A Scenario Based Notation for Specifying Temporal Properties

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

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, 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 that, in our opinion, fixes the highlighted lacks of these notations. We propose an extended graphical notation of a subset of UML 2.0 *Interaction Sequence Diagrams*. A precise semantics of this new language, called Property Sequence Chart (PSC), is provided via translation, by means of an algorithm implemented as a plugin of our CHARMY tool, into Büchi automata. Expressiveness of PSC has been validated with respect to well known *property specification patterns*.

Categories and Subject Descriptors

D.2.1 [**Requirements/Specifications**]: Languages; D.2.4 [**Software/Program Verification**]: Model Checking; Formal methods

General Terms

Design, Verification

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 formalisms. For instance, the high level of inherent complexity of Linear-time Temporal Logic formulae (LTL) [16, 19] may induce to specify properties in a wrong way.

Properties that are simply captured within the context of interest and that are easily described by natural language may result very hard to specify in LTL. In other words, there is a substantial gap between natural language and the LTL syntax. Although in this paper we focus on formulae expressed in LTL notation, other similar formalisms (such as CTL, $ACTL$) suffer the same problems. In [22] 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".

Many 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 [8], other works propose the specification of temporal properties through graphical formalisms [21], [7], [25], [1], and [14]. The main problems of the latter approaches is in balancing the *expressive power* and the *simplicity* of the graphical property description language. GIL [7] is sufficiently expressive but lacks in user friendliness. For example, GIL achieves expressive power by allowing nesting but its formulae become potentially difficult to understand. This difficulty comes from the fact that its graphical notation is too close to temporal logic. On the contrary TimeEdt [21] (also called *Time Line Editor*) considers a more intuitive notation but it was specifically developed to capture long running on complex chains of dependent events (the specification patterns people [8] call them "chain patterns"). Moreover, even if intuitive, TimeEdt introduces a graphical language that is not close to the ones commonly used in current industrial practice.

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 (e.g., the set of the *property specification patterns* [8]) while keeping in mind that easy use and simplicity

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SCESM'06, May 27, 2006, Shanghai, China.
Copyright 2006 ACM 1-59593-085-X/06/0005 ...$5.00.

are mandatory requirements to make a formalism adopted by industries.

In this paper we aim to provide developers with a simple but expressive tool for specifying temporal properties. We focus on the graphical formalisms for scenario-based specification that are commonly and extensively used within industrial software development practice: Message Sequence Charts (MSCs) [12], and UML 2.0 Interaction Sequence Diagrams [18].

We propose a scenario-based visual language that is an extended graphical notation of a subset of the UML 2.0 *Interaction Sequence Diagrams* language [18]. We call this language *Property Sequence Chart* (PSC).

Within our 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.

It is well known that a LTL formula can be translated into a Büchi automaton [3] that is then used by model checkers [11] or component assemblers [17]. Although this representation looks more intuitive, it can be very difficult to correctly and directly represent a property into a Büchi automaton. Therefore, in order to overcome this problem and to provide PSC with a formal 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 [5] which is a framework (based on the model checker Spin [11]) for software architecture design and validation with respect to temporal properties.

We measured the expressiveness of our approach with respect to the set of *property specification patterns* proposed in [8] that captures recurring solutions to design and coding problems.

The paper is organized as follows: Section 2 analyzes MSC and UML 2.0 Interaction Diagrams in order to understand where they lack when used for expressing temporal properties. Section 3 presents PSC as a possible solution to overcome these lacks. Section 4 discusses the PSC expressive power. The PSC plugin is briefly introduced in Section 5 and related work is discussed in Section 6. Finally, we give conclusion and future work in Section 7.

2. INSPECTING MSC AND UML 2.0 INTERACTION SEQUENCE DIAGRAMS

The main goal of this work is to propose a scenario-based visual language for specifying temporal properties which balances *expressive power* and *simplicity* of use.

Several works have been proposed in these years to help in writing and understanding properties by providing properties templates and visual or natural language support [22, 6]. In this section we focus on the visual formalisms for scenario-based specification that are commonly and extensively used within industrial software development practice: Message Sequence Charts (MSCs) [12], and UML 2.0 Interaction Sequence Diagrams [18].

Other approaches, which are not commonly used in industries, are discussed later in Section 6.

We decided to remain close to the graphical notation of these two languages to satisfy the requirement of *simplicity* of use. For *expressiveness* we analyze them in order to identify the aspects where they lack expressivity 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 3, 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.

MSC Focusing on Message Sequence Charts (MSC), the Telecommunication Standardization Sector of Internal Telecommunication Union (ITU) proposes the MSC standardization through the Recommendation Z.120 [12]. Messages are used to perform communication between system components. The sending and the consumption of messages are two asynchronous events. The time is running from top to bottom along each instance axis and a MSC imposes a partial ordering on the contained events. Except for *coregions* (that allow the specification of unordered messages) and *inline expressions* (parallel composition, iteration, exception and optional regions), there are rules defining the messages ordering: a total time ordering of events is assumed along each instance axis; 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 expressing power. In the following we summarize the MSC missing features:

(i) as pointed out by [6], 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 can only be possible to define a set of *liveness property* 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 execution of a system;

(ii) it is not possible to state that a message must be strictly followed by another message;

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

(iv) given the set of messages potentially exchanged by the system, it is useful to be able to state that the system can exchange all the messages except a specified one. MSC language does not permit to do this;

(v) for two contiguous messages $m1$ and $m2$ (within a lifeline, $m2$ follows $m1$), it is not well defined if $m2$ must necessarily follow $m1$ or if $m1$ and $m2$ can also happen simultaneously. Often, explicitly selecting whether a set of messages are simultaneous or not, might be profitable.

UML 2.0 Many of the above 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 [10] 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, UML 2.0 adds some features that are useful for our purposes: **(i)** *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 can be applied to a set of messages. These operators are graphically represented as a frame box with a compartment displaying its name. If an operator has two or more operands, they are divided by dashed line. 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 [23], 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 the neg and assert features 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 clearly and intuitively usable. Moreover, **(ii)** while a partial ordering is assumed by default, a designer can also define a strict ordering between messages by using the operator *strict*. On the contrary, both in UML 2.0 and MSC, there is no direct and simple way to deal with **(iii)**, **(iv)**, and **(v)** above.

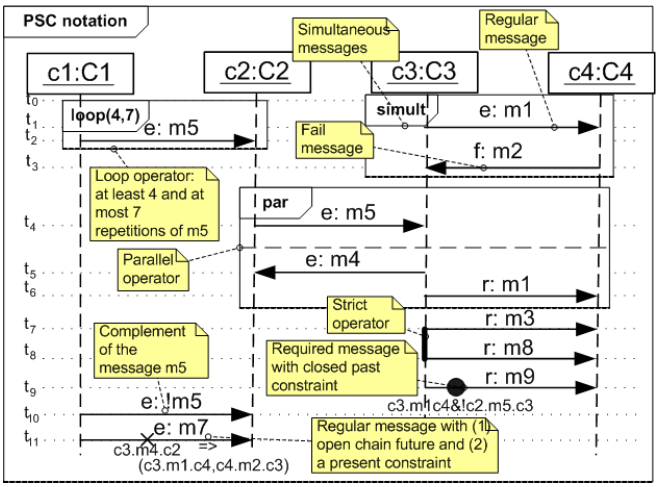

Figure 1: PSC **graphical notation**

3. PSC: PROPERTY SEQUENCE CHART

A PSC describes a finite interaction between a collection of interacting components that can be simultaneously executed. Components communicate by message passing and the acts of sending and receiving a message are considered to be atomic events. We assume a synchronous communication and hence send and receive events of the same message are considered to occur simultaneously. Thus, we can restrict to only send-events. It is worthwhile noticing that, as pointed out in [24], "a bounded asynchronous communication can be modeled by introducing buffer abstraction to decouple message passing". In the remaind of the paper the terms *component* and *component instance* are used interchangeably.

Hereafter, we show the PSC graphical notation and give an informally description of all the statements in the PSC language.

The graphical notation we define is based on UML 2.0 Interaction Sequence Diagrams suitably extended to properly cover all the deficiencies previously identified.

3.1 PSC graphical notation

As it is showed in Figure 1, each component instance is represented as a named rectangle with a vertical dashed-line, called *lifeline*, which extends downward. The time runs from top to bottom. A labeled horizontal arrow represents a message referred as *arrowMSG*. 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.

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)	Complement	No direct counterpart	No direct counterpart	All of messages but one
(v)	Simultaneity	No direct counterpart	No direct counterpart	Simultaneous messages
	Loop	Loop	Loop	Iteration construct
	Par	Par	Par	Parallel operator

Table 1: Comparison between PSC**, UML 2.0 Interaction Sequence Diagrams and MSC**

Since we are focussing on expressing properties for specifying desired execution sequence of a system in terms of messages exchange, 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 to real-time systems for which modeling time is relevant. As shown in Figure 1, we identify a set of horizontal dotted lines $t_0, \ldots, t_n$. These lines are called *structural time-lines* (or simply *time-lines*) and only one *arrowMSG* for each *time-line* is allowed, except for *time-line* t_0 that cannot have associated any message. Time-lines are totally ordered from top to bottom. Note that, the pair time-line and its associated *arrowMSG* uniquely identifies the sender and the receiver (and hence the corresponding send-event). The use of *time-lines* is a means for structuring the lifelines. Even though time-lines induce a total ordering of *arrowMSGs* among the entire *PSC*, a designer can also specify a strict and/or partial ordering of *arrowMSGs* by using **constraints** on the message and **operators** that we shall define later.

Furthermore, let m be a message on a time-line t_i, we refer to t_i as the *present* of m, 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 particular, in order to well define the *past* (*future*) for the message at the line t_1 (t_n), we implicitly assume the presence of a *time-line* t_0 (t_{n+1}) that has no message. The messages possibly exchanged in the past, present and future are referred as *intraMSGs* of m. The introduction of *intraMSGs* allow one to specify conditions 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 but stipulate restrictions on *intraMSGs* of m.

In order to identify the sender and the receiver components, the labels for *intraMSGs* 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. In the remaind of the paper, when there is no ambiguity, the terms *message* and *message label* are used interchangeably.

By referring to the above itemized aspects, **(i)** PSC distinguishes among three different types of *arrowMSGs* (see Figure 1):
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, however, if it 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 "**r:**" prefixed to the labels. It is mandatory for the system to exchange this type of messages. 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: the labels are 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 invalid traces.

(ii) 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 the time-lines t_i and t_{i+1} respectively, the *strict* operator specifies that no other messages can occur. Graphically, the strict operator is a thick line that links the pair of messages (as in Figure 1, between messages $m3$ and $m8$). 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 2. Differently from us, in UML 2.0 it is also permitted to specify strict ordering within more than one lifeline. The specification that the standard for UML 2.0 gives to this operator can lead to ambiguous semantics. Therefore, we believe that the strict operator is better identified as a relation between two contiguous messages within a single lifeline.

(iii) As already said *constraints* are associated to a single *arrowMSG* m and permit to specify "restrictions" for *intraMSGs* of m. Restrictions specify either a chain of *intraMSGs* or a boolean formula (over a set of *intraMSG* labels) limited to only contain the logical connectives and (&), or (||), not (!) in order to keep simpler the language. Informally, a chain represents a sequence of *intraMSGs* of m and it is satisfied if the messages are exchanged following the ordering imposed by the chain itself. Informally, a boolean formula f is satisfied iff there exists an evaluation in terms of *intraMSGs* passing that makes it true. For instance, let m_i and m_j be two messages: $m_i \& m_j$ is evaluated to true iff m_i and m_j are exchanged simultaneously; $m_i || m_j$ is evaluated to true iff at least one of m_i and m_j is exchanged; $!m_i$ is true iff m_i is not exchanged.

We distinguish between three super-classes of constraints: *past constraints*, *present constraints*, and *future constraints*. *past constraints* and *future constraints* can be both chains and boolean formulae. On the contrary *present constraint* can be only boolean formulae.

As it is intuitive, *future constraints* cannot be associate to a fail message: the system as no future after such a message has been exchanged. Furthermore, 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 *present* and *past constraints* and m_j can only have *present* and *future constraints*.

The super-classes *past constraints* and *future constraints* can be specialized to constitute the classes *open* and *closed past constraints* and *open* and *closed future constraints*, respectively. Within an *arrowMSG* m, a closed past (future) constraint applies the specified restrictions both on the *past*(*future*) and the *present* of m, conversely an open past (future) constraint applies the restriction only on the *past*(*future*) of m.

Closed and open constraints, which have a boolean formula as argument (so called *boolean formula constraints*), are graphically represented as filled and empty circles respectively, see Figure 1. The boolean formula is specified in the textual form and is placed just under the circles. Instead closed and open constraints, which have a chain as argument (so called *chain constraints*), are graphically represented as filled and empty arrows respectively. A chain over the set of messages $\{m_1, \ldots, m_n\}$ is specified by the ordered tuple $(l_1, \ldots, l_n)$ of the messages' labels. The tuple is placed just under the arrow. In particular, for *chain constraints*, it is also possible to specify "unwanted" chains. In other words, it is possible to specify that the messages in the chain are exchanged following any ordering different from the one represented by the chain itself. These kind of constraints are called open and closed *"unwanted" chain constraints* and are graphically represented by filled and empty barred arrows.

For both boolean formulae and chains, past constraints are placed near to the message arrow source. On the contrary future constraints are placed near to the arrow target.

While we allow a message to have both an open and a closed *boolean formula constraint*, we disallow a message to have a *boolean formula constraint* and a *chain constraint* at the same time because the semantics could not be really intuitive.

Finally the graphical element used for a *present constraint* is a cross positioned on the middle of an *arrowMSG* message and only a boolean formula is allowed as argument.

(iv) Moreover, from our experience in specifying temporal properties, we found out that it is helpful to have the notion of complement to refer the set of all messages potentially

exchanged by the system except one designed *arrowMSG* (regular, required or fail message). By applying the PSC *complement* operator to a regular, required or fail message, m is it possible to state that the set of all messages except m can, must or cannot be exchanged, respectively.

The notation used for the complement is the symbol "!" infixed between the message type and its label. In Figure 1, e: $!m5$ is the complement of the regular message $m5$. For example, let us suppose that all the exchangeable messages of a system are $\{m1, m2, m3\}$, the message labeled by $!m1$ matches with $m2$ and $m3$. There is no direct mapping for the complement of a message in MSC and UML 2.0. It is possible to indirectly express this concept by encapsulating each message in the complement set of a given message into the alt operator of both MSC and UML 2.0. For systems with a high number of exchanged messages, this can be tedious or unfeasible.

(v) In Figure 1 we can see that, *parallel*, *loop*, and *simultaneous* operators are introduced with a UML 2.0 like graphical notation. As previously mentioned, the UML 2.0 standard has not a well defined semantics. Thus, it is not clear whether having the power of recursion is a good thing in the first place, since it introduces an extra level of complexity into any semantics by implying fixed points. As noticed in [23], similar problems have not been addressed in the standard, and it is likely that they have been overlooked. By sharing the same concern, we prefer to limit the power of recursion by allowing only sequences of the three types of messages (subject to some limitations) to be valid arguments of parallel, loop, and simultaneous operands. By imposing this restriction, our visual language preserves a good level of expressing power without compromising its intuitive understanding. For the lack of space in the following we briefly describe these operators and we refer to [20] for more details.

Informally, the *parallel* operator allows a parallel merge between the behaviors of the two operands. The messages arguments of the operands can be interleaved in any way as long as the ordering imposed by each operand as such is preserved. The *loop* operator allows the operand to be repeated a given number of times. As it is shown in Figure 1, it is also possible to specify a lower and an upper number of repetitions. The *simultaneous* operator has been introduced to have the possibility of specifying that two or more messages can be potentially exchanged at the same time.

To summarize, in Table 1 we report a comparison between MSC, UML 2.0 Interaction Sequence Diagrams and PSC, restricted to the PSC features.

3.2 PSC semantics

The PSC operational semantics is given in terms of Büchi Automata [3] that are an operational representation for LTL formulae [9]. For each PSC, the denotational semantics will be the languages accepted by the associated automaton. The translation algorithm from PSC to Büchi automata is called Psc2Ba.

We recall that the automata-based model checking uses automata for both representing the system and the property. More precisely the model checker requires having the complement of the languages recognized by the automaton of the property. Since it is an expensive task to negate a Büchi automaton, we directly express in the Büchi automaton the negation of the desired temporal property.

It is out of the purpose of this paper to go through a detailed description of the translation rules used by the algorithm and we entirely refer to [20] for them. An example of basic translation rules for Required messages is given in Figure 2.

4. 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.

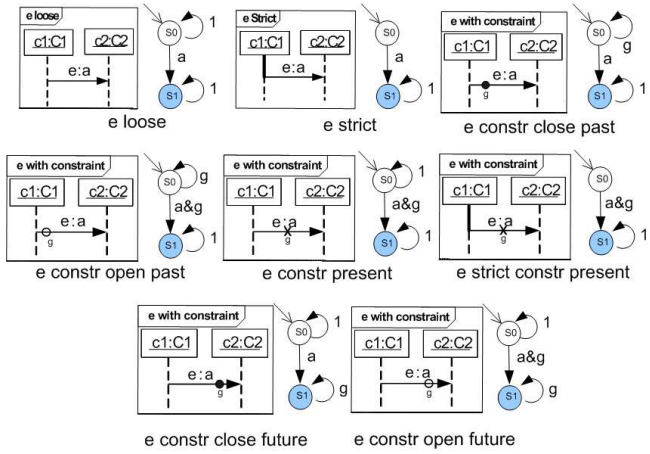

Figure 2: Regular Messages

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 into 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 [8]. 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 [20]. 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 [20] for a description of all patterns. In Figure 3 we report two examples, a Precedence Chain 1 cause-2 effects (p precedes s and t) and a particular instance of the *Bounded Existence Pattern*, where the bound is at most 2 designated messages.

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 same scope, the considered instance of the Bounded Existence Pattern states that after q, an error arises if r is exchanged and, between q and r, the message p is exchanged more than two times.

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 Bounded existence:

$$G((q\&Fr) -> ((!p\&!r)U(r||((p\&!r)U(r||((!p\&!r)U (r||((p\&!r)U(r||(!pUr))))))))))$$

While the LTL formulas are not easily understandable, the same properties expressed in the PSC formalism (Figure 3) appear more intuitive and closer to their natural language description.

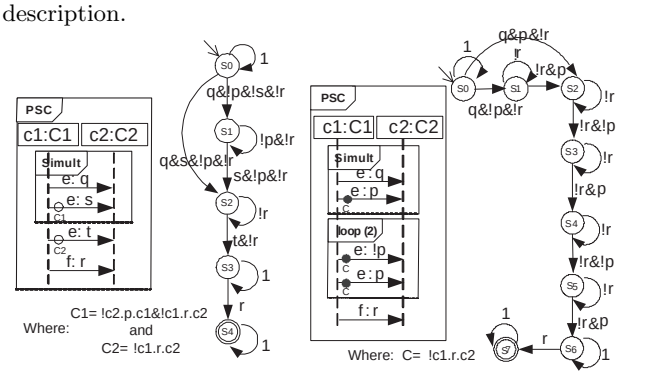

Figure 3: Left-hand side: precedence chain 1 cause-2 effects. Right-hand side: bounded existence pattern with bound at most 2 and scope between q and r

Focusing on the Precedence Chain, within the chosen scope, the "after q" notion is represented as a regular message. If the message s happens before r (the scope) and p, and if t happens before r, then we build the error condition. Since we are in the scope between q and r, the error is raised iff the message r is exchanged. It is worth mention that the simultaneous operator is used because the scope definition itself states that s and q can be exchanged simultaneously. By recalling that the Büchi automaton expresses the negation of the desired temporal property it is simple to recognized the negation of the Precedence Chain in the automaton on the left-hand side of Figure 3.

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 constraint of the messages s and t, and we have to add before the last fail message r a new regular message z.

In the other example, the bounded existence, after a precondition composed of the message q, the scope after q, three messages p alternated by messages different from p (i.e., $!p$) and no occurrences of r (represented as constraints), an error is raised when and if the message r happens (i.e., while in the scope *Between q and r*). Note that the simultaneous operator is used because q and p can happen simultaneously and we use the loop operator in order to compactly write the formula and to write a scalable formula. If we want to instantiate the bounded existence pattern to a bound equal to n, we need only to change to n the number of repetitions of the loop operator.

5. TOOL

The Psc2Ba algorithm has been implemented as a plugin for Charmy. The Psc2Ba plugin implementation is currently available in [5]. 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. More details the Psc2Ba plugin can be found in [5].

We also proposed a wizard called W_PSC [2] that, by using a set of sentences (classified according to temporal properties main keywords), helps the user while writing PSC scenarios.

6. RELATED WORK

The Psc2Ba algorithm follows some ideas and the terminology of the Timeline Editor (TimeEdt) [21]. The timeline editor was developed to capture long running and complex chains of dependent events (called "chain patterns" from the specification patterns people). These chains are very hard to write in LTL thus the TimeEdt people developed syntax for writing them as timelines. Even though we recognize chains to be very useful when specifying properties and even though TimeEdt is a powerful means for writing them, its usefulness is not convincing when used in a more general context. For example, TimeEdt 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 3, by means of one chain pattern, highlights the ability of PSC.

Many works in the literature propose algorithms for temporal properties generation [13, 14, 25] starting from Live Sequence Charts (LSC) [6]. LSCs are an extension of MSCs with the aim to deal 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 in specifying *intraMsg* and especially chain constraints. In fact a constraint allows the specification of what can be performed before, during 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 [20, 8]. 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. In addition open and closed constraints of PSC permit to define with high accuracy these kinds of relations. For example, considering a future chain constraint, we could be interested in establishing whether the first message of the chain can happen or not. 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 think can be specified by using regular messages.

The translation algorithm proposed by Ghezzi et al. [25] gets in 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, allow for the verification of systems. The paper [14] proposes a translation into CTL* while the work [13] offers a solution for timed Büchi automata generation. Other approaches [4, 15, 1] define graphical languages that appeared to be not easily comprehensible and not easily integrable into industrial software development processes.

7. CONCLUSION AND FUTURE WORK

In this paper we propose a formalism for specifying temporal properties aimed at being simple, (sufficiently) powerful and user-friendly. After having examined Message Sequence Charts [12], and UML 2.0 Interaction Sequence Diagrams [18], we present a scenario-based graphical language that is an extended notation of a selected subset of the UML 2.0 Interaction Sequence Diagrams. We call 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. Psc2Ba 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* [8]. We showed that with our PSC language it is possible to represent all these patterns. Moreover, we have defined 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 [5] 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 define a lower and an upper bound between two subsequent messages on one instance. Consequently, the algorithm Psc2Ba will be update in order to produce timed Büchi automata. We plan also to investigate if PSC can be proposed as a UML 2.0 profile.

8. REFERENCES

[1] A. Alfonso, V. Braberman, N. Kicillof, and A. Oliviero. Visual timed event scenarios. In *26th ICSE'04. Edinburgh, Scotland, UK*, May 2004.

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

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

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

[5] Charmy Project. Charmy web site. http://www.di.univaq.it/charmy, February 2004.

[6] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

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

[8] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, pages 411–420, 1999.

[9] R. Gerth, D. Peled, M. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *5th IFIP WG6.1*. Chapman & Hall, Ltd., 1996.

[10] Ø. Haugen. Comparing UML 2.0 Interactions and MSC-2000. In *SAM*, pages 65–79, 2004.

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

[12] ITU-T Recommendation Z.120. Message Sequence Charts. ITU Telecom. Standardisation Sector.

[13] J. Klose and H. Wittke. An automata based interpretation of live sequence charts. In *TACAS 2001. LNCS 2031, pp. 512-527*, 2001.

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

[15] I. Lee and O. Sokolsky. A Graphical Property Specification Language. In *High-Assurance Systems Engineering Workshop, Washington, DC*, Aug 11 - 12.

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

[17] M.Tivoli and M.Autili. Synthesis: a tool for synthesizing "correct" and protocol-enhanced adaptors. *RSTI - L'OBJET JOURNAL 12/2006. WCAT04*, pages 77–103.

[18] Object Management Group (OMG). UML: Superstructure version 2.0.

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

[20] PSC Project. PSC web site. http://www.di.univaq.it/psc2ba, April 2005.

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

[22] R. L. Smith, G. S. Avrunin, L. A. Clarke, and L. J. Osterweil. PROPEL: An Approach Supporting Property Elucidation. In *ICSE2002*, pages 11–21, May 19-25 2002.

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

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

[25] L. Zanolin, C. Ghezzi, and L. Baresi. An approach to model and validate publish/subscribe architectures. In *SAVCBS*, Sept. 2003.