Extending the Object-Process Methodology to Handle Real Time Systems

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; Fax 972-4-8235194 , Tel: 972-4-8294409 / 2921

Abstract

There is general consensus in the software literature that real-time systems are difficult to model, specify, and design. It is an important and challenging task to develop an intuitive and easy-to-use, yet coherent and concise method for specifying such systems. The Object-Process Methodology (OPM) graphically specifies systems in a single unified model that describes the static-structural and behavioral-procedural aspects of a system by a set of Object-Process Diagrams. In this research paper we present OPM/T, as an extension of OPM for specification of reactive and real-time systems. A detailed telephone call example demonstrates the power of OPM/T to express such notions as timing constraints, events, conditions, exceptions, and control flow constructs.

Pertinent Subjects — analysis and design methods, real-time systems, design patterns.

1. Introduction

Reactive systems and real-time systems play an important role in many technological advents. The Real time aspect is critical in such systems as chemical processing, road-traffic, nuclear reactions, satellite control, and airplanes and missile navigation. There is general consensus in the software and control systems literature [9] that real-time systems are difficult to model, specify, and design. This is due to the fact that issues such as concurrency, synchronization among processes, and real-time constraints must be expressed explicitly and unambiguously. It is an important and challenging task to develop an intuitive and easy-to-use, yet comprehensive and concise method for specifying such systems.

Reactive systems [5] are systems which are event-driven, continuously having to react to external and internal stimuli. The behavior of reactive systems cannot be specified by merely giving the inputs and outputs of the system. There is a need to represent the **control** component which determines the order and timing of processes. One customary way of expressing a system's response to events is by applying a set of Event-Condition-Action (ECA) rules. These rules specify that a process (action) in a reactive system is executed when the triggering event occurs, provided the conditions guarding the process execution are fulfilled.

Real-time systems constitute a subset of reactive systems, in which timing constraints may be **quantitative**. The constraints may be on the time between an event and the system's response, on the execution time of a process, on the time the system stays in a specific state, etc.

This research paper presents OPM/T — a real-time extension to the Object-Process Methodology (OPM) [1]. The paper is organized as follows. Section 2 briefly surveys specification methods for reactive and real-time systems. Section 3 introduces and demonstrates the basics of OPM and OPM/T. Section 4 describes the details of the OPM/T specification of a telephone call process, whichdemonstrates the use and benefits of OPM/T. A discussion concludes the paper.

2. Specification Methods for Reactive and Real-Time Systems

Specification of reactive and real-time systems is currently done in a number of methods, which can be roughly divided into two groups [9]. The first group consists of graphical methods, while the second one is based on logics and algebras. Following is a comparison of the different specification methods. The methods are compared in terms of their ability to model real-time systems and verify the specification, and their usefulness as providers of a sound basis for design and implementation.

Graphical methods used for specification of real-time systems can be categorized as (1) system structure, function, and dynamics modeling methods, (2) methods that specify system dynamics only, and (3) real-time design methods. The first group of methods includes, among others, Object Oriented Analysis (OOA), Object Modeling Technique (OMT) [11], Object Oriented Software Engineering (OOSE), Object Life Cycles, and Object Oriented System Analysis (OSA) [4], reviewed in [2]. The methods belonging to this category describe a system from three different aspects: structural — objects and the relationships among them, using mainly ERDs—Entity-Relationship Diagrams of various forms, functional — the processes executed in the system, including their inputs and outputs, using mainly DFDs—Data Flow Diagrams, and dynamic — control of process execution and changes in an object's state, using methods based on Finite State Machines and their elaborations, such as Statecharts.

While these methods are easy to use, and their resulting specification can be understood by non-experts, they suffer from two main disadvantages. First, they do not fully support expression of temporal constraints and reference to events. Second, to specify a system, these methods use a combination of at least three models, each describing a different aspect of the system. Incompatibility problems, such as mismatches among names of objects and processes are more likely to occur when more than one model is used. Much more problematic is the integration of the different models that describe different aspects of the system which is seldom explicit. Hence, the difficult task of mental integration has to be done by the analyst and the audience to which the analysis is intended. Except for OSA, all system structure, function, and dynamics modeling methods have no formal semantics and no support for formal verification. OSA [4] can support validation based on prototype execution of analysis application models. Some of the modeling methods discussed above are supported by CASE tools, which are supposed to facilitate the transition from specification to design, and from design into executable code.

Statecharts [5], Modecharts [8] and Petri Nets [10] are methods that specify system dynamics only. Both Statecharts and Modecharts are based on Finite State Machines, but they are also capable of hierarchical and structural decomposition into sub-states, which may coexist in parallel or exhibit an exclusive-or (XOR) relationship. Statecharts have been extended in [3] so as to express quantitative timing constraints. STATEMATE [6], the graphical tool which implements Statecharts, has an automated simulation tool that allows the user to execute his/her model. Modecharts [8] are capable of expressing quantitative sporadic and periodic timing constraints. Their specification is done in Real Time Logic (RTL) [9]. Although both Statecharts and Modecharts are formal methods, formal verification is not yet supported for their resulting specifications.

Petri Nets [10] is a formal graphical language that can express concurrency, nondeterminism, and cause-and-effect relationships between events and states. Timed Petri Nets were developed to express temporal constraints [9]. Petri Nets are especially useful for performance evaluation of modeled systems. They can be formally checked for boundedness, safety, and freedom from deadlock. Invariants can be checked for small systems. But Petri Nets can overkill if the system is too simple, while if the system is too complex, timing can become obscured.

Jackson System Development (JSD) [7] and Sanden's Entity-Life Modeling [12] are real-time design methods that can also be used for specification. Both methods focus on the implementation domain. The graphical diagrams these methods yield present the different tasks performed by objects and the communication among them. In addition to the graphical diagrams, these methods also rely heavily on pseudocode, which is hard to follow and use. The advantage of using pseudocode is that the specification can potentially be converted to executable code.

Logics and algebras used for specification of real-time systems are reviewed by Ostroff in [9]. The main advantages of using logics and algebras in specification of systems are that any temporal property can be specified, and that the specification can be verified for correctness by mathematical methods. 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. Mechanical theorem provers have failed to be of much help due to the inherent complexity of testing validity for even the simplest logics. There are no available tools yet for design and implementation of specifications given in logics and algebras.

3. The Object-Process Methodology (OPM) and OPM/T

The Object-Process Methodology (OPM) [1] incorporates the static-structural and dynamic-procedural aspects of a system into a **single** unifying model. OPM achieves this by treating both **objects** and **processes** as **things** (entities) which have equal status. OPM handles complex systems by using recursive seamless scaling. It is suitable both for system analysis and system design, and enables smooth transition between theses phases.

In OPM, objects are viewed as persistent entities interacting with each other through processes—transient entities that affect objects by changing their state. Object-Process Diagrams

(OPDs) enable us to describe things (objects and processes) and how they interact with each other. Things can be simple or compound. A compound thing is a thing which is a generalization of other things, or an aggregation of other things, or is characterized by other things. Objects may serve as enablers — instruments or intelligent agents, which are involved in a process without changing their state, or they may be affected (or generated or consumed) by a process. OPDs can depict both sequential and parallel processes, and accommodate expressions of branching.

OPM/T is an extension of OPM for specification of reactive and real-time systems. The OPM/T functionality includes triggering events, guarding conditions, temporal constraints, and timing exceptions. Triggering events can be explicitly represented in OPDs by adding information to the procedural link directed from the triggering object to the corresponding triggered process. As shown in Figure 1, the letter *e* (for "event") added within the circle of an enabling link (agent or instrument), or next to the arrowhead of an effect link, specifies the fact that the link is a triggering event. The event can also be explicitly specified in text (e.g., "*e: digit dialed*"), which is recorded along the corresponding procedural link.

Guarding conditions are specified in a manner similar to that of events. The letter "*c*" is recorded within the circle of an enabling link or next to the arrowhead of an effect link connecting the object state that serves as a condition guarding the execution of a process. Figure 1 shows an OPD featuring a triggering event and guarding condition of a *Dialing* process. It specifies the fact that on the event of a digit dialed by the Caller, the *Dialing* process is triggered under the condition that the Caller's Line is in the "*dial tone*" state. The *Dialing* process changes the state of the Caller's Line to "*trying to connect*".

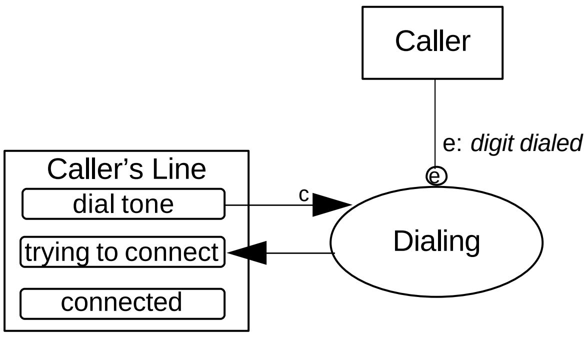

Figure 1: An OPD featuring the triggering event and guarding conditions of the *Dialing* process.

Temporal constraints are expressed by specifying an interval "(x,y)", where $0 \leq x \leq y$. x and y represent the lower and upper bounds of the constraint, respectively. There are three different kinds of temporal constraints:

- process duration constraint, in which the interval (x,y) is recorded inside the constrained process, as in Figure 2(a).
- state duration constraint, in which the interval (x,y) is recorded inside the constrained object state, as in Figure 2(b).
- reaction time constraint, in which the interval (x,y) is recorded above the procedural link connecting the triggering object to its triggered process, as in Figure 2(c):

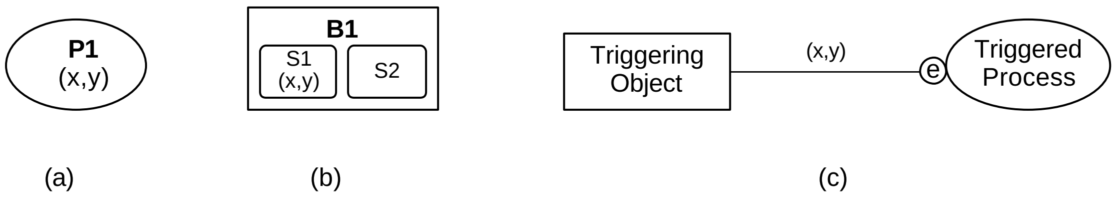

Figure 2: Expression of temporal constraints in OPDs. (a) process duration constraint; (b) state duration constraint; (c) reaction time constraint.

A *timing exception* is a violation of a temporal constraint. Certain exceptions are quite common and acceptable, while others may be rare, albeit hazardous. In case such an exception occurs, it is desirable to perform a suitable exception handling process. In order to represent exception triggers, the Exception Link, denoted as —|—, adopted from [4], is used to connect the violated constraint to the exception triggered process.

4. The Phone Call Process and Its OPM/T Specification

Figures 3 through 6 constitute the OPD set specifying a simplified process of a phone call. Figure 3 is a top-level OPD, which presents the dialing and connecting process. It shows all the objects which take part in this process: Caller, Callee, Switchboard, and Operator, which enable the *Dialing and Connecting* process without being affected by it, along with Caller's Line and Dialed Line, which are affected by that process. The paths marked by the letters 'a' and 'b' show the execution thread, i.e., the order of effect links which connect the *Dialing and Connecting* process to the two affected objects Caller's Line and Dialed Line. The structural links between Caller and Caller's Line, between Callee and Dialed Line, between Caller's Line and Switchboard, and between Dialed Line and Switchboard are also shown in this top-level OPD and in lower level OPDs with their name recorded along the links. Processes with a bold contour are zoomed in (scaled up) in other, lower level, OPDs.

Figure 4 is a blow-up of the *Dialing and Connecting* process. This process is initiated when the caller lifts up the receiver. The letter *e*, depicted at the process end of the agent link emanating from the object Caller to the process *Dial Tone Sending* (P1), symbolizes this triggering event. The event *e* is also explicitly expressed along the agent link as "*e: receiver lifted*". The interval (0,2), recorded next to this event name, specifies the constraint on the reaction time between the event and the triggered process. The meaning of this constraint is that the event *e* triggers the

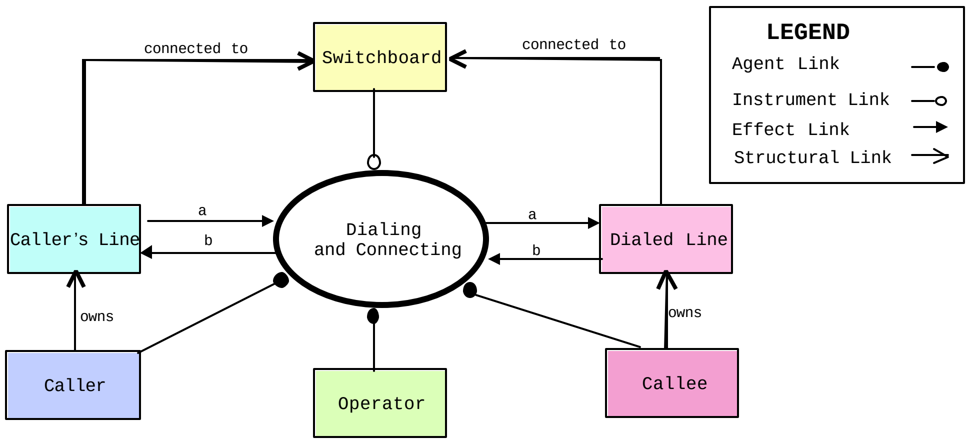

Figure 3: A top-level OPD of the *Dialing and*

process *Dial Tone Sending* (P1) within 2 seconds of its occurrence. When invoked, process P1 changes the Caller’s Line state from "*free*" to "*dial tone*", which is a sub-state within the "*dialing*" state, which, in turn, is a substate of the "*busy*" state of Caller’s Line. The "*dial* tone" state has a (0, 30) duration constraint attached to it, requiring that once the Caller’s Line has entered the "*dial tone*" state, it must remain there for no longer than 30 seconds. Otherwise, an exception, symbolized by the Exception Link, occurs. The Exception triggers the *Reorder Tone Sending* process (P6), which sends the reorder tone and changes the Caller’s Line state to "*faulty*". If the caller does dial a digit within 30 seconds after the Caller’s Line has entered the "*dial tone*" state, then the *Dialing* process (P2) is initiated. This process has a duration constraint restricting its total duration to a maximum of 120 seconds. If the process does not fulfill this restriction, an exception occurs, which activates the *Reorder Tone Sending* process (P6) discussed above. Note that the default logical relation among two or more triggering events of a single process is OR, meaning that any one of them alone can trigger the process. Thus, for example, the occurrence of any one of the triggering events of P6 is sufficient for its activation.

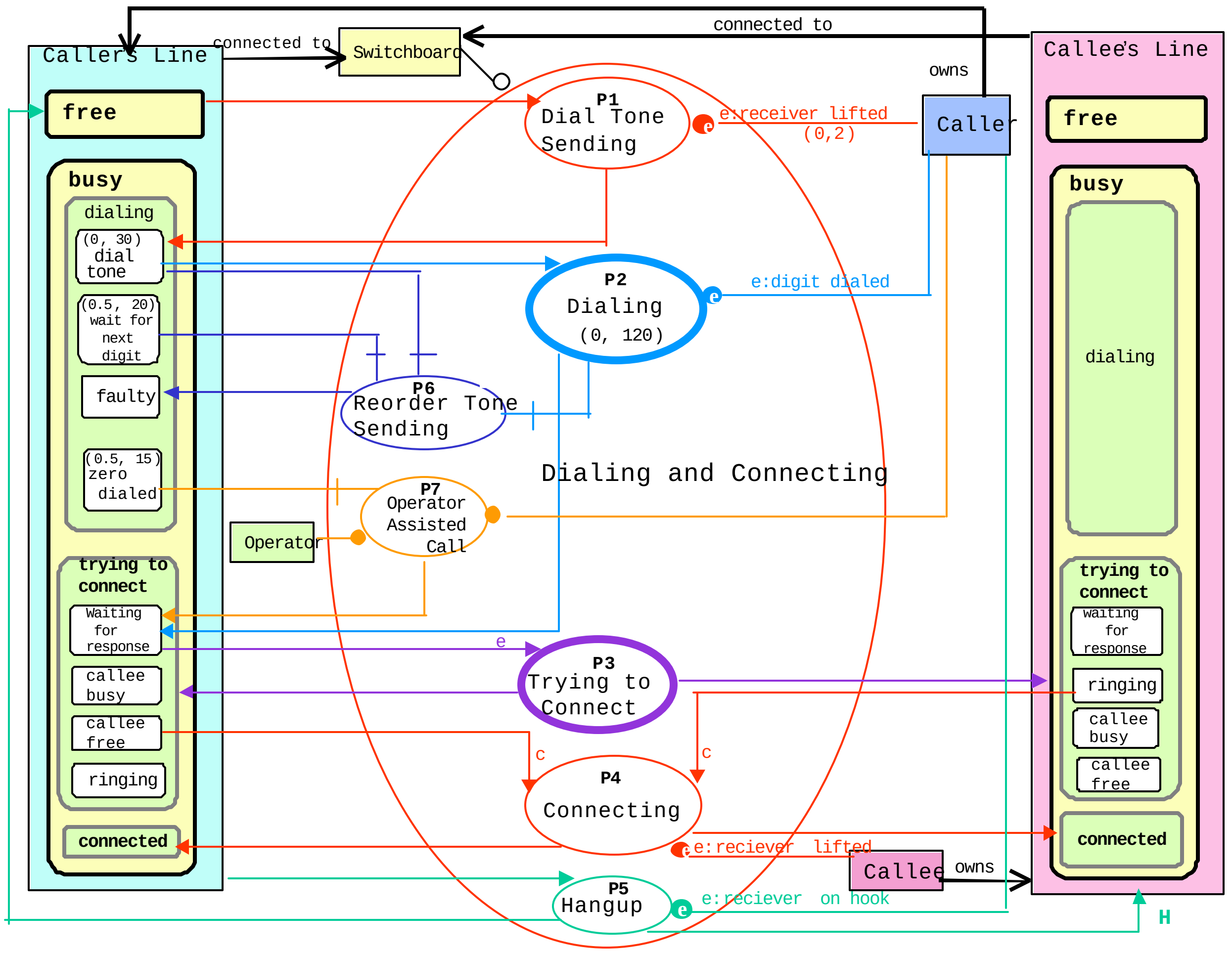

Figure 4: Blow-up of the *Dialing and Connecting* process

The *Dialing* process (P2) is blown-up in Figure 5. Dialing a digit when the Caller’s Line is in the "*dial tone*" state creates an event which triggers the *Dialing Initiation* process (P2.1). This process Initializes to 1 the object Counter, which counts the number of digits dialed, saves the dialed digit in the object Accumulated Dialed Number, and changes the Caller's Line state to the "*waiting for operator*" state, if the dialed digit was zero, or to the "*wait for next digit*" state, otherwise. For simplification, we assume that the dialed number must have 7 digits if it is local (does not start with a zero) or 9 digits otherwise. Therefore, the *Dialing Initiation* process sets the Stopping Condition value to v = 9 or to v = 7, depending on the first digit dialed. The next three processes are executed in a loop, which collects the rest of the digits which constitute the telephone number. The first process, *Comparison* (P2.2), is invoked by the termination of process

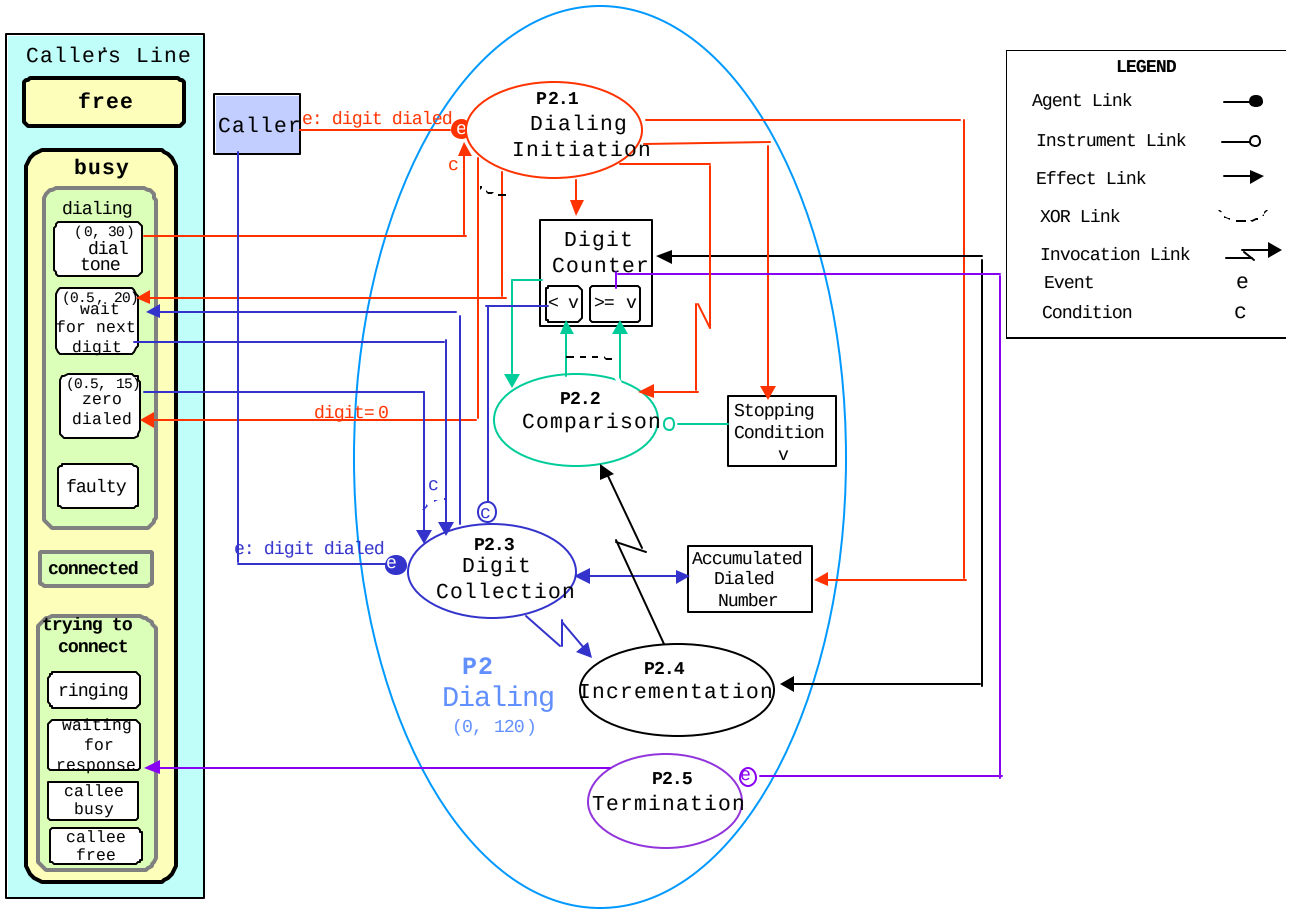

Figure 5: Blow-up of the *Dialing (P2)* process.

P2.1. This is marked by the invocation link emanating from P2.1 to P2.2. *Comparison* (P2.2) compares the Digit Counter's value, *v*, which holds the number of dialed digits, to the Stopping Condition value, which holds the expected number of digits (7 or 9) in the complete phone number that is being called. If the Digit Counter's value is equal to the Stopping Condition's value, *v*, then the *Termination* process (P2.5) sets the Caller's Line to the "waiting for response" state, marking the termination of the Dialing process (P2). If the Digit Counter's value is smaller than the Stopping Condition's value, then the event of dialing a digit, under the conditions that the digit was dialed either between 0.5 seconds and 20 seconds after entering the Caller's Line "*wait for next digit*" state, or between 0.5 seconds and 15 seconds after entering the Caller's Line "*wait for operator*" state, triggers the *Digit Collection* process (P2.3). This process adds the dialed digit to the Accumulated Dialed Number and sets the Caller's Line to the "*wait for next digit*" state. Note that setting an Object into a state initiates the state's duration measurement. Upon its termination, process P2.3 triggers, via the invocation link, the *Incrementation* process (P2.4). The *Incrementation* process (P2.4), increments the Digit Counter by 1. The Invocation Link now passes the control back to the *Comparison* process (P2.2), the output of which decides whether the dialing process should continue or stop. The duration constraint on the Caller's Line "*wait for next digit*" state, (0.5, 20), constrains the duration of time spent at that state. Dialing a digit too fast after the dialing of a previous digit (i.e., less than 0.5 seconds after the entrance to the "*wait for next digit*" state) generates an event that is ignored. Failure to dial the next digit within 20 seconds creates an exception, denoted by the Exception Link, shown in Figure 4 emanating from the "*wait for next digit*" state. This Exception triggers the *Reorder Tone Sending* process (P6). Similarly, the duration constraint of the Caller's Line "*wait for operator*" state, (0.5, 15), constrains the duration of time spent at that state.

Again, dialing a digit too fast after the dialing of a previous digit (less than 0.5 seconds after the entrance to the "*wait for operator*" state) generates an event that is ignored. Failure to dial the next digit within 15 seconds creates an exception, denoted by the Exception Link shown in Figure 4 emanating for the "*wait for operator*" state. This Exception triggers the *Operator Assisted Call* process (P7).

As can be seen in the OPD of Figure 4, the entrance of the Caller's Line to the "*waiting for response*" sub-state in the "*trying to connect*" state, marking the successful termination of the *Dialing* process (P2), is an event which triggers the *Trying to Connect* process (P3). As a simplification, it is assumed that every dialed number is legal. The *Trying to Connect* process (P3) is blown up in the OPD of Figure 6.

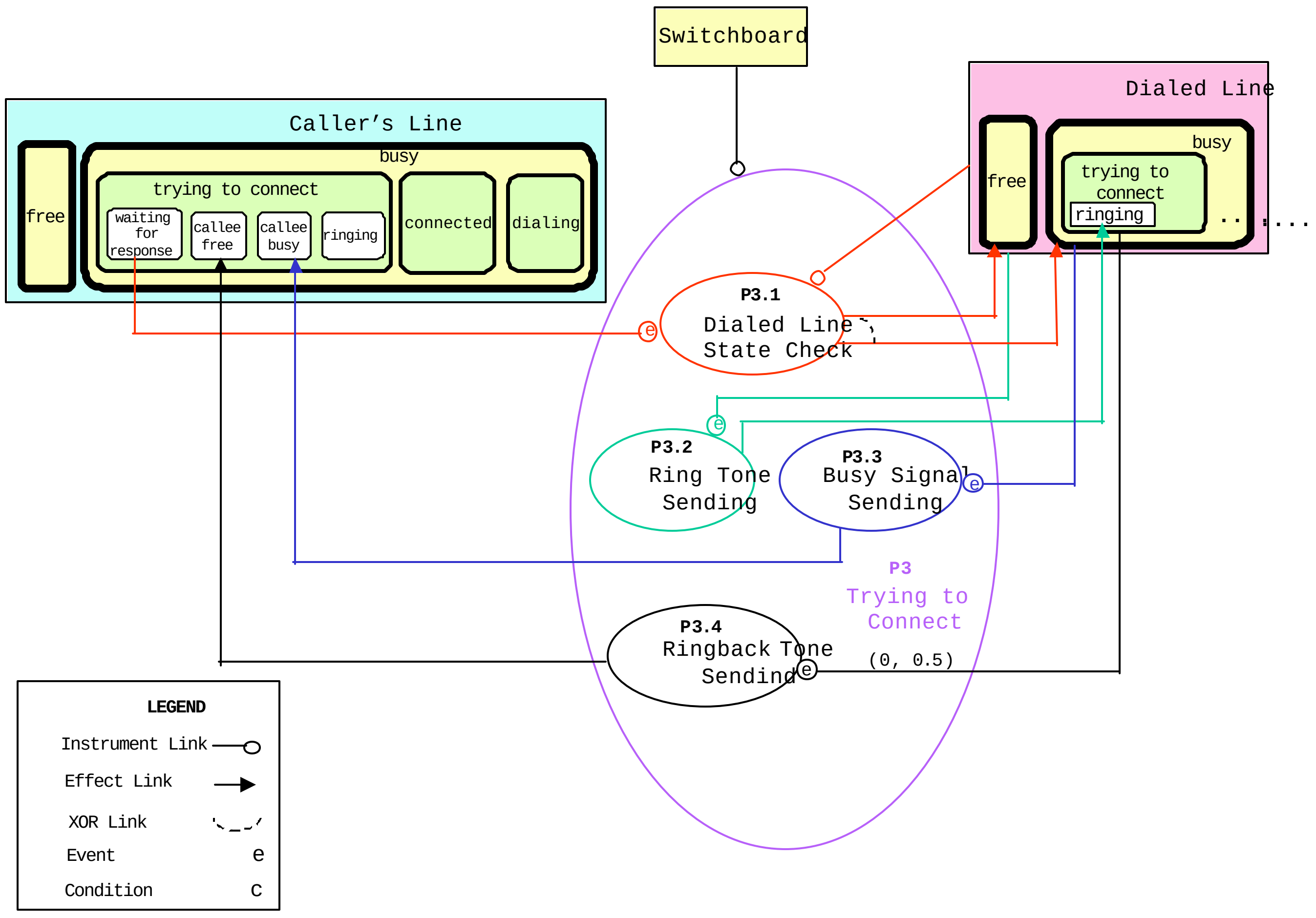

Figure 6: Blow-up of the *Trying to Connect (P3)* process

As shown, the first process occurring is *Dialed Line State Check* (P3.1) which checks whether the Dialed Line state is free or busy. This process is carried out only if the Caller's Line is in the "*waiting for response*" state, which results from the successful termination of the *Dialing* process (P2). If the Dialed Line is in the "*busy*" state, then the *Busy Signal Sending* process (P3.3) changes the Caller's Line into the "*callee busy*" state. If the state of the Dialed Line is "*free*", then the *Ring Tone Sending* process (P3.2) changes the Dialed Line into the "*ringing*" state. Note that since the Dialed Line "*busy*" and "*free*" states are mutually exclusive, only one of the processes P3.2 and P3.3 can be executed, so this is in fact an example of branching. Entering the "*ringing*" state generates an event which triggers the *Ringback Tone Sending* process (P3.4), which sets the Caller's Line into the "*callee free*" state within 0.5 seconds. This is specified by the reaction time constraint (0, 0.5), depicted above the instrument link emanating from the Dialed Line "*ringing*" state to the *Ringback Tone Sending* process (P3.4).

Turning back to Figure 4, when the Callee lifts up the receiver, then if the Caller's Line is in the "*callee free*" state and the Dialed Line is in the "*ringing*" state, then an event occurs which triggers the *Connecting* process (P4). This process sets the Caller's Line and the Dialed Line into the "*connected*" states. On the event of "*receiver on hook*", which happens when the Caller's Line is at any state, the *Hang-up* process (P5) is triggered. This process changes the Caller's Line to the "*free*" state, and sets the Dialed Line into its previous state, as denoted by the letter "H" (short for History) next to the effect link emanating from the *Hang-up* process towards the Dialed Line Object. This notation is borrowed from Harel's Statecharts [5].

Discussion

OPM/T is a real-time extension of OPM — a visual methodology which describes both the static-structural and the dynamic-procedural aspects of systems in a single model. OPM/T is designed to express triggering events, guarding conditions, timing constraints, timing exceptions, and flow of control constructs. The telephone call process example demonstrates how a real-time system can be explicitly specified, and clearly understood and communicated among analysts, designers, and implementors, in

OPM/T. OPM/T helps clarifying and communicating the structure and behavior of the entire system in the following ways:

1. Being as extension of OPM, the system's static-structural and functional-behavioral aspects are incorporated into a single model. Therefore, no model switching and mental transformations are required in order to understand the system as a whole. The only passages are made between scaling levels in order to see more or less details of some parts of the system. Furthermore, working within a single model, it is less likely to generate and encounter incompatibilities
2. Following the default scenario (normal sequence of processes and events) is easy, since it is reflected by the top-to-bottom order in which the processes in OPDs are depicted.
3. For each process, all relevant information is readily visible. This includes the triggering events, guarding conditions, timing constraints, all the affected or consumed objects which serve as the process inputs, the affected or generated output objects, and the enablers—objects that participate in the process without changing their state. State changes of the affected objects that result from the process execution are also shown explicitly.
4. The system's static-structural and functional-behavioral aspects are presented from a top-level view, and zoomed in (scaled up) to provide more and more details. Scaling makes it possible to depict only those objects that participate in the processes shown in an OPD. These objects are scaled up to any desired level of detail that exposes all the relevant parts and/or specializations and/or features (attributes and methods) of each object.
5. Processes in OPDs group together all objects which are transformed by the same event (which may be simple or compound). For example, the triggering event of the Connecting process, (P4) affects

both the Caller's Line and the Dialed Line. This grouping is also useful for expressing synchronization of the different reactions that trigger the process and/or are triggered by it.

OPM and OPM/T are especially suitable for specifying systems which contain many objects and/or processes, and in particular if they need to be specified at different detail levels which are taken care of by OPM's scaling option.

OPM/T supplies some very useful design patterns. One example of such a design pattern is triggering of an internal system process by an external event — in our case, the *Dial Tone Sending* process is triggered by the event of the receiver being lifted, and the *Dialing* process is triggered by the event of dialing a digit. A distinction between different types of triggering events (external events, events that mark the change in an object's state, events that mark the termination of a process, and exception events) makes it easier for analysts and/or designers of systems to specify their intended meaning.

Another example of a very useful design pattern is a for-loop iteration, shown in the OPD of Figure 5, which is a blow-up of the Dialing process. This process behaves like a loop which iterates 7 or 9 times depending on the number dialed being local or long-distance.

Refinement of OPM/T, including identification, characterization, and classification of real-time design patterns is currently under way. The implementation of real-time extensions is being incorporated into OPCAT—Object Process CAse Tool, such that version 2.0 is designed to feature significant real-time functionality.

References

1. Dori, D. Object-Process Analysis: Maintaining the Balance Between System Structure and Behavior. *Journal of Logic and Computation* 5, 2, (April 1995), 227-249.

2. Dori, D. and Goodman M. On Bridging the Analysis-Design and Structure-Behavior Grand Canyons with Object Paradigms. *Report on Object Analysis and Design* 2,5, (January-February 1996), 25-35.

3. Drusinsky, D. and Harel, D. Using Statecharts for Hardware Description and Synthesis. *IEEE Transactions on Computer-Aided Design* 8, 7 (July 1989), 798-807.

4. Embley, D. W., Jackson, R.B. and Woodfield, S. N. Object-Oriented Systems Analysis: Is It or Isn't It?, *IEEE Software* 12,4 (July 1995), 19-33.

5. Harel, D., Statecharts: a Visual Formalism for Complex Systems, *Sci. Comput. Program,* Vol. 8 (1987), 231-274.

6. Harel, D., Lachover, H., Naamad, A., Pnueli, A., Politi, M., Sherman, R., Shtull-Trauring, A. and Trakhtenbrot, M., STATEMATE: A Working Environment for the Development of Complex Systems. *IEEE Transactions on Software Engineering* 16, 4 (April 1990), 403-414.

7. Jackson, M., *System Development*. Prentice-Hall International, 1983.

8. Jahanian, F. and Mok. A.K. Modechart: A specification Language for Real-Time Systems. *IEEE Transactions on Software Engineering* 12, 12 (December 1994), 933-947.

9. Ostroff, J.S. Formal Methods for the Specification and Design of Real-Time Safety Critical Systems. *The Journal of Systems and Software* 18, 1(April 1992), 33-60.

10. Peterson, J.L. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall Englewood Cliffs, NJ, 1981.

11. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorenson, W. *Object-Oriented Modeling and Design*, Prentice-Hall, Englewood Cliffs, NJ, 1991.

12. Sanden, B. Entity-Life Modeling and Structured Analysis in Real-Time Software Design A Comparison, *Communications of the ACM* 32, 12 (December 1989), 1458-1466.