Specifying and Verifying Business Processes Using PPML

Germán Regis[1], Nazareno Aguirre[1], and Tom Maibaum[2]

[1] Departamento de Computación, FCEFQyN, Universidad Nacional de Río Cuarto and CONICET, Ruta 36 Km. 601, Río Cuarto (5800), Córdoba, Argentina
{gregis,naguirre}@dc.exa.unrc.edu.ar
[2] Department of Computing & Software, McMaster University, 1280 Main St. West, Hamilton, Ontario, Canada L8S 4K1
tom@maibaum.org

Abstract. The *Product Process Modeling Language* (PPML) is a formal language for the specification of business processes, which has a formal semantics based on timed transition systems. As opposed to other business process modeling languages, PPML puts an emphasis on *products* (not only processes), allowing the specifier to describe properties of these, and how processes affect them. This facilitates modeling of business processes, and combined with other characteristics of the language, most notably timing constraints in the form of time bounds associated with processes, makes it an expressive vehicle for modeling business processes.

PPML is more a formalism than an actual modeling language, since no syntax was ever defined for the formalism. In this paper, we define a suitable syntax for PPML models, and provide a formal semantics for the extended language in terms of timed automata. The formal semantics is given as a translation from PPML into UPPAAL. This formal semantics enables us to straightforwardly employ the UPPAAL model checker in order to verify real time properties of PPML specifications.

We show some of the benefits of a product-oriented language for business process modeling, the details of our translation and the results of the use of the UPPAAL model checker for PPML specifications via a simple case study, regarding a motherboard production line.

1 Introduction

The constant effort of different organizations for improving their business and manufacturing processes for efficiency and control has led to the development of languages and methods for business process modeling and analysis. Currently, there exist several business process modeling languages, such as BPEL, WS-CDL [14], etc. Most of these have been defined with a significant emphasis on modeling service oriented systems [1], and generally lack a formal semantics, which makes them less suitable for automated analysis.

PPML [19], on the other hand, is a *formal* business process modeling language based on timed state transition systems [18]. PPML models are composed of *processes*, and their effects on *products*. Also, processes may include temporal

© Springer-Verlag Berlin Heidelberg 2009

bounds, which enable one to specify timing constraints. These features make the language appropriate for modeling concurrency related restrictions, process synchronizations, etc. The main difference with other languages is that, in PPML, *products* are explicit referents of the model. This provides us with greater flexibility, compared to other business process modeling formalisms, particularly when describing models in which products are complex, and their description is as important as that of processes. We believe that having the possibility of describing products and their structure is essential in cases in which the information flow of processes and how products evolve in these processes need to be explicitly specified, e.g., for stating invariants, properties describing relationships between different products (or different states of the same product), etc. Some situations in which this is clearly observed are the specification of certain industrial processes, protocol descriptions such as CORBA, etc. PPML also allows us to describe the structural state of products in particular moments in time; for instance, one can describe the state of a product before and after a process is executed on it. This facilitates the description of properties regarding product traceability, and other properties not directly associated with the processes, but with the products and their evolution in the system.

In the last two decades, the development of algorithmic methods for software/hardware verification has led to powerful analysis mechanisms, such as model checking [6]. These mechanisms have been enhanced by increasing computer power, and, in the last decade or so, various tools for automated analysis/verification have been developed, and are being used in practice. Many systems have requirements associated with real time (e.g., requirements associated with response within some preestablished bounds, etc.). For these kinds of systems and properties, there exist special model checking tools, most notably the tools Kronos [8] and UPPAAL [3]. Various kinds of timing constraints are often found in business process descriptions (cf. [15] page 3, [26] Section 1.7), and therefore, as it will be made clearer later on, we can benefit from the use of model checking tools for real-time for analyzing business process specifications.

We are interested in formally specifying business processes using a richer language for specifying products and their characteristics, as opposed to what is normally found in business process modeling notations. Moreover, we are also interested in verifying properties of these models, in particular real time properties.

The PPML formalism has been carefully defined, and various of its features have been thoroughly studied [19]. We refer both to the language and the logic as PPML, as opposed to [19], where PPML is the logic underlying the language, and the language is called *Mensurae*. Since no formal, precise syntax has been provided for PPML, we define a suitable syntax for it, extending the original language. Moreover, we also provide an encoding of the extended language into the language associated with the UPPAAL model checking tool. This translation provides the language with a formal semantics based on timed automata, the semantic formalism behind UPPAAL. We describe the above mentioned encoding, and develop a case study, based on a simplified version of a motherboard production line. This will enable us to justify the usefulness of the encoding, for verifying interesting real time properties associated with business process specification.

The paper proceeds as follows. First we describe the PPML language and provide an overview of the UPPAAL language and tool. We then present our extension of PPML, its formal syntax as well as the proposed semantics, as a translation from PPML into the language of the UPPAAL model checker. We use our case study as a reference for the presentation. We also use the translation in order to verify properties associated with the case study. Finally, we discuss related work in the area and draw some conclusions.

2 An Overview of PPML

PPML is a formal language which can be used to model business processes [19]. The basics of the method underlying the language are described in [17]. PPML has three basic constructs: *products*, *processes* and *gates*. *Products* are entities characterized by a set of measurable attributes. Products can be manipulated by processes. *Processes* are entities that represent behaviours, which are not necessarily instantaneous, i.e, they can take some time to be completed. They are modeled via "single input, single output" tasks. If multiple inputs are necessary, these have to be put together in a composite product. The main element employed for composing/decomposing products, to be processed by processes, is the *gate*. Basically, there exist three types of gate, namely, the *multiplexer*, the *demultiplexer* and the *semaphore*.

One can also associate timing constraints with processes. This is done in PPML via two bounds associated with processes: a lower bound (minimum time that the process needs to fulfill its task) and an upper bound (maximum time that the process can spend to complete its work).

2.1 Products

Products represent empirical referent objects (i.e., "things" in the world being modeled). Products are characterized by their measurable attributes, e.g., length, weight, color, etc. These characteristics may be directly observable or can be calculated by functions applied to values of other existing features. The characteristics associated with a product entity must be given in a suitable measurement scale [9].

In order to define products, we assume a first order theory presentation $\langle \Sigma, A \rangle$, called a proto-product, which defines the basic types (e.g., for attributes) necessary for products, including a sort of codes ($Code$), a sort of names ($Name$), and a sort for instants ($Time$, which corresponds to a discrete totally ordered set with a first element).

Products can then be either *atomic*, *composite* or *structured*. An *atomic product* P is a tuple $\langle code\_number_P, product\_name_P, time_P, attributes_P \rangle$, where $code\_number_P$ is a $Code$ constant, used to identify products; $product\_name_P$ is a constant of sort $Name$, which allows one to refer to particular products; $time_P$ is a constant of sort $Time$, and it is the *time stamp* of the product P, indicating the last time that the product's attributes have been updated; $attributes_P \subseteq \Sigma$ is a set of attributes, the measurable characteristics of the corresponding empirical referent (i.e., the entity in the real world being modeled by the product). Certain attributes, called $direct\_attributes_P$, may be directly measured by means of appropriate measurement procedures, while others, $derived\_attributes_P$, are calculated using rules and laws governed by the *axioms* A. As an example, suppose that we need to model a memory bank with some basic characteristics such as the memory's size and a flag indicating if the memory was tested or not:

$$\langle code\_memory, memory, 0, \{size : nat, tested : bool\}\rangle.$$

A *composite product* P is either:

- a pair $P = \langle code\_number_P, \otimes(P_{c1}, .., P_{cn})\rangle$ where $\otimes(P_{c1}, .., P_{cn})$ is an injection of the components P_{ci} into the cartesian product. This type of composite product is used to blend products emerging from several previous processes. This is useful, in particular, for synchronizing products in time, to be consumed as inputs by other processes,
- a pair $P = \langle code\_number_P, \iota(\mathcal{P})\rangle$ where $\mathcal{P}$ is a finite set of products and $\iota(\mathcal{P}) \in \mathcal{P}$. This type is the *choice* product and is used in situations where an input from any one of some previous processes is chosen based on some defined condition.

Products that need to be treated as atomic artifacts, but whose definitions are given in terms of constituent parts, are not the same as composite products. These are a particular kind of product, called a *structured* product.

A *structured product* P may be *refined* (or specialized), which corresponds in logic to extending the presentation $\langle \Sigma, A\rangle$ to some new proto-product $\langle \Sigma', A'\rangle$ by adding some new constants, functions and relations, or *aggregate*, which corresponds to a type of product that allows us to aggregate several constituent products into a new atomic product.

As it can be observed in the above specification, structured product descriptions are akin of classes in object orientation. The *instances* of these product descriptions will represent the individual referents in the real world. That is, the instances of product descriptions will be involved in the executions of processes.

2.2 Processes

A process models an empirical referent process (i.e., some real world process or procedure) that transforms an input product into an output one. As for products, the processes may be *atomic* (a process without internal constituent "subprocesses") or *structured*. They model input/output transformations. In order to carry out these transformations, each process has a *virtual machine* that interprets its basic commands. Intuitively, a virtual machine is an object system with routines, representing basic actions that it is capable of doing, such as assignments in a conventional programming language.

In order to specify a process, we define what input/output transformations are required using basic actions or some combinations of these, via some control structure of the virtual machine internal to the process. Not all actions of the virtual machine are under the control of the process. *Environmentally* controlled actions may appear. Another control structure that can be useful is parallel execution.

The formalization of the concept of the virtual machine is based on *object specifications* [10] and consists of a logical framework based on timed transition systems, called RETOOL [4]. The RETOOL semantics is based on the notion of computation over a timed object frame. The specification (theory presentation) describing the virtual machine is a pair consisting of a signature and a collection of sentences describing the behaviour of the system [19].

The definition of a process is given as a *transaction* defining a computation segment of the underlying virtual machine. It is specified by five elements, namely, the initial and final conditions (q, p), the invariant I, and the lower and upper bounds (l, u). The initial condition q specifies the states in which the transaction can be initiated; the final condition describes the states in which the transaction finishes (i.e., a kind of postcondition); the invariant I is a property that is supposed to hold throughout the execution of the transaction (e.g., requiring that the equipment being used for the process is not unplugged during the execution of the process!); finally, the lower and upper bounds l and u state the minimum and maximum time that the execution of the transaction can take, in order to be completed.

A process behaviour, i.e., its associated transfer function (how the input product is transformed into the output product), can then be formally characterized by a formula $(q, I)_l\Delta^u p$, which is interpreted with respect to a timed state sequence $\langle\sigma, T\rangle$ for a timed object frame, (where σ is an infinite sequence of states and T is an infinite sequence of corresponding times), and an instant i of time (the current time), in the following way: $\sigma, T, i \vDash (q, I)_l\Delta^u p$ iff, for some k such that $i + l < k \leq i + u$; $\sigma, T, i \vDash q$, $\sigma, T, k \vDash p$ and $\sigma, T, n \vDash I$ hold, for every $i \leq n \leq k$.

An *atomic process* p is a pair $\langle proc, VM\rangle$, where VM is the process' virtual machine (an object specification [4]) and $proc = \langle process\_code, process\_name, P_I, P_O, (q, I)_l\Delta^u p\rangle$; $process\_code, process\_name$ are state variables of the sorts used for process codes and names, respectively. These variables are rigid (i.e., their interpretations are immutable along computations), and their sorts are assumed to be defined and specified in the proto-product $\langle\Sigma, A\rangle$. P_I, P_O are the input and output products and $(q, I)_l\Delta^u p$ is the specification of the properties of the process (i.e., its associated *transfer function*). Consider, for instance, the following tuple describing an atomic tester process:

$$\langle code, tester, memory, memory, (\neg memory.tested, true)_5\Delta^{10} memory.tested\rangle$$

This tester process takes as input a non tested memory bank, and after i units of time ($5 \leq i \leq 10$), the process returns a tested memory bank. For the sake of simplicity, we skip the description of the virtual machine for this process.

2.3 Gates

In order to make processes interact, by interconnecting them via products, it is often necessary to combine products to build composite ones, or decompose products, for instance for feeding other processes with the parts. In order to do this, PPML provides the concept of gate. Besides gates for composing/decomposing products, there is a third kind of gate, the semaphore, which is useful for synchronization. Gates are useful for modeling transfer functions that can be regarded as instantaneous because the time taken is trivial and where the single input/single output constraint is not met by purely trivial marshalling activities. The types of gates are formally defined as follows:

- A multiplexer is a tuple $M = \langle \mathit{multiplexer\_code}, \mathcal{P}, P, F\rangle$, where *multiplexer_code* is a fixed value used to identify the gate, $\mathcal{P}$ is the set of input products, P is the output product of the multiplexer and F is the multiplexer action function defining P explicitly in terms of the set of input products $\mathcal{P}$.
- A demultiplexer is the dual of a multiplexer. In this case, the output products are defined as projections of the input product.
- A semaphore is a tuple $S = \langle \mathit{semaphore\_code}, P, S\rangle$, where *semaphore_code* is a fixed value used to identify the semaphore, P is the input/output product and S is the condition that must be satisfied to continue.

Graphically, gates are depicted as shown in Fig. 1 2 3, As an example, consider a multiplexer gate that receives a memory bank and a processor and returns a composite product putting together the input products. This is specified as follows:

$$\langle code_M, \{mem, proc\}, \langle code_P, \otimes(p_1, p_2)\rangle, \{(mem, p_1), (proc, p_2)\}\rangle$$

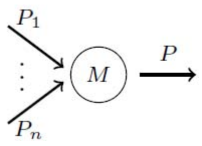

Fig. 1. Multiplexer

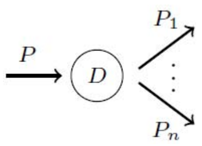

Fig. 2. Demultiplexer

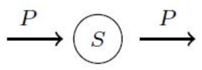

Fig. 3. Semaphore

2.4 Framework Processes

When modeling complex empirical referents, it is often the case that one needs mechanisms providing us with *abstraction* and *encapsulation*, in order to deal with complexity. This is the usual situation when one decides to model processes in a bottom up way, i.e., modeling simpler processes first and composing these later on, as well as in top down approaches, i.e., modeling complex processes abstractly first and later on refining these into more detailed subprocesses [2]. PPML provides facilities for dealing with abstraction and encapsulation, particularly the notion of *framework process*. Framework processes are defined via process combinators. These are the following: Let $p1 = \langle p1\_code, p1\_name, p1_I, p1_O,$

$(q_{p1}, I_{p1})_{l_{p1}} \Delta^{u_{p1}} p_{p1}\rangle$ and $p2 = \langle p2\_code, p2\_name, p2_I, p2_O, (q_{p2}, I_{p2})_{l_{p2}} \Delta^{u_{p2}} p_{p2}\rangle$ be two processes,

- *Sequential combination*: Denoted as $p_1; p_2$, combines two processes into a new one $p = \langle p\_code, p\_name, p1_I, p2_O, (q_{p1}, I_p)_{l_{p1}+l_{p2}} \Delta^{u_{p1}+u_{p2}} p_{p2}\rangle$, in the expected way. The *p_code* and *p_name* "fields" have new unique *code* and *name*, respectively. The invariant I_p requires that the initial condition of the first of the processes holds, while the invariant I_{p1} holds for some time not exceeding u_{p1} units of time; after that, the final condition p_{p1} becomes true in at least l_{p1} units of time. Then, eventually the initial condition q_{p2} holds, while the invariant I_{p2} holds, from that point onwards, for at most u_{p2} units of time; after that, the final condition p_{p2} becomes true in at least l_{p2} units of time.
- *Semaphore (conditional) combination*: The semaphore composition of $p1$ and $p2$, denoted by $p1;_s p2$, is defined as for the sequential composition, but the invariant of the transfer function is strengthened in the sense that s (semaphore condition) must be true in order to start the second process.
- *Parallel combination*: Let $m_O = \langle multiplexer\_code, \mathcal{P}_O, P_O, F_O\rangle$ and $d_I = \langle demultiplexer\_code, P_I, \mathcal{P}_I, F_I\rangle$ be a multiplexer and a demultiplexer, respectively, each defined over the set of input and output products of $p1$ and $p2$. Then, the parallel composition p of $p1$ and $p2$ with respect to m_O and d_I, denoted by $[p1; p2](d_I, m_O)$ is defined as follows: The *process_code* and *process_name* "fields" of p have new unique *code* and *name* respectively. The input of p is the input of d_I and its output is the output of m_O. The transfer function τ of p is $(q_{p1} \wedge q_{p2}, I_p)_{max((l_{p1}, l_{p2}))} \Delta^{max((u_{p1}, u_{p2}))} (p_{p1} \wedge p_{p2})$, where the invariant I_p is defined as $(q_{p1} \wedge q_{p2}) \Rightarrow (\tau_1[(p_{p1} \mathcal{U} (p_{p1} \wedge p_{p2}))/p_{p1}] \wedge (\tau_2[(p_{p2} \mathcal{U} (p_{p1} \wedge p_{p2}))/p_{p2}]))$ (where $\tau[(p\mathcal{U}q)/r]$ denotes the replacement of the final condition r in τ by the condition $(p\mathcal{U}q)$). The symbol "$\mathcal{U}$" denotes the well known strong until temporal operator. This requires that, when the two processes are ready to start, they are executed in parallel. Further, when one of them finishes, it must wait for the other process to reach its final state.

A *framework process p* is an *atomic* process composed of a set of constituent processes $\{p_1, ..., p_n\}$ using the combinators defined above. We denote framework processes by pairs of the form $\langle p, fw.exp\rangle$, where p is the standard PPML definition of process and $fw.exp$ is the specification of the constituent processes in terms of the combinators.

3 UPPAAL

UPPAAL is a toolbox for the verification of real-time systems. It is based on the theory of timed automata, and provides a subset of CTL (computational tree logic) as a query language to specify properties to be checked. A model in UPPAAL is a set of instances of *templates* which can be communicated by means of various kinds of communication channels.

An UPPAAL specification consists of three parts: *global declarations*, *schemas* (automata templates) with their corresponding *local declarations*, and the *system specification.*

3.1 Declarations

The global declaration, or the local declaration of a template, may include the definition of variables, arrays, registers or types (as in the C programming language). There exist four predefined types: int (integers), bool (booleans), clock (clocks), and chan (communications channels). The communications channels can be *basic*, *urgent* or *broadcast.* Constants can also be defined, using the *const* keyword. UPPAAL provides a rich language for the declaration of functions that can be invoked in the templates. Parameters, conditional sentences and iterative sentences, such as 'while' or 'for' statements, can also be specified.

3.2 Templates

Templates are defined as *extended timed automata.* An automaton consists of *locations* and *edges*, and can also have local declarations and parameters (by value or by reference).

Locations can be labeled (reference names). We can specify *invariants* in the location, indicating that some condition must hold in the state. The invariant expressions can only be conjunctions of simple conditions over clocks, or boolean expressions without clock variables. Conditions involving lower bounds on clocks are not allowed. There are three modifiers for a template's locations: *initial* (each template must have exactly one initial state), *urgent* (time stops while a process is in one of these states), and *committed* (time stops while a process is in one of these states, as for urgent locations, but they also bind the system scheduler to choose one of the committed locations in the next transition).

Locations are connected by *edges.* The edges can be annotated with *selections*, *guards*, *synchronizations* or *updates.*

- *Selections*: Selections non-deterministically bind a given identifier to a value in a given range. The other three labels of an edge are within the scope of this binding.
- *Guards*: An edge is enabled in a state if and only if the guard in it evaluates to true.
- *Synchronization*:Processes can synchronize over channels. Edges labeled with complementary actions over a common channel synchronize.
- *Updates*: When an edge is "executed", the update expression of the edge is evaluated. The side effect of this expression changes the state of the system.

When two processes are synchronized, both synchronized edges are *fired* at the same time. Their corresponding updates are performed in an ordered manner: first the update of the *sending* process, and then the update of the *receiver.* Notice that the edges allow only for a single synchronizing channel. *Broadcast* channels represent one-to-many synchronizations, since the sender and all the receiver edges are fired at the same time.

3.3 System Specification

The specification of a system model consists of one or more concurrent processes (template instances), variables and communication channels. The variables, channels and functions defined at this level are not available in the templates.

3.4 Temporal Properties

UPPAAL provides a CTL temporal logic [20], with some restrictions (only one path quantifier), as a query language. Thus, the allowed query formulas are the following:

- $E\Diamond q$: evaluates to true for a timed transition system if and only if there is a sequence of alternating delay transitions and action transitions $s_0 \rightarrow \cdots \rightarrow s_n$, where s_0 is the initial state and s_n satisfies q.
- $A\Box q$: evaluates to true if and only if every reachable state satisfies q.
- $E\Box q$: evaluates to true for a timed transition system if and only if there is a sequence of alternating delay or action transitions $s_0 \rightarrow s_1 \rightarrow \cdots \rightarrow s_i \rightarrow \cdots$ for which q holds in all states s_i.
- $A\Diamond q$: evaluates to true if and only if all possible transition sequences eventually reach a state satisfying q.

In the above formulas, q is a well formed logical expression. The variables or states of a process in an expression can be referenced. For instance,

$$A\Box Motherboard.End \rightarrow Motherboard.hasProcessor$$

expresses that, for all execution sequences, it is always the case that, if a motherboard is in its end state, then it must have a processor (its *hasProcessor* variable is set to true).

4 PPML Syntax and Extensions

In previous work on PPML, all the elements that are part of business process specifications are formally defined, but no actual syntax for the specifications is proposed. In order to provide a suitable high level syntax for specifying PPML models, we propose a syntax for products, processes and gates. This syntax has two objectives, namely, it allows us to provide a more flexible and user friendly way of writing PPML specifications (as in other business description languages), and to standardize the syntax so that tools for the language, such as parsers and analyzers, can be built.

For the sake of simplicity, we will mainly show the proposed syntax for PPML using a case study as a reference. The case study is the following. Suppose that we need to model a simplified version of part of a production line of a company that assembles motherboards. In this simplification, the assembly process receives as initial source products base motherboards, with two (empty) slots, one for a processor, and the other for a memory bank. Once assembled, each base motherboard is complemented with a processor and a memory bank, each seated in its corresponding slot. At the end of the manufacturing process, the assembled motherboard is tested, more precisely, the motherboard is tested in combination with the memory bank, and in combination with the processor. In order to reduce the manufacturing time, these tests can be done in parallel by two independent testing processes.

Motherboards are structured products. Consider the definition of the *Motherboard* product shown in Fig. 4, and illustrating the syntax of products.

```
Product MotherBoard {
   Proccessor MProccessor,
   Memory MMemory,
   int Proccessor_Socket,
   boolean hasProcessor,
   boolean hasMemory,
   boolean tested
}
```

Fig. 4. Structured Product *MotherBoard*

As it can be observed, structured product descriptions are akin to classes in object orientation (although is not shown in the example, the only kind of "method" allowed in product specifications are the definitions of derived attributes). The instances of products will be involved in the executions of processes.

When modeling business processes, one often finds situations in which certain products are built or transformed in several steps. In these cases, sometimes the state of some of the products' attributes being built or transformed are unknown, e.g., when these have not yet been assigned a particular value. In order to model these situations, it is necessary to introduce *null values*.

In order to illustrate a *process* definition, let us model part of the motherboard production line, namely the process that takes a motherboard without processor and a processor, and returns the partially assembled motherboard resulting from seating the processor in its corresponding slot in the motherboard. The input product is a composite product, consisting of a motherboard without processor, and a processor. The initial condition requires the compatibility of sockets and the processor slot in the motherboard being empty. The invariant for this process should specify that the processor cannot be assembled in parallel with the seating of the memory for this motherboard. The output product is simply the original motherboard with the processor put in its corresponding slot. We might associate time bounds with this process, for instance saying that the process cannot take less than 5 units of time to be performed, and it takes 10 units of time or less to be completed. This process is specified, in our proposed syntax, in Fig. 5.

Due to space restrictions, we include here only the formal syntax of atomic and structured products (see Fig. 6) and atomic processes (see Fig. 7). More complex processes, composed of simpler ones, are easier to express using a graphical notation, as we will see later on in the paper.

```
Process assem_1 {
  input: [Motherboard Mother_in; Proccessor Proc_in],
  output: MotherBoard Mother_out,
  invariant: Mother_in.HasMemory == false,
  requires: Mother_in.Proccessor_Socket == Proc_in.Proccessor_Socket
           && Mother_in.HasProccessor=false,
  ensure: Mother_out == Mother_in && Mother_in.MProcessor == Proc_in
         && Mother_in.hasProcessor == true,
  l_time: 5 ,
  u_time:10
}
```

Fig. 5. Process that assembles a motherboard and a processor

```
Product              → ’Product’ Product_name ’{’ Product_body ’}’
Product_body         → Product_refs ’,’ Product_atts | Product_atts
Product_refs         → Product_ref ’,’ Product_refs | Product_ref
Product_atts         → Product_att ’,’ Product_atts | Product_att
Product_ref          → Product_ref_name Product_ref_ID
Product_att          → Product_direct_att | Product_derived_att
Product_direct_att   → Product_att_type Product_att_ID
Product_derived_att  → Product_att_type Product_att_ID ’=’ Expression
Product_att_type     → ’int’ | ’boolean’
Expression           → Expression Binary_op Expression | Unary_op Expression
                       | Product_att_ID | Product_ref_name’.’Product_att_ID
                       | ’(’Expression’)’ | NAT_const | BOOLEAN_const
Binary_op            → ’+’ | ’-’ | ’*’ | ’/’ | ’==’ | ’and’ | ’or’ | ’&&’ | ’||’
Unary_op             → ’-’ | ’not’ | ’!’
```

Fig. 6. BNF for the syntax of atomic and structured products

```
Process        → ’Process’ Process_name ’{’ Process_body ’}’
Process_body   → ’input:’ Input_def ’,’ ’output:’ Output_def ’,’
                 ’invariant:’ Invariant_def ’,’ ’requires:’ Requires_def ’,’
                 ’ensures:’ Ensure_def ’,’ Times_def
Input_def      → Product_ref | Composite_product_ref
Output_def     → Product_ref | Composite_product_ref
Invariant_def  → Expression
Requires_def   → Expression
Ensure_def     → Expression
Times_def      → ’l_time :’ NAT ’,’ ’u_time :’ NAT
```

Fig. 7. BNF for the syntax of atomic processes

5 From PPML to UPPAAL

With the aim of verifying temporal properties of PPML specifications, and providing a semantics for PPML in terms of timed automata, we propose a translation from PPML into UPPAAL. This will enable us, in particular, to employ the UPPAAL model checker for verifying real time properties of PPML models.

Essentially, the encoding of PPML into UPPAAL is defined in the following way. Each product class will correspond to a template, which will represent the product states in the system (i.e., all states that the product can have in the system) and a type that represents the product configuration, i.e., a structure that describes the product's attributes.

Processes are encoded as templates that mimic the PPML processes' behaviours in the system.

Gates are encoded as arrays of integers, declared as global variables. The values of an array representing a gate will correspond to the presence of the expected product at a given instant in the system. More precisely, if the array has in position i a value $n \neq 0$, then n is the code of a product available as input for the gate at instant i. Code 0 represents the absence of products. Since codes are unique for each product, they can be interpreted as references (with 0 being the null reference).

The encoding of gates is merged with those of products and process templates, in the following way:

- For a multiplexer gate M, the process that waits for the gate's output will have a conditional transition, checking whether $M[i] \neq 0$, for every i. All the products to be collected at the gate will also have in their corresponding templates a conditional transition checking whether the values in their corresponding positions in the gate are 0. If the condition holds and the transition is fired, the products update the array values with their corresponding codes.
- For a demultiplexer gate D, each process waiting for D's output checks if its input product is available.
- Semaphore gates are encoded as multiplexers, but with the process transition including an extra condition corresponding to the semaphore pass condition.

As is usual in model checking, we are forced to consider finite state systems, and thus we have to consider a maximum number of instances for product classes. For each product class (e.g., motherboard in our case study), we declare an array whose length is the maximum number of instances of the class, and which will hold the values of the attributes of these instances. For each process, we declare a broadcast channel, which is shared between the process and its input products, and is used to synchronize the start and finish tasks. The process manipulates products by updating their attributes, stored in the corresponding data array.

The independence of templates for processes and products enables us to provide a flexible parametrization of the system domain. For example, we can change the number of instances of products very easily, without altering the other constituents of the system. In our case study, for instance, we exploit this flexibility in order to "test" the system using different numbers of motherboards, processors and memory banks.

In order to illustrate the encoding, let us consider the process depicted in Fig. 8.

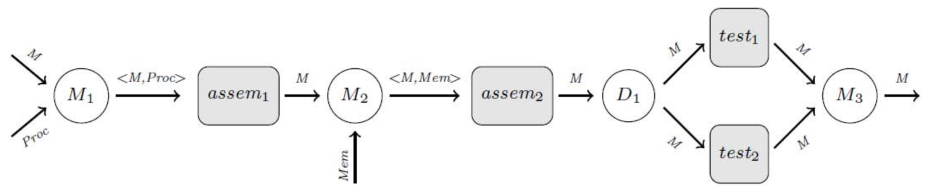

Fig. 8. Diagram for PPML process: $assem_1;assem_2;[test_1;test_2](D_1,M_3)$

The assembly process begins with the process $assem_1$ that takes a motherboard and a processor, and returns them assembled with a delay between 5 and 10 units of time. The next process is $assem_2$, which receives the output of the previous process and a memory bank, and assembles them with the same temporal bounds as the previous process. After that, the components are tested in parallel by the processes $test_1$ and $test_2$, whose lower and upper time bounds are 3 and 5 units of time.

5.1 Translation of Products

For each PPML product we generate:

– As global declarations, a range definition $0..n$ of integers, which specifies n instances of the product. These values will be used as codes for each of the system products, with 0 representing the *null* product. For each product class, a *struct type* declaration containing the data information of the product is also defined. In the case of a structured product, for each of its components, its type will have a variable of the corresponding product type. These variables will have the code of the composite product, or the value 0 (null, for incomplete products). Finally we declare an array of the product class type, whose length is the number of product instances. This array will contain the data values associated with these instances.
– A template with:
  - A clock variable declaration (time of the product in the system).
  - A function definition that updates its derived attributes.
  - A *constant*, used for representing the codes of the instances of products in the system.
  - *Locations*: an initial location, a location for each gate or process that manipulates it, and a final location.
  - *Edges*: without loss of generality, let us assume that we have a gate before every process. Thus we have two possible scenarios for edges: the edges go from a gate location to a process location, or from process locations to gate locations. Edges of the form (e_{gate}, e_{proc}), corresponding to the first of the scenarios described, are labeled with a conditional array gate update, where the update data is its code, and the condition expressing that the gate's product port is empty. Edges of the form (e_{proc}, e_{gate}), corresponding to the second kind of scenario described, are labeled with

a waiting message in the channel corresponding to the process whose code is in the gate. It also includes a call to the function that describes the updates of its calculated attributes.

When products are "copied" by a demultiplexer, all possible interleavings of processes modifying the product are taken into consideration.

– The instances of templates are declared in the system declarations.

As an example of product encoding, consider the specification given in Fig. 9, which is the result of encoding the memory product.

Global Declarations

```
//number of instances 2
 typedef int[1,2] n_memories;
//number of instances 2 + null
 typedef int[0,2] n_memories0;
//Prod. Structures declarations
 typedef struct {
   int size;
   bool assembled;
   bool tested ;
 } TMemory;
//Arrays for Prod. attr. values
 TMemory
     DataMemories[n_memories];
```

Template Declarations

```
clock x;
//Calculated Attr. Update
void UpdateAttr(){}
```

System Declarations

```
Memories (const n_memories id)
  := Memory(id);
```

Template states and edges

$$\begin{tikzcd} e_1 \arrow[r, "l_1"] & e_2 \arrow[r, "l_2"] & e_3 \arrow[r, "l_3"] & e_4 \arrow[r, "l_4"] & e_5 \end{tikzcd}$$

e1: Initial State (M_2)
l1: Guard: gMult2[1]== 0;
Update: gMult2[1]:= id
e2: $assem_2$
l2: Guard: gMultiplexer2[1]== id
Sync.: cassem2?
Update: UpdateAttr()
e3: D_1
l3: Guard: gMultiplexer3[1]== 0;
Update: gMult3[1]:=id
e4: $test_2$
l4: Guard: gMult3[1]== id
Sync.: ctest2?
Update: UpdateAttr()
e5: Final State (M_3)

Fig. 9. *Memory* Product translation

5.2 Translation of Processes

For each PPML process we generate:

– As global system declarations, channels to communicate the process with the products it is fed with, i.e., one channel and a variable of the corresponding code type are declared for each product involved in the process. The marshalled input products are simulated by the synchronizations of the above declared channels. We also generate a *broadcast* channel used by the process to "inform" the products that they are being processed.

– A template with:
  - A clock variable declaration (time within current process).
  - An initial location.
  - An update function, which expresses the action that the process performs on the products being processed. This is defined as an update of attributes of the products involved, defined as global shared variables.
  - The locations corresponding to: the idle process's location (a location for awaiting the arrival of products), a ready to process location (a location for the state in which all required products are queued for processing), and a final location (for the post-process state).
  - An edge connecting the idle location with the ready location, labeled with the condition that every array cell (codes of necessary products to start) be nonzero, and resetting its internal clock.

    The edge from the ready location to the post-processing location represents the activity of the process. So, it is labeled with the time constraints over the internal process clock, a broadcasting signal to the products involved and the update function call described above. Finally, an edge connecting the post-processing and idle locations is included, for resetting the process to receive a new job.

– In the system declarations, we create an instance for each process template.

As an example clarifying the process encoding, consider the specification shown in Fig. 10, which is the result of encoding the process of assembling a motherboard with a memory bank.

Global Declarations

```
//processing communication channel
broadcast chan cAssem2;
```

Template Declarations

```
clock x;
//Process Updates
void pAssem2(){
   //MotherBoard Update;
   DataMotherB[gMult2[0]].id_Memory:=
  gMult2[1];
}
```

System Declarations

```
Assem20 = Assem2();
```

States and Edges

$$\begin{tikzcd} e_1 \arrow[r, "l_1"] & e_2 \arrow[r, "l_2"] & e_3 \arrow[ll, bend right, "l_3"'] \end{tikzcd}$$

e1: Initial State (M_2)
l1: Guard: gMult2[0]!=0 and
gMult2[1]!=0
Update: x:=0;
e2: Pre-assem2
l2: Guard: x >=3 && x<=5
Sync.: cassem2!
e3: Pos-assem2
l3: Update: x:=0;
gMult2[0]:= 0;
gMult2[1]:= 0;

Fig. 10. assem2 Process translation

5.3 System Specification and Verification of Real Time Properties

We finish the specification of the system by defining the product and process templates. Once this is done, we have a complete UPPAAL specification, and we can employ the UPPAAL model checker in order to verify real time properties of our PPML model.

For our case study, we considered the following properties for verification:
1) *Processors and memory chips cannot be shared by (different) motherboards.*

```
A[] forall (i:n_motherboards) forall (j:n_motherboards)
 i != j imply ((MotherBoards(i).End and MotherBoards(j).End)
 imply (DataMotherBoards[i].id_Memory != DataMotherBoards[j].id_Memory
 and DataMotherBoards[i].id_Processor!=DataMotherBoards[j].id_Processor))
```

2) *Every motherboard that reaches the final location has its processor and memory already tested.*

```
A[] forall (i:n_motherboards) (MotherBoards(i).End imply
  DataMotherBoards[i].tested)
```

3) *The processing of the motherboards can be completed in less than 14 units of time.*
Notice that this corresponds to the testing phases being performed in parallel.

```
E<> exists(i:n_motherboards)(MotherBoards(i).End and
  MotherBoards(i).x<=13)
```

4) *There exists the possibility that all motherboards can be assembled and tested successfully.*

```
E<> forall(i:n_motherboards)(DataMotherBoards[i].tested)
```

The above properties were verified in a PC with an 2.33GHz Intel Core2 Duo CPU processor, with 2GB of DDR2 memory, running GNU/Linux with kernel version 2.6.28-11. The UPPAAL version employed was 4.1.1. The details of the verifications are summarized in the following time table:

	Test 1	Test 2	Test 3	Test 4	Test 5
1)	0,231 s	4,651 s	22,093 m	2,694 m	-
2)	0,013 s	2,583 s	14,371 m	2,413 m	-
3)	0,001 s	0,792 s	1,642 s	1,931 s	20,053 s
4)	0,001 s	3,970 s	13,339 m	6,017 m	-

In the above table, (s) indicates seconds, (m) minutes and (-) indicates that the verification process was stopped after the system memory was exhausted, i.e., when the amount of virtual memory used doubled the size of real (hard) memory. At this point the trashing process made the verification infeasible. The experiments associated with the columns of the table, referred to as “tests”, are the following:

- Test 1: 2 instances of each of the products, i.e., two motherboards, two processors and two memories.
- Test 2: 3 instances of each of the products.
- Test 3: 4 instances of each of the products.
- Test 4: three instances of motherboards and, five instances of processors and memories.
- Test 5: 6 instances of each of the products.

Notice that the experiments were carried out with quantities of processors and memories sufficient for assembling all the instances of Motherboard, i.e., *at least* one processor and one memory per motherboard. If we run the verification with fewer instances of memories or processors than instances of motherboards, some properties, such as, for instance, properties 2 and 4, may not be satisfied by the model.

These experiments are not only provided for illustration purposes, but also to show how limited the straightforward real time model checking is, for these kinds of specifications. Indeed, even though our case study is rather small, the number of instances of products that the model checker is able to deal with (in a standard desktop computer) is also quite small. The reader might argue that the inefficiency might be due to our translation; however, checking an ad hoc UPPAAL characterisation of the described case study yielded similar analysis results.

This fact shows that abstraction mechanisms, such as those supported by PPML via framework processes, are crucial for scaling up the analysis tasks. We plan, as future work, to take advantage of framework processes in order to improve the analyzability of PPML specifications (most likely, employing abstraction techniques).

6 Related Work and Conclusions

There are several approaches proposing formalizations for business process languages and their web services extensions, such as BPEL and WS-BPEL. A survey of formal verification for business process modelling approaches can be found in [22], where a classification of proposals for formally analyzing business processes is presented. Three kinds of formal semantics are the focus of the survey, namely automata, Petri nets and process algebra based semantics. Exhaustive reviews of approaches in the area of business process modeling and analysis are also presented in [11,12], where a translation from UML web services into FSP is proposed, so that web service specifications can be analyzed using LTSA.

Some approaches focus on providing formal semantics for the Business Process Modeling Notation (BPMN), such as for instance the works presented in [27,29]. Such semantics allows for the analysis of compatibility of business collaborations at design level. It also enables a pattern-based approach to the specification of behavioural properties (which can be verified), using Dwyer et al.'s approach. The same authors present a relative-timed semantic model for BPMN [28], and show how properties can be automatically verified using model checking via the

FDR tool. Another related approach is that presented in [16], where a semi automated translation from business process diagrams (BPD) into TLA+ is presented, using Petri nets as an intermediate formalism for the translation. This (conservative) translation allows one to model check properties of business processes, expressed as TLA formulas, using the TCL model checker.

A BPEL formalization, closely related to ours, is proposed in [23], mapping BPEL into timed automata. This formalization allows for checking deadlock freedom and reachability properties via the UPPAAL model checker, and is integrated into the ActiveBPEL tool. Other related approaches based on BPEL and BPEL4WS are the business process formalizations associated with the study of transactions and fault handling via compensations [24].

Other attempts at formalizing and analyzing workflow languages, such as UML activity diagrams, have been proposed; an example of this is that presented in [13], translating these kinds of diagrams into PROMELA (the language of the SPIN model checker). There exist various approaches providing formal semantics for workflow languages based on Petri nets or timed extensions of Petri nets [25].

A primary difference of our approach with respect to the ones described above is that PPML puts an emphasis on product description that, as far as we are aware of, is not available in any other business process modeling language. This capability enables the specifier to "balance" the description of business processes adequately, using a rich language for describing products in order to make process descriptions simpler. The language also offers timing constraints, given in the form of two bounds associated with processes, the lower and upper bounds. These are useful features with an intuitive meaning, that enable the specifier to annotate activities with timing restrictions, so that timing related properties, such as throughput or response time, can be analyzed. We took into consideration these characteristics of the language, and proposed a translation from PPML into UPPAAL, so that real time properties of PPML models can be verified using model checking. The query language (the language for expressing the properties to be checked) is a rather expressive language (computational tree logic with certain restrictions), enabling one to specify a wide range of properties, including safety and liveness properties. We have introduced a syntax for the language (the constituent elements of the formalisms have been formally defined in previous work, but no appropriate syntax for the language was provided), and an encoding of the language into UPPAAL, that provides, indirectly, a timed automata semantics of the language, and the direct possibility of model checking specifications. We have also illustrated the verification of some sample properties, using the proposed translation into UPPAAL.

Currently, we are developing a software tool to assist in the creation of PPML models, and the translation of PPML models into UPPAAL is being developed as a plug-in of this tool. We believe that PPML is a language that is useful for the formal specification (and now the analysis) of industrial processes. Directions for future work include developing abstract interpretation mechanisms associated with PPML models, so that the verification via model checking can be improved (by tackling the well known state explosion problem). More precisely, we plan to work on predicate abstraction [7] techniques for PPML analysis, exploiting the framework processes available in the language.

We also plan to prove that the new semantics, that is indirectly provided for the language via the encoding into UPPAAL, is in fact compatible with the original timed transition systems semantics of PPML given in [19].

Acknowledgements

The first two authors were partially supported by CONICET, the Argentinian Agency for Scientific and Technological Promotion (ANPCyT) and the Ministry of Science and Technology of the Province of Córdoba in Argentina. The third author was partially supported by McMaster University, the Canada Research Chair programme, and the Natural Sciences and Engineering Council of Canada.

References

1. Andrews, T., et al.: Business Process Execution Language for Web Services version 1.1, `http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel/ws-bpel.pdf`
2. Baum, G., Frias, M.F., Maibaum, T.S.E.: A Logic for Real-Time Systems Specification. In: Algebraic Semantics, and Equational Calculus, AMAST, pp. 91–105 (1998)
3. Bengtsson, J., et al.: UPPAAL- A Tool Suite for the Automatic Verification of Real-time Systems. In: Alur, R., Sontag, E.D., Henzinger, T.A. (eds.) HS 1995. LNCS, vol. 1066, pp. 232–243. Springer, Heidelberg (1996)
4. Carvalho, S., Fiadeiro, J., Haeusler, E.: A Formal Approach to Real-Time Object Oriented Software. In: Proceedings of the Workshop on Real-Time Programming IFAP/IFIP, Lyon, France, pp. 91–96 (1997)
5. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Transactions on Programming Languages and Systems 8, 244–263 (1986)
6. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (2000)
7. Das, S., Dill, D., Park, S.: Experience with Predicate Abstraction. In: 11th International Conference on Computer-Aided Verification, pp. 160–172. Springer, Heidelberg (1999)
8. Daws, C., et al.: The Tool KRONOS. In: Alur, R., Sontag, E.D., Henzinger, T.A. (eds.) HS 1995. LNCS, vol. 1066, pp. 208–219. Springer, Heidelberg (1996)
9. Fenton, N., Pfleeger, S.L.L.: Software Metrics: A Rigorous and Practical Approach, Course Technology 2nd edn. (1998)
10. Fiadeiro, J., Maibaum, T.: Temporal Theories as Modularisation Units for Concurrent System Specification. In: Formal Aspects of Computing, pp. 239–272 (1992)
11. Foster, H., et al.: LTSA-WS: A Tool for Model-based Verification of Web Service Compositions and Choreography. In: 28th International Conference on Software Engineering (ICSE 2006), pp. 771–774 (2006)
12. Foster, H., et al.: WS-Engineer: A Model-Based Approach to Engineering Web Service Compositions and Choreography. In: Test and Analysis of W.S., pp. 87–119 (2007)

13. Guelfi, N., Mammar, A.: A Formal Semantics of Timed Activity Diagrams and its PROMELA Translation. In: APSEC, pp. 283–290 (2005)
14. Kavantzas, N., et al.: Web Services Choreography Description Language Version 1.0, `http://www.w3.org/2002/ws/chor/edcopies/cdl/cdl.html`
15. Koehler, J., Tirenni, G., Kumaran, S.: From Business Process Model to Consistent Implementation: A Case for Formal Verification Methods. In: 6th International Enterprise Distributed Object Computing Conference (EDOC 2002), pp. 96–106. IEEE Computer Society, Los Alamitos (2002)
16. Masalagiu, C., et al.: A Rigorous Methodology for Specification and Verification of Business Processes. In: Formal Aspects of Computing. Springer, London (2009)
17. Myers, M., Kaposi, A.: A First Systems Book: Technology and Management, 2nd edn. Imperial College Press, London (2004)
18. Henzinger, T.A., Manna, Z., Pnueli, A.: Timed Transition Systems (1996)
19. Maibaum, T.S.E.: An Overview of The Mensurae Language: Specifying Business Processes. In: Rigorous Object-Oriented Methods, BCS (2000)
20. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems - Specification. Springer, Heidelberg (1991)
21. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems -Safety. Springer, Heidelberg (1995)
22. Morimoto, S.: A Survey of Formal Verification for Business Process Modeling. In: ICCS 2008, pp. 514–522 (2008)
23. Qian, Y., et al.: Tool Support for BPEL Verification in ActiveBPEL Engine. In: Australian Software Engineering Conference, pp. 90–100. IEEE Computer Society, Los Alamitos (2007)
24. Qiu, Z., et al.: Semantics of {BPEL4WS}-Like Fault and Compensation Handling. In: Proceedings of the International Symposium on Formal Methods 2005, pp. 350–365. Springer, Heidelberg (2005)
25. Van der Aalst, W., Ter Hofstede, A.: YAWL: Yet Another Workflow Language. Information Systems 30, 245–275 (2003)
26. W3C, Web Service Choreography Interface 1.0 (2002), `http://www.w3.org/TR/wsci`
27. Wong, P.Y.H., Gibbons, J.: A Process Semantics for BPMN. In: Liu, S., Maibaum, T., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp. 355–374. Springer, Heidelberg (2008)
28. Wong, P.Y.H., Gibbons, J.: A Relative Timed Semantics for BPMN. In: Proceedings of 7th International Workshop on the Foundations of Coordination Languages and Software Architectures. ENTCS (2008)
29. Wong, P.Y.H., Gibbons, J.: Property Specifications for Workflow Modelling. In: Leuschel, M., Wehrheim, H. (eds.) IFM 2009. LNCS, vol. 5423, pp. 56–71. Springer, Heidelberg (2009)