵ being a constant of sort T).
4. If $R = S^*$, then $\mathsf{l}(R) = 0$ and $\mathsf{u}(R) = \infty$.
5. If $R = S+T$, then $\mathsf{l}(R) = \min\{\mathsf{l}(S), \mathsf{l}(T)\}$ and $\mathsf{u}(R) = \max\{\mathsf{u}(S), \mathsf{u}(T)\}$.
6. If $R = S \cdot T$, then $\mathsf{l}(R) = \max\{\mathsf{l}(S), \mathsf{l}(T)\}$ and $\mathsf{u}(R) = \max\{\mathsf{u}(S), \mathsf{u}(T)\}$.
7. If $R = S;T$, then $\mathsf{l}(R) = \mathsf{l}(S)$ and $\mathsf{u}(R) = \mathsf{u}(S) + \mathsf{u}(T)$.
8. If $R = \alpha?$ with $\alpha \in For(\mathcal{S})$ quantifier free and with free variables $\vec{x}$, $\mathsf{l}(R) = \mathsf{l}(\alpha)$ and $\mathsf{u}(R) = \mathsf{u}(\alpha)$.
9. If $\alpha = p(t_1, \ldots, t_k)$, then $\mathsf{l}(\alpha) = l_p \in \mathsf{T}$ and $\mathsf{u}(\alpha) = u_p \in \mathsf{T}$, with $l_p \leq u_p$.
10. If $\alpha = \neg\beta$, then $\mathsf{l}(\alpha) = \mathsf{l}(\beta)$ and $\mathsf{u}(\alpha) = \mathsf{u}(\beta)$.
11. If $\alpha = \beta \mathbf{op} \gamma$ with $\mathbf{op} \in \{\vee, \wedge, \rightarrow\}$, then $\mathsf{l}(\alpha) = \min\{\mathsf{l}(\beta), \mathsf{l}(\gamma)\}$ and $\mathsf{u}(\alpha) = \max\{\mathsf{u}(\beta), \mathsf{u}(\gamma)\}$.
12. If $\alpha = \left\langle \vec{x}\ {}_l R^u\ \vec{y} \right\rangle \beta$, then $\mathsf{l}(\alpha) = \mathsf{l}(R)$ and $\mathsf{u}(\alpha) = \mathsf{u}(R) + \mathsf{u}(\beta)$.

Given a set of sorts $S = \{s_1, \ldots, s_k\}$ and domains $\mathsf{S} = \{\mathsf{s}_1, \ldots, \mathsf{s}_k\}$ for these sorts, by a *valuation of the individual variables of sort* s_i we refer to a function $\nu : IndVar_{s_i} \rightarrow \mathsf{s}_i$. A valuation of the relational variables is a function $\mu : RelVar \rightarrow \mathcal{P}(\mathsf{S}^* \times \mathsf{S}^*)$.

Definition 6. Given a valuation of the individual variables ν and an array of variables $\vec{x} = x_1, \ldots, x_n$, by $\nu(\vec{x})$ we denote the tuple $\langle \nu(x_1), \ldots, \nu(x_n) \rangle$.

Let $\mathcal{A}$ be an object structure and μ a valuation of the relational variables. Given valuations of the individual variables ν and ν' and a timed action term $\vec{x}\ {}_l R^u\ \vec{y}$, by $\nu \left(\vec{x}\ {}_l R^u\ \vec{y} \right) \nu'$ we denote the fact that: $\left\langle \nu(\vec{x}), \nu'(\vec{y}) \right\rangle \in R^{\mathcal{A}}_{\mu}$ (the denotation of the relational term R, formally defined in Def. 7), for every variable z not occurring in $\vec{y}$, $\nu'(z) = \nu(z)$, and, $\nu(l) \leq \mathsf{l}(R)$ and $\nu(u) \geq \mathsf{u}(R)$.

The semantics of formulas is now defined relative to valuations of individual variables and relational variables. In the following definition, the notation $\mathcal{A} \models_{P/PML} \alpha[\nu][\mu]$, is to be read "*The formula* α *is satisfied in the object structure* $\mathcal{A}$ *by the valuations* ν *and* μ".

Definition 7. Let us have an object signature $\mathcal{S} = \langle A, \langle S, F, P \rangle \rangle$ and an object structure $\mathcal{A} = \langle \mathbf{S}, \mathbf{A}, \mathbf{F}, \mathbf{P} \rangle$. Let ν be a valuation of individual variables and μ a valuation of relational variables. Then:

1. If $a \in A$ then $a^{\mathcal{A}}_{\mu}$ is the element with index a in $\mathbf{A}$.
2. If $R \in RelVar$, then $R^{\mathcal{A}}_{\mu} = \mu(R)$.
3. If $R = 1'_t$ with $t = s_1 \ldots s_k$, $R^{\mathcal{A}}_{\mu} = \{\langle \langle a_1, \ldots, a_k \rangle, \langle a_1, \ldots, a_k \rangle \rangle : a_i \in \mathsf{s}_i\}$.

[1] We will only consider quantifier-free formulas, since these are the ones used for building actions of the form $\alpha?$.

4. If $R = S^*$, with $S \in RT(\mathcal{S})$, then $R_\mu^{\mathcal{A}}$ is the reflexive-transitive closure of the binary relation $S_\mu^{\mathcal{A}}$.
5. If $R = S+T$, with $S,T \in RT(\mathcal{S})$, then $R_\mu^{\mathcal{A}} = S_\mu^{\mathcal{A}} \cup T_\mu^{\mathcal{A}}$.
6. If $R = S \cdot T$, with $S,T \in RT(\mathcal{S})$, then $R_\mu^{\mathcal{A}} = S_\mu^{\mathcal{A}} \cap T_\mu^{\mathcal{A}}$.
7. If $R = S;T$, with $S,T \in RT(\mathcal{S})$, then $R_\mu^{\mathcal{A}}$ is the composition of the binary relations $S_\mu^{\mathcal{A}}$ and $T_\mu^{\mathcal{A}}$.
8. If $R = \alpha?$ with $\alpha \in For(\mathcal{S})$ quantifier free and with free variables $\vec{x} = x_1, \ldots, x_n$, then $R_\mu^{\mathcal{A}} = \left\{ \left\langle \nu(\vec{x}), \nu(\vec{x}) \right\rangle : \mathcal{A} \models_{P/PML} \alpha[\nu][\mu] \right\}$.
9. If $\varphi = p(t_1, \ldots, t_n)$ with $p \in P$, $\mathcal{A} \models_{P/PML} \varphi[\nu][\mu]$ if $\langle t_{1\nu}^{\mathcal{A}}, \ldots, t_{n\nu}^{\mathcal{A}} \rangle \in p^{\mathcal{A}}$.
10. If $\varphi = \neg\alpha$, then $\mathcal{A} \models_{P/PML} \varphi[\nu][\mu]$ if $\mathcal{A} \not\models_{P/PML} \alpha[\nu][\mu]$.
11. If $\varphi = \alpha \vee \beta$, $\mathcal{A} \models_{P/PML} \varphi[\nu][\mu]$ if $\mathcal{A} \models_{P/PML} \alpha[\nu][\mu]$ or $\mathcal{A} \models_{P/PML} \beta[\nu][\mu]$.
12. If $\varphi = (\exists x : s)\alpha$, then $\mathcal{A} \models_{P/PML} \varphi[\nu][\mu]$ if there exists $a \in \mathsf{s}$ such that $\mathcal{A} \models_{P/PML} \alpha[\nu_x^a][\mu]$ (ν_x^a, as usual, denotes the valuation that agrees with ν in all variables but x, and satisfies $\nu_x^a(x) = a$).
13. If $\varphi = \left\langle \vec{x}\ {}_l R^u\ \vec{y} \right\rangle \alpha$, then $\mathcal{A} \models_{P/PML} \varphi[\nu][\mu]$ if there exists a valuation ν' such that $\nu \left(\vec{x}\ {}_l R^u\ \vec{y} \right) \nu'$ and $\mathcal{A} \models_{P/PML} \alpha[\nu'][\mu]$.

Example 1. The example shows how the real-time features of the specification language (P/PML) play a decisive role in the election of the implementations of processes. A manufacturer of candy vending machines wants to manufacture machines with the following characteristics. If the machine has candy, then, after money has been deposited, at most a time K_1 passes before candy is delivered. If the machine is empty, then at most a time K_2 can pass before the transaction is finished. If the machine can be fully replenished in time to meet the K_2 upper bound, then it should be replenished, otherwise, the money should be given back.

Let us model the part of the behavior of the machine after money has been introduced and until candy has been delivered or the money was given back[2].

$$\begin{aligned}
&\forall m, l, u, x\, (\$_in?(m) = \mathsf{t} \wedge x = \#candy(m) > 0 \wedge u \geq K_1 \wedge l = 0 \\
&\quad \Rightarrow [m_l\, VM\, {}^u m]\, (\#candy(m) = x - 1 \wedge delivered?(m) = \mathsf{t})) \\
&\quad \forall m, l, u\, (\$_in?(m) = \mathsf{t} \wedge \#candy(m) = 0 \wedge u \geq K_2 \wedge l = 0 \\
&\Rightarrow [m_l\, VM\, {}^u m]\, (\#candy(m) = max_candy - 1 \wedge delivered?(m) = \mathsf{t})) \\
&\quad \forall m, l, u\, (\$_in?(m) = \mathsf{t} \wedge \#candy(m) = 0 \wedge u < K_2 \wedge l = 0 \\
&\quad \Rightarrow [m_l\, VM\, {}^u m]\, (delivered?(m) = \mathsf{f} \wedge money_back?(m) = \mathsf{t}))
\end{aligned}$$

If the manufacturer believes that a consumer can wait for candy 3 minutes without loosing his patience, then K_2 can be set to 3 minutes in the specification.

[2] Given an object m of the class "*vending machine*", the method $\$\_in?$ tests if money has been deposited. Method $\#candy$ retrieves the amount of candy left in the machine. The method *delivered*? tests if candy has been delivered, and *money_back*? tests if the money has been returned to the customer. The constant *max_candy* stands for the maximum amount of candy the machine can contain. A formal specification of the class is not given by lack of space.

Let us assume that as a constraint, this part of the machine must be built using some of the following processes:

- *RETURN PRODUCT* (that returns candy provided the machine is not empty. Its lower time bound is 0 and the upper time bound is 3 seconds).
- *REPLENISH* (that fully replenishes the machine. Its lower time bound is 0 and the upper time bound will be discussed later).
- *RETURN MONEY* (that gives the customer its money back. Its lower time bound is 0 and the upper time bound is 3 seconds)

If the machine is to be placed inside a convenience store, then as soon as the machine is emptied a clerk will replenish it, and therefore, a reasonable upper time bound for the replenishing action might be 2 minutes. Then, the following action shows a feasible implementation:

$$(\#candy(m) > 0)?; RETURN\ PRODUCT$$
$$+ (\#candy(m) = 0)?; REPLENISH; RETURN\ PRODUCT\ .$$

If the machine is to be placed in a subway station, then it may be expected that it will not be replenished more than twice a day. Then, the upper time bound for the replenishing action might for instance be 12 hours. In this case, the previously described process does not satisfy the specification, but the following one does:

$$(\#candy(m) > 0)?; RETURN\ PRODUCT$$
$$+ (\#candy(m) = 0)?; RETURN\ MONEY\ .$$

4 Omega Closure Fork Algebras

Equational reasoning based on substitution of equals for equals is the kind of manipulation that is performed in many information processing systems. The role of equational logics in development of formal methods for computer science applications is increasingly recognized and various tools have been developed for modeling user's systems and carrying through designs within the equational framework (Gries and Schneider [10], Gries [9]).

In this section we present the *calculus for closure fork algebras* (CCFA), an extension of the *calculus of relations* (CR) and of the *calculus of relations with fork* [5]. Because of the non enumerability of the theory of dynamic logic, the CCFA cannot provide an adequate algebraization. In order to overcome this restriction we will define the calculus ω-CCFA by adding an infinitary equational inference rule. From the calculus we define the class ω-CFA of the *omega closure fork algebras* and a representation theorem is presented, showing that the Kleene star as axiomatized, indeed characterizes reflexive-transitive closure.

In the following paragraphs we will introduce the *Calculus for Closure Fork Algebras* (CCFA).

Definition 8. Given a set of relation symbols R, the set of CCFA terms on R is the smallest set $T\mathsf{CCFA}(R)$ satisfying: $R \cup RelVar \cup \{ 0, 1, 1' \} \subseteq T\mathsf{CCFA}(R)$. If $x \in T\mathsf{CCFA}(R)$,then $\{ \breve{x}, x^*, x^\diamond \} \subseteq T\mathsf{CCFA}(R)$. If $x, y \in T\mathsf{CCFA}(R)$,then $\{ x+y, x \cdot y, x;y, x \nabla y \} \subseteq T\mathsf{CCFA}(R)$.

The symbol $^\diamond$ denotes a *choice* function (see [12, §3], which is necessary in order to prove Thm. 1.

Definition 9. Given a set of relation symbols R, the set of CCFA formulas on R is the set of identities $t_1 = t_2$, with $t_1, t_2 \in T\mathsf{CCFA}(R)$.

Definition 10. Given terms $x, y, z, w \in T\mathsf{CCFA}(R)$, the identities defined by the following conditions are axioms:

Identities axiomatizing the relational calculus [13],

The following three axioms for the fork operator:

$$x \nabla y = (x;(1' \nabla 1)) \cdot (y;(1 \nabla 1'))\,,$$
$$(x \nabla y)\,;(z \nabla w)^\smile = (x;\breve{z}) \cdot (y;\breve{w})\,,$$
$$(1' \nabla 1)^\smile \nabla (1 \nabla 1')^\smile \leq 1'.$$

The following three axioms for the choice operator, taken from [12, p. 324]:

$$x^\diamond;1;\breve{x^\diamond} \leq 1', \qquad \breve{x^\diamond};1;x^\diamond \leq 1',$$
$$1;(x \cdot x^\diamond)\,;1 = 1;x;1.$$

The following two axioms for the Kleene star:

$$x^* = 1' \,+\, x;x^*, \qquad x^*;y \leq y \,+\, x^*;(\overline{y} \,\cdot\, x;y)\,.$$

Let us denote by $1'_U$ the partial identity $Ran\left(\overline{1 \nabla 1}\right)$. Then, the axiom $1;1'_U;1 = 1$ (which states the existence of a nonempty set of non-splitting elements) is added.

The rules of inference for the calculus CCFA are those of equational logic. Note that x^* is the smallest reflexive and transitive relation that includes x.

Definition 11. We define the calculus ω-CCFA as the extension of the CCFA obtained by adding the following inference rule[3]:

$$\frac{\vdash 1' \leq y \qquad x^i \leq y \vdash x^{i+1} \leq y}{\vdash x^* \leq y}$$

Definition 12. We define the class of the *omega closure fork algebras* (ω-CFA) as the models of the identities provable in ω-CCFA.

The standard models of the ω-CCFA are the *Proper Closure Fork Algebras* (PCFA for short). In order to define the class PCFA, we will first define the class $\bullet$PCFA.

[3] Given $i > 0$, by x^i we denote the relation inductively defined as follows: $x^1 = x$, and $x^{i+1} = x;x^i$.

Definition 13. Let E be a binary relation on a set U, and let R be a set of binary relations. A $\bullet$PCFA is a two sorted structure with domains R and U $\langle R, U, \cup, \cap, ^{-}, \emptyset, E, ;, Id, ^{\smile}, \nabla, ^{\diamond}, ^{*}, \star \rangle$ such that

1. $\bigcup R \subseteq E$,
2. $\star : U \times U \to U$ is an injective function when its domain is restricted to the set E,
3. If we denote by Id the identity relation on the set U, then $\emptyset$, E and Id belong to R,
4. R is closed under set choice operator defined by the condition:

$$x^{\diamond} \subseteq x \qquad \text{and} \qquad |x| = 1 \iff x \neq \emptyset.$$

5. R is closed under set union ($\cup$), intersection ($\cap$), complement relative to E ($^{-}$), composition of binary relations ($;$), converse ($^{\smile}$), reflexive-transitive closure (*) and fork (∇), defined by $S \nabla T = \{ \langle x, \star(y, z) \rangle : x S y \text{and} x T z \}$.

Note that $x^{\diamond}$ denotes an arbitrary pair in x, then $x^{\diamond}$ is called a *choice* operator.

Definition 14. We define the class PCFA as **Rd**$\bullet$PCFA where **Rd** takes reducts to structures of the form $\langle R, \cup, \cap, ^{-}, \emptyset, E, ;, Id, ^{\smile}, \nabla, ^{\diamond}, ^{*} \rangle$.

Note that given $\mathbf{A} \in$ PCFA, the terms $(1' \nabla 1)^{\smile}$ and $(1 \nabla 1')^{\smile}$ denote respectively the binary relations $\{ \langle a \star b, a \rangle : a, b \in A \}$ and $\{ \langle a \star b, b \rangle : a, b \in A \}$. Thus, they behave as projections with respect to the injection $\star$. We will denote these terms by π and ρ, respectively.

From the operator fork we define $x \otimes y = (\pi; x) \nabla (\rho; y)$. The operator $\otimes$ (*cross*), when interpreted in an proper closure fork algebra behaves as a parallel product: $x \otimes y = \{ \langle a \star b, c \star d \rangle : \langle a, c \rangle \in x \ \wedge \ \langle b, d \rangle \in y \}$.

A relation R is *constant* if satisfies: $\breve{R}; R \leq 1'$, $1; R = R$, and $R; 1 = 1$. Constant relations are alike constant functions, i.e., they relate every element from the domain to a single object[4]. We will denote the constant whose image is the value a by C_a.

Definition 15. We denote by FullPCFA the subclass of PCFA in which the relation E equals $U \times U$ for some set U and R is the set of all binary relations contained in E.

Similarly to the relation algebraic case, where every proper relation algebra (PRA) $\mathbf{A}$ belongs to[5] **ISP**FullPRA, it is easy to show that every PCFA belongs to **ISP**FullPCFA. We finally present the representation theorem for ω-CFA.

Theorem 1. *Given* $\mathbf{A} \in \omega$-CFA*, there exists* $\mathbf{B} \in$ PCFA *such that* $\mathbf{A}$ *is isomorphic to* $\mathbf{B}$.

[4] This comment is in general a little strong and applies to *simple* algebras, but is nevertheless useful as an intuitive aid for the non specialist.

[5] By **I**, **S** and **P** we denote the closure of an algebraic class under isomorphic copies, subalgebras and direct products, respectively.

5 Interpretability of P/PML in ω-CCFA

In this section we will show how theories on P/PML can be interpreted as equational theories in ω-CCFA. This is very useful because allows to reason equationally in a logic with variables over two different sorts (individuals and relations).

Definition 16. Let S, F and P be sets consisting of sort, function and relation symbols, respectively. By $\omega\text{-CCFA}^+(S, A, F, P)$ we denote the extension of ω-CCFA obtained by adding the following equations as axioms.

1. For each $s, s' \in S$ $(s \neq s')$, the equations $1'_s + 1'_U = 1'_U$ and $1'_s \cdot 1'_{s'} = 0$ (elements from types do not split, and different types are disjoint).
2. For each $a \in A$ with $ia(a) = s_1 \dots s_k$ and $oa(a) = s'_1 \dots s'_n$, the equation $(1'_{s_1} \otimes \dots \otimes 1'_{s_k})\,;a\,;\left(1'_{s'_1} \otimes \dots \otimes 1'_{s'_n}\right) = a$.
3. For each $f : s_1 \dots s_k \to s \in F$, $\breve{f};f + 1'_s = 1'_s$ and $(1'_{s_1} \otimes \dots \otimes 1'_{s_k})\,;f = f$, stating that f is a functional relation of the right sorts.
4. For each p of arity $s_1 \dots s_k$ in P, the equation $(1'_{s_1} \otimes \dots \otimes 1'_{s_k})\,;p;1 = p$, stating that p is a right-ideal relation expecting inputs of the right sorts.

Definition 17. A *model* for the calculus $\omega\text{-CCFA}^+(S, A, F, P)$ is a structure $\mathcal{A} = \left\langle \left\langle \mathbf{A}, S^{\mathcal{A}}, A^{\mathcal{A}}, F^{\mathcal{A}}, P^{\mathcal{A}} \right\rangle, m \right\rangle$ where: $\mathbf{A} \in \omega\text{-CFA}$. $S^{\mathcal{A}}$ is a set of disjoint partial identities, one for each sort symbol in S. $A^{\mathcal{A}}$ is a set of binary relations, one for each action symbol $a \in A$. Besides, if $ia(a) = s_1 \dots s_k$ and $oa(a) = s'_1 \dots s'_n$, then $a^{\mathcal{A}}$ satisfies the condition in item 2 of Def. 16. $F^{\mathcal{A}}$ is a set of functional relations, one for each function symbol in F. Besides, if $f : s_1 \dots s_k \to s$, then $f^{\mathcal{A}}$ satisfies the conditions in item 3 of Def. 16. $P^{\mathcal{A}}$ is a set of right ideal relations, one for each predicate symbol $p \in P$. Besides, if p has arity $s_1 \dots s_k$, then $p^{\mathcal{A}}$ satisfies the conditions in item 4 of Def. 16. $m : RelVar \to \mathbf{A}$.

Noce that the mapping m in a $\omega\text{-CCFA}^+(S, A, F, P)$ model extends homomorphically to arbitrary relational terms. For the sake of simplicity, we will use the same name for both.

In the following paragraphs we will define a function mapping formulas from $P/PML(S, A, F, P)$ to $\omega\text{-CCFA}^+(S, A, F, P)$ formulas. In the next definitions, σ is a sequence of numbers increasingly ordered. Intuitively, the sequence σ contains indices of those individual variables that appear free in the formula (or term) being translated. By $Ord(n, \sigma)$ we will denote the position of the index n in the sequence σ, by $[\sigma \oplus n]$ we denote the extension of the sequence σ with the index n, and by $\sigma(k)$ we denote the element in the k-th position of σ. In what follows, $t^{;n}$ is an abbreviation for $t;\dots;t$ (n times). For the sake of completeness, $t^{;0}$ is defined as 1'. We will denote by $IndTerm(F)$ the set of terms from P/PML built from the set of constant and function symbols F. By $RelDes(K)$ we denote the set of terms from ω-CCFA that are built from the set of relation constants K.

Definition 18. The function $\delta_\sigma : IndTerm(F) \to RelDes(F)$, mapping individual terms into relation designations, is defined inductively by the conditions:

1. $\delta_\sigma(v_i) = \begin{cases} \rho^{;Ord(i,\sigma)-1};\pi \text{ if} i \text{ is not the last index in } \sigma, \\ \rho^{;Length(\sigma)-1} \quad \text{if} i \text{ is the last index in } \sigma. \end{cases}$
2. $\delta_\sigma(f(t_1,\ldots,t_m)) = (\delta_\sigma(t_1)\nabla\cdots\nabla\delta_\sigma(t_m));f$ for each $f \in F$.

Given a sequence σ such that $Length(\sigma) = l$ and an index n $(n < \omega)$ such that v_n has sort s, we define the term $\Delta_{\sigma,n}$ $(n < \omega)$ by the condition[6]

$$\Delta_{\sigma,n} = \begin{cases} \delta_\sigma(v_{\sigma(1)})\nabla\cdots\nabla\delta_\sigma(v_{\sigma(k-1)})\nabla 1_s \nabla\delta_\sigma(v_{\sigma(k+1)})\nabla\cdots\nabla\delta_\sigma(v_{\sigma(l)}) \\ \qquad\qquad \text{if } k = Ord(n,[\sigma\oplus n]) < l, \\ \delta_\sigma(v_{\sigma(1)})\nabla\cdots\nabla\delta_\sigma(v_{\sigma(l-1)})\nabla 1_s \qquad \text{if } Ord(n,[\sigma\oplus n]) = l. \end{cases}$$

Notation 1 Let σ be a sequence of indices of individual variables of length n. Let $\vec{x} = \langle x_1,\ldots,x_k\rangle$ be a vector of variables whose indices occur in σ. We will denote by $\Pi_{\sigma,\vec{x}}$ the relation that given a tuple of values for the variables whose indices appear in σ, projects the values corresponding to the variables appearing in $\vec{x}$. For example, given $\sigma = \langle 2,5,7,9\rangle$ and $\vec{x} = \langle v_2, v_7\rangle$, $\Pi_{\sigma,\vec{x}} = \{\langle a_1 \star a_2 \star a_3 \star a_4, a_1 \star a_3\rangle : a_1,a_2,a_3,a_4 \in A\}$. Similarly, $Arrange_{\sigma,\vec{x}}$ denotes the relation that, given two tuples of values (one for the variables with indices in σ and the other for the variables in $\vec{x}$), produces a new tuple of values for the variables with indices in σ updating the old values with the values in the second tuple. For the previously defined σ and $\vec{x}$, we have $Arrange_{\sigma,\vec{x}} = \{\langle (a_1 \star a_2 \star a_3 \star a_4) \star (b_1 \star b_2), b_1 \star a_2 \star b_2 \star a_4\rangle : a_1,a_2,a_3,a_4,b_1,b_2 \in A\}$. Note that these two relations can be easily defined using the projections π and ρ previously defined.

Definition 19. The mappings $M : RT(\mathcal{S}) \to RelDes(A)$ and $T_\sigma : For(\mathcal{S}) \to RelDes(A \cup F \cup P)$ are mutually defined by

$M(a) = a$ for each $a \in A \cup RelVar$, $\quad M(1'_{s_1\ldots s_k}) = 1'_{s_1} \otimes \cdots \otimes 1'_{s_k}$,
$M(R^*) = M(R)^*$, $\quad M(R+S) = M(R)+M(S)$,
$M(R\cdot S) = M(R)\cdot M(S)$, $\quad M(R;S) = M(R);M(S)$,
$M(\alpha?) = T_{\sigma_\alpha}(\alpha)\cdot 1'$,
$T_\sigma(p(t_1,\ldots,t_k)) = (\delta_\sigma(t_1)\nabla\cdots\nabla\delta_\sigma(t_k));p$, $\quad T_\sigma(\neg\alpha) = \overline{T_\sigma(\alpha)}$,
$T_\sigma((\exists v_n : s)\,\alpha) = \Delta_{\sigma,n};T_{[\sigma\oplus n]}(\alpha)$, $\quad T_\sigma(\alpha\vee\beta) = T_\sigma(\alpha)+T_\sigma(\beta)$,
$T_\sigma\left(\left\langle \vec{x}\ {}_l R^u\ \vec{y}\right\rangle \alpha\right) =$

$$\begin{pmatrix} 1' \\ \nabla \\ \Pi_{\sigma,\vec{x}};M(R) \end{pmatrix};Arrange_{\sigma,\vec{y}};T_\sigma(\alpha)\cdot\left((v_l;\leq)\cdot C_{l(R)}\right);1\cdot\left((v_u;\geq)\cdot C_{u(R)}\right);1.$$

We will denote by $\vdash_{\omega\text{-CCFA}}$ the provability relation in the calculus ω-CCFA. The next theorem states the interpretability of theories from P/PML as equational theories in ω-CCFA.

Theorem 2. *Let $\Gamma\cup\{\varphi\}$ be a set of P/PML formulas without free individual variables. Then, $\Gamma \models_{P/PML} \varphi \iff \{T_\langle\rangle(\gamma) = 1 : \gamma\in\Gamma\} \vdash_{\omega\text{-CCFA}} T_\langle\rangle(\varphi) = 1$.*

[6] By 1_s we denote the relation $1;1'_s$.

6 Conclusions

We have presented a logic (P/PML) for formal real–time systems specification and construction. This logic is an extension of dynamic logic by considering arbitrary atomic actions, an operator for putting processes in parallel, and explicit time. We have also presented an equational calculus in which theories of P/PML can be interpreted, thus enabling the use of equational inference tools in the process of systems construction.

References

1. Carvalho, S.E.R., Fiadeiro, J.L. and Haeusler, E.H., *A Formal Approach to Real–Time Object Oriented Software*. In Proceedings of the Workshop on Real–Time Programming, pp. 91–96, sept/1997, Lyon, France, IFAP/IFIP.
2. Fenton, N. E., *Software Metrics. A Rigorous Approach*, International Thomson Computer Press, 1995.
3. Fiadeiro, J. L. L. and Maibaum, T. S. E., *Temporal Theories as Modularisation Units for Concurrent System Specifications*, Formal Aspects of Computing, Vol. 4, No. 3, (1992), 239–272.
4. Fiadeiro, J. L. L. and Maibaum, T. S. E., *A Mathematical Toolbox for the Software Architect*, in Proc. 8th International Workshop on Software Specification and Design, J. Kramer and A. Wolf, eds., (1995) (IEEE Press), 46–55.
5. Frias M. F., Baum G. A. and Haeberer A. M., *Fork Algebras in Algebra, Logic and Computer Science*, Fundamenta Informaticae Vol. 32 (1997), pp. 1–25.
6. Frias, M. F., Haeberer, A. M. and Veloso, P. A. S., *A Finite Axiomatization for Fork Algebras*, Logic Journal of the IGPL, Vol. 5, No. 3, 311–319, 1997.
7. Frias, M. F. and Orlowska, E., *A Proof System for Fork Algebras and its Applications to Reasoning in Logics Based on Intuitionism*, Logique et Analyze, vol. 150–151–152, pp. 239–284, 1995.
8. Frias, M. F. and Orlowska, E., *Equational Reasoning in Non–Classical Logics*, Journal of Applied Non Classical Logic, Vol. 8, No. 1–2, 1998.
9. Gries, D., *Equational logic as a tool*, LNCS 936, Springer–Verlag, 1995, pp. 1-17.
10. Gries, D. and Schneider, F. B., *A Logical Approach to Discrete Math.*, Springer–Verlag, 1993.
11. Kaposi, A. and Myers, M., *Systems, Models and Measures*, Springer–Verlag London, Formal Approaches to Computing and Information Technology, 1994.
12. Maddux, R.D., *Finitary Algebraic Logic*, Zeitschr. f. math. Logik und Grundlagen d. Math. vol. 35, pp. 321–332, 1989.
13. Maddux, R.D., *Relation Algebras*, Chapter 2 of Relational Methods in Computer Science, Springer Wien New York, 1997.
14. Roberts, F. S., *Measurement Theory, with Applications to Decision–Making, Utility and the Social Sciences*, Addison–Wesley, 1979.