Graphical Scenarios for Specifying Temporal Properties: an Automated Approach†

M. Autili (marco.autili@di.univaq.it), P. Inverardi (inverard@di.univaq.it) and P. Pelliccione (pellicci@di.univaq.it)
Dipartimento di Informatica, University of L'Aquila, I-67010 L'Aquila, Italy

Abstract. Temporal logics are commonly used for reasoning about concurrent systems. Model checkers and other finite-state verification techniques allow for automated checking of system model compliance to given temporal properties. These properties are typically specified as linear-time formulae in temporal logics. Unfortunately, the level of inherent sophistication required by these formalisms too often represents an impediment to move these techniques from "research theory" to "industry practice". The objective of this work is to facilitate the non trivial and error prone task of specifying, correctly and without expertise in temporal logic, temporal properties.

In order to understand the basis of a simple but expressive formalism for specifying temporal properties we critically analyze commonly used in practice visual notations. Then we present a scenario-based visual language called Property Sequence Chart (PSC) that, in our opinion, fixes the highlighted lacks of these notations by extending a subset of UML 2.0 *Interaction Sequence Diagrams*. We also provide PSC with both denotational and operational semantics. The operational semantics is obtained via translation into Büchi automata and the translation algorithm is implemented as a plugin of our Charmy tool. Expressiveness of PSC has been validated with respect to well known *property specification patterns*.

Keywords: Scenario based notation, System requirements specification, Temporal properties specification

1. Introduction

Temporal logics are commonly used for reasoning about concurrent systems. Model checkers and other finite-state verification techniques allow for automated checking of system model compliance to given temporal properties. These properties are typically specified as linear-time formulae in suitable temporal logics. However, it is a difficult task to accurately and correctly express properties in these logics. For instance, the high level of inherent complexity of Linear-time Temporal Logic formulae (LTL) (Pnueli, 1977; Manna and Pnueli, 1991) may cause users to specify properties incorrectly.

† Preliminary results appeared in (Autili et al., 2006).

Properties, that are simply captured within the context of interest and easily expressed in natural language, could be very hardly specified in LTL.

In other words, there is a substantial gap between natural language and the LTL language. Holzmann in (Holzmann, 2002) states, for example, that one of the "most underestimated problems in applications of automated tools to software verification" is "the problem of accurately capturing the correctness requirements that have to be verified". In the same paper Holzmann shows that writing LTL formulae is an error prone task.

Although in this paper we focus on formulae expressed in LTL notation, other similar formalisms (such as CTL, $ACTL$) suffer the same problems. In (Smith et al., 2002) the authors notice that these problems are not only related to the chosen notation, in fact "no matter what notation is used, however, there are often subtle, but important, details that need to be considered". For this reason, the introduction of temporal logic-based techniques in an industrial software life-cycle requires specific skills and good tool support. As a matter of fact, industries are not willing to use the above mentioned techniques and this slows down the transition of software verification tools from "research theory" to "industry practice". In order to mitigate this problem, in (Smith et al., 2002) the authors propose PROPEL that, by building upon property patterns previously identified, introduces pattern templates which are represented using both disciplined natural language and finite state automata.

Many other works in the last years propose solutions to overcome this problem. While one proposal is to construct a library of predefined LTL formulae from which a user can choose (Dwyer et al., 1999), other works propose the specification of temporal properties through graphical formalisms (Smith et al., 2001), (Dillon et al., 1994), (Zanolin et al., 2003), (Alfonso et al., 2004; Braberman et al., 2005), and (Kugler et al., 2005). Any of these solutions have advantages and disadvantages.

Based on these considerations, we believe that an accurate analysis is necessary in order to understand what is required in a formalism to express a *"useful set"* of temporal properties while keeping in mind that easy use and simplicity are mandatory requirements to make a formalism adopted by industries. Thus, in our opinion, the "perfect" language should find the "right" balance between *expressive power* and *simplicity of use.*

In this paper we present a scenario-based visual language called PSC that represents a first step toward the "perfect" one. PSC retains many features of other formalisms and builds on them trying to improve the "performance" of the formalism along the two dimensions namely expressive power and simplicity of use.

Within the PSC language, a property is seen as a relation on a set of exchanged system messages, with zero or more constraints. Our language may be used to describe both positive scenarios (i.e., the "desired" ones) and negative scenarios (i.e., the "unwanted" ones) for specifying interactions among the components of a system. For positive scenarios, we can specify both mandatory and provisional behaviors. In other words, it is possible to specify that the run of the system *must* or *may* continue to complete the described interaction. In order to unambiguously determine which execution sequences are allowed or not, we formally define a trace-based denotational semantics of PSC by associating to each PSC the set of all the *invalid traces*.

It is well known that an LTL formula can be translated into a Büchi automaton (Buchi, 1960) that can be used by model checkers (Holzmann, 2003) or component assemblers (M.Tivoli and M.Autili, 2004). Although this representation looks more intuitive, it can be very difficult to correctly and directly represent a property as a Büchi automaton. Therefore, in order to overcome this problem and to provide PSC also with an operational semantics, we propose an algorithm, called Psc2Ba, to translate PSC specifications into Büchi automata. The algorithm has been implemented as a plugin of our tool Charmy (Charmy Project, 2004) which is a framework (based on the model checker Spin (Holzmann, 2003)) for software architecture design and verification with respect to temporal properties.

We measured the expressiveness of our language with respect to the set of *property specification patterns* proposed in (Dwyer et al., 1999) that captures recurring solutions to design and coding problems.

The paper is organized as follows: Section 2 sets the context with respect to properties specification. Section 3 analyzes MSC and UML 2.0 Interaction Diagrams when used for expressing temporal properties. Section 4 gives the state of the art in languages for properties specification. Section 5 presents the PSC's graphical and textual notations. Section 6 gives the PSC operational semantics by means of the Psc2Ba translation algorithm and Section 7 gives the PSC denotational semantics. The equivalence among these two semantics is presented in Section 7.6. Section 8 discusses the PSC expressive power and Section 9 presents the considered case study. The PSC tool is introduced in Section 10 and finally, conclusion and future work are discussed in Section 11.

2. Setting the context: our point of view in properties specification

The main goal of this work is to propose a scenario-based visual language for specifying temporal properties. Even though PSC is yet another properties specification language, it aims at proposing a language that, building on results and experience of already existent and valuable works in the Literature, aims at balancing *expressive power* and *simplicity* of use.

To this purpose, we analyzed existing solutions in order to figure out graphical notations and associated semantics that might be appropriate to facilitate the introduction of our language in an industrial software life-cycle while retaining enough power to express the targeted useful set of temporal properties, i.e., the one identified by the properties specification patterns (Dwyer et al., 1999) (in the middle of Figure 1). The properties that we want to specify express temporal relations between messages exchanged among parts of the system. For this reason the starting point of our analysis has been the tools that are commonly used in industries for specifying component-based systems that interact by message passing. Visual formalisms for scenario-based modeling that are commonly and extensively used within industrial software development practice are Message Sequence Charts (MSCs) (ITU-T Recommendation Z.120., 1999) and UML 2.0 Sequence Diagrams (Object Management Group (OMG), 2004) (left-hand side of Figure 1).

We decided to remain close to the graphical notation of these two languages to satisfy the requirement of *simplicity* of use. The fundamental graphical characteristics that we consider appealing are:

– bi-dimensional time-space representation;

– time running from top to bottom giving the idea of a system execution;

– system components placed along the space-axis and message exchanging clearly represented by sequence of arrows from the sender component to the receiver component (this outlines the high-level architecture of the system and which system components are involved in providing the intended system behavior);

– implicit component interfaces description (by illustrating which messages are being sent and received by each component).

We analyzed UML sequence diagrams and MSC (see Section 3) also from the point of view of the *expressiveness* in order to identify the aspects where they lack in expressive power and those ones where they are "too powerful" (and often tricky to use) for our purposes. In other words, we "*filter*" out of these languages the features that we consider useful and, in Section 5, we extend them by adding the ones that we have identified as necessary to deal with the class of temporal property specifications we are targeting.

Concerning the features that had to be added, if other languages in the Literature (different from UML and MSC) already had one of these features we inherit it from them. For instance, our notion of *constraints* (that we will introduce in Section 5) has been inspired by Timeedit (Smith et al., 2001) and we have directly inherited part of its graphical notation. We analyze these other languages in Section 4.

In Figure 1 we show formalisms that we have considered as the ingredients of our approach. In the left hand side box we list the industrially-valuable ones while in the right side box the more academic-valuable ones. PSC has been influenced by both sides and its expressivity has been validated against the properties specification patterns, as shown in the middle of the figure.

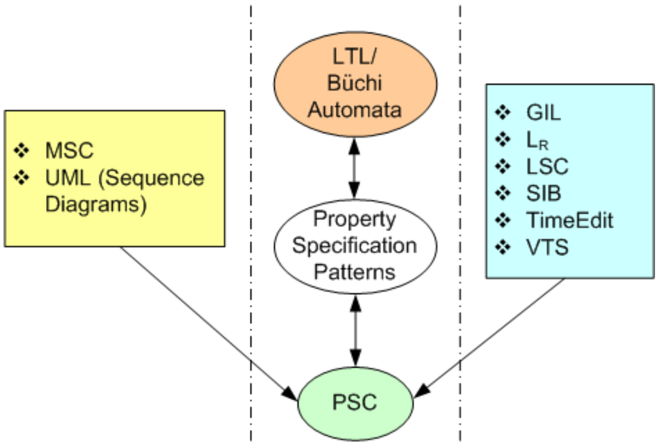

Figure 1. The approach

It is important to recall that these languages are not only syntax but also semantics and users of a language attach meaning to the diagrams they produce. Thus, one way to propose a new language is to extend an existing one retaining its original semantics. In the case of UML and MSC it is not possible to completely follow this way since these languages have neither a native formal semantics nor a widely accepted one despite the several attempts to define formal semantics for these two languages (see for instance (Störrle, 2003) for UML sequence diagrams and (Uchitel et al., 2004) for MSCs).

However UML and MSC have an informal semantics that people associate to them according to the specification native documents (Object Management Group (OMG), 2004) and (Object Management Group (OMG), 2004; ITU-T Recommendation Z.120., 1999), respectively. Thus, we inherit graphical elements from UML sequence diagrams and MSC only when these elements have a native informal semantics that is consistent with the formal semantics we define, substituting those graphical elements having an ambiguous semantics. This is for instance the case of message types for which we used Timeedit graphical elements instead of UML *assert*, *optional*, and *neg* frames (see Section 3 for further details).

3. Inspecting MSC and UML 2.0 Sequence Diagrams

3.1. MSC

The Telecommunication Standardization Sector of International Telecommunication Union (ITU) proposes the Message Sequence Charts (MSC) standardization through the Recommendation Z.120. The objective of the MSC language is to describe the interaction between a set of independent message-passing instances. The sending and the consumption of messages are two asynchronous events. MSC is a scenario-based language that describes the order in which communication events flow takes place. Instances are graphically represented as named rectangles connected to descending vertical lines, called *instance axis*. The time runs from top to bottom along each instance axis and an MSC imposes a partial ordering on the "attached" events. Except for *coregions* (that allow the specification of unordered messages) and *inline expressions* (parallel composition, iteration, exception, and optional regions), a total time ordering of events is assumed along each instance axis and a message must first be sent before it is consumed.

Many works in the literature propose a formal semantics useful to determine unambiguously which execution traces are allowed by an MSC. However, for the purpose of using MSC to describe temporal properties, the MSC language lacks in expressive power as discussed and itemized in the following:

(i) Message types: as pointed out by (Damm and Harel, 2001; Harel and Marelly, 2002), it is not clear if the system has to carry out all the indicated events in a scenario or it can stop at some point without continuing. In other words it is not possible to clearly distinguish between mandatory messages and provisional ones. Furthermore, since MSCs can only represent desired exchanging of messages (i.e., positive scenarios), it is only possible to define a set of *liveness properties* to stipulate that *"good things"* do (eventually) happen during the execution of a system. On the contrary, often, it is necessary to express forbidden scenarios (i.e., negative ones) to specify *safety properties* which stipulate that *"bad things"* do not happen during the execution of a system;

(ii) Strict ordering: it is not possible to state that a message can be only followed by a specific message;

(iii) Constraints: it is not possible to impose restrictions on additional messages that can be potentially exchanged between a given pair of messages;

(iv) Alternative: it is useful to be able to specify two or more different sequences of messages that can be unconditionally chosen. The MSC language deals with alternatives by means of *high-level MSCs* (*hMSC*);

(v) Parallel: it is useful to specify two or more different sequences of messages that represent parallel executions of the system. Even though without a clear semantics, MSCs can specify parallel execution by using hMSC;

(vi) Loop: sometimes a sequence of messages has to be repeated several times. MSC deals with repetitions by also using hMSC but it is not possible to specify a lower and upper bound to the number of repetitions.

3.2. UML 2.0

Many of the previously identified features have been added in the UML 2.0 Interaction Sequence Diagrams. UML 2.0 is the major revision of all the previous versions of UML. In particular, UML Sequence Diagrams have been thoroughly revisited and revised leading to UML 2.0 Interaction Sequence Diagrams. MSC and UML 2.0 Interaction Sequence Diagrams are so similar that in (Haugen, 2004) the authors propose that either MSC should be retired or should become a profile of UML 2.0. By referring to the above itemized missing aspects of MSC, UML 2.0 adds some features that are useful for our purposes:

(i) Message types: *assert* is used to specify mandatory messages; *neg* is used to describe forbidden scenarios. Both of them are defined as operators (i.e., InteractionOperators), they support nesting and they can be applied to a set of messages. These operators are graphically represented as a frame box with a compartment displaying the name. If an operator has two or more operands, they are divided by dashed lines. This graphical notation can be very expressive when dealing with more than one message and with nesting, but UML 2.0 has yet again not provided a formal semantics. Specifically, it is unclear what happens if there are several neg/assert operators nested or intermixed with other operators. Thus, as noticed in (Störrle, 2003), neg and assert should be modeled as attributes of a single message rather than operators. In accordance with this idea, as we will see later on, the graphical notation we use for neg and assert is different from the one used by UML 2.0. This has been done in order to be closer to the notion of attribute and to make the notation more clear and intuitive;

(ii) Strict ordering: while a partial ordering is assumed by default, designers can also define a strict ordering between messages by using the *strict* operator;

(iii) Constraints: UML 2.0 has no direct and simple way to deal with constraints;

(iv) Alternative: the solution of UML 2.0 for alternative choices in the execution of the system is the *alternative choice* operator;

(v) Parallel: UML 2.0 has the *parallel* operator for expressing parallel sequences of messages;

(vi) Loop: the UML 2.0 *loop* operator allows sequences of messages to be executed several times.

4. State of the art in languages for Properties specification

Many works in Literature propose languages for specifying temporal properties.

Graphical Interval Logic (GIL) (Dillon et al., 1994) is sufficiently expressive but its formulae become potentially difficult to understand. This difficulty comes from the fact that its graphical notation is very close to temporal logic syntax. On the contrary, Timeedit (Smith et al., 2001) (also called *TimeLine Editor*) has a more intuitive notation but it was specifically developed to capture long running on complex chains of dependent events (the specification patterns people (Dwyer et al., 1999) call them "chain patterns"). These chains are very hard to write in LTL thus the Timeedit people propose an easy way for writing them as timelines. Even though we recognize chains to be very useful when specifying properties and even though Timeedit is a powerful means for writing them, its expressive power could have limitations when used for specifying more general properties. For example, Timeedit does not allow partial ordering to be specified. PSC, thanks also to the chain constraints, is able to describe all the different kind of chain patterns. Figure 16, by means of one chain pattern, highlights the ability of PSC. PSC follows some ideas, terminology, and graphical elements of Timeedit.

Visual Timed event Scenarios (VTS) (Braberman et al., 2005; Alfonso et al., 2004) is a visual language for expressing event-based requirements. In this language system events are considered any observable and interesting changes (from the point of view of the verification) during the system execution. Thus events are considered to be more abstract and general than message exchanging (e.g., an event can be a key press or an internal state change). Consequently, differently from us, they do not a priori consider component-based systems (where interaction is modeled by message passing) and it makes no sense to have specific graphical notations for representing the involved components, lifeline, exchanged messages, etc. Basing on these considerations authors propose a new language with novel graphical notation (composed of few graphical elements) to express complex properties for real-time systems. The declarative semantics of VTS is simple and compact, and allows the language to be easily engineered and used for model checking of real-time properties.

Many works in the literature propose algorithms for temporal properties generation (Klose and Wittke, 2001; Kugler et al., 2005; Zanolin et al., 2003) starting from Live Sequence Charts (LSC) (Damm and Harel, 2001; Harel and Marelly, 2002). LSCs are an extension of MSCs with the aim of dealing with liveness. This is done by introducing the difference between mandatory and provisional messages that have the same meaning of our Regular and Required messages respectively. LSC is a project started before UML 2.0 and it played an important role in suggesting features of UML 2.0 Interaction Diagrams. In fact, many LSC features are today parts of UML 2.0 Interaction Diagrams. For this reason we have developed the translation algorithm defining PSC as a conservative extension of this language. The main advantage of PSC with respect to LSC is its ability to specify $intraMsg$ and especially chain constraints. In fact a constraint allows the specification of what can be performed before and after a message exchange. This is very useful to describe causes, effects and precedence and response relations. This motivation is amplified in the case of precedence or when a response is a chain (PSC Project, 2005; Dwyer et al., 1999). Note that a chain is different from a sequence of messages because several repetitions of the same message before the next element of the chain are not allowed. Furthermore a chain is a sequence of messages to be considered in its entirety. For instance we would express that there is a system error if a chain does not precede a message m. This means that if m happens anyway in between messages of the chain, the system has to reach a state of error.

One of the most interesting features of LSC is the ability to establish when an LSC starts i.e. when the system starts or when the pre-chart is detected one or more times. In PSC the same can be specified by using regular messages.

The translation algorithm proposed by Ghezzi et al. (Zanolin et al., 2003) gets as input a LSC and produces an automaton and a LTL formula, both necessary to express the correct temporal properties. The automaton and the LTL formula are translated into Promela code that, introduced into the proposed process, allows for the verification of systems. The paper (Kugler et al., 2005) proposes a translation into CTL* while the work (Klose and Wittke, 2001) offers a solution for timed Büchi automata generation.

Other approaches (C. André and M-A. Peraldi-Frati and J-P. Rigault, 2001; Lee and Sokolsky, 1997) define graphical languages that appear to be not easily comprehensible and not easily integrable into industrial software development processes.

5. PSC: Property Sequence Chart

PSC is a scenario-based language for expressing temporal properties to be checked against component-based system models specifying the interaction among component instances that can be concurrently executed. Component instances[1] communicate by message passing and sending/receiving a message is considered to be an atomic event. We also assume components to communicate by synchronous communication channels and hence, send and receive events of the same message are considered to occur simultaneously. Thus, we can restrict to send-events only and, hereafter, we uniquely associate a message (and hence its label) to its send-event.

It is worthwhile noticing that, PSC can also be used to specify properties for asynchronous component-based systems since, as pointed out in (Uchitel et al., 2004), "*a bounded asynchronous communication can be modeled by introducing buffer abstraction to decouple message*

[1] In the remaining of the paper the terms *component* and *component instance* are used interchangeably.

passing". Of course, this could lead to ugly and cluttered specification of both the system models and the properties themselves.

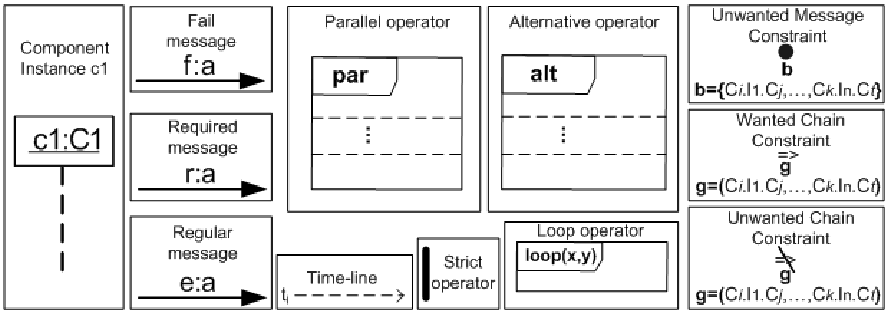

Figure 2. PSC graphical elements

In Figure 2 we show all the PSC graphical elements and in Figure 3 we show an example of a property expressed by PSC over an ATM system. Each involved component instance is graphically represented as a named rectangle with a vertical dashed-line, called *lifeline*, which extends downward (see also *UserInterface*, *ATM*, and *BankDB* in Figure 3). The time runs from top to bottom. A labeled horizontal arrow represents a message that we call *arrowMSG* (e.g., in figure the message labeled *login* is an arrowMSGs). The output of a message from one instance is represented by the arrow source on the instance lifeline and the input of the message is represented by the arrow target.

Since we are interested in expressing properties for specifying execution sequences of a system in terms of message passing, we define an ordering relation among the system messages by abstracting with respect to the absolute time. This abstraction is acceptable since, at the moment, we are not interested in real-time systems for which modeling time becomes relevant (Alfonso et al., 2004).

Within a PSC scenario we identify a set of horizontal dotted lines $t_0, \ldots, t_{n+1}$ (see Figures 2 and 3). These lines are called *structural time-lines* (or simply *time-lines*) and identify a point in time. For each *time-line* only one *arrowMSG* is allowed, except for *time-line* t_0 and t_{n+1} that cannot have associated messages. Time-lines are totally ordered from top to bottom and the pair time-line and its associated *arrowMSG* uniquely identifies the sender and the receiver (and hence the corresponding send-event). Note that, the use of *time-lines* is only a means for structuring the lifelines. In fact, time-lines are totally ordered but this ordering is only (graphical-)structural. That is a designer can also specify a partial ordering of messages by using **constraints** and **operators** that we shall define later.

Let m be a message on a time-line t_i: in absence of operators, we refer to the temporal space from t_{i-1} to t_i as the *past* of m, and the temporal space from t_i to t_{i+1} as the *future* of m; in case of operators, *past* (*future*) of m is the temporal space between t_i and the "previous" ("next") timeline according to the partial ordering deriving from the application of the operator (See Section 5.1).

In order to have well defined *past* and *future* of the message at the time-lines t_1 and t_n, respectively, we have introduced the time-lines t_0 and t_{n+1} that cannot have associated messages. The messages possibly exchanged in the past and future are referred as intra-messages (*intraMSGs*) of m. They are messages that can be exchanged between *arrowMSGs* and their introduction allows one to specify restrictions on "additional" messages that are not explicitly specified as *arrowMSGs*. In fact, as it will be clear later, constraints are associated to a single *arrowMSG* m and stipulate restrictions on *intraMSGs* of m. It is worth noticing that we distinguish between *arrowMSGs* and *intraMSGs* since by *arrowMSGs* we want to emphasize the main structure of the property under specification and by *intraMSGs* we can refine the main structure. PSC constraints and hence the distinction between *arrowMSGs* and *intraMSGs* have been inspired by Timeedit (Smith et al., 2001) and have been extended in order to deal with the properties set we are targeting (Dwyer et al., 1999).

In order to identify the sender and the receiver components, the *intraMSG* labels are prefixed by the name of the sender component and postfixed by the name of the receiver component. For example, the label $C_i.l.C_j$ denotes the message labeled by l sent from the component C_i to the component C_j[2].

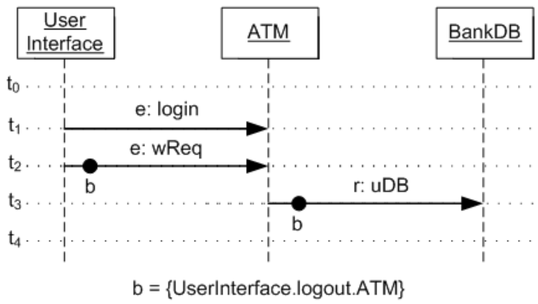

Figure 3. PSC example

To better understand the usefulness of distinguishing between *arrowMSGs* and *intraMSGs*, we discuss the example in Figure 3. The

[2] In the remaining part of the paper, when there is no ambiguity, the terms *message* and *message label* are used interchangeably.

property states that "*if the user has logged in* (login) *and if the withdraw request has been satisfied* (wReq)*, the bank DB must be updated* (uDB)*; the withdraw request is allowed only if the user has not logged out* (logout)".

The filled circles are two constraints that (by the identifier b) reference the message label *UserInterface.logout.ATM*. The constraint associated to *wReq* states that the pair of messages *login* and *wReq* is a valid precondition for the *uDB* request iff the intra-message *logout* is not exchanged after *login* and before *wReq*. The same holds for the other constraint associated to *uDB*. It is useful to note that, while the messages *login*, *wReq* and *uDB* create the main structure of the property, the constraint over the unwanted message *logout* is used to refine the main structure by imposing restrictions on what can happen over the time intervals between *login* and *wReq*, and between *wReq* and *uDB*. To deal with optional behaviors ("i*f the user has logged in ...*") and to deal with mandatory behaviors ("*... the bank DB must be updated ...*") PSC makes use of different types of messages that we are going to introduce.

By referring to the comparison between UML 2.0 and MSC in Section 3, in the following we detail how PSC deals with the aspects **(i)-(vi)** itemized in Sections 3.1 and 3.2 for MSC and UML 2.0, respectively.

(i) Message types: PSC distinguishes among three different types of *arrowMSGs* (see Figure 2):

- *Regular messages:* the labels of such messages are prefixed by "**e:**". They denote messages that constitute the precondition for a desired (or an undesired) behavior. It is not mandatory for the system to exchange a Regular message (or a set of sequential Regular messages), however, if it (or they all) happens the precondition for the continuations has been verified. This kind of messages can be mapped into UML 2.0 and MSC provisional messages (i.e., non mandatory messages graphically represented by simple arrows);
- *Required messages:* are identified by the "**r:**" label prefix. It is mandatory for the system to exchange this type of messages provided that their (possible) precondition is met. By means of these messages we can specify *liveness* properties. Required messages have the same meaning of UML 2.0 assert messages that are used to identify the only valid continuations. All the other continuations result in an invalid trace. No similar kinds of messages exist in the MSC specification;

- *Fail messages:* their label is prefixed by "**f:**". They identify messages that should never be exchanged. Fail messages are used to express undesired behaviors and hence *safety* properties. UML 2.0 neg operator expresses the same notion. In fact, the operator neg is used to represent unwanted traces.

(ii) Strict ordering: in order to explicitly choose a *strict ordering* between a pair of messages, we define the *strict* operator. A *loose ordering* is assumed otherwise. Within a lifeline, between a pair of messages m and m' on time-lines t_i and t_{i+1} respectively, the *strict* operator specifies that no other messages can be exchanged. Graphically, the strict operator is a thick line that links the pair of messages (see Figure 4, between the messages m and m').

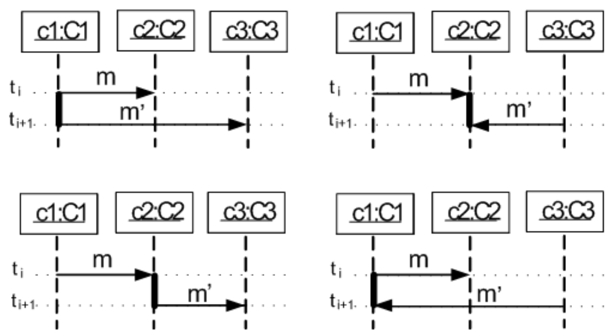

Figure 4. Strict Operator

For this operator UML 2.0 uses the same graphical notation (i.e., a named frame box) as the one used for the *neg* and *assert* operators described in Section 3. Differently from us, in UML 2.0 it is also permitted to specify strict ordering within more than one lifeline. We think that the strict operator is well defined as a relation between two contiguous messages within a single lifeline (see Figure 4). In fact, by referring to Figure 5.a, it does not make sense to state that the message m between $C1$ and $C2$ must be strictly followed by the message m' between $C3$ and $C4$. On the contrary, as it showed in Figure 5.b a loose ordering of two messages m and m' exchanged between independent pairs of components is allowed. It is worth to remark that PSC is a scenario based formalism for specifying temporal properties to be checked against a system model. That is, a property specified as in Figure 5.b holds iff in the system model there exists at least one message (after m and before m') "synchronizing" the pairs of components. The insertion of a strict operator as in Figure 5.a prohibits any synchronization message between the component pairs and hence the property might make no sense.

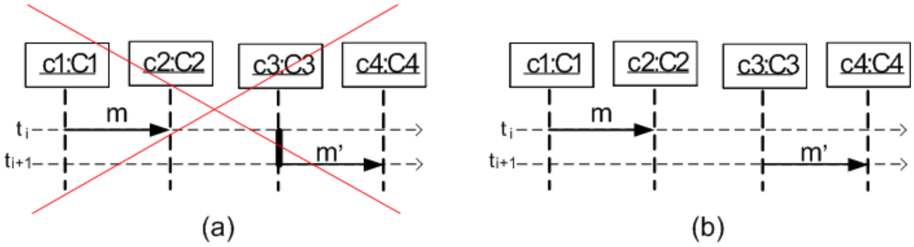

Figure 5. (a) illegal strict ordering; (b) legal loose ordering

(iii) Constraints: *constraints* of an *arrowMSG* m impose "restrictions" on *intraMSGs* of m. Restrictions specify either a chain of *intraMSGs* (*chain constraints*) or a set of *intraMSGs* that the system must not exchange (*unwanted messages constraints*). Informally, an *unwanted messages constraint* is satisfied iff all the set of *intraMSGs* specified as unwanted messages are not exchanged.

As noticed in (Dwyer et al., 1999) chains are very useful for describing a relationship between a single *arrowMSG* m and a sequence of *intraMSGs* $m_1, \ldots, m_n$. Informally, a chain predicates on a sequence of dependent *intraMSGs* and it is satisfied if the messages are exchanged following the loose ordering imposed by the chain specification. For instance it is possible to specify that the message m must be followed or must be preceded by the sequence of messages $m_1, \ldots, m_n$. It is also possible to specify "unwanted" chains to require that the messages in the chain specification are exchanged following any ordering different from the one imposed by the corresponding wanted chain. Unwanted chains are useful when dealing with whole sequences of undesired messages.

As will be clear later, both wanted and unwanted chains are different from sequences of required, regular, or fail messages since chains consider the sequence of messages as a whole. For instance, in the case of an unwanted chain we have an error only if the chain is completely exchanged. It would be impossible to express the same property by using a sequence of fail messages since we would have an error the first time one of the fail messages is exchanged. On the other hand, also sequence of required or regular messages would be inappropriate.

We distinguish between two types of constraints: *past constraints* and *future constraints*. They can be both chain constraints and unwanted messages constraints. As it is intuitive, *future constraints* cannot be associated to a fail message since the system has no future after such a message has been exchanged. Furthermore, as direct consequence, in the case of *strict ordering* between a pair of messages m_i and m_j (m_j follows m_i), m_i can only have *past constraints* and m_j can only have *future constraints*.

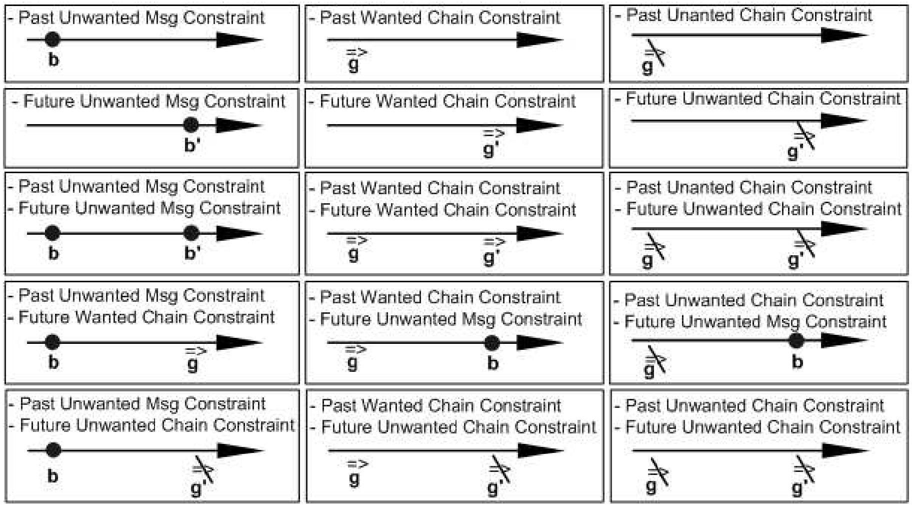

Figure 6. Constraints Combinations

Unwanted messages constraints are graphically represented as filled circles (see Figures 2 and 3). Wanted and unwanted chain constraints are graphically represented as arrows and crossed arrows, respectively (see Figure 2). Unwanted message constraints are specified as sets of *intraMSG* labels, i.e., $\{C_1.l_1.C'_1, \ldots, C_m.l_m.C'_m\}$, and chain constraints are specified as tuples of *intraMSG* labels, i.e., $(C_1.l_1.C'_1, \ldots, C_m.l_m.C'_m)$. The sets and the tuples are referenced by an *id* placed just under the filled circles (see the identifier b in Figures 2 and 3) and the arrows (see the identifier g in Figures 2), respectively.

Past constraints are placed near the message arrow source and future constraints are placed near to the arrow target (for both *unwanted messages constraints* and *chain constraints*). We disallow a message to have an *unwanted messages constraint* and a *chain constraint* at the same time. Even though it is possible to define semantics for such a combination of constraints, for the sake of simplicity, we impose this limitation because it could not be really intuitive for an end-user to easily understand its usage. Figure 6 shows the allowed constraints combinations;

(iv) Alternative: the *alternative* operator has been introduced in PSC to have the possibility of specifying that one or more alternative sequences of messages can be non-deterministically chosen;

(v) Parallel: informally, the *parallel* operator allows a parallel merge of the message sequence within its operands. The messages posing as arguments of the operands can be interleaved in any way as long as the ordering imposed by each operand as such is preserved;

(vi) Loop: the *loop* operator allows the sequence of messages within its operand to be repeated a given number of times. It is also possible to specify a lower and an upper number of repetitions.

Table I. Comparison between PSC, UML 2.0 Interaction Sequence Diagrams and MSC

Ref	**PSC**	**UML 2.0**	**MSC**	**Comments**
(i)	Fail	Neg	No direct counterpart	Undesired message
	Required	Assert	No direct counterpart	Mandatory message
	Regular	Default message	Default message	Provisional message
(ii)	Strict	Strict	No direct counterpart	Strict sequencing
	Loose	Seq	Seq	Weak sequencing
(iii)	Constraint	No direct counterpart	No direct counterpart	Restrictions on *intraMSGs*
(iv)	Alternative	Alt	Solved by by h-MSC	Alternative choices
(v)	Parallel	Par	Solved by by h-MSC	Parallel operator
(vi)	Loop	Loop	Solved by by h-MSC	Iteration construct

To summarize, in Table I we report a comparison between MSC, UML 2.0 Interaction Sequence Diagrams and PSC, restricted to the PSC features we are considering.

In order to avoid syntactic and semantic misinterpretation, in Section 5.1 we give the formal textual definition, in Section 6 we provide PSC with an operational semantics in terms of Büchi Automata, and in Section 7 we formalize the PSC denotational semantics. By formally defining the syntax and the two semantics we aim at providing the researchers, designers, and developers community with a means to build upon PSC. Moreover, with the prospect of developing a good PSC tool support and as a base for automatic formal analysis, absence of ambiguity is required.

5.1. PSC textual notation

In this section we begin by introducing the basic concept of *Structural linearization* to give to PSC a linear structure based on time-lines; then we proceed by presenting PSC's formal syntax definition that makes use of this structure.

Definition 1 (structural linearization) Let $A = \{a_0, \ldots, a_n\}$ be a countable set of elements and $a_0 \prec a_1 \prec \ldots \prec a_n$, where $\prec \subseteq A \times A$ is a strict total order. We refer to $<a_0 \cdot a_1 \cdot \ldots \cdot a_n>$ as the *structural linearization* (or simply *linearization*) of the elements $a_0, a_1, \ldots, a_n$ induced by $\prec$. Given a *linearization* $<a_0 \cdot a_1 \cdot \ldots \cdot a_n>$, we refer to $<a_i \cdot \ldots \cdot a_j>$, $0 \leq i \leq j \leq n$, as a *sub-linearization* of it. Note that a *sub-linearization* is itself a *linearization* and when there is no ambiguity the two terms will be used interchangeably. The number of elements of a (*sub-*)*linearization* $<a_0 \cdot a_1 \cdot \ldots \cdot a_n>$ is denoted as $|<a_0 \cdot a_1 \cdot \ldots \cdot a_n>|$.

Given two *linearizations* $sl{=}<a_0 \cdot a_1 \cdot \ldots \cdot a_n>$ and $sl'{=}<a'_0 \cdot a'_1 \cdot \ldots \cdot a'_m>$, their concatenation $sl \cdot sl'$ is $<a_0 \cdot a_1 \cdot \ldots \cdot a_n \cdot a'_0 \cdot a'_1 \cdot \ldots \cdot a'_m>$. We use sl^k to denote the concatenation of sl with itself k times.

Definition 2 (Property Sequence Charts) A Property Sequence Chart (PSC) is a structure $psc = (L, I, T, \prec, arrowMSGs, t2m, SLO)$ where:

- ► $L = \{l_1, l_2, \ldots, l_m\}$ is a set of message labels.
- ► I is a set of component instance names.
- ► $T = \{t_0, t_1, \ldots, t_{n+1}\}$ is a countable set of time-lines and $t_0 \prec t_1 \prec \ldots \prec t_{n+1}$.
- ► $\prec \subseteq T \times T$ is a strict total order.

Let $G = \{(C'_1.l_1.C''_1, \ldots, C'_t.l_t.C''_t) \mid l_i \in L,\ C'_i, C''_i \in I,\ 1 \leq i \leq t\}$ be a set of tuples over *intraMSG* labels given as arguments to chain constraints (refer to Section 5 point (iii)). We use $g, g', g'', \ldots$ to denote elements in G.

Let $B = \{\{C'_1.l_1.C''_1, \ldots, C'_t.l_t.C''_t\} \mid l_i \in L,\ C'_i, C''_i \in I,\ 1 \leq i \leq t\}$ be a set of sets over *intraMSG* labels given as arguments to unwanted messages constraints (refer to Section 5 point (iii)). We use $b, b', b'', \ldots$ to denote elements in B.

Thus, pfC=$\{\Diamond(g) \mid \Diamond\in\{\Rightarrow,\not\Rightarrow\}, g\in G\}\cup\{\bullet(b) \mid b\in B\}$ is a set of past and future constraint attributes.

► $arrowMSGs$ is a set of messages and $|arrowMSGs|=|T|-2$. A message is a structure $msg = (type, label, sender, receiver, past, future)$ where: $type \in \{\mathbf{e:}, \mathbf{r:}, \mathbf{f:}\}$ is the message type, $label \in L$, $sender \in I$, $receiver \in I$, $future \in pfC\cup\lambda$, and $past \in pfC\cup\lambda$. We use λ to denote absence of constraints (refer to Section 5 for a straightforward mapping to the corresponding PSC graphical elements).

► $t2m : T \rightarrow arrowMSGs \cup \{\boldsymbol{noMSG}\}$ is a mapping between time-lines $\{t_0, t_1, \ldots, t_{n+1}\}$ and *arrowMSGs* or the element ***noMSG***. Each time-line in $\{t_0, t_1, \ldots, t_{n+1}\}$ is uniquely associated to exactly one *arrowMSG*, except for the time-lines t_0 and t_{n+1} that have no messages and are associated to the element ***noMSG***.

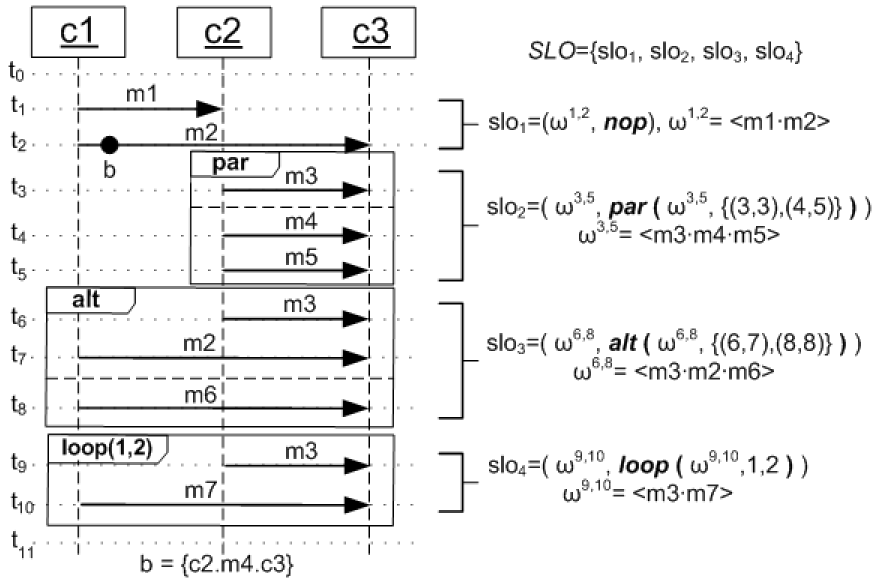

Figure 7. PSC structural linearization

Let $\omega^{1,n}$=$<t2m(t_1) \cdot \ldots \cdot t2m(t_n)>$=$<m_1 \cdot \ldots \cdot m_n>$ be the *structural linearization* of all the messages in $arrowMSGs$ induced by the ordering $t_0{\prec}t_1{\prec}\ldots{\prec}t_{n+1}$. In Figure 7 we have $\omega^{1,10}$=$<m_1, m_2, m_3, m_4, m_5, m_3, m_2, m_6, m_3, m_7>$.

- SL=$\{\omega^{1,r}, \omega^{r+1,s}, \ldots, \omega^{j+1,n}\}$ is a set of strictly sequential *sub-linearizations* of $\omega^{1,n}$. Note that, $<\omega^{1,r}{\cdot}\omega^{r+1,s}{\cdot}\ldots{\cdot}\omega^{j+1,n}>$ is a *structural linearization* of all the *sub-linearizations* in SL and this linearization is only a means for providing each PSC scenario with a linear structure (that will be later used in the semantics definition).

- $\mathtt{sp}(h,k)=\{(h,q),(q+1,z),\ldots,(t+1,k)\}$ is a set of strictly sequential pair of indexes between h and k, $1\leq h\leq q<z<\ldots<t+1\leq k\leq|T|-1$. The set $\mathtt{sp}(h,k)$ is used to split the messages within $\omega^{h,k}$ in more then one strictly sequential (*sub-*)*sub-linearization* $\omega^{h,r}=<m_h\cdot\ldots\cdot m_r>$, $\omega^{r+1,s}=<m_{r+1}\cdot\ldots\cdot m_s>,\ldots,\omega^{j+1,k}=<m_{j+1}\cdot\ldots\cdot m_k>$. The set $\mathtt{sp}(h,k)$ is needed to identify (*sub-*)*sub-linearizations* that are arguments of the operands of parallel and alternative operators.

- $O=\{\boldsymbol{strict}(\omega^{h,h})\ |\ \omega^{h,h}\in SL\}\cup\{\boldsymbol{par}(\omega^{h,k},\mathtt{sp}(h,k))|\ \omega^{h,k}\in SL\}\cup\{\boldsymbol{loop}(\omega^{h,k},l,u\})|\omega^{h,k}\in SL,\ 1\leq l\leq u\}\cup\{\boldsymbol{alt}(\omega^{h,k},\mathtt{sp}(h,k))|\omega^{h,k}\in SL\}\cup\{\boldsymbol{nop}\}$ is the set of *operators* applied to *sub-linearizations* in SL (refer to Section 5 points (iv), (v), and (vi) for an informal description of *operators*).

► $SLO\subseteq SL\times O$ is a binary relation that associates *sub-linearizations* in SL to *operators* in O. SLO is an ordered set and the ordering is induced by SL. See Figure 7 for a straightforward mapping between graphical elements and pairs in SLO.

In the following we enumerate a set of properties that must hold for a PSC to be *valid*:

1. *arrowMSGs consistency*:
   a) $\forall$ pair $m_{i+1}=(t,l,s,r,p,f)\in arrowMSGs$ and $m_i=(t',l',s',r',p',f')\in arrowMSGs$, $(p\neq\lambda)\Rightarrow(f'=\lambda)$ and $(f'\neq\lambda)\Rightarrow(p=\lambda)$, $i\in\{1,\ \ldots,\ n-1\}$. In other words, for two sequential messages the future and the past constraints cannot overlap: there could be a contradiction and it could make no sense.
   b) $\forall(t,l,s,r,p,f)\in arrowMSGs$, $(t=\mathbf{f\!:})\Rightarrow(f=\lambda)$. In others words, *future constraints* cannot be applied to a fail message since the system has no future after such a message has been exchanged.
2. *sub-linearization-operators uniqueness and completeness, and consistency*:
   a) (*uniqueness and completeness*)
   $\forall sl\in SL$ there exists one, and only one, $o\in O$ such that $(sl,o)\in SLO$ (i.e., each *sub-linearization* in SL must be associated to only one *operator* in O).
   b) (*consistency*)
   $(sl,\boldsymbol{strict}(\omega^{h,h}))\in SLO\Rightarrow sl=\omega^{h,h}$;

$(sl,\boldsymbol{par}(\omega^{h,k},\mathtt{sp}(\mathrm{h,k})))\in SLO \Rightarrow sl=\omega^{h,k}$;
$(sl,\boldsymbol{loop}(\omega^{h,k}, \mathrm{l,u}\}))\in SLO \Rightarrow sl=\omega^{h,k}$;
$(sl,\boldsymbol{alt}(\omega^{h,k},\mathtt{sp}(\mathrm{h,k})))\in SLO \Rightarrow sl=\omega^{h,k}$;

In other words, each *sub-linearization* in SL must be consistently associated to the *sub-linearization* taken as input by each *operator* in O. In particular the ***strict*** operator is associated to *sub-linearizations* with only one message.

3. *operators consistency*:

   a) $\boldsymbol{strict}(\omega^{i,i})$, where $\omega^{i,i}$=$<m_i>\in SL$, is the application of the *strict* operator to $m_i \in arrowMSGs$. That is, the *strict* operator has one operand and can be applied to only one message to make it strict with the previous message. Since there cannot be other messages between two strict messages and the strict operator is well defined only as a relation between two contiguous messages within a single lifeline (see Figures 4 and 5), this operator requires that if $i\geq 2$, $m_{i+1} = (t, l, s, r, p, f)$ and $m_i = (t', l', s', r', p', f')$ then (p=λ and f'=λ) and ($s = s'$ or $s = r'$ or $r = r'$ or $r = s'$). If i=1 then p=λ (we recall that the time-line t_0 has no message).

   b) $\boldsymbol{par}(\omega^{i,k},\mathtt{sp}(i, k))$, $\omega^{i,k}\in SL$, $i<k$, is the application of the *parallel* operator to the messages $m_i, \ldots, m_k \in arrowMSGs$. For this operator we require that:

      i) $|\mathtt{sp}(i, k)|\geq 2$. Recalling that $\mathtt{sp}(i, k)$ is used to split $\omega^{i,k}$, the *parallel* operator has two or more operands and can only be applied to sequential (*sub-*)*sub-linearizations* (one for each operand).

      ii) $\forall h\in\{i, \ldots, k\}$, if $m_h = (t, l, s, r, p, f)$ then p=λ and f=λ. In other words, within the *parallel* operator, messages cannot have past and future constraints.

   c) $\boldsymbol{loop}(\omega^{i,j}, l, u)$, $\omega^{i,j}\in SL$, $1\leq l\leq u$, is the application of the *loop* operator to the messages $m_i, \ldots, m_j \in arrowMSGs$. For this operator we require that:

      i) if $m_i = (t, l, s, r, p, f)$ and $m_j = (t', l', s', r', p', f')$ then ($f'\neq\lambda \Rightarrow p$=$\lambda$) and ($p\neq\lambda \Rightarrow f'$=$\lambda$).

Definition 3 (valid Property Sequence Charts) A Property Sequence Chart is valid *iff* the above properties 1, 2, and 3 hold.

6. PSC operational semantics

In this section we give the PSC operational semantics in terms of Büchi automata that can be seen as operational representations of PSC scenarios. In fact, it is well-known (Gerth et al., 1995; Clarke et al., 2001) that all LTL formulas, and hence also all the PSC scenarios, can be translated into a Büchi automaton.

Before giving the PSC operational semantics we briefly recall the definition of Büchi Automata (BA) and how they can be used to perform automata-based model checking (Gerth et al., 1995; Clarke et al., 2001).

6.1. Büchi automata and the automata-based model checking problem

A well known method for describing sets of acceptable or unacceptable behaviors of a system is by using automata over infinite words and Büchi automata represent a popular formulation of infinite word automata.

A Büchi automaton $\mathcal{B}$ is a 5-tuple $< S, A, \triangle, q_0, F >$, where S is a finite set of *states*, A is a set of *actions*, $\triangle \subseteq S \times A \times S$ is a set of *transitions*, $q_0 \in S$ is the *initial state*, and $F \subseteq S$ is a set of *accepting states*. An *execution* of $\mathcal{B}$ over an infinite word $w = a_0 a_1 \ldots$ over A is an infinite sequence $\sigma = q_0 q_1 \ldots$ of elements of S, where $(q_i, a_i, q_{i+1}) \in \triangle, \forall i \geq 0$. An execution of $\mathcal{B}$ is *accepting* if it contains one accepting state in F an infinite number of times. $\mathcal{B}$ *accepts* a word w if there exists an accepting execution of $\mathcal{B}$ over w (Gerth et al., 1995; Clarke et al., 2001).

An accepting state is graphically represented with a double-circled state. An initial state is represented with an entering arrow. For our purposes, $A = A' \cup \{1\} \cup \{\varepsilon\}$ where the transition label *1* represents *true* (i.e., all possible messages) and ε represents a hidden transition. The transition labels in A' are boolean formulae over the sets of *arrowMSGs* and *intraMSGs* labels. In particular, $C_i.!l_i.C'_i$ (or equivalently $!C_i.l_i.C'_i$) represents the logical *"NOT"* and denotes an undesired message transition label. It is evaluated to true (i.e., the transition happens) when any possible message different from the undesired message itself, is exchanged. $C_i.l_i.C'_i || C_j.l_j.C'_j$ is the logical *"OR"* of $C_i.l_i.C'_i$ and $C_j.l_j.C'_j$. It is evaluated to true if either $C_i.l_i.C'_i$ or $C_j.l_j.C'_j$ are exchanged. $C_i.!l_i.C'_i \& C_j.!l_j.C'_j$ is the logical *"AND"* of the undesired messages $C_i.!l_i.C'_i$ and $C_j.!l_j.C'_j$. It is evaluated to true when any message different from $C_i.l_i.C'_i$ and $C_j.l_j.C'_j$ is exchanged. We only allow *"AND"* clauses of undesired messages since it does not make sense to require that two or more messages must occur simultaneously.

By referring to Section 5, we recall that an unwanted messages constraint is specified as a set $\{C_1.l_1.C'_1, \ldots, C_m.l_m.C'_m\}$ of *intraMSG* labels referenced by an *id* placed just under the filled circles (i.e., the graphical elements for representing unwanted messages constraints). Within a transition label, such a constraint is the logical *"AND"* $C_1.!l_1.C'_1 \,\&\, \ldots \,\&\, C_m.!l_m.C'_m$ of the unwanted message labels in the set. Abusing notation, in the figures, which we are going to show, we only use the underlined *id* (see $\underline{b}$) in the place of the logical *"AND"* formulae. Moreover, we use $\underline{l_i}$ in the place of $C_i.l_i.C'_i$ if l_i is an *arrowMSG* label.

The problem of model checking using automata assumes that both the system model and the property are specified by automata. Let us suppose that we want to model check the system $\mathcal{M}$. Let $\mathcal{A}$ be the automaton representing the system $\mathcal{M}$ such that the behavior of $\mathcal{M}$ is the language $\mathcal{L}(\mathcal{A})$. Let $\mathcal{S}$ be an automaton such that $\mathcal{L}(\mathcal{S})$, the corresponding language, contains the set of allowed behaviors. The system $\mathcal{A}$ satisfies $\mathcal{S}$ when $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{S})$. That is, if the intersection $\mathcal{L}(\mathcal{A}) \bigcap \overline{\mathcal{L}(\mathcal{S})}$ contains behaviors, each of them corresponds to a counterexample. Therefore the model checker requires to have the negation of $\mathcal{S}$ to perform the analysis. Since to negate a Büchi automaton is an expensive task, and we use Büchi automata to give semantics to PSC, the translation algorithm from PSC to Büchi automata directly derives the Büchi automaton corresponding to the negation of the desired temporal property. That is, when an execution of the automata is accepting then the property is violated.

In the following, we present the translation rules and the pseudocode of the algorithms used to translate a PSC into its corresponding Büchi automaton.

6.2. Psc2Ba: PSC to Büchi Automata

The translation algorithm Psc2Ba makes use of *fundamental translation rules* (Section 6.2.1) and *subroutines* (Section 6.2.2). In Section 6.2.3 we report the Psc2Ba *algorithm* that gets as input a PSC and returns the corresponding Büchi automata.

6.2.1. *Fundamental Translation Rules*

The fundamental translation rules are used for directly deriving the Büchi automaton corresponding to a single *arrowMSG* (*Regular*, Sections 6.2.1.1 and 6.2.1.4, *Required*, Sections 6.2.1.2 and 6.2.1.5, and *Fail* message, Sections 6.2.1.3 and 6.2.1.6) subjected to a constraint and/or a strict operator. More precisely, for each message type we consider only a single *arrowMSG* that can be loose, strict and constrained by either an unwanted message constraint or a chain constraint.

In the following, we show the automata obtained from the application of the fundamental translation rules[3]. We refer to these automata as *basic automata*. In each figure it is possible to see states graphically represented with filled circles. These states are called ***final states*** and represent those states reached when correct behaviors happen (i.e., valid continuations). For example, in the case of regular and required messages the correct behavior is performed when the relative message is exchanged (see Figures 8 and 9). In the case of a fail message, the final state is reached when the message is not exchanged (Figure 10). As we will see in Section 6.2.2, *final states* play a crucial role to achieve compose-ability. That is, the fundamental rules derive "chunks" of Büchi automaton that, by means of final states, concur to the construction of more complex automata through compositional rules within the *subroutines*. By composing two *basic automata*, the resulting automaton (called *composite automaton*) can be seen as a new basic automaton and the process of composition can be iterated.

In the following we show the basic automata directly derived by applying the fundamental translation rules.

6.2.1.1. *Regular Message* (Figure 8)

The regular messages represent the construction of a precondition. If a regular message does not happen then the system is not in error (i.e., the property is still valid) but if a regular message (or a set of) happens then a precondition has been satisfied and the continuation of the PSC must be explored. Therefore, the obtained automata do not contain any accepting states but they contain final states that are reached when the regular messages are exchanged. We recall that the translation algorithm directly derives the Büchi automaton corresponding to the negation of the desired temporal property.

The *e loose* rule in Figure 8 represents the weakest rule for regular messages in which if a happens then it causes a transition to a final state. The self-transition labeled 1 in the state $S0$ means that we are waiting for an occurrence of a, not necessarily the first one (note that the generated Büchi automaton is non-deterministic) and other messages can be exchanged before a.

[3] Note that the simple scenarios shown in each figure are only for presentation purpose and the translation rules are valid for any *arrowMSG* positioned on an arbitrary time-line.

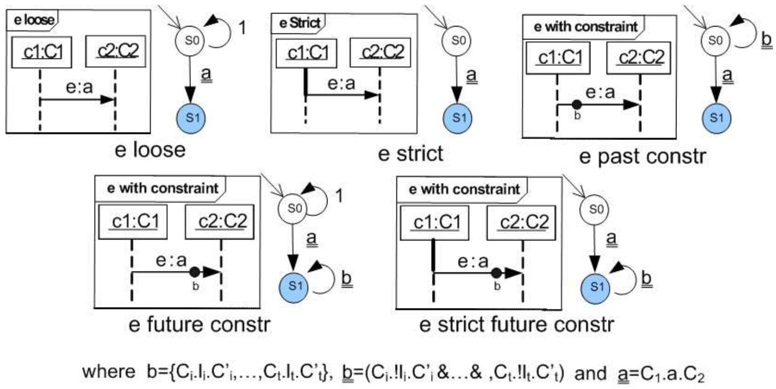

Figure 8. Regular Messages

The rule *e strict* is for the strict operator. Here, differently from the *e loose* rule, after having reached the state $S0$, we have a valid continuation to be explored iff the next exchanged message is exactly a. In fact, we recall that, since there are no others exiting transitions from $S0$, any other message but a will immediately lead to a non valid continuation.

The rule *e past constr* is the combination of the message a with the unwanted messages constraint b. The idea is that we have a valid continuation if a happens and during its past no message $m \in b$ has been exchanged, i.e., the construction of the valid continuation vanishes if in the past of a one message $m \in b$ has been exchanged. This is obtained by means of the self loop labeled $\underline{b}$ on the state $S0$.

The *e future constr* rule is for the future constraint. In this case we want to have one message $m \in b$ after having reached the valid continuation on $S1$. More precisely, the valid continuation is no longer valid if one message $m \in b$ is exchanged.

The last rule is *e strict future constr* that is exactly the intuitive combination of *e strict* with *e future constr*.

6.2.1.2. *Required Message* (Figure 9)

A required message is a message that must be exchanged. In the case of the weakest rule *e loose*, if the message never happens the automaton cycles infinitely often on the accepting state $S0$ and the property is not valid. A valid continuation can be reached if a happens. Note that, due to the weakness of this rule, all the other messages have no restrictions and the valid continuation can be reached even when a is not the first message from the state $S0$. On the contrary, the rule *r strict* shows that if any other message but a (i.e. $!\underline{a}$ in figure) happens while in the state $S0$ then the sink accepting state $S2$ is immediately reached and there are no more chances for satisfying the property.

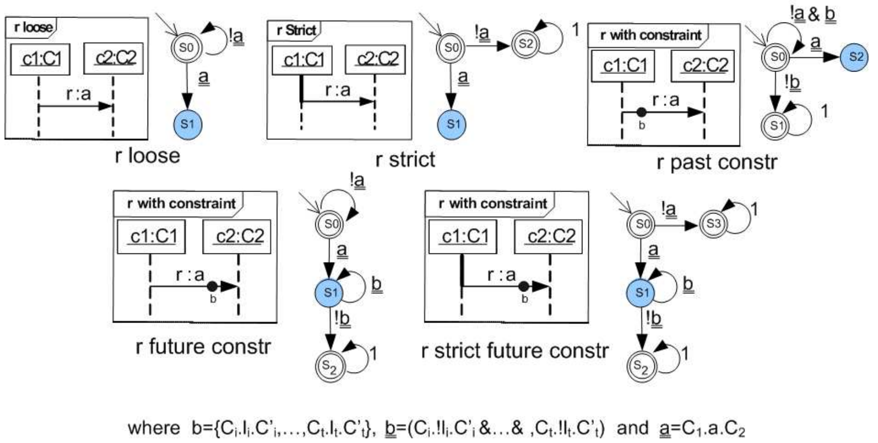

Figure 9. Required Messages

The rule *r past constr* raises an error if infinitely often $!\underline{a}\&\underline{b}$ happens in the state $S0$ and hence both message a and all messages in b are not exchanged. While in this state, we still have a chance of reaching the valid continuation on $S2$. In fact, while b is true, the past constraint is satisfied and, since we do not have strict ordering, we can wait for the message a. Conversely, immediately when b is not true (i.e., a message $m \in b$ is exchanged) there is the unavoidable sink accepting state $S1$.

The *r constr future* rule is for future constraints in which b is imposed on the future. The last rule is *r strict future constr* and it is exactly the combination of *r strict* and *r future constr*.

6.2.1.3. *Fail Message* (Figure 10)

A fail message is a message that should never occur. The *f loose* rule shows that if the message a never happens, the automaton cycles infinitely often on the final state $S0$ (the valid continuation) and the property is not violated; exactly when the first message a happens the automaton reaches an accepting sink node. Note that, the automaton is non-deterministic and if a happens the error is raised since there is at least one run leading to an accepting sink node (i.e., the state $S1$).

Considering the *f strict* rule there is an error only if a happens as first message. An important consideration here is that, differently from regular and required messages, the rule with the strict attribute is the weakest one since the first message different from a is enough for avoiding the failure.

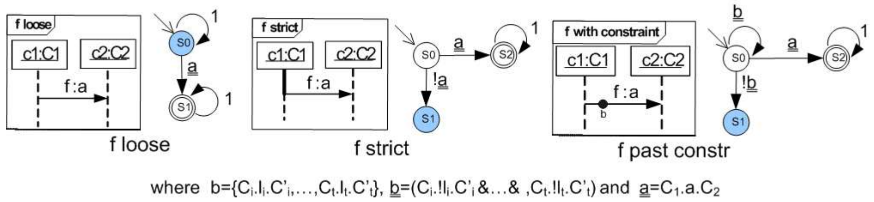

Figure 10. Fail Messages

The same considerations about the strict ordering, and the constraints scope, already done for regular and required messages, hold for fail messages (we recall that fail messages cannot have future constraints). In particular, in this case the past constraints represent restrictions that should hold in the past in order to have a failure with the fail message. For the past constraint, if b is false before a happens then the "precondition" for the failure is falsified. Then we do not have an undesired behavior but we reach the valid continuation on $S1$ (see the transition labeled $!b$ in the *f past constr* rule).

In the following three sub-sections we report the semantics of chain constraints for each type of message.

6.2.1.4. *Chain constraint on Regular messages* (Figure 11)

Before giving a detailed description of the chain constraints, which are used to express relationships between chains of *intraMSGs* and *arrowMSGs*, we recall that we distinguish between *wanted* and *unwanted chains* (Section 5.1). Informally, a wanted chain constraint (either in the past or in the future) related to an *arrowMSG* a represents a sequence of *intraMSGs* $(m_1, \ldots, m_n)$ of a and it is satisfied if the messages are exchanged following the loose ordering imposed by the chain itself. Conversely, unwanted chain constraints specify that the messages in the chain are exchanged following any ordering different from the one imposed by the corresponding wanted chain.

Focusing on the rule *e future unwanted chain constr*, we note that the valid continuation is in every state after a has been exchanged and the chain is not completely performed. In fact, since we do not want the messages $(m_1, \ldots, m_n)$ to be exchanged in the order of the corresponding wanted chain, before that the corresponding wanted chain has been completely accomplished, each state is a final state (and hence, a valid continuation). It is worth noticing that the "last chance" state

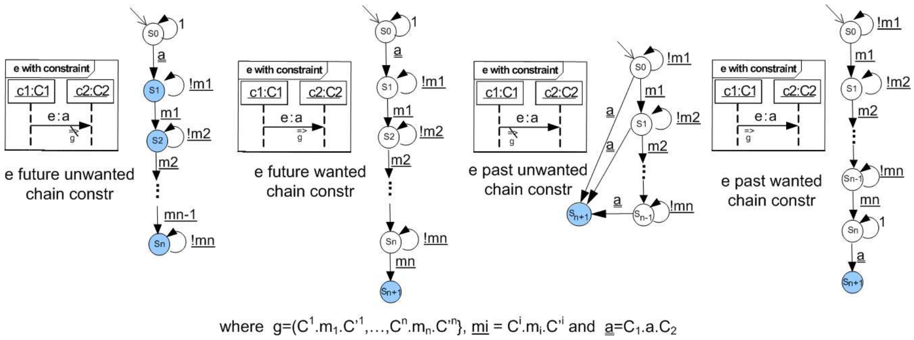

Figure 11. Chain constraint on Regular messages

for keeping a valid continuation is the one before the last message m_n happens.

In *e past unwanted chain constr* we have a valid continuation only if a happens before the chain has been completely accomplished. Conversely, in the case of *e future wanted chain constr*, after a happens, the valid continuation is reached when the chain has been completely accomplished. Complementarily, in the rule *e past wanted chain constr*, the valid continuation is obtained when a happens after the chain.

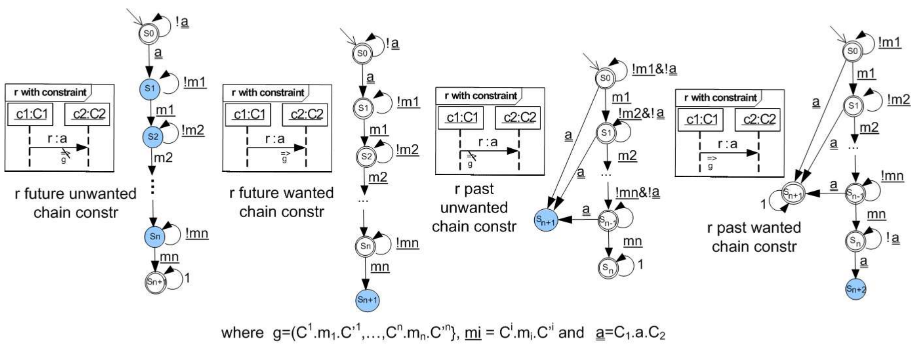

Figure 12. Chain constraint on Required messages

6.2.1.5. *Chain constraint on Required messages* (Figure 12)

The rule *r future unwanted chain constr* has, as valid continuations, every state reached after a has been exchanged and the chain has not been completely accomplished. The property is not satisfied in two accepting states: the first one is $S0$, in which we are still waiting for a (a is required); the second one is the sink accepting state $Sn{+}1$ reached if the chain is completely accomplished (remember that we do not want the chain).

The rule *r past unwanted chain constr* contains only one valid continuation state reached if a happens before the chain has been completely accomplished. Each other state is accepting since it is mandatory to exchange the message a. The state Sn is a sink node and it is reached when the chain has been completely accomplished and a has not been exchanged yet.

For the *r future wanted chain constr* we have a valid continuation if the chain is accomplished after a. All the other states are accepting because both a and the chain are required (now we want the chain). The last rule *r past wanted chain constr* has the valid continuation (on the state $Sn{+}2$) when a happens after the chain has been completely accomplished. We have an error if a happens before the chain has been completely accomplished and/or it never happens.

6.2.1.6. *Chain constraint on Fail messages* (Figure 13)

Fail messages have only two chain constraint rules since the system has no future after a fail message has been exchanged.

Considering the rule *f past unwanted chain constr* we have an error if a happens after the chain has been completely accomplished since, in this case, the "precondition" for the error is satisfied.

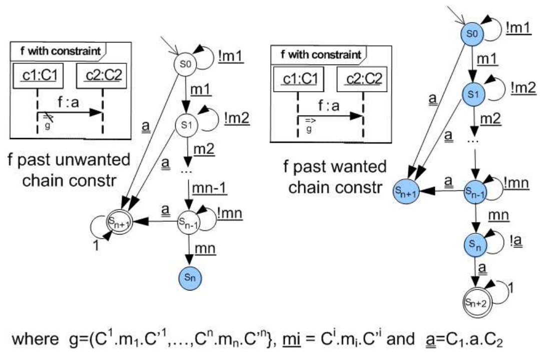

Figure 13. Chain constraint on Fail messages

The rule *f past wanted chain constr* contains only one accepting state ($Sn{+}2$) reached when a happens after the chain has not been completely accomplished. All the other states are valid continuations since the error precondition is not verified. In fact, similarly to the messages constraints for fail messages (see b in Figure 10), if the chain is not completely accomplished before a happens then the precondition for the failure is falsified and we do not have an undesired behavior. The state $Sn{+}1$ is a particular valid continuation since it is reached when a happens before the chain is accomplished.

6.2.2. Psc2Ba *subroutines*

In this section we present subroutines used for composing Büchi automata and special composition rules for PSC operators: *parallel*, *loop* and *alternative*. These rules are *meta-rules* because they are used for translating the mentioned operators that do not have a prefixed number of messages as arguments. For this reason there is no direct mapping into Büchi automata but they require a dedicated translation algorithm.

6.2.2.1. *Composing Büchi automata*

The function *compose()* takes as input two Büchi automata b_1 and b_2 and returns the automaton b' obtained by composing b_1 and b_2. Note that, this function adds ε-transitions going from all the final states of b_1 to the initial state of b_2, but an equivalent automaton without ε-transitions can be obtained through standard automata operations (Gerth et al., 1995). These operations are implemented by the function *collapse()*.

Büchi **function compose** {
Given two Büchi automata b_1=$<S_1, A_1, \triangle_1, q_0^1, F_1>$ and b_2=$<S_2, A_2, \triangle_2, q_0^2, F_2>$:

let $Final_1$ be the set of final states of b_1;
$\triangle'$=$\triangle_1 \cup \triangle_2$;
for each $q \in Final_1$ **do**
  $\triangle'$=$\triangle' \cup \{< q, \varepsilon, q_0^2 >\}$;
end for
b'=$<S_1 \cup S_2, A_1 \cup A_2, \triangle', q_0^1, F_1 \cup F_2>$;
collapse(b');
set the final states of b' as the final states of b_2;
return b'
}

6.2.2.2. PSC *operators translation*

Before showing the meta-rules for the parallel, loop, and alternative operators we introduce two subroutines: *subl2Büchi* and *mergeIFA*.

The function *subl2Büchi()* takes as input a sub-linearization subjected to no operators and returns its associated Büchi automaton. The sub-routine *basicAutomata()* takes as input a message and returns its associated basic automaton by implementing the fundamental translation rules.

Büchi **function subl2Büchi** {
Given a sub-linearization $\omega^{i,j}$:
let $\omega^{i,j}$ be equal to $<m_i \cdot m_{i+1} \cdots m_j>$;
$b := basicAutomata(m_i)$;
for each $k := i+1$ **to** j **do**
    $b' := basicAutomata(m_k)$;
    $b := compose(b, b')$;
end for
return b;
}

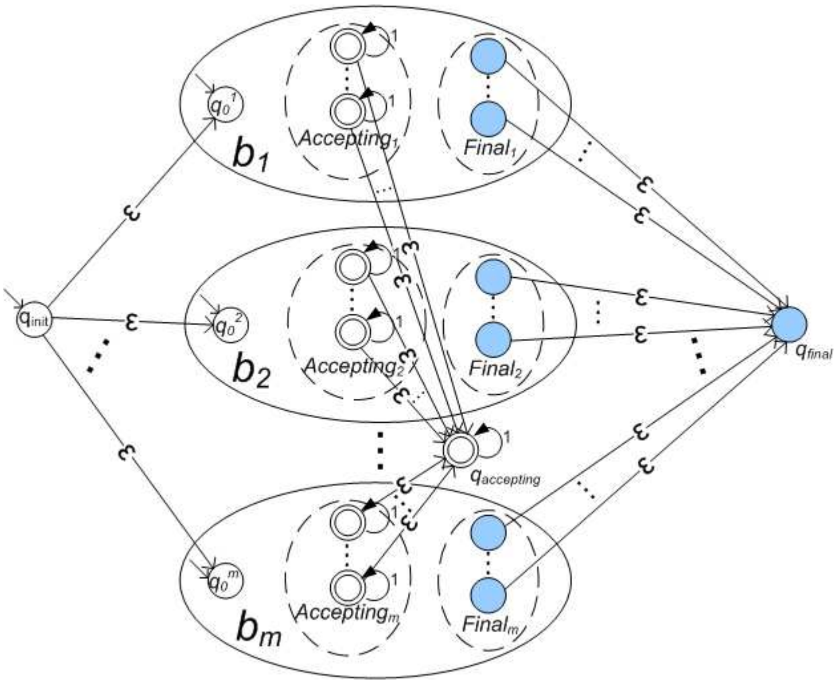

Figure 14. The mergeIFA() function

The function *mergeIFA()* takes as input a list of Büchi automata and, by means of ε-transitions, merges all the initial states, all the final state, and all the sink accepting states having a self-transition labeled by 1. Refer to Figure 14 for a clear description of the algorithm before the *collapse()* function invocation (i.e., till row 23).

Büchi **function mergeIFA** {
Given a list of Büchi automata
b_1=<$S_1, A_1, \triangle_1, q_0^1, F_1$>, . . ., b_m=<$S_m, A_m, \triangle_m, q_0^m, F_m$>:

$\forall_{i=1}^{m}$ let $Final_i$ be the set of final states of b_i;
$\forall_{i=1}^{m}$ let $Accepting_i$ be the set of accepting states of b_i having a self-transition labeled by 1;
$\triangle'=\emptyset$;
for each $i := 1$ **to** m **do**
  $\triangle'=\triangle' \cup \triangle_i$;
end for
$S'=(\bigcup_{i=1}^{m} S_i)\cup\{q_{final}\}$;
$A'=\bigcup_{i=1}^{m} A_i$;
$F'=\{q_{accepting}\}$;
for each $i := 1$ **to** m **do**
  for each $q \in Final_i$ **do**
    $\triangle'=\triangle' \cup \{< q, \varepsilon, q_{final} >\}$;
  end for
end for
for each $i := 1$ **to** m **do**
  for each $q \in Accepting_i$ **do**
    $\triangle'=\triangle' \cup \{< q, \varepsilon, q_{accepting} >\}$;
  end for
end for
for each $i := 1$ **to** m **do**
  $\triangle'=\triangle' \cup \{< q_{init}, \varepsilon, q_0^i >\}$;
end for
b'=<$S', A', \triangle', q_{init}, F'$>;
collapse(b');
set q_{final} as the unique final state of b';
return b'
}

Parallel: the next function permits to derive the Büchi automaton corresponding to the parallel operator. The parallel operator ***par***$(\omega^{h,k}$, $\mathtt{sp}(h,k))$ with $\mathtt{sp}(h,k)=\{(h,q),(q+1,z),\ldots,(t+1,k)\}$ interleaves the linearizations $\omega^{h,q}$, $\omega^{q+1,z}$, . . . , $\omega^{t+1,k}$ in any way as long as the ordering imposed by each linearization as such is preserved. That is, for this operator we need to generate a number, let's say m, of new linearizations ω^{h^1,k^1}, ω^{h^2,k^2}, . . ., ω^{h^m,k^m}. These new linearizations are called *"parallel" linearizations* and each of them has length equal to $|\omega^{h,q}|+|\omega^{q+1,z}|+\ldots+|\omega^{t+1,k}|$. It is worth noticing that $\forall_{j=1}^{m}\ \omega^{h^j,k^j}$=<$m_{h^j}\cdot \ldots \cdot m_{k^j}$> and $h^j,\ldots,k^j$ is a simple permutation of $h,\ldots,k$ according to the concept of interleaving.

For instance, for $\boldsymbol{par}(\omega^{2,5}, \{(2,3),(4,5)\})$ the function generates the linearizations: $<m_2{\cdot}m_3{\cdot}m_4{\cdot}m_5>$, $<m_2{\cdot}m_4{\cdot}m_3{\cdot}m_5>$, $<m_2{\cdot}m_4{\cdot}m_5{\cdot}m_3>$, $<m_4{\cdot}m_5{\cdot}m_2{\cdot}m_3>$, $<m_4{\cdot}m_2{\cdot}m_5{\cdot}m_3>$, and $<m_4{\cdot}m_2{\cdot}m_3{\cdot}m_5>$.

Büchi **function parallel2BA** {
Given the sub-linearization $\omega^{h,k}$ and the set of pairs $\mathtt{sp}(h,k)$=$\{(h,q),(q+1,z),\ldots,(t{+}1,k)\}$ that are arguments of the *"parallel"* operator:

let ω^{h^1,k^1}, ω^{h^2,k^2}, $\ldots$, ω^{h^m,k^m} be the set of *"parallel" linearizations* obtained through the interleaving of $\omega^{h,q}$, $\omega^{q+1,z}$, $\ldots$, $\omega^{t+1,k}$.
for each $j := 1$ **to** m **do**
  $b_j := \mathit{subl2B\ddot{u}chi}(\omega^{h^j,k^j})$;
end for
$b := \mathit{mergeIFA}(b_1, ..., b_m)$;
return b;
}

Loop: the next function permits to derive the Büchi automaton corresponding to the loop operator $\boldsymbol{loop}(\omega^{h,k}, l, u)$. Since this operator repeatedly executes its operand $\omega^{h,k}$ at least l times and at most u times, the function firstly calculates a set of new linearizations $(\omega^{h,k})^l$, $(\omega^{h,k})^{l+1}$, $\ldots$, $(\omega^{h,k})^u$ called *"loop" linearizations* (i.e., concatenations of $\omega^{h,k}$ with itself). Then, it translates these loop linearizations into the corresponding Büchi automaton. Finally, the algorithm collapses the obtained automata by means of the function *mergeIFA*().

Büchi **function loop2BA** {
Given the sub-linearization $\omega^{h,k}$, a lower bound l and an upper bound u that are arguments of the *loop* operator:

let $(\omega^{h,k})^l$, $(\omega^{h,k})^{l+1}$, $\ldots$, $(\omega^{h,k})^u$ be the set of *"loop" linearizations* obtained through the concatenations of $\omega^{h,k}$ with itself from l to u times.
for each $t := l$ **to** u **do**
  $b_t := \mathit{subl2B\ddot{u}chi}((\omega^{h,k})^t)$;
end for
$b := \mathit{mergeIFA}(b_l, ..., b_u)$;
return b;
}

Alternative: the alternative operator $\boldsymbol{alt}(\omega^{h,k}, \mathtt{sp}(h,k))$ with $\mathtt{sp}(h,k)$= $\{(h,q),(q+1,z),\ldots,(t+1,k)\}$ admits the choice of one of the sub-linearizations al_1=$\omega^{h,q}$, al_2=$\omega^{q+1,z}$, $\ldots$, al_m=$\omega^{t+1,k}$, called *"alternative" linearizations*. That is, the following function constructs a Büchi automaton for each sub-linearization and makes use of the *mergeIFA*() function for obtaining a unique Büchi automaton encoding the different alternatives.

Büchi **function alt2BA** {
Given the sub-linearization $\omega^{h,k}$ and the set of pairs $\mathtt{sp}(h,k)$=$\{(h,q),(q+1,z),\ldots,(t+1,k)\}$ that are arguments of the *alternative* operator:

let al_1=$\omega^{h,q}$, al_2=$\omega^{q+1,z}$, ..., al_m=$\omega^{t+1,k}$ be the set of *"alternative" linearizations* obtained by splitting the sub-linearization $\omega^{h,k}$.
for each $j := 1$ **to** m **do**
  b_j := *subl2Büchi*(al_j);
end for
b := *mergeIFA*$(b_1, ..., b_m)$;
return b;
}

6.2.3. Psc2Ba *algorithm*

Büchi **function PSC2BA** {
Given a PSC ***psc*** $= (L, I, T, \prec, arrowMSGs, t2m, SLO)$:

let SLO=$\{(sl_1, o_1),(sl_2, o_2),\ldots,(sl_n, o_n)\} \subseteq SL \times O$ be the set of pairs of sub-linearizations in SL and operators in O.
b := <$\{q_0\}, \emptyset, \emptyset, q_0, \emptyset$>;
for each $i := 1$ **to** n **do**
  Let sl_i be equal to $\omega^{h,k}$ for some h and k;
  if (o_i=***par***($\omega^{h,k}$, $\mathtt{sp}(h,k)$) & $\mathtt{sp}(h,k)$=$\{(h,q),(q+1,z),\ldots,(t+1,k)\}$) **then**
    b'=*parallel2BA*$(\omega^{h,k}, \mathtt{sp}(h,k))$;
    b := *compose*(b, b');
  else
    if (o_i=***loop***($\omega^{h,k}$,l,u)) **then**
      b'=*loop2BA*$(\omega^{h,k}, l, u)$;
      b := *compose*(b, b');
    else
      if (o_i=***alt***($\omega^{h,k}$, $\mathtt{sp}(h,k)$) & $\mathtt{sp}(h,k)$=$\{(h,q),(q+1,z),\ldots,(t+1,k)\}$) **then**
        b'=*alt2BA*$(\omega^{h,k}, \mathtt{sp}(h,k))$;
        b := *compose*(b, b');
      else
        if ($(sl_i, o_i) = (\omega^{h,k},$ ***nop***$)$ & $\omega^{h,k}$=<$m_h \cdot m_{h+1} \cdots m_k$>) **then**
          for each $j := h$ **to** k **do**
            b' := *basicAutomata*(m_j);
            b := *compose*(b, b');
          end for
        end if
      end if
    end if
  end if
end for
return b;
}

7. PSC denotational semantics

In this section we begin by introducing some basic definitions coming from formal languages theory; then we proceed by defining the trace-based semantics of PSC that associates to each PSC the language of all the *invalid traces*. By unambiguously determine which execution sequences are not allowed, this semantics will provide a better understanding of a specific PSC without needing to apply the algorithm to translate it into a Büchi automaton and then to interpret the automaton.

Definition 4 (trace and language of traces) Let $\Lambda = \{a_0, \ldots, a_n\}$ be a countable set of symbols, a *trace* over Λ is a (possible infinite) sequence of elements of Λ obtained by juxtaposition (e.g., $a_2 \cdot a_3 \cdot a_1$ is a trace over Λ). Given a trace s, $|s|$ is its length (the number of symbols). By definition, we write $|s| \doteq \infty$ when s is infinite and $|s| \lessdot \infty$ when s has finite length. Moreover, ε is the empty trace and $|\varepsilon|=0$. Denoting by Λ^* the set of all traces over Λ having finite length, a *language of finite traces* is a subset of Λ^*. Denoting by Λ^∞ the set of all traces over Λ having infinite length, a *language of infinite traces* is a subset of Λ^∞. The empty language is denoted by $\varnothing$.

Definition 5 (concatenation of traces and concatenation of languages) Given two traces s and s', $|s| \lessdot \infty$ and either $|s'| \lessdot \infty$ or $|s'| \doteq \infty$, the *concatenation of s and s'* is the trace denoted $s \cdot s'$ obtained by juxtaposition of s and s'.
Given two languages $L_1 \subseteq \Lambda^*$ and either $L_2 \subseteq \Lambda^*$ or $L_2 \subseteq \Lambda^\infty$, the *concatenation of L_1 and L_2* is the language $L_1 \cdot L_2 = \{w | w = s \cdot s', s \in L_1, s' \in L_2\}$. Note that, the concatenations $L_1 \cdot \varnothing = L_1$, $\varnothing \cdot L_1 = L_1$, $\varnothing \cdot L_2 = L_2$ are well defined but, if $|s'| \doteq \infty$ and $L_2 \subseteq \Lambda^\infty$ then both the concatenations $s' \cdot s$ and $L_2 \cdot L_1$ are not well defined since s' is an infinite trace and L_2 contains infinite traces.

Definition 6 (closure, positive closure, and infinite closure of a language) Given a language $L \subseteq \Lambda^*$, the *closure of L* is the language $L^* = \bigcup_{n \geq 0} L^n$, where $L^0 = \{\varepsilon\}$ and $L^n = L \cdot L^{n-1}$, $n \geq 1$.
The *positive closure of L* is the language $L^+ = \bigcup_{n \geq 1} L^n = L^* \setminus \{\varepsilon\}$.
If $\varepsilon \notin L$, the *infinite closure of L* is the infinite language L^∞ of traces having infinite length obtained by concatenating, an infinite number of times, words from L in every possible way. Now it is clear because we have used Λ^* and Λ^∞ to denote the sets of all finite and infinite traces over Λ, respectively.

7.1. PSC invalid trace-semantics

To define the semantics of a ***psc*** = (L, I, T, $\prec$, $arrowMSGs$, $t2m$, SLO) we need:

1. $\mathcal{L}$=$\{C'_i.l_i.C''_i | (C'_1.l_1.C''_1, \ldots, C'_i.l_i.C''_i, \ldots, C'_t.l_t.C''_t) \in G,\ 1 \leq i \leq t\} \cup$
   $\{C'.l.C'' |\ C'.l.C'' \in b, b \in B\} \cup$
   $\{C.l.C' |\ (type, l, C, C', past, future) \in arrowMSGs\}$
   the alphabet of labels used within arguments of unwanted messages constraints and chain constraints, and *arrowMSG* labels enriched by the sender and the receiver components.

2. $[\![\cdot]\!]^{it} \subseteq (\mathcal{L}^* \cup \mathcal{L}^\infty)$ the set of *invalid traces* that represents unwanted system behaviors.

3. $[\![\cdot]\!]^{vc} \subseteq \mathcal{L}^+$ the set of traces representing *valid continuations* of the system execution. This set contains traces that represent correct behaviors with respect to the specified temporal property. As it will be clear later, valid continuations are used for defining invalid traces.

The denotational semantics of PSC is defined in a compositional way by means of:

1. *single message semantics:* the semantics for a linearization constituted by a single *arrowMSG* possibly subjected to a constraint and/or a ***strict*** operator (Section 7.2). In other words, for each message type we consider only a single *arrowMSG* that can be associated to the ***nop*** operator or to the ***strict*** operator and constrained by either an unwanted messages constraint or a chain constraint. Proceeding in this way, on one hand we will have several semantic rules, *i.e.*, one for each "combination" of message type, constraints and strict operator; on the other hand it will be very simple to achieve the set of invalid traces of a given combination by simply choosing the proper rule. Moreover, this allows us to easily achieve compose-ability for ***nop*** *sequential messages semantics* and *operators semantics* we are going to introduce.

2. ***nop*** *sequential messages semantics:* the semantics for a linearization of messages associated to the ***nop*** operator (Section 7.3). This semantics is defined by composing the semantics of each single message. Figure 15 shows the so called ***"nop"*** *linearization* $<m_1 \cdot m_2>$ for the messages m_1 and m_2 without operators. The semantics of $<m_1 \cdot m_2>$ is defined by composing the *single message semantics* of m_1 and m_2.

3. *operators semantics:* the semantics for a (sub-)linearization of messages subjected to the ***par***, ***loop***, or ***alt*** operators (Section 7.4). This semantics is defined by means of the *"parallel" linearizations*, *"loop" linearizations*, and *"alternative" linearizations* as defined in Section 6.2.2.2. Figure 15 shows the *"parallel" linearizations* $<m_3{\cdot}m_4{\cdot}m_5>$, $<m_4{\cdot}m_3{\cdot}m_5>$, and $<m_4{\cdot}m_5{\cdot}m_3>$ generated by the parallel operator; the *"alternative" linearizations* $<m_3{\cdot}m_2>$ and $<m_6>$ generated by the alternative operator; the *"loop" linearizations* $<m_3{\cdot}m_7>$ and $<m_3{\cdot}m_7{\cdot}m_3{\cdot}m_7>$ generated by the loop operator.

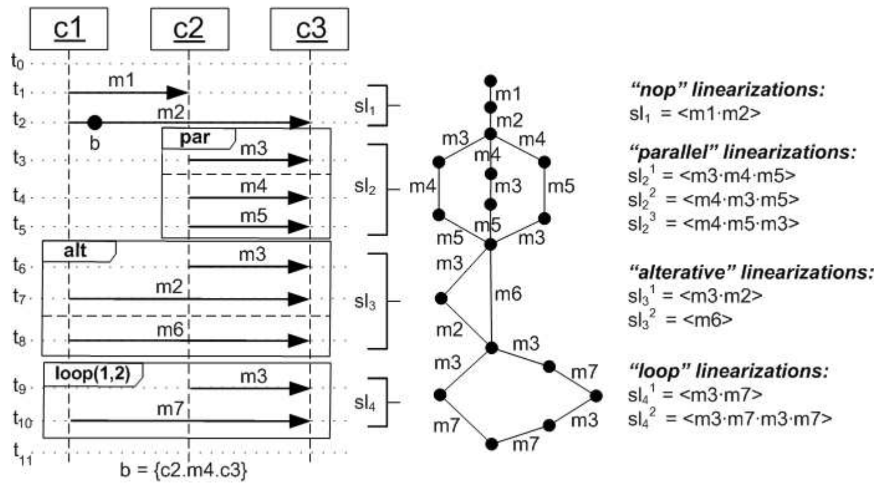

Figure 15. Operators linearizations

Now, by considering all these "additional" sub-linearizations deriving from the sub-linearizations in SL we define SLO'=$\{(sl,\boldsymbol{nop}) \mid sl$ *is an additional linearization*$\}$ as the set of pairs of all the additional linearizations associated to the ***nop*** operator. For instance, by referring to Figure 15, from the sub-linearization $sl_3 = <m_3{\cdot}m_2{\cdot}m_6>$ in SL (the argument of the ***alt*** operator), we derive the pairs $(<m_3{\cdot}m_2>,\boldsymbol{nop})$ and $(<m_6>,\boldsymbol{nop})$. Note that the additional sub-linearizations in SLO' have a partial order induced by the total order of the sub-linearizations in SL.

The semantics of PSC is obtained by considering both the set of *valid continuations* and the set of *invalid traces* denoted by $[\![(sl,o)]\!]^{vc}$ and $[\![(sl,o)]\!]^{it}$, respectively ($(sl,o) \in SLO \cup SLO'$).

The semantics of a single sub-linearization might depend on the messages within the subsequent sub-linearization. That is, to give the semantics of $(sl,o) \in SLO \cup SLO'$ we need to define a function $\mathfrak{F}$:

$SLO \cup SLO' \rightarrow \mathcal{L}$ that, given $(sl', o') \in SLO \cup SLO'$, the subsequent linearization of (sl, o), returns the set of message labels that must be accounted to give the semantics of (sl, o). Formally:

$$
\mathfrak{F}(s) = \begin{cases}
\varnothing & \text{if } \begin{array}[t]{l} s = (<(\mathbf{e:}, l, C, C', \lambda, \lambda)>, \mathbf{nop}) \text{ or} \\ s = (<(\mathbf{e:}, l, C, C', \lambda, c)>, \mathbf{nop}) \text{ or} \\ s = (<(\mathbf{f:}, l, C, C', \lambda, \lambda)>, \mathbf{nop}) \text{ or} \\ s = (<\omega^{h,h}>, \boldsymbol{strict}(\omega^{h,h})) \text{ or} \\ s = (<\mathbf{noMSG}>, \mathbf{nop}) \end{array} \\[2ex]
\{C.l.C'\} & \text{if } \begin{array}[t]{l} s = (<(\mathbf{r:}, l, C, C', \lambda, \lambda)>, \mathbf{nop}) \text{ or} \\ s = (<(\mathbf{r:}, l, C, C', \lambda, c)>, \mathbf{nop}) \end{array} \\[2ex]
g & \text{if } \begin{array}[t]{l} s = (<(\mathbf{e:}, l, C, C', \bullet(b), \lambda)>, \mathbf{nop}) \text{ or} \\ s = (<(\mathbf{f:}, l, C, C', \bullet(b), \lambda)>, \mathbf{nop}) \end{array} \\[2ex]
g \cup \{C.l.C'\} & \text{if } s = (<(\mathbf{r:}, l, C, C', \bullet(b), \lambda)>, \mathbf{nop}) \\[2ex]
\{l_1\} & \text{if } \begin{array}[t]{l} s = (<(\mathbf{e:}, l, C, C', \Rightarrow(l_1, l_2, \ldots, l_n), \lambda)>, \mathbf{nop}) \text{ or} \\ s = (<(\mathbf{e:}, l, C, C', \not\Rightarrow(l_1, l_2, \ldots, l_n), \lambda)>, \mathbf{nop}) \text{ or} \\ s = (<(\mathbf{r:}, l, C, C', \Rightarrow(l_1, l_2, \ldots, l_n), \lambda)>, \mathbf{nop}) \text{ or} \\ s = (<(\mathbf{f:}, l, C, C', \Rightarrow(l_1, l_2, \ldots, l_n), \lambda)>, \mathbf{nop}) \text{ or} \\ s = (<(\mathbf{f:}, l, C, C', \not\Rightarrow(l_1, l_2, \ldots, l_n), \lambda)>, \mathbf{nop}) \end{array} \\[2ex]
\{l_1\} \cup \{C.l.C'\} & \text{if } s = (<(\mathbf{r:}, l, C, C', \not\Rightarrow(l_1, l_2, \ldots, l_n), \lambda)>, \mathbf{nop}) \\[2ex]
\{l^h, l^{q+1}, \ldots, l^{t+1}\} & \text{if } \begin{array}[t]{l} s = (<\omega^{h,k}>, \boldsymbol{par}(\omega^{h,k}, \mathtt{sp}(h,k))) \text{ or} \\ s = (<\omega^{h,k}>, \boldsymbol{alt}(\omega^{h,k}, \mathtt{sp}(h,k))), \\ \\ \mathtt{sp}(h,k) = \{(h,q), (q+1,z), \ldots, (t+1,k)\}, \\ t2m(h) = <(\mathbf{t}, l^h, C, C', \lambda, \lambda)>, \\ t2m(q+1) = <(\mathbf{t}, l^{q+1}, C, C', \lambda, \lambda)>, \\ \ldots, \\ t2m(t+1) = <(\mathbf{t}, l^{t+1}, C, C', \lambda, \lambda)>, \\ \mathbf{t} \in \{\mathbf{e:}, \mathbf{r:}, \mathbf{f:}\} \end{array}
\end{cases}
$$

To obtain the proper subsequent linearization we define a function $succ : SLO \cup SLO' \rightarrow SLO \cup SLO'$ that takes as input an $slo \in SLO \cup SLO'$ and returns $slo' \in SLO \cup SLO'$ that is the subsequent sub-linearization of slo.

$$succ(s) = \begin{cases} s_{i+1} & \text{if} \quad s{=}s_i,\ SLO = s_1,\ldots,s_i,s_{i+1},\ldots,s_k \\ (<m_{j+1}>,\boldsymbol{nop}) & \text{if} \quad s{=}(<m_j>,\boldsymbol{nop}), \\ & \quad\ (<m_i\cdots m_j\cdots m_k>,\boldsymbol{nop}){\in}SLO',\ i{\leq}j{<}k \\ s_{i+1} & \text{if} \quad s{=}(<m_j>,\boldsymbol{nop}),\ SLO{=}s_1,\ldots,s_i,s_{i+1},\ldots,s_k, \\ & \quad\ (<m_i\cdots m_j>,\boldsymbol{nop}){\in}SLO' \text{is an additional} \\ & \quad\ \text{linearization derived from } s_i \end{cases}$$

For instance, referring to Figure 15, $succ(sl_1) = sl_2$ and $succ(sl_2) = sl_3$; $succ((<m3>, \boldsymbol{nop}))=$ $(<m5>, \boldsymbol{nop})$ for sl_2^2 $=<m_4{\cdot}m_3{\cdot}m_5>$ and $succ((<m7>, \boldsymbol{nop}))=$ $(<m3>, \boldsymbol{nop})$ for $sl_4^2 =<m_3{\cdot}m_7{\cdot}m_3{\cdot}m_7>$; $succ((<m5>, \boldsymbol{nop}))=$ sl_3 for $sl_2^2 =<m_4{\cdot}m_3{\cdot}m_5>$.

7.2. Single message semantics

7.2.1. *Regular, Required, and Fail messages without constraints and strict ordering relation*

$[\![(<(\mathbf{e:}, l, C, C', \lambda, \lambda)>, \boldsymbol{nop})]\!]^{vc}{=}\{\alpha{\cdot}C.l.C' \mid \alpha \in \mathcal{L}^*\}$
$[\![(<(\mathbf{e:}, l, C, C', \lambda, \lambda)>, \boldsymbol{nop})]\!]^{it}{=}\varnothing$
$[\![(<(\mathbf{r:}, l, C, C', \lambda, \lambda)>, \boldsymbol{nop})]\!]^{vc}{=}\{\alpha{\cdot}C.l.C' \mid \alpha \in (\mathcal{L} \setminus (\{C.l.C'\} \cup \mathfrak{F}(succ(\omega^{h,h}))))^*,\ \omega^{h,h}{=}<(\mathbf{r:}, l, C, C', \lambda, \lambda)>\ \}$
$[\![(<(\mathbf{r:}, l, C, C', \lambda, \lambda)>, \boldsymbol{nop})]\!]^{it}{=}\{\alpha \mid \alpha \in (\mathcal{L} \setminus \{C.l.C'\})^\infty\}$
$[\![(<(\mathbf{f:}, l, C, C', \lambda, \lambda)>, \boldsymbol{nop})]\!]^{vc}{=}\{\alpha \mid \alpha \in (\mathcal{L} \setminus \{C.l.C'\})^*\}$
$[\![(<(\mathbf{f:}, l, C, C', \lambda, \lambda)>, \boldsymbol{nop})]\!]^{it}{=}\{\alpha{\cdot}C.l.C' \mid \alpha \in (\mathcal{L} \setminus (\{C.l.C'\} \cup \mathfrak{F}(succ(\omega^{h,h}))))^*,\ \omega^{h,h}{=}<(\mathbf{f:}, l, C, C', \lambda, \lambda)>\}$

7.2.2. *Regular, Required, and Fail messages with past constraints*

$$[\![(<(\mathbf{e:}, l, C, C', \bullet(b), \lambda)>, \boldsymbol{nop})]\!]^{vc} = \{\alpha \cdot C.l.C' \mid \alpha \in (\mathcal{L} \setminus b)^*\}$$

$$[\![(<(\mathbf{e:}, l, C, C', \bullet(b), \lambda)>, \boldsymbol{nop})]\!]^{it} = \varnothing$$

$$[\![(<(\mathbf{r:}, l, C, C', \bullet(b), \lambda)>, \boldsymbol{nop})]\!]^{vc} = \{\alpha \cdot C.l.C' \mid \alpha \in (\mathcal{L} \setminus (b \cup \{C.l.C'\}))^*\}$$

$$[\![(<(\mathbf{r:}, l, C, C', \bullet(b), \lambda)>, \boldsymbol{nop})]\!]^{it} = \{\alpha \mid \alpha \in (\mathcal{L} \setminus (b \cup \{C.l.C'\}))^\infty\} \bigcup \{\alpha \cdot l' \mid \alpha \in (\mathcal{L} \setminus (b \cup \{C.l.C'\}))^*, l' \in b\}$$

$$[\![(<(\mathbf{f:}, l, C, C', \bullet(b), \lambda)>, \boldsymbol{nop})]\!]^{vc} = \{\alpha \cdot l' \mid \alpha \in (\mathcal{L} \setminus (b \cup \{C.l.C'\}))^*, l' \in b\}$$

$$[\![(<(\mathbf{f:}, l, C, C', \bullet(b), \lambda)>, \boldsymbol{nop})]\!]^{it} = \{\alpha \cdot C.l.C' \mid \alpha \in (\mathcal{L} \setminus (b \cup \{C.l.C'\}))^*\}$$

7.2.3. *Regular and Required messages with future constraints*

$$[\![(<(\mathbf{e:}, l, C, C', \lambda, \bullet(b))>, \boldsymbol{nop})]\!]^{vc} = \{\alpha \cdot C.l.C' \cdot \beta \mid \alpha \in \mathcal{L}^*, \beta \in (\mathcal{L} \setminus b \cup \mathfrak{F}(succ(\omega^{h,h})))^*, \omega^{h,h} = <(\mathbf{e:}, l, C, C', \lambda, \bullet(b))>\}$$

$$[\![(<(\mathbf{e:}, l, C, C', \lambda, \bullet(b))>, \boldsymbol{nop})]\!]^{it} = \varnothing$$

$$[\![(<(\mathbf{r:}, l, C, C', \lambda, \bullet(b))>, \boldsymbol{nop})]\!]^{vc} = \{\alpha \cdot C.l.C' \cdot \beta \mid \alpha \in (\mathcal{L} \setminus \{C.l.C'\})^*, \beta \in (\mathcal{L} \setminus b \cup \mathfrak{F}(succ(\omega^{h,h})))^*, \omega^{h,h} = <(\mathbf{r:}, l, C, C', \lambda, \bullet(b))>\}$$

$$[\![(<(\mathbf{r:}, l, C, C', \lambda, \bullet(b))>, \boldsymbol{nop})]\!]^{it} = \{\alpha \mid \alpha \in (\mathcal{L} \setminus \{C.l.C'\})^\infty\} \cup \{\alpha \cdot C.l.C' \cdot \beta \cdot l' \mid \alpha \in (\mathcal{L} \setminus \{C.l.C'\})^*, \beta \in (\mathcal{L} \setminus b \cup \mathfrak{F}(succ(\omega^{h,h})))^*, \omega^{h,h} = <(\mathbf{r:}, l, C, C', \lambda, \bullet(b))>, l' \in b\}$$

7.2.4. *Regular, Required, and Fail messages with past wanted chain constraints*

$$[\![(<(\mathbf{e:}, l, C, C', \Rightarrow(l_1, l_2, \ldots, l_n), \lambda)>, \boldsymbol{nop})]\!]^{vc} = \{\beta_1 \cdot l_1 \cdot \beta_2 \cdot l_2 \cdot \ldots \cdot \beta_n \cdot l_n \cdot \gamma \cdot C.l.C' \mid \beta_i \in (\mathcal{L} \setminus \{l_i\})^*, 1 \leq i \leq n, \gamma \in \mathcal{L}^*\}$$

$$[\![(<(\mathbf{e:}, l, C, C', \Rightarrow(l_1, \ldots, l_n), \lambda)>, \boldsymbol{nop})]\!]^{it} = \varnothing$$

$$[\![(<(\mathbf{r:},l,C,C',\Rightarrow(l_1,l_2,\ldots,l_n),\lambda)>,\boldsymbol{nop})]\!]^{vc}=\{\beta_1\cdot l_1\cdot\beta_2\cdot l_2\cdot\ldots\cdot\beta_n\cdot l_n\cdot\gamma\cdot C.l.C' \mid \beta_i\in(\mathcal{L}\backslash\{l_i\})^*,\ 1\leq i\leq n,\ \gamma\in(\mathcal{L}\backslash\ \{C.l.C'\}\ )^*\}$$

$$[\![(<(\mathbf{r:},l,C,C',\Rightarrow(l_1,\ldots,l_n),\lambda)>,\boldsymbol{nop})]\!]^{it}=(\textstyle\bigcup_{0\leq i\leq n}\mathcal{A}_i)\cup(\bigcup_{1\leq i\leq n}\mathcal{B}_i)\cup(\bigcup_{1\leq i\leq n}\mathcal{C}_i)$$

where

$\mathcal{A}_i=\{\alpha\cdot l_1\cdot\beta_1\cdot l_2\beta_2\cdot\ldots\cdot l_{i-1}\cdot\beta_{i-1}\cdot C.l.C'|\alpha\in(\mathcal{L}\backslash\{l_1\})^*,\ \forall_{1\leq j\leq i-1}\ \beta_j\in(\mathcal{L}\backslash\{l_{j+1}\})^*\}$

$\mathcal{B}_i=\{\beta_1\cdot l_1\cdot\beta_2\cdot l_2\cdot\ldots\cdot l_{i-1}\cdot\beta_i \mid \forall_{1\leq j\leq i-1}\ \beta_j\in(\mathcal{L}\backslash\{l_j\})^*,\ \beta_i\in(\mathcal{L}\backslash\{l_i\})^\infty\}$

$\mathcal{C}_i=\{\beta_1\cdot l_1\cdot\beta_2\cdot l_2\cdot\ldots\cdot\beta_i\cdot l_i\cdot\alpha \mid \forall_{1\leq j\leq i}\ \beta_j\in(\mathcal{L}\backslash\{l_j\})^*,\ \alpha\in(\mathcal{L}\backslash\{C.l.C'\})^\infty\}$

$$[\![(<(\mathbf{f:},l,C,C',\Rightarrow(l_1,l_2,\ldots,l_n),\lambda)>,\boldsymbol{nop})]\!]^{vc}=(\textstyle\bigcup_{1\leq i\leq n}\mathcal{A}_i)\cup(\bigcup_{1\leq i\leq n}\mathcal{B}_i)\cup(\mathcal{C})$$

where

$\mathcal{A}_i=\{\beta_1\cdot l_1\cdot\beta_2\cdot l_2\cdot\ldots\cdot l_{i-1}\cdot\beta_i\cdot C.l.C' \mid \forall_{1\leq j\leq i}\ \beta_j\in(\mathcal{L}\backslash\{l_j\})^*\}$

$\mathcal{B}_i=\{\beta_1\cdot l_1\cdot\beta_2\cdot l_2\cdot\ldots\cdot l_{i-1}\cdot\beta_i \mid \forall_{1\leq j\leq i}\ \beta_j\in(\mathcal{L}\backslash(\{l_j\}\cup\mathfrak{F}(succ(\omega^{h,h}))))^*,\ \omega^{h,h}=<(\mathbf{f:},l,C,C',\Rightarrow(l_1,l_2,\ldots,l_n),\lambda)>\}$

$\mathcal{C}=\{\beta_1\cdot l_1\cdot\beta_2\cdot l_2\cdot\ldots\cdot\beta_n\cdot l_n\cdot\alpha \mid \beta_i\in(\mathcal{L}\backslash\{l_i\})^*\ 1\leq i\leq n,\ \alpha\in(\mathcal{L}\backslash(\{C.l.C'\}\cup\mathfrak{F}(succ(\omega^{h,h}))))^*,\ \omega^{h,h}=<(\mathbf{f:},l,C,C',\Rightarrow(l_1,l_2,\ldots,l_n),\lambda)>\}$

$$[\![(<(\mathbf{f:},l,C,C',\Rightarrow(l_1,\ldots,l_n),\lambda)>,\boldsymbol{nop})]\!]^{it}=\{\beta_1\cdot l_1\cdot\beta_2\cdot l_2\cdot\ldots\cdot\beta_n\cdot l_n\cdot\gamma\cdot C.l.C' \mid \beta_i\in(\mathcal{L}\backslash(\{l_i\}\cup\mathfrak{F}(succ(\omega^{h,h}))))^*,\ 1\leq i\leq n,\ \gamma\in(\mathcal{L}\backslash\ (\{C.l.C'\}\cup\mathfrak{F}(succ(m))))^*,\ \omega^{h,h}=<(\mathbf{f:},l,C,C',\Rightarrow(l_1,\ldots,l_n),\lambda)>\}$$

7.2.5. *Regular, Required, and Fail messages with past unwanted chain constraints*

$$[\![(<(\mathbf{e:},l,C,C',\not\Rightarrow(l_1,l_2,\ldots,l_n),\lambda)>,\boldsymbol{nop})]\!]^{vc}=\textstyle\bigcup_{1\leq i\leq n}\mathcal{VC}_i$$

where

$\mathcal{VC}_i=\{\alpha\cdot l_1\cdot\beta_1\cdot l_2\beta_2\cdot\ldots\cdot l_{i-1}\cdot\beta_{i-1}\cdot C.l.C'|\alpha\in(\mathcal{L}\backslash\{l_1\})^*,\forall_{1\leq j\leq i-1}\ \beta_j\in(\mathcal{L}\backslash\{l_{j+1}\})^*\}$

$$[\![(<(\mathbf{e:},l,C,C',\not\Rightarrow(l_1,\ldots,l_n),\lambda)>,\boldsymbol{nop})]\!]^{it}=\varnothing$$

$$[\![(<(\mathbf{r:},l,C,C',\not\Rightarrow(l_1,l_2,\ldots,l_n),\lambda)>,\boldsymbol{nop})]\!]^{vc}=\textstyle\bigcup_{1\leq i\leq n}\mathcal{VC}_i$$

where

$\mathcal{VC}_i=\{\beta_1\cdot l_1\cdot\beta_2\cdot l_2\beta_3\cdot\ldots\cdot l_{i-1}\cdot\beta_i\cdot C.l.C' \mid \forall_{1\leq j\leq i}\ \beta_j\in(\mathcal{L}\backslash(\{l_j\cup\{C.l.C'\}\})\})^*\}$

$$[\![(<(\mathbf{r:},l,C,C',\not\Rightarrow(l_1,\ldots,l_n),\lambda)>,\boldsymbol{nop})]\!]^{it}=(\textstyle\bigcup_{0\leq i\leq n}\mathcal{A}_i)$$

$\cup\ \mathcal{B}$ where

$\mathcal{A}_i=\{\beta_1\cdot l_1\cdot\beta_2\cdot l_2\cdot\ldots\cdot l_{i-1}\cdot\beta_i \mid \forall_{1\leq j\leq i-1}\ \beta_j\in(\mathcal{L}\backslash(\{l_j\}\cup\{C.l.C'\}))^*,$
$\beta_i\in(\mathcal{L}\backslash(\{l_j\}\cup\{C.l.C'\}))^\infty\}$ $\mathcal{B}=\{\beta_1\cdot l_1\cdot\beta_2\cdot l_2\cdot\ldots\cdot\beta_n\cdot l_n \mid \beta_i\in(\mathcal{L}\backslash(\{l_i\}\cup\{C.l.C'\}))^*,\ 1\leq i\leq n\}$

$[\![(<(\mathbf{f:}, l, C, C', \not\Rightarrow(l_1, l_2, \ldots, l_n), \lambda)>, \boldsymbol{nop})]\!]^{vc}=\{\beta_1 \cdot l_1 \cdot \beta_2 \cdot l_2 \beta_3 \cdot \ldots \cdot \beta_n \cdot l_n \mid \beta_i \in (\mathcal{L} \backslash \{l_i\})^*,\ 1 \leq i \leq n\}$
$[\![(<(\mathbf{f:}, l, C, C', \not\Rightarrow(l_1, \ldots, l_n), \lambda)>, \boldsymbol{nop})]\!]^{it}=(\bigcup_{1 \leq i \leq n} \mathcal{A}_i)$ where $\mathcal{A}_i=\{\beta_1 \cdot l_1 \cdot \beta_2 \cdot l_2 \cdot \ldots \cdot l_{i-1} \cdot \beta_i \cdot C.l.C' \mid \forall_{1 \leq j \leq i}\ \beta_j \in (\mathcal{L} \backslash \{l_j\})^*\}$

7.2.6. *Regular and Required messages with future wanted chain constraints*

$[\![(<(\mathbf{e:}, l, C, C', \lambda, \Rightarrow(l_1, l_2, \ldots, l_n))>, \boldsymbol{nop})]\!]^{vc}=\{\alpha \cdot C.l.C' \cdot \beta_1 \cdot l_1 \cdot \beta_2 \cdot l_2 \cdot \ldots \cdot \beta_n \cdot l_n \mid \beta_i \in (\mathcal{L} \backslash \{l_i\})^*,\ 1 \leq i \leq n,\ \alpha \in \mathcal{L}^*\}$
$[\![(<(\mathbf{e:}, l, C, C', \lambda, \Rightarrow(l_1, \ldots, l_n))>, \boldsymbol{nop})]\!]^{it}=\varnothing$

$[\![(<(\mathbf{r:}, l, C, C', \lambda, \Rightarrow(l_1, l_2, \ldots, l_n))>, \boldsymbol{nop})]\!]^{vc}=\{\alpha \cdot C.l.C' \cdot \beta_1 \cdot l_1 \cdot \beta_2 \cdot l_2 \cdot \ldots \cdot \beta_n \cdot l_n \mid \beta_i \in (\mathcal{L} \backslash \{l_i\})^*,\ 1 \leq i \leq n,\ \alpha \in (\mathcal{L} \backslash\ C.l.C')^*\}$
$[\![(<(\mathbf{r:}, l, C, C', \lambda, \Rightarrow(l_1, \ldots, l_n))>, \boldsymbol{nop})]\!]^{it}=\bigcup_{1 \leq i \leq n} \mathcal{IT}_i \cup \{\alpha \mid \alpha \in (\mathcal{L} \backslash \{C.l.C'\})^\infty\}$ where $\mathcal{IT}_i=\{\alpha \cdot C.l.C' \cdot l_1 \cdot \beta_1 \cdot l_2 \beta_2 \cdot \ldots \cdot l_i \cdot \beta_i \mid \alpha \in (\mathcal{L} \backslash \{C.l.C'\})^*,\ \forall_{1 \leq j \leq i-1}\ \beta_j \in (\mathcal{L} \backslash \{l_j\})^*,\ \beta_i \in (\mathcal{L} \backslash \{l_i\})^\infty\}$

7.2.7. *Regular and Required messages with future unwanted chain constraints*

$[\![(<(\mathbf{e:}, l, C, C', \lambda, \not\Rightarrow(l_1, l_2, \ldots, l_n))>, \boldsymbol{nop})]\!]^{vc}=\bigcup_{0 \leq i \leq n} \mathcal{VC}_i$ where $\mathcal{VC}_i=\{\alpha \cdot C.l.C' \cdot \beta_1 \cdot l_1 \beta_2 \cdot l_2 \cdot \ldots \cdot l_{i-1} \cdot \beta_i \mid \alpha \in \mathcal{L}^*,\ \forall_{1 \leq j \leq i} \beta_j \in (\mathcal{L} \backslash (\{l_j\} \cup \mathfrak{F}(succ(\omega^{h,h}))))^*,\ \omega^{h,h}=<(\mathbf{e:}, l, C, C', \lambda, \not\Rightarrow(l_1, l_2, \ldots, l_n))>\}$
$[\![(<(\mathbf{e:}, l, C, C', \lambda, \not\Rightarrow(l_1, \ldots, l_n))>, \boldsymbol{nop})]\!]^{it}=\varnothing$

$[\![(<(\mathbf{r:},l,C,C',\lambda,\not\Rightarrow(l_1,l_2,\ldots,l_n))>,\boldsymbol{nop})]\!]^{vc}=\bigcup_{0\le i\le n}\mathcal{VC}_i$ where $\mathcal{VC}_i=\{\alpha\cdot C.l.C'\cdot\beta_1\cdot l_1\beta_2\cdot l_2\ \cdot\ldots\cdot l_{i-1}\cdot\beta_i \mid \alpha\in(\mathcal{L}\backslash\{C.l.C'\})^*,$ $\forall_{1\le j\le i}\ \beta_j\in(\mathcal{L}\backslash(\{l_j\}\cup\mathfrak{F}(succ(\omega^{h,h}))))^*,$ $\omega^{h,h}=<(\mathbf{r:},l,C,C',\lambda,\not\Rightarrow(l_1,l_2,\ldots,l_n))>\}$
$[\![(<(\mathbf{r:},l,C,C',\lambda,\not\Rightarrow(l_1,\ldots,l_n))>,\boldsymbol{nop})]\!]^{it}=\{\alpha\mid\alpha\in(\mathcal{L}\backslash\{C.l.C'\})^{\infty}\}\cup$ $\{\alpha\cdot C.l.C'\cdot\beta_1\cdot l_1\beta_2\cdot l_2\ \cdot\ldots\cdot\beta_n\cdot l_n \mid \alpha\in(\mathcal{L}\backslash\{C.l.C'\})^*,\ \beta_i\in(\mathcal{L}\backslash(\{l_i\}\cup$ $\mathfrak{F}(succ(\omega^{h,h}))))^*,\ 1\le i\le n,\ m=(\mathbf{r:},l,C,C',\lambda,\not\Rightarrow(l_1,\ldots l_n))\}$

7.2.8. *Regular, Required, and Fail messages with strict operator*

$[\![(<(\mathbf{e:},l,C,C',\lambda,\lambda)>,\boldsymbol{strict}(\omega^{h,h}))]\!]^{vc}=\{C.l.C'\}$
$[\![(<(\mathbf{e:},l,C,C',\lambda,\lambda)>,\boldsymbol{strict}(\omega^{h,h}))]\!]^{it}=\varnothing$ where $\omega^{h,h}=<(\mathbf{e:},l,C,C',\lambda,\lambda)>$

$[\![(<(\mathbf{r:},l,C,C',\lambda,\lambda)>,\boldsymbol{strict}(\omega^{h,h}))]\!]^{vc}=\{C.l.C'\}$
$[\![(<(\mathbf{r:},l,C,C',\lambda,\lambda)>,\boldsymbol{strict}(\omega^{h,h}))]\!]^{it}=\{\alpha\mid\alpha\in\mathcal{L}\backslash\{C.l.C'\}\}$ where $\omega^{h,h}=<(\mathbf{r:},l,C,C',\lambda,\lambda)>$

$[\![(<(\mathbf{f:},l,C,C',\lambda,\lambda)>,\boldsymbol{strict}(\omega^{h,h}))]\!]^{vc}=\{\alpha\mid\alpha\in\mathcal{L}\backslash\{C.l.C'\}\}$
$[\![(<(\mathbf{f:},l,C,C',\lambda,\lambda)>,\boldsymbol{strict}(\omega^{h,h}))]\!]^{it}=\{C.l.C'\}$ where $\omega^{h,h}=<(\mathbf{f:},l,C,C',\lambda,\lambda)>$

7.2.9. *Regular and Required messages with future constraints and strict operator*

$[\![(<(\mathbf{e:},l,C,C',\lambda,\bullet(b))>,\boldsymbol{strict}(\omega^{h,h}))]\!]^{vc}=\{C.l.C'\cdot\beta\mid\beta\in(\mathcal{L}\setminus(b\cup\mathfrak{F}(succ(\omega^{h,h}))))^*,$ $\omega^{h,h}=<(\mathbf{e:},l,C,C',\lambda,\bullet(b))>\}$
$[\![(<(\mathbf{e:},l,C,C',\lambda,\bullet(b))>,\boldsymbol{strict}(\omega^{h,h}))]\!]^{it}=\varnothing$ where $\omega^{h,h}=<(\mathbf{e:},l,C,C',\lambda,\bullet(b))>$

$$[\![(<(\mathbf{r{:}},l,C,C',\lambda,\bullet(b))>,\boldsymbol{strict}(\omega^{h,h}))]\!]^{vc}=\{C.l.C'\cdot\beta|\beta\in(\mathcal{L}\setminus(b\cup\mathfrak{F}(succ(\omega^{h,h}))))^{*},\ \omega^{h,h}=<(\mathbf{r{:}},l,C,C',\lambda,\bullet(b))>\}$$

$$[\![(<(\mathbf{r{:}},l,C,C',\lambda,\bullet(b))>,\boldsymbol{strict}(\omega^{h,h}))]\!]^{it}=\{\alpha|\alpha\in\mathcal{L}\backslash\{C.l.C'\}\}\cup \{C.l.C'\cdot\beta\cdot l' \mid \beta\in(\mathcal{L}\setminus b\cup\mathfrak{F}(succ(\omega^{h,h})))^{*},\ \omega^{h,h}=<(\mathbf{r{:}},l,C,C',\lambda,\bullet(b))>,\ l'\in b\}$$

where

$\omega^{h,h}=<(\mathbf{r{:}},l,C,C',\lambda,\bullet(b))>$

7.2.10. *Regular and Required messages with future wanted chain constraints and strict operator*

$$[\![(<(\mathbf{e{:}},l,C,C',\lambda,\Rightarrow(l_1,l_2,\ldots,l_n))>,\boldsymbol{nop})]\!]^{vc}=\{C.l.C'\cdot\beta_1\cdot l_1\cdot\beta_2\cdot l_2\cdot\ldots\cdot\beta_n\cdot l_n \mid \beta_i\in(\mathcal{L}\backslash\{l_i\})^{*},\ 1\leq i\leq n\}$$

$$[\![(<(\mathbf{e{:}},l,C,C',\lambda,\Rightarrow(l_1,\ldots,l_n))>,\boldsymbol{nop})]\!]^{it}=\varnothing$$

$$[\![(<(\mathbf{r{:}},l,C,C',\lambda,\Rightarrow(l_1,l_2,\ldots,l_n))>,\boldsymbol{nop})]\!]^{vc}=\{C.l.C'\cdot\beta_1\cdot l_1\cdot\beta_2\cdot l_2\cdot\ldots\cdot\beta_n\cdot l_n \mid \beta_i\in(\mathcal{L}\backslash\{l_i\})^{*},\ 1\leq i\leq n\}$$

$$[\![(<(\mathbf{r{:}},l,C,C',\lambda,\Rightarrow(l_1,\ldots,l_n))>,\boldsymbol{nop})]\!]^{it}=\bigcup_{1\leq i\leq n}\mathcal{IT}_i\cup\{\alpha|\alpha\in(\mathcal{L}\backslash\{C.l.C'\})\}$$

where

$\mathcal{IT}_i=\{C.l.C'\cdot l_1\cdot\beta_1\cdot l_2\beta_2\cdot\ldots\cdot l_i\cdot\beta_i|\ \forall_{1\leq j\leq i-1}\ \beta_j\in(\mathcal{L}\backslash\{l_j\})^{*},\ \beta_i\in(\mathcal{L}\backslash\{l_i\})^{\infty}\}$

7.2.11. *Regular and Required messages with future unwanted chain constraints and strict operator*

$$[\![(<(\mathbf{e{:}},l,C,C',\lambda,\not\Rightarrow(l_1,l_2,\ldots,l_n))>,\boldsymbol{nop})]\!]^{vc}=\bigcup_{0\leq i\leq n}\mathcal{VC}_i$$

where

$\mathcal{VC}_i=\{C.l.C'\cdot\beta_1\cdot l_1\beta_2\cdot l_2\cdot\ldots\cdot l_{i-1}\cdot\beta_i \mid \forall_{1\leq j\leq i}\ \beta_j\in(\mathcal{L}\backslash(\{l_j\}\cup\mathfrak{F}(succ(\omega^{h,h}))))^{*},\ \omega^{h,h}=<(\mathbf{e{:}},l,C,C',\lambda,\not\Rightarrow(l_1,l_2,\ldots,l_n))>\}$

$$[\![(<(\mathbf{e{:}},l,C,C',\lambda,\not\Rightarrow(l_1,\ldots,l_n))>,\boldsymbol{nop})]\!]^{it}=\varnothing$$

$[\![(<(\mathbf{r:}, l, C, C', \lambda, \not\Rightarrow(l_1, l_2, \ldots, l_n))>, \boldsymbol{nop})]\!]^{vc} = \bigcup_{0\leq i\leq n} \mathcal{VC}_i$ where $\mathcal{VC}_i = \{C.l.C' \cdot \beta_1 \cdot l_1 \beta_2 \cdot l_2 \cdot \ldots \cdot l_{i-1} \cdot \beta_i \mid \forall_{1\leq j\leq i}\ \beta_j \in (\mathcal{L}\setminus(\{l_j\} \cup \mathfrak{F}(succ(\omega^{h,h}))))^*,$ $\omega^{h,h} = <(\mathbf{r:}, l, C, C', \lambda, \not\Rightarrow(l_1, l_2, \ldots, l_n))>\}$
$[\![(<(\mathbf{r:}, l, C, C', \lambda, \not\Rightarrow(l_1, \ldots, l_n))>, \boldsymbol{nop})]\!]^{it} = \{\alpha \mid \alpha \in (\mathcal{L}\setminus\{C.l.C'\})\} \cup$ $\{C.l.C' \cdot \beta_1 \cdot l_1 \beta_2 \cdot l_2 \cdot \ldots \cdot \beta_n \cdot l_n \mid$ $\beta_i \in (\mathcal{L}\setminus(\{l_i\} \cup \mathfrak{F}(succ(\omega^{h,h}))))^*,\ 1\leq i\leq n,$ $m = (\mathbf{r:}, l, C, C', \lambda, \not\Rightarrow(l_1, \ldots l_n))\}$

Hereafter, abusing notation, we assume that for all operator $o \in O$ and for all the sub-linearization $\omega^{h,k}$ both the languages of valid continuations $[\![(\omega^{h,k}, o)]\!]^{vc}$ and invalid traces $[\![(\omega^{h,k}, o)]\!]^{it}$ are empty if $h<k$.

7.3. **nop** SEQUENTIAL MESSAGES SEMANTICS

$[\![(\omega^{i,k}, \boldsymbol{nop})]\!]^{vc} = [\![(\omega^{i,i}, \boldsymbol{nop})]\!]^{vc} \cdot [\![(\omega^{i+1,i+1}, \boldsymbol{nop})]\!]^{vc} \cdot \ldots \cdot [\![(\omega^{k,k}, \boldsymbol{nop})]\!]^{vc}$
$[\![(\omega^{i,k}, \boldsymbol{nop})]\!]^{it} = \bigcup_{i\leq j\leq k} ([\![(\omega^{i,j-1}, \boldsymbol{nop})]\!]^{vc} \cdot [\![(\omega^{j,j}, \boldsymbol{nop})]\!]^{it})$

7.4. OPERATORS SEMANTICS

7.4.1. *Parallel operator*

By referring to the description of the function *parallel2BA()* in Section 6.2.2.2, we recall that for the parallel operator $\boldsymbol{par}(\omega^{h,k}, \mathtt{sp}(h,k))$ with $\mathtt{sp}(h,k) = \{(h,q), (q+1,z), \ldots, (t+1,k)\}$, we need to generate new linearizations, called *"parallel" linearizations*, $pl_1 = \omega^{h^1,k^1}$, $pl_2 = \omega^{h^2,k^2}$, ..., $pl_m = \omega^{h^m,k^m}$ that are all the possible interleaving of $\omega^{h,q}$, $\omega^{q+1,z}$, ..., $\omega^{t+1,k}$.

$[\![(\omega^{h,k}, \boldsymbol{par}(\omega^{h,k}, \mathtt{sp}(h,k)))]\!]^{vc} = \bigcup_{1\leq i\leq m} [\![(pl_i, \boldsymbol{nop})]\!]^{vc}$
$[\![(\omega^{h,k}, \boldsymbol{par}(\omega^{h,k}, \mathtt{sp}(h,k)))]\!]^{it} = \bigcup_{1\leq i\leq m} [\![(pl_i, \boldsymbol{nop})]\!]^{it}$,   where pl_i is the *i-th* parallel linearization.

7.4.2. *Loop operator*

By referring to the description of the function *loop2BA()* in Section 6.2.2.2, we recall that this operator **loop**($\omega^{h,k}$, l,u) allows its operand $\omega^{h,k}$ to be repeated a given number of times (at least l and at most u). Thus, it generates a set of *loop linearizations* $(\omega^{h,k})^l$, $(\omega^{h,k})^{l+1}$, ..., $(\omega^{h,k})^u$ that are concatenations of $\omega^{h,k}$ with itself from l to u times. For example, loop$(\omega^{2,3}, 2, 4)$ generates the linearizations: $<m_2 \cdot m_3 \cdot m_2 \cdot m_3>$, $<m_2 \cdot m_3 \cdot m_2 \cdot m_3 \cdot m_2 \cdot m_3>$, and $<m_2 \cdot m_3 \cdot m_2 \cdot m_3 \cdot m_2 \cdot m_3 \cdot m_2 \cdot m_3>$. Formally:

$[\![(\omega^{h,k},\boldsymbol{loop}(\omega^{h,k}, \mathrm{l,u}))]\!]^{vc} = \bigcup_{l \le j \le u} [\![((\omega^{h,k})^j, \boldsymbol{nop})]\!]^{vc}$
$[\![(\omega^{h,k},\boldsymbol{loop}(\omega^{h,k}, \mathrm{l,u}))]\!]^{it} = \bigcup_{l \le j \le u} [\![((\omega^{h,k})^j, \boldsymbol{nop})]\!]^{it}$

7.4.3. *Alternative operator*

By referring to the description of the function *alt2BA()* in Section 6.2.2.2, we recall that for the alternative operator **alt**($\omega^{h,k}$, $\mathtt{sp}(h,k)$) with $\mathtt{sp}(h,k)$= $\{(h,q),(q+1,z),\dots,(t+1,k)\}$, we split $\omega^{h,k}$ into the *"alternative" linearizations* al_1=$\omega^{h,q}$, al_2=$\omega^{q+1,z}$, ..., al_m=$\omega^{t+1,k}$ that can be alternatively chosen.

$[\![(\omega^{h,k},\boldsymbol{alt}(\omega^{h,k}, \mathtt{sp}(h,k)))]\!]^{vc} = \bigcup_{1 \le j \le m} [\![(al_j,\ \boldsymbol{nop})]\!]^{vc}$
$[\![(\omega^{h,k},\boldsymbol{alt}(\omega^{h,k}, \mathtt{sp}(h,k)))]\!]^{it} = \bigcup_{1 \le j \le m} [\![(al_j,\ \boldsymbol{nop})]\!]^{it}$

7.5. PSC semantics

Let SL=$\{\omega^{1,r}, \omega^{r+1,s}, \dots, \omega^{h+1,j}, \omega^{j+1,|T|-1}\}$ be the set of sub-linearizations and let sl_i denotes the *i-th* sub-linearization of the sequence $\omega^{1,r}$, $\omega^{r+1,s}, \dots, \omega^{h+1,j}, \omega^{j+1,|T|-1}$, the semantics of the psc = (L, I, T, $\prec$, $arrowMSGs$, $t2m$, SLO) is:

$[\![psc]\!]^{it} = \bigcup_{1 \le i \le \lvert SLO \rvert} [\![(sl_1, o_1)]\!]^{vc} \cdot [\![(sl_2, o_2)]\!]^{vc} \cdot \ldots \cdot [\![(sl_{i-1}, o_{i-1})]\!]^{vc} \cdot [\![(sl_i, o_i)]\!]^{it}$

7.6. Operational and denotational semantics consistency

In this section we enunciate the theorem asserting the consistency between the operational and denotational semantics and we sketch its straightforward proof. In other words, the theorem states that the two semantics are equivalent and hence a message sequence is accepted by the Büchi automaton iff it represents an invalid trace of the PSC. By recalling the definition of acceptance of a Büchi automaton $\mathcal{B}$ in Section 6.1, we say that the language accepted by $\mathcal{B}$, denoted as $L(\mathcal{B})$, consists of all traces accepted by $\mathcal{B}$.

Theorem 1 (operational and denotational semantics equivalence)

Let $\boldsymbol{psc} = (L, I, T, \prec, arrowMSGs, t2m, SLO)$ be a PSC and let $\mathcal{B}$ be the Büchi automaton associated to it derived by the algorithm Psc2Ba presented in Section 6.2.3, then $[\![\boldsymbol{psc}]\!]^{it}$=$L(\mathcal{B})$.

Proof: The proof is organized in three steps:

(i) the first step concerns the proof of equivalence for single *arrowMSGs* possibly subjected to a constraint (either unwanted messages constraint or chain constraint) and/or a strict operator. For this point we refer to the *fundamental translation rules* of the operational semantics (in Section 6.2.1) and to the *single message denotational semantics* (in Section 7.2). This part of the proof is trivial and achieved by construction. In fact the language of invalid traces associated to each single *arrowMSG* m (subjected to the ***nop*** operator) by the denotational semantics is exactly the language of traces accepted by the corresponding Büchi automaton obtained by the *fundamental translation rule* for m. That is, considering the basic Büchi automaton b for m in Section 6.2.1 (by taking into account the definition of acceptance given in Section 6.1) it is straightforward to recognize the equivalence of $L(b)$ and the language of invalid traces $[\![(<m>, \boldsymbol{nop})]\!]^{it}$ in Section 7.2.

(ii) the second step concerns the proofs of the equivalence for *sub-linearizations subjected to operators* (parallel, alternative or loop). For this point we refer to the *PSC operators translation* meta-rules of the operational semantics (in Section 6.2.2.2) and to the denotational *operators semantics* (in Sections 7.4). Both the semantics of sub-linearizations subjected to operators are given by means of ad-hoc *"additional" sub-linearizations* (namely "parallel" linearizations, "alternative" linearizations, and "loop" linearizations). We

recall that these additional sub-linearizations are subjected to no operators.

For sub-linearizations subjected to operators, the operational semantics makes use of the functions *parallel2BA()*, *loop2BA()* and *alt2BA()*. By means of the subroutine *subl2Büchi()* (Section 6.2.2.2), these functions firstly construct a dedicated Büchi automaton for each additional sub-linearization. Then, by means of the subroutine *mergeIFA()* (Section 6.2.2.2) they merge the initial, final and sink accepting states of all the dedicated automata, and a single Büchi automaton is given as output. This single automaton accepts the union of the languages accepted by the dedicated automata (see Figure 14).

In the same way, the denotational semantics constructs the languages of invalid traces for sub-linearizations subjected to operators (see the *operators semantics* in Section 7.4) making the union of the languages of invalid traces derived from the additional sub-linearizations associated to the ***nop*** operator (see the ***nop*** *sequential messages semantics* in Section 7.3).

That is, to prove the equivalence of the operational and denotational semantics for sub-linearizations subjected to operators, it is sufficient to prove the equivalence of the two semantics for additional sub-linearizations. In order to do this, we recall that, by means of the subroutine *subl2Büchi()*, the operational semantics works in a compositional way by concatenating the basic Büchi automata derived for each single $arrowMSG$ m_i in the sub-linearization $<m_i \cdot m_{i+1} \cdot \ldots \cdot m_j>$ given as input to the subroutine. This simple composition process makes use of the *final states* that indicate the valid continuations of the Büchi automata (see the function *compose()* in Section 6.2.2.1). The language accepted by the composed automaton (and hence the language of invalid traces) is the union of the languages accepted by the ("sub-") composed automata for $<m_i>$, $<m_i \cdot m_{i+1}>$, $\ldots$, $<m_i \cdot m_{i+1} \cdot \ldots \cdot m_j>$.

The proof is then straightforward since the denotational semantics works similarly. It calculates the invalid traces (i.e., $[\![(\omega^{i,k}, \boldsymbol{nop})]\!]^{it}$ in Section 7.3) by making use of the valid continuation semantics (i.e., $[\![(\omega^{i,k}, \boldsymbol{nop})]\!]^{vc}$ in Section 7.3). Also for this semantics the language of invalid traces is the union of the invalid traces of $<m_i>$, $<m_i \cdot m_{i+1}>$, $\ldots$, $<m_i \cdot m_{i+1} \cdot \ldots \cdot m_j>$.

(iii) the last step concerns the proof of the equivalence among the overall composition mechanisms of the two semantics. For this proof we refer to the PSC2BA *algorithm* (implemented by the function

PSC2BA() in Section 6.2.3), for the operational semantics, and to the PSC *denotational semantics* ($[\![psc]\!]^{it}$ in Section 7.5). Both the semantics consider the set of ordered pairs of sub-linearizations in SL and operators in O (i.e., SLO). Similarly to what we have discussed in point *(ii)*, the composition is carried on by means of the valid continuations (and hence final states) and the operational semantics simply composes the automata obtained from each pair (see the invocations of the function *compose()* within the function *PSC2BA()*). Thus, the language accepted by the composed automaton (and hence the set of invalid traces) is the union of the languages accepted by the automata obtained for the single pair (sl_1, o_1), for the two subsequential pairs (sl_1, o_1) (sl_2, o_2), for the three subsequential pairs (sl_1, o_1) (sl_2, o_2) (sl_3, o_3), etc.

To complete the proof is sufficient to consider that the denotational semantics simply performs the union of all the invalid trace languages $[\![(sl_1, o_1)]\!]^{it}$, $[\![(sl_1, o_1)]\!]^{vc} \cdot [\![(sl_2, o_2)]\!]^{it}$, $[\![(sl_1, o_1)]\!]^{vc} \cdot [\![(sl_2, o_2)]\!]^{vc} \cdot [\![(sl_3, o_3)]\!]^{it}$, etc.

■

8. Evaluation

As already discussed, since scenario specifications are less informative with respect to LTL formulae, the set of properties that can be specified in this way is just a subset of LTL properties. However, this does not appear to be a significant restriction since the subset of specifiable properties, as confirmed by several case studies we have considered so far, appears sufficiently expressive for a software designer.

More precisely, Psc2Ba can graphically express a useful set of both liveness and safety properties:

Liveness: by means of required messages we are able to express that a message is mandatory.

Safety: by means of fail messages we can express that a message should not happen. By means of constraints we can raise an error when a message in a constraint happens before the message containing the constraint.

In order to better validate the expressivity of the PSC language we refer to the specification patterns system introduced by the Kansas State University (Dwyer et al., 1999). Dwyer et al. define a repository with the intent of collecting patterns that commonly occur in the specification of concurrent and reactive systems. The patterns are defined for various logics and specification formalisms and we refer to the mappings for property patterns in LTL. A specification pattern has a *scope* that defines the range in which the pattern must hold; for example while *global* means that the pattern must hold everywhere, *between q and r* means that the pattern must hold from the first occurrence of q to the first occurrence of r only and only if r happens.

We are able to represent in PSC all the defined patterns (PSC Project, 2005). Since PSC is an event-based formalism in terms of exchanged messages among components, and since in event-based formalisms the underlying model does not allow two events to coincide (Dwyer et al., 1999), we disallow the specification of simultaneous events.

Similarly to what done in the specification patterns system, in PSC a scope can be represented once and instantiated for each pattern. See the PSC web page (PSC Project, 2005) for a description of all patterns.

In Figure 16 we report two examples, a *Precedence Chain 1 cause-2 effects* (p precedes s and t) and a particular instance of the *Precedence Chain 2 causes-1 effect*. Considering a *Between q and r* scope a Precedence Chain states that after q, an error arises if r is exchanged and, between q and r, s and t are exchanged without having p before.

Within the scope *After q*, *Precedence Chain 2 causes-1 effect* states that after q, an error arises if p is exchanged before having the chain s and t.

The LTL formulae to describe these patterns are the following:

Precedence Chain 1 cause-2 effects:

$$G((q\&Fr)->((!(s\&(!r)\&X(!rU(t\&!r))))U(r||p)))$$

and Precedence Chain 2 causes-1 effect:

$$(G!q)||(!qU(q\&Fp->(!pU(s\&!p\&X(!pUt))))$$

While the LTL formulas are not easily understandable, the same properties expressed in the PSC formalism (Figure 16) appear more intuitive and closer to their natural language description.

Focusing on the Precedence Chain 1 cause-2 effects within the between q and r scope, the "after q" notion is represented as a regular message. Now, we have an error if the scope is recognized (an occurrence of r) and before the sequence s and t (the 2 effects) happens before having p (the cause). In this case the error condition is recognized as an error state, thus the error is represented with the fail message r. The precondition of this error is an occurrence of the sequence s, t without having before p. Thus s and t are represented as regular messages and p is an unwanted message constraint since it is something that should not happen before s (for having an error). Following the definition of

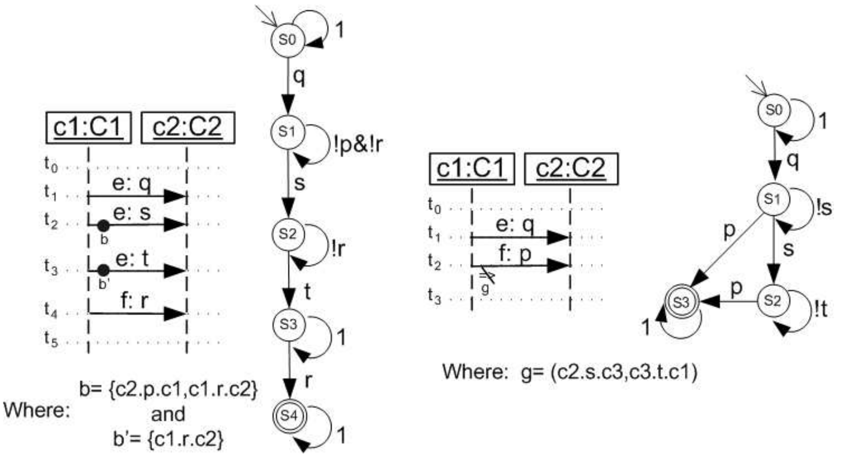

Figure 16. Left-hand side: Precedence Chain 1 cause-2 effects within the between q and r scope. Right-hand side: Precedence Chain 2 causes-1 effect within the after q scope

the between q and r scope, the precondition of the error is valid if r does not happen. Consequently, r is an unwanted message constraint of both s and t. By recalling that the Büchi automaton expresses the negation of the desired temporal property it is simple to recognize the negation of the Precedence Chain in the automaton on the left-hand side of Figure 16. Note that the PSC formula is scalable, in fact, if we want to write a 1 cause-3 effects, we have to add the third effect, z, as a new regular message z with unwanted message constraint b' before the fail message r.

In the other example, Precedence Chain 2 causes-1 effect within the after q scope, the two causes are s and t and the effect is p. The "after q" scope is represented as a regular message. There is an error if we have the effect without having the chain of causes. Thus, the error is represented as a fail message p with a past unwanted chain constraint of s and t. Note that the PSC formula is scalable, in fact, if we want to write a 3 causes-1 effect, we have to add the third cause, z, as a third element in the tuple g.

9. Case Study

In Section 8 we described the application of PSC on the specification patterns system in order to validate the expressivity of the PSC language. Contrarily, the aim of this section is to put in practice PSC in order to validate its usability in industrial projects. Thus, we report our experience in using PSC for defining properties to be checked by CHARMY in the context of a project developed by Selex Communications (refer to (Colangelo et al., 2006) for further details). Selex Communications mainly operates in a naval communication environment.

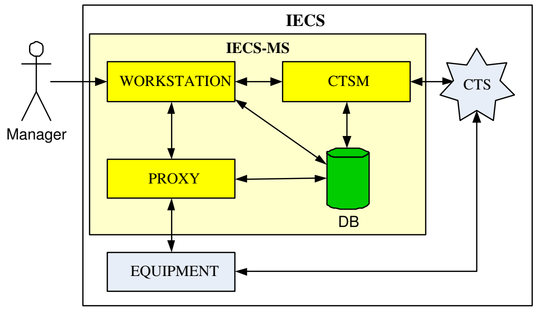

Figure 17. IECS Software Configuration

The considered system is the Integrated Environment for Communication on Ship (IECS) that provides heterogeneous services on board of the ship. The purpose of the system is to fulfil the following main functionalities: *i)* provide voice, data and video communication modes; *ii)* prepare, elaborate, memorize, recovery and distribution of operative messages; *iii)* configuration of radio frequency, variable power control and modulation for transmission and reception over radio channel; *iv)* remote control and monitoring of the system for detection of equipment failures in the transmission/reception radio chain and for the management of system elements; *v)* data distribution service; *vi)* implement communication security techniques to the required level of evaluation and certification.

The SA is composed of the IECS Management System (IECS-MS), CTS, and EQUIPMENT components as highlighted in Figure 17.

In the following we focus on the IECS-MS, the more critical component since it coordinates different heterogeneous subsystems, both software and hardware. Indeed, it controls the IECS system providing both internal and external communications. The IECS-MS complexity and heterogeneity need the definition of precise software architecture to express its coordination structure. The system involves several operational consoles that manage the heterogeneous system equipment including the ATM based Communication Transfer System (CTS) through Proxy computers. For this reason the high level design is based on a manager-agent architecture that is summarized in Figure 17, where the Workstation (WS) component represents the management entity while the PROXY and the Communication Transfer System

Manager (CTSM) components represent the interface to control the managed equipment and the CTS, respectively.

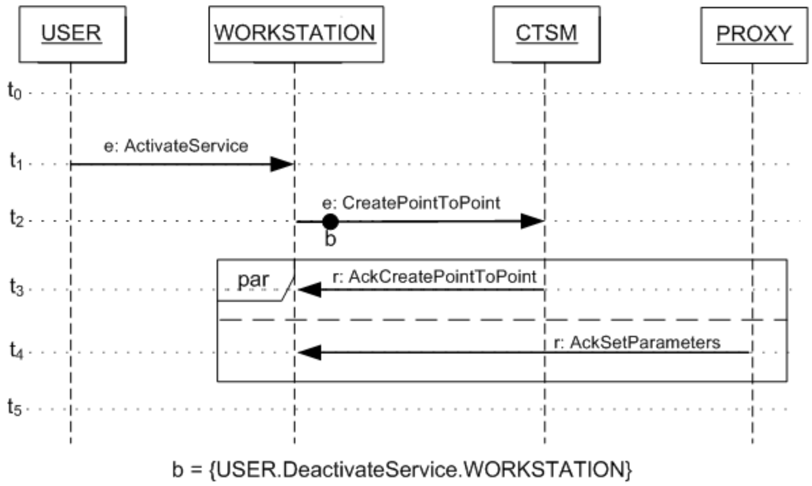

Figure 18. Property: Service Activation

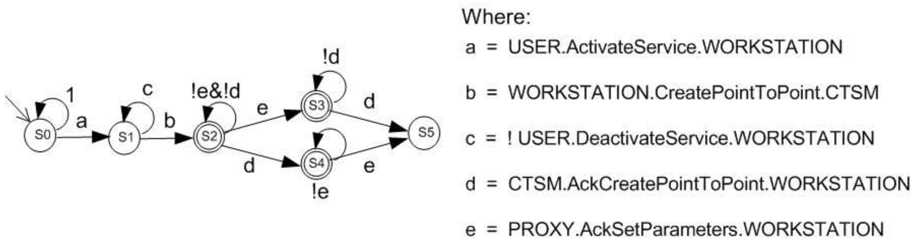

Figure 19. Büchi Automaton of the Service Activation property

The functionalities of interest of the IECS-MS are: *i)* service activation; *ii)* service deactivation; *iii)* service reconfiguration; *iv)* equipment configuration; *v)* control equipment status; *vi)* fault CTS. A *service*, in this context, denotes a unit base of planning and the implementation of a logic channel of communication through the resources of communications on the ship. All the above described functionalities are "atomics", since it is not possible to execute two different functionalities at the same time on the system.

We verified several properties on this system and in this paper we show how some of them have been modeled by using the PSC tool presented in Section 10. In Figure 18 the considered property concerns the service activation. The property has two regular messages that realize the service activation request and represent the precondition. When such a precondition is satisfied, if the USER does not deactivate the service, after the *ActivateService* request and before the *CheckCTSMStatus* message (see the past unwanted messages constraint over the second regular message that references the message $USER.DeactivateService.WORKSTATION$), the service must be activated. The activation is notified to the WORKSTATION component by the last two required messages *AckCreatePointToPoint* and *AckSetParameters*. Note that both these acknowledge messages are mandatory but the ordering between them is not significant. Hence we use the parallel operator to interleave them. Figure 19 shows the Büchi automaton corresponding to this property.

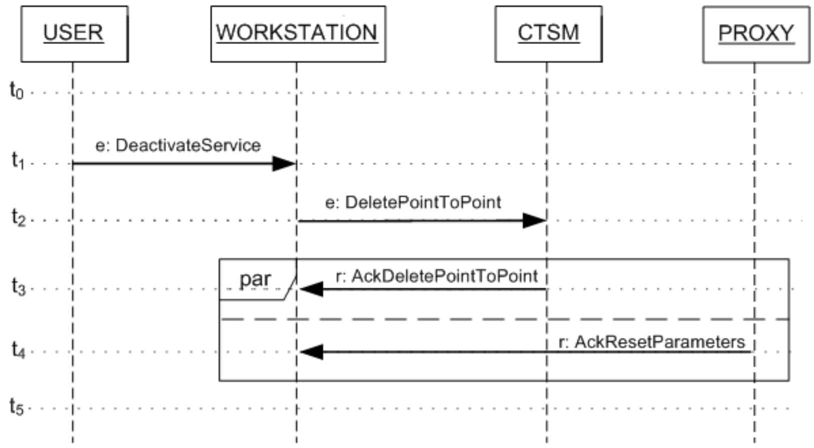

Figure 20. Property: Service Deactivation

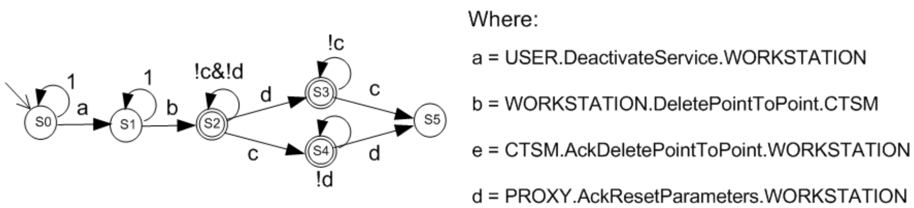

Figure 21. Büchi Automaton of the Service Deactivation property

In Figure 20 the property concerns the service deactivation. After having understood the previous property on the service activation the comprehension of the property on service deactivation is straightforward. Note that, scenarios for service activation and deactivation are asymmetric. The only difference is that, during activation it is important that there is no intervening deactivation message from the user; during the deactivation scenario it is not important to specify that there is no intervening activation message since the dedicated handler is switched off and possible activation messages are ignored. The generated Büchi automata is reported in Figure 21.

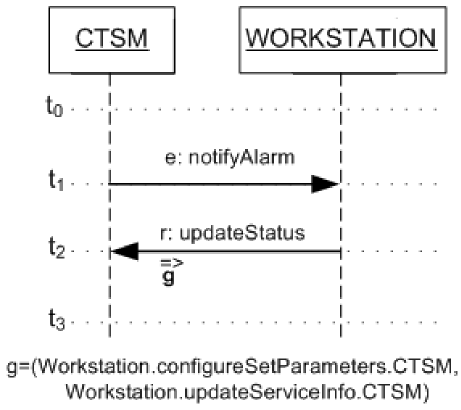

Figure 22. Property: Alarm Notification

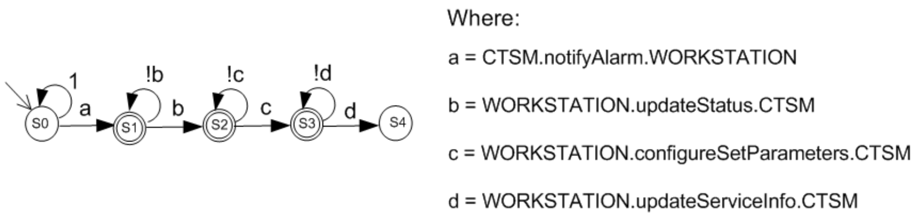

Figure 23. Büchi Automaton of the Alarm Notification property

In Figure 22 we report the last property concerning the management of an alarm notification and in Figure 23 we show its corresponding Büchi automaton. In case of an alarm must be notified we have to be sure that in each behavior of the system the triggered sequence of events responds to the alarm notification stimulus. The precondition for the sequence of response events is the $notifyAlarm$ regular message. After that, it is required to have the sequence of events *updateStatus*, $configureSetParameters$, and *updateServiceInfo* that is represented as the *updateStatus* required message with a future wanted chain constraint with the following tuple as parameter

$$g = (WORKSTATION.configureSetParameters.CTSM,$$
$$WORKSTATION.updateServiceInfo.CTSM)$$

10. Tool

The PSC2BA algorithm has been implemented as a plugin for CHARMY. The PSC2BA plugin implementation is currently available in (Charmy Project, 2004). The plugin permits to design the PSC scenarios and to produce the corresponding Büchi automata. The current implementation produces a Büchi automaton in the form of *never claim*, which is the syntactical textual representation of Büchi automata.

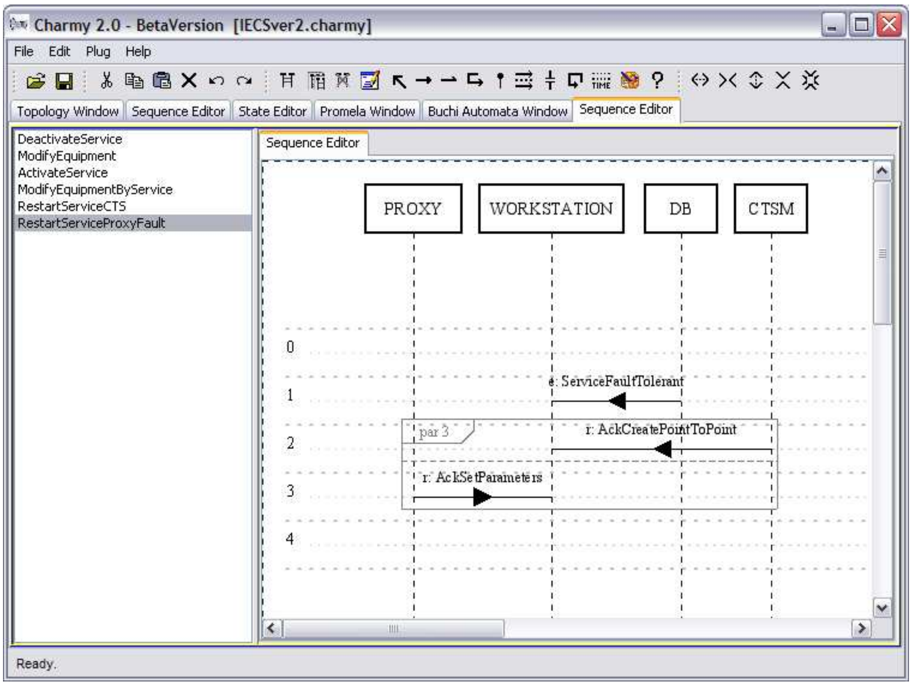

Figure 24. PSC tool: a screenshot

Figure 24 shows a screenshot of the PSC tool.

We also proposed a wizard called W_PSC (Autili and Pelliccione, 2006) that, by using a set of sentences (classified according to temporal properties main keywords), helps the user while writing PSC scenarios.

11. Conclusion and Future Work

In this paper we proposed a formalism for specifying temporal properties aimed at being simple, (sufficiently) powerful and user-friendly. After having examined Message Sequence Charts (ITU-T Recommendation Z.120., 1999), and UML 2.0 Interaction Sequence Diagrams (Object Management Group (OMG), 2004), we presented a scenario-based graphical language that is an extended notation of a selected subset of the UML 2.0 Interaction Sequence Diagrams. We called this language *Property Sequence Chart* (PSC).

Within PSC a property is seen as a relation on a set of exchanged system messages, with zero or more constraints. More precisely, our language is used to describe both positive scenarios (i.e., the "desired" ones) and negative scenarios (i.e., the "unwanted" ones) for describing interactions among the components of a system. PSC can graphically express a useful set of both liveness and safety properties.

We validated the expressiveness of our formalism with respect to the set of the *property specification patterns* (Dwyer et al., 1999). We showed that with our PSC language it is possible to represent all these patterns. We also provided PSC with both denotational and operational semantics. The operational semantics is obtained via an algorithm, called Psc2Ba to translate our visual language specification into Büchi automata thus providing a precise semantics. The algorithm has been implemented as a plugin of our tool Charmy (Charmy Project, 2004) that is a framework for software architecture design and validation with respect to temporal properties.

As future work we plan to introduce timing constraints in order to be able to specify properties also for real-time systems. Consequently, the algorithm Psc2Ba will be updated in order to produce timed Büchi automata.

Acknowledgments

This work is partially supported by the POPEYE project: Peer to Peer Collaborative Working Environments over Mobile Ad-Hoc Networks. POPEYE is part-funded by the EU under the 6th Framework Program, IST priority Contract No. IST-2006-034241. http://www.ist-popeye.org.

The work is also partially supported by ARTDECO (Adaptive infRasTructure for DECentralized Organizations), an Italian FIRB (Fondo per gli Investimenti della Ricerca di Base) 2005 Project.

We want to acknowledge the anonymous reviewers of this paper for the valuable contributions in providing quality and constructive comments. Furthermore, we acknowledge Massimo Tivoli for his valuable comments.

References

Alfonso, A., V. Braberman, N. Kicillof, and A. Olivero: May 2004, 'Visual Timed Event Scenarios'. In: *26th ICSE'04. Edinburgh, Scotland, UK.*

Autili, M., P. Inverardi, and P. Pelliccione: Shanghai, China, May 27, 2006., 'A Scenario Based Notation for Specifying Temporal Properties'. In: *5th International Workshop on Scenarios and State Machines: Models, Algorithms and Tools (SCESM'06).*

Autili, M. and P. Pelliccione: 2006, 'Towards a Graphical Tool for Refining User to System Requirements'. In: *5th GT-VMT'06 - ETAPS'06, to appear in ENTCS.*

Braberman, V., N. Kicillof, and A. Olivero: 2005, 'A Scenario-Matching Approach to the Description and Model Checking of Real-Time Properties'. *IEEE Transactions on Software Engineering* **31**(12), 1028–1041.

Buchi, J., R.: 1960, 'On a decision method in restricted second order arithmetic.'. In: *Proc. of the Int. Congress of Logic, Methodology and Philosophy of Science.*

C. André and M-A. Peraldi-Frati and J-P. Rigault: 2001, 'Scenario and Property Checking of Real-Time Systems Using a Synchronous Approach'. In: *4th IEEE Int. Symp. on OO Real-Time Distributed Computing.*

Charmy Project: 2004, 'Charmy web site'. http://www.di.univaq.it/charmy.

Clarke, E. M., O. Grumberg, and D. A. Peled.: 2001, *Model Checking*. The MIT Press, Massachusetts Institute of Technology.

Colangelo, D., D. Compare, P. Inverardi, and P. Pelliccione: 2006, 'Reducing Software Architecture Models Complexity: a Slicing and Abstraction Approach'. In: *FORTE 2006. September 26-29 2006, Paris, France. LNCS. 4229. pp. 243258, 2006.*

Damm, W. and D. Harel: 2001, 'LSCs: Breathing Life into Message Sequence Charts'. *Formal Methods in System Design* **19**(1), 45–80.

Dillon, L. K., G. Kutty, L. E. Moser, P. M. Melliar-Smith, and Y. S. Ramakrishna: 1994, 'A graphical interval logic for specifying concurrent systems'. *ACM TOSEM* **Vol. 3, Issue 2**.

Dwyer, M. B., G. S. Avrunin, and J. C. Corbett: 1999, 'Patterns in Property Specifications for Finite-State Verification.'. In: *ICSE*. pp. 411–420.

Gerth, R., D. Peled, M. Vardi, and P. Wolper: 1995, *Simple on-the-fly automatic verification of linear temporal logic*, pp. 3–18. Chapman and Hall.

Harel, D. and R. Marelly: 2002, 'Playing with Time: On the Specification and Execution of Time-Enriched LSCs'. *mascots* **00**, 0193.

Haugen, Ø.: 2004, 'Comparing UML 2.0 Interactions and MSC-2000.'. In: *SAM*. pp. 65–79.

Holzmann, G. J.: 2002, 'The Logic of Bugs'. *In Proc. Foundations of Software Engineering (SIGSOFT 2002/FSE-10).*

Holzmann, G. J.: Sept. 2003, *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley.

ITU-T Recommendation Z.120.: 1999, 'Message Sequence Charts'. ITU Telecom. Standardisation Sector.

Klose, J. and H. Wittke: 2001, 'An Automata Based Interpretation of Live Sequence Charts'. In: *TACAS 2001. LNCS 2031, pp. 512-527.*

Kugler, H., D. Harel, A. Pnueli, Y. Lu, and Y. Bontemps: 2005, 'Temporal Logic for Scenario-Based Specifications'. In: *11th Int. Conf. TACAS'05*. Springer-Verlag.

Lee, I. and O. Sokolsky: 1997, 'A Graphical Property Specification Language'. In: *High-Assurance Systems Engineering Workshop, Washington, DC.*

Manna, Z. and A. Pnueli: 1991, *The Temporal Logic of Reactive and Concurrent Systems*. New York: Springer-Verlag.

M.Tivoli and M.Autili: 2004, 'SYNTHESIS: a tool for synthesizing "correct" and protocol-enhanced adaptors'. *RSTI - L'OBJET JOURNAL 12/2006. WCAT04* pp. 77–103.

Object Management Group (OMG): 2004, 'UML: Superstructure version 2.0.'.

Pnueli, A.: 1977, 'The temporal logic of programs.'. In: *Proc. 18th IEEE Symposium on Foundation of Computer Science*. pp. pp. 46–57.

PSC Project: 2005, 'PSC web site'. http://www.di.univaq.it/psc2ba.

Smith, M. H., G. J. Holzmann, and K. Etessami: Aug. 2001, 'Events and constraints: a graphical editor for capturing logic properties of programs'. In: *5th International Symposium on Requirements Engineering.*

Smith, R. L., G. S. Avrunin, L. A. Clarke, and L. J. Osterweil: 2002, 'PROPEL: An Approach Supporting Property Elucidation'. In: *ICSE2002*. pp. 11–21.

Störrle, H.: 2003, 'Semantics of Interactions in UML 2.0'. In: *VLFM'03 Intl. Ws. Visual Languages and Formal Methods, at HCC'03, Auckland, NZ.*

Uchitel, S., J. Kramer, and J. Magee: 2004, *Incremental elaboration of scenario-based specifications and behavior models using implied scenarios*. ACM TOSEM, Vol. 13 , Issue 1.

Zanolin, L., C. Ghezzi, and L. Baresi: 2003, 'An Approach to Model and Validate Publish/Subscribe Architectures'. In: *SAVCBS*.