From Object-Process Diagrams to a Natural Object-Process Language

Mor Peleg and Dov Dori

Faculty of Industrial Engineering and Management
Technion—Israel Institute of Technology
Haifa 32000, Israel
{mor, dori}@ie.technion.ac.il

Abstract. As the requirements for system analysis and design become more complex, the need for a natural, yet formal way of specifying system analysis findings and design decisions are becoming more pressing. We propose the Object-Process Language as a textual natural language means for systems specification that is complementary to the graphic description through Object-Process Diagrams.

1 Natural Languages and Graphic Representations of Systems

As the complexity of computer-based systems grows, precise and concise specification is in increasing demand, calling for new approaches of systems development. One very common and powerful modeling technique is simple prose, or natural language. Natural languages are very powerful because they are the result of thousands of years of evolution, during which humans who use them have developed the ability to make subtle observations and distinctions, such as those made in this sentence. However, natural languages are also frequently ambiguous and always linear, i.e., they must be read and spoken in a linear way – the "reading order". Systems in general and reactive systems in particular, feature things that exist or happen concurrently, making it difficult to model them with prose alone.

Formal textual languages based on logics and algebras are often used to specify complex dynamic systems [1]. Their formality makes it possible to verify desired system properties and check them for internal consistency. These languages are precise, amenable to verification and can potentially also serve as a basis for automatic conversion into executable code. Nevertheless, the task of specifying a system in logic is very difficult, and the resulting specification is hard to follow and understand by non-experts. Formal methods, notation and tools do not yet adequately support the development of large and complex systems [2]. Logics and algebras suffer from one more disadvantage; in addition to their being linear too, they are not intuitive and cannot be understood by non-experts. To demonstrate this, Fig. 1 presents specifications of the system requirement "Following the event of a repairman arriving at an elevator, the action Checking begins its execution within 1 minute" in three different logic languages.

The specification in Fig. 1(a) is done in Real-Time Logic (RTL) [3], the one in Fig. 1(b) is done in Metric Temporal Logic (MTL) [4], and that of Fig. 1(c) – in Timed Transition Models/Real Time Temporal Logics (TTM/RTTL) [5]. While some humans may understand a particular specification more than the other, it is evident from this small example that formal logic-based languages are far from being natural and intuitive, and therefore require a considerable amount of learning, training and adjustment that very few people would be willing to invest. In this work, we propose Object-Process Language (OPL) as a language that is both formal and intuitive, thereby catering to the need of humans on one hand and machines on the other hand. Fig. 1(d) presents the OPL specification of the same system requirement discussed above, which resembles a sentence in English.

$\forall x$ [@(ΩREPAIRMAN_ARRIVED_AT_ELEVATOR, x) ≤ @(↑CHECKING, x) ∧ @(↑CHECKING, x) ≤ @(ΩREPAIRMAN_ARRIVED_AT_ELEVATOR, x) + 1]
Legend: Ω - event occurrence; @ - time of event occurrence; ↑ - beginning of an action;

(a)

Repairman_arrived_at_elevator → Eventually$_{\leq 1}$ checking

(b)

$\forall$T [(repairman_arrived_at_elevator ∧ t = T) → Eventually (checking ∧ t = T+1)]

(c)

Event **RepairmanArrivedAtElevator**[external] triggers **Checking** with a reaction time of (**0, 1 m**).

(d)

Fig. 1. Specification by logics vs. specification by OPL. (a) RTL; (b) MTL; (c) TTM/RTTL; and (d) OPL. All four specifications specify that following the event of a repairman arriving at an elevator, the action Checking begins its execution within 1 minute.

Diagrams are often invaluable for describing models of abstract things, especially complex systems. An accepted diagramming method has the potential of becoming a powerful modeling tool, provided that it really constitutes an unambiguous language, or a *visual formalism* [6]. A visual formalism is valuable if each symbol in the diagram has a defined semantics and the links among the symbols unambiguously convey some meaningful information that is clearly understood to those who are familiar with the formalism. Object-Process Diagrams (OPDs) provide such a concise visual formalism for specification of systems of all domains and complexity levels analyzed by the Object-Process Methodology (OPM) [7]. OPM is founded on an ontology that distinguishes between objects and processes as two types of things of basically equal status and importance in the specification of a system. The OPM model shows how objects interact with each other via processes such that both the structural and the procedural system aspects are adequately represented. OPM/T [8] is the extension of OPM that handles reactive and real-time systems by specifying system dynamics, including triggering events, guarding-conditions, temporal constraints and exception handling.

2 The Object-Process Language

The Object-Process Language is the textual, natural language-like equivalent of the graphical representation of the system being studied or developed through the OPD set. OPL is designed such that it is very close to English as a natural language, but with much more stringent and limited syntax. The similarity of OPL to natural language makes it readable and understandable to humans without the need to learn any programming or pseudo-code-like language. The system's OPL specification, resulting from an OPD set, is thus amenable to being checked by domain experts, who need not be software experts. While humans can also inspect diagrams, they are not expected to master all the OPD symbols and their semantics. The information provided in a natural language through OPL complements the graphic information and provides for a self-learning tool that can be consulted when in doubt as to the meaning of some graphic construct. Reviewing both the OPD set and is corresponding OPL sentence set decreases the likelihood of missing specification assertions.

This synergy between text and graphics is instrumental in closing the gap between the requirement specification, which is usually expressed in prose, and the actual system specification, resulting from the OPM analysis and design. The syntax of OPL is well defined and unambiguous. This eliminates the problem of the fuzziness of natural languages and provides a firm basis for automated implementation of executable code generation and database schema definition.

Through a detailed walkthrough of a simple case study, this paper presents and demonstrates the principles and syntax of the Object-Process Language, its equivalence to the graphic description done by Object-Process Diagrams and the production of OPL sentences with the OPL grammar.

3 The Elevator Checkup System

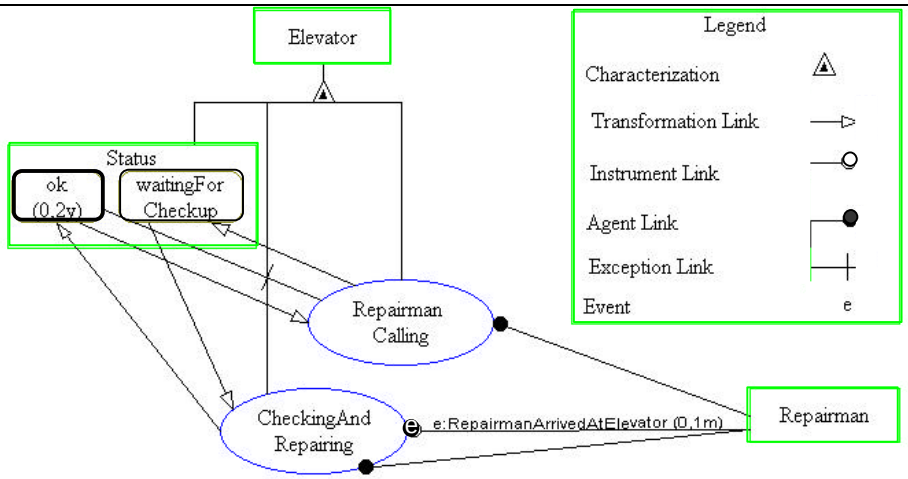
The Elevator Checkup and Repair is a simplified case study of a reactive system. The system requirement specification (SRS) document for this small example is listed in Fig. 2. Fig. 3 shows an OPD of the Elevator Checkup and Repair case study and its equivalent OPL text. The OPL text contains OPL sentences, which specify the system in Depth First Search (DFS) order. This ordering means, for example, that after listing the features of a thing *T*, we specify in detail the features and/or parts of each feature of *T* before moving on to specify the parts of *T*, if such parts exist.

> The elevator of a building has to be checked every two years. When the time for a checkup of the elevator arrives, the repairman is summoned. Within one minute of the repairman's arrival to check the elevator, the elevator is checked, and if necessary, repaired, so that it is again in a usable condition.

Fig. 2. The formal system requirement specification (SRS) document for the Elevator Checkup and Repair case study.

The top of the OPD of Fig. 3, as well as the first OPL sentence specify exactly what is understood naturally from reading it, namely the features that characterize Elevator: its object **Status**, and its two characterizing processes (methods, or services)

RepairmanCalling and **CheckingAndRepairing**. The "features" clause corresponds to the characterization symbol, denoted by the black triangle within a white triangle, as shown in the legend of Fig. 3.



Object **Elevator** features **Status**, Process **RepairmanCalling** and Process **CheckingAndRepairing**.

Object **Status** has states **waitingForCheckup** and **ok**[timed (**0**, **2 y**), in tial].

Event **Status_ok**[state timeout] of **Status** of **Elevator** triggers **RepairmanCalling** of **Elevator**.

Process **RepairmanCalling** is **guarded** by "**Status** is **ok**".

RepairmanCalling is enabled by **Repairman**[agent].

RepairmanCalling affects **Status** of **Elevator** from **ok** to **waitingForCheckup**.

Event **RepairmanArrivedAtElevator**[external] of **Elevator** triggers **CheckingAndRepairing** of **Elevator** with a reaction time of (0, 1 m).

Process **CheckingAndRepairing** is guarded by "**Status** is **waitingForCheckup**".

CheckingAndRepairing is enabled by **Repairman**[agent].

CheckingAndRepairing affects **Status** of **Elevator** from **waitingForCheckup** to **ok.**

Fig. 3. An OPD and equivalent OPL text of the Elevator Checkup case study.

The following OPL sentences specify the details of the features of Elevator. The **Status** attribute of **Elevator** is specified as an object, which has two states: **waitingForCheckup** and **ok**. State **ok** is the initial state of **Status**, as can be seen in the OPD by its wider contour, and in the OPL sentence, by the use of the keyword initial. State **ok** is time constrained, meaning, in this case, that there is an upper time limit to being in that state.

The maximum time that can be spent in that state is two years (while the minimum time is zero). This is specified in the OPD as the interval (**0**, **2 year**), written within state **ok** of object **Status**, which represents the duration constraint associated with this state. The analogous OPL sentence "Object **Status** has states **waitingForCheckup** and **ok**[[timed (**0**, **2 year**), initial]." specifies these facts just as well.

Once **Status** has been in state **ok** for more than **2** years, a timeout event is triggered, which triggers the **RepairmanCalling** process. This timeout event is specified in the OPD by the Exception link emanating from state **ok** of **Status** to process **RepairmanCalling.** It is also specified by the OPL sentence "Event **Status_ok**[state timeout]

manCalling. It is also specified by the OPL sentence "Event **Status_ok**[state timeout] of **Status** of **Elevator** triggers **RepairmanCalling** of **Elevator**.".

The name of this event is **Status_ok**, which is a combination of the name of the state that is duration constrained (**ok**), preceded by the name of the object containing this state (**Status**). The event name is followed by the event type, which, in this case is [state timeout], and by the path that leads from state **ok** of object **Status** to its Root Object. The Root Object of an object B_1 is defined as an ancestor of B_1 that has no ancestors. In our example, the Root Object is **Elevator**.

The process **RepairmanCalling** can take place only if object **Status**, characterizing the same **Elevator**, is in state **ok**. This is specified by the fact that in the OPD, the transformation link (denoted as an arrow), emanates from this state toward the process **RepairmanCalling**. This same guarding condition is specified by the corresponding OPL sentence

Process **RepairmanCalling** is guarded by "**Status** is **ok**".

The process **RepairmanCalling** is enabled by the agent **Repairman**, who has to participate in the process. This is specified by the OPL-sentence "**RepairmanCalling** is enabled by **Repairman**[agent].".

The process **RepairmanCalling**, which characterizes **Elevator**, affects **Status** of that same **Elevator**. The effect is clearly expressed by the OPL-sentence "**RepairmanCalling** affects **Status** of **Elevator** from **ok** to **waitingForCheckup**.". In the OPD, it is reflected by the transformation link emanating from state **ok** of **Status** to process **RepairmanCalling** and the transformation link emanating from **RepairmanCalling** to state **waitingForCheckup** of **Status**. Note that the **RepairmanCalling** process is not only responsible for calling the **Repairman** but also for changing the state of the **Elevator Status**. In OPM, only processes change the state of an object. There are no direct transitions between object states in which no process is involved.

When **Elevator** is at state **waitingForCheckup**, it can respond to the external event marking the arrival of a **Repairman** at the **Elevator**. Within 1 minute, this external event triggers the **CheckingAndRepairing** process of **Elevator**. In the OPD this is specified by the agent link from **Repairman** to **CheckingAndRepairing** that has the letter *e*, for event, written inside the agent link circle, along which the label "e: **Repairman Arrived at Elevator (0,1 m)**" is written. The corresponding OPL sentence is

Event **RepairmanArrivedAtElevator**[external] of **Elevator** triggers **CheckingAndRepairing** of **Elevator** with a reaction time of (**0, 1 m**).

This sentence specifies the constraint on the reaction time elapsed from the occurrence of the event and the start of the execution of the process. This reaction time constraint is reflected in the OPD by the interval (**0, 1 m**) attached to the event link that connects the triggering object **Repairman** to the triggered process **CheckingAndRepairing**. The same constraint was specified by different logic languages in Fig. 1.

The OPL sentences

Process **CheckingAndRepairing** is guarded by "**Status** is **waitingForCheckup**".
CheckingAndRepairing is enabled by Repairman[agent].
CheckingAndRepairing affects **Status** from **waitingForCheckup** to **ok.**

specify that (a) the **CheckingAndRepairing** process can only take place if the object **Status** is in state **waitingForCheckup**; (b) **CheckingAndRepairing** is enabled by the agent

Repairman; and (c) **CheckingAndRepairing** affects **Status** by changing its state from **waitingForCheckup** to **ok**. The OPD of Fig. 3 expresses this graphically.

4 OPL Production Rules

The syntax of OPL is defined as a set of production rules through a context-free grammar [9] G_{OPL}. Like all context-free grammars, $G_{OPL} = \{S, P, s, T\}$, where
S is the *Starting* symbol, from which all OPL sentences are produced;
P is the set of *Production rules*;
s is the set of *Non-terminals;* and
T is the set of *Terminals.*

Due to lack of space, we cannot list the entire set of production rules. Instead, in Fig. 4, we provide a subset of rules that are used to produce the OPL sentence "Object **Elevator** features **Status**, Process **RepairmanCalling** and Process **Checking**." of Fig. 3 and show how this production is obtained using these rules.

A production rule (e.g., $A \rightarrow B$) consists of a non-terminal, called the *left side* of the production rule, an arrow, and a sequence of terminals and/or non-terminals called the *right side* of the production rule. Time New Roman Italics are used for representing non-terminals, or expressions, which can be decomposed into other non-terminals and/or terminals, by applying other production rules. Non-italics Arial font letters stand for terminal symbols, which are identifiers (e.g., the object-name **Elevator**) or reserved words (e.g., for). All OPL sentences use the **Arial** font. Identifiers (e.g., **Elevator**) are marked in **bold**, whereas reserved words are in regular font style. The non-terminal S is designated as the *start* symbol, from which all OPL-sentences are constructed. The '|' symbol represents the logical OR relation. It can be used to combine two production rules that have the same *left side* (e.g., the production rules $A \rightarrow B$ and $A \rightarrow$ c can be combined into the production $A \rightarrow B \mid$ c).

The first OPL production rule, R1, specifies that an OPL sentence, produced from the initial symbol *S*, represents an Object expression, a Process expression, an Event expression or a Relationship expression (Relationship expression is not relevant to our example). R2 states that an object may be simple or complex. R3 states that a complex-object expression begins with the reserved word "Object", followed by the expression *object-statement**s***. R4 states that an *object-statement**s*** expression can simply be an *object-statement* expression, or can be decomposed into an *object-statement* expression followed by an *object-statement**s*** expression. R5 specifies that an *object-statement* expression can specify object features, parts, states, base class, or derived classes. Rules R6 and R7 are used to derive the *Object-names* expression. Rules R8 through R10 define an *Object-name* as a regular expression that begins by a *Capital Letter*, followed by a sequence of any number (* stands for 0 to many consecutive occurrences) of letters. Rules R11 through R14 specify how the *object-process-list* and *process-list* expressions are expanded.

Fig. 5 shows how the subset of production rules listed in Fig. 4 was used to derive the OPL sentence Object **Elevator** features **Status**, process **RepairmanCalling** and process **Checking**. The rule numbers are listed above the production rule arrows. To enhance

readability, terminals in Fig. 5 are surrounded by frames, such that in the beginning no frame appears, and gradually they become larger and more frequent, until finally the entire sentence is surrounded by one frame.

(R1) S → *Object* | *Process* | *Event* | *Relationship*
(R2) *Object* → *simple-object* | *complex-object*
(R3) *complex-object* → Object *object-statements*
(R4) *object-statements* → *object-statement* | *object-statement object-statements*
(R5) *object-statement* → *Object-name* features *object-process-list***.** |
Object-name consists of *aggregate-list***.** | *Object-name* has states *state-list* . |
Object-name specializes *Object-names***.** |*Object-name* generalizes *Object-names***.**
(R6) *Object-names* → *Object-name* |
Object-names-within-commas and *Object-name*
(R7) *Object-names-within-commas* → *Object-name* |
Object-name, Object-names-within-commas
(R8) *Object-name* → *CapitalLetter* (*letter)**
(R9) *CapitalLetter* → [**A..Z**] (R10) *letter* → [**a..z, A..Z**]
(R11) *object-process-list* → *Object-names* |
Object-names-within-commas and Process *Process-name* |
*Object-names-within-commas***,** *process-list*
(R12) *Process-name* → *CapitalLetter* (*letter* | *digit*)*
(R13) *process-list* → Process *Process-name* |
process-names-with-commas and Process *Process-name*
(R14) *process-names-with-commas* → Process *Process-name* |
process-names-with-commas , Process *Process-name*

Fig. 4. The subset of OPL production rules required to produce the OPL sentence Object **Elevator** features **Status**, Process **RepairCalling** and Process **Checking**.

5 Summary and Work in Progress

We have presented the principles of the Object-Process Language as the textual equivalent of the corresponding Object-Process Diagrams. We show how both the OPD and its OPL equivalent specify a system that is being analyzed. Although, for the sake of brevity, the example is simple, it does contain most of the elements required to model systems at any level of complexity, including reactive and real-time systems. The specification reader, when presented with both the graphic and the textual specification modes, enjoys the synergy of this combination. When in doubt as to the semantics of one mode, the other one can always be consulted, and so the two modes support and reinforce each other. The OPDs can, for example, be read in any sequence, and not just in the linear "reading order", which is helpful for presenting concurrent processes. Some people feel more comfortable with one representation while other – with the other.

The resulting OPL text is used for two major purposes: one is to provide a concise specification of the analyzed and designed system feedback to the prospective system customer, while the other is to automate the application generation, i.e., the executable code and database schema generation. Work in progress tackles these two tasks and we already have a working program for C++ code generation that is in line with [10] and database schema generation for relational, object-oriented and object-

relational database types. The OPL compiler that generates C++ code creates full code, including a header file and a code file implementing object methods. It relies upon predefined modules that define, among others, classes of processes, states, events and an event queue. We are also improving the readability of OPL by eliminating non-natural elements like parentheses of various kinds.

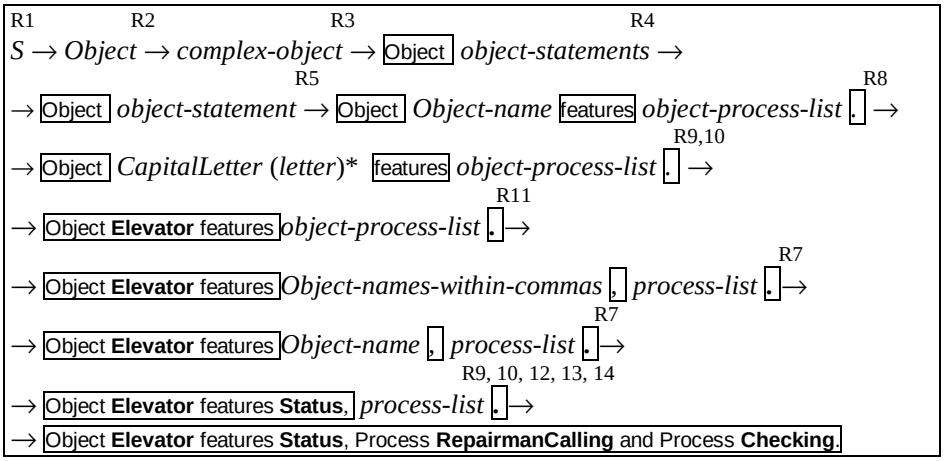


Fig. 5. Production of the OPL sentence "Object **Elevator** features **Status**, Process **RepairCalling** and Process **Checking**" by applying the OPL production rules listed in Fig. 4.

References

1. Ostroff, J.S. : Formal Methods for the Specification and Design of Real-Time Safety Critical Systems. The Journal of Systems and Software 18, 1, (1992) 33-60
2. Luqi and Goguen J.A.: Formal Methods: Promises and Problems. IEEE Software 1 (1997) 73-85
3. Jahanian, F. and Mok, A.: Safety Analysis of Timing Properties in Real-Time Systems. IEEE Transactions on Software Engineering Vol. SE-12, No. 9 (1986) 890-904
4. Koymans, R.: Specifying Message Passing and Time Critical Systems With Temporal Logic. Lecture Notes in Computer Science, Vol. 651. Springer-Verlag , Berlin Heidelberg New York (1992)
5. Ostroff, J.L., and Wonham, W.M.: A Framework for Real-Time Discrete Event Control. IEEE Transactions on Automatic Control, 35, 4. (1990) 386-397.
6. Harel, D.: Statecharts: a Visual Formalism for Complex Systems. Science of Computer Programming (1987) 231-274.
7. Dori D.: Object-Process Analysis: Maintaining the Balance between System Structure and Behavior. Journal of Logic and Computation. 5, 2 (1995) 227-249.
8. Peleg, M. and Dori, D.: Extending the Object-Process Methodology to Handle Real-Time Systems. Journal of Object-Oriented Programming 11, 8, (1999) 53-58.
9. Aho, A.V., Sethi, R. and Ullman, J.D.: Compilers/Principles, Techniques, and Tools, Addison-Wesley, 1986.
10. Dori, D. and Goodman, M.: From Object-Process Analysis to Object-Process Design, Annals of Software Engineering, Vol. 2: Object-Oriented Software Engineering: Foundations and Techniques (1996) 25-20.