Safety-Liveness Semantics for UML 2.0 Sequence Diagrams*

Radu Grosu and Scott A. Smolka

Department of Computer Science
State University of New York at Stony Brook
grosu@cs.sunysb.edu, sas@cs.sunysb.edu

Abstract

We provide an automata-theoretic solution to one of the main open questions about the UML standard, namely how to assign a formal semantics to a set of sequence diagrams without compromising refinement? *Our solution relies on a rather obvious idea, but to our knowledge has not been used before in this context: that bad and good sequence diagrams in the UML standard should be regarded as safety and liveness properties, respectively. Proceeding in this manner, we obtain a semantics that essentially complements the set of behaviors associated with the set of sequence diagrams, thereby allowing us to use the standard notion of refinement as language inclusion. We show that refinement in this setting is compositional with respect to sequential composition, alternative composition, parallel composition, and star+ composition.*

1. Introduction

Scenario-based descriptions of the interaction among the components of a distributed reactive system have long attracted the attention of the software-engineering and the computer-aided verification communities. Originally standardized in the telecommunication industry as message sequence charts [18], they played a key role in the software-development methods SDL [8] and ROOM [25], and have become a central component of the UML standard [10] under the name *sequence diagrams* (SDs).

Various researchers have realized the importance of equipping SDs with a formal semantics, thereby providing SDs with a precise and unambiguous intention, and forming the basis for powerful analysis algorithms and tools [5, 7, 3, 17, 20, 21, 12, 15, 14, 26, 27, 28, 16].

*R. Grosu was partially supported by the NSF Faculty Early Career Development Award CCR01-33583.

In spite of these successes, there is still no unanimous agreement on one of the main questions about SDs, namely *how can one assign a formal meaning to a set of SDs without compromising refinement?* The standard notion of refinement via simulation or language inclusion requires that each observable behavior of an implementation is also an observable behavior of the specification. An SD, however, may be composed with other SDs via sequential composition, alternative composition, parallel composition and star+ composition (looping), and a direct translation of SDs to automata (or partial orders) leads to implementations with a larger number of behaviors than their corresponding specifications.

In this paper, we provide a solution to this paradox by defining a formal semantics for SDs that relies on a rather obvious idea, but to our knowledge has not been used before: that bad and good SDs in the UML standard should be regarded as safety and liveness properties, respectively. The time-proven theory and practice of the safety/liveness classification gives us additional confidence in the usefulness of our approach. To simplify the presentation, most examples in the paper consider SDs in the context of synchronous concatenation [7]. Such SDs define regular languages having associated (hierarchic) automata or regular definitions. Our semantic construction is, however, independent of this choice, and, as shown in Section 4.3, can be easily extended to the case of asynchronous concatenation of bounded SDs.

Our semantic treatment of SDs proceeds roughly as follows: (i) Given a set of SDs, we translate each SD to a hierarchic automaton that may contain negative states. (ii) We separate the negative states from the positive ones by defining a construction that extracts a negative hierarchic automaton and a positive hierarchic automaton from the original hierarchic automaton. (iii) We construct a safety Büchi automaton from the negative hierarchic automaton and a liveness Büchi automaton from the positive one. The safety and liveness automata ensure that a trace that may either lead to the completion of a bad scenario or prevent the com-

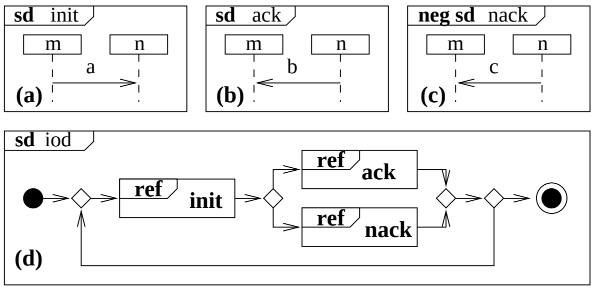

Figure 1. (a)–(c) Basic sequence diagrams. (d) High-level sequence diagram.

pletion of a good scenario is rejected. (iv) We take the product of these two automata as the operational semantics of the original set of SDs and the corresponding language as the denotational semantics. (v) We define refinement in terms of language inclusion; there is no need to consider finer (more behavioral) notions of refinement as our automata are input enabled. (vi) We show that refinement is compositional with respect to sequential composition, alternative composition, parallel composition and star+ composition.

The rest of the paper is organized as follows. Section 2 reviews the syntax of UML SDs. Section 3 shows how to interpret SDs as hierarchic automata, while Section 4 shows how to extract negative and positive automata from such a hierarchic automaton. Section 4 also explains how such a pair of automata can be transformed into the product of a safety and a liveness automaton, which is subsequently used to define a compositional notion of refinement. Section 5 presents our compositionality results while Section 6 discusses related work. Section 7 contains our concluding remarks and directions for future work.

2. Syntax of UML 2.0 SDs

The visual notation for a UML 2.0 SD is a solid rectangle containing two parts: the *name* of the SD which occurs after the keyword **sd** in the pentagon in the upper-left corner of the rectangle, and the *body* of the SD which occurs in the center of the rectangle. SDs are classified according to their body type as either *basic SDs*, as exemplified by the set of basic SDs depicted in Figures 1(a)–(c), or *high-level SDs*, as exemplified by the SD in Figure 1(d). High-level SDs are referred to as "interaction overview diagrams" in the UML 2.0 standard.

A basic SD focuses on the message interchange among a set of processes. Each process is represented by a *life-line* drawn as a (dashed) vertical line beneath a box containing the name of the process, and each message is represented by an arrow drawn from the sender to the receiver and decorated with a message identifier. For example, the basic SD `init` of Figure 1(a) contains two life-lines associated with processes `m` and `n`, and one message decorated with message identifier `a`.[1]

A high-level SD is intended to serve as an overview of the flow of control among its constituent SDs, and is essentially a flowchart whose nodes are either *initial*, *final* or *decision/merge* points, or non-recursive *references* to basic or high-level SDs. To increase readability, high-level SDs may also contain *fork* and *join* points which allow one to compose SDs in parallel.

The sequencing or *concatenation* of SDs implicit in a high-level SD—e.g., the execution of `init` precedes that of `ack` or `nack` in the high-level SD of Figure 1(b)—comes in two varieties: *synchronous* and *asynchronous*. For the synchronous concatenation of two SDs S_1 and S_2, an event in S_2 is assumed to happen only after all the events in S_1; in the asynchronous case, this restriction is lifted.

Basic SDs may also be visually partitioned into arbitrarily nested interaction *fragments*, drawn also as rectangles and containing an *interaction operator* and a set of *interaction operands*. The main UML 2.0 interaction operators include sequencing, alternation, option (representing a choice between the enclosed operand and an invisible empty operand), parallel, looping, and *negative*, which designates an invalid execution sequence. Since most interaction operators, with the notable exception of negative, can be captured by high-level SDs, we assume for simplicity that basic SDs are flat (non-nested), but allow the negative operator (denoted `neg`) to be applied to SDs; see, for example, SD `nack` of Figure 1(a), which is a negative SD.

3. Closed-World Semantics of UML SDs

The formal semantics of SDs is traditionally given in terms of a process algebra or partial orders. If, however, SDs are bounded (see Section 4.3), one can equivalently use nondeterministic finite automata as a semantic framework. Such semantics are *closed-world* in the sense that given a set of SDs, no new SDs may be added to it without compromising refinement. However, the advantage of the automata-theoretic approach is that it allows us (Section 4) to respectively map negative and positive SD connotations into "attempting an undefined transition" and "reaching a state infinitely often" in the setting of Büchi automata. This mapping forms the basis of our safety-liveness semantics for UML SDs, and consequently allows us to prove that refinement is compositional in this setting.

Definition 1 (NFA) A *nondeterministic finite automaton* (NFA) $A = \langle \Sigma, S, \delta, S_0, F \rangle$ over an input alphabet Σ is a 5-tuple such that:

[1] SD names, message identifiers, arrowheads, etc. may additionally contain information such as parameters and types. For simplicity, we do not consider such extensions.

- Σ is the *input alphabet*.
- S is a finite set of *states*.
- $\delta \subseteq S \times \Sigma \cup \{\epsilon\} \times S$ is the *transition relation*.
- $S_0 \subseteq S$ is a set of *initial states*.
- $F \subseteq S$ is a set of *accepting states*.

An input sequence $\sigma = \sigma_1 \ldots \sigma_n$ is *accepted* by A if there is a state sequence $s = s_0 \ldots s_n$ such that $s_0 \in S_0$, $s_n \in F$ and for each i, the tuple $(s_i, \sigma_{i+1}, s_{i+1})$ is in δ. $L(A)$, the set of all accepted input sequences, is called the *language* of A. □

NFAs shall be used to provide a closed-world semantics for basic SDs and, as such, we shall henceforth refer them as *basic* NFAs. We shall do the same for high-level SDs, and for this purpose we need a notion of a high-level NFA.

Definition 2 (High-level NFA) A *high-level NFA* is an NFA such that each state is a basic NFA or another, non-recursive, high-level NFA. Also, each transition of a high-level NFA is labeled by ϵ. A formal definition of a high-level NFA can be found in [7, 4]. □

The UML 2.0 semantics for SDs is based on an asynchronous interpretation of communication, where the sending and receiving of a message are considered to be distinct events. A natural way of capturing this semantics is to associate a partial order (PO) with each basic SD, where events are ordered from top to bottom along process life-lines and where the sending event at the tail of a message arrow is uniquely related to and precedes the receiving event at the head of the message arrow. Given the PO, one can obtain an NFA from it by applying the classical PO-to-NFA translation scheme of [7]: (1) The states of the automaton correspond to *cuts*, which are sets closed w.r.t. the partial order; the empty cut is the initial state and the cut with all events is the accepting state. (2) If cut d equals cut c plus a single event e, then there is an edge from c to d labeled with $l(e)$. The labeling function l is given with respect to an input alphabet Σ, which is partitioned into sending tuples `m:n!a` and receiving tuples `n:m?a`.

Example 1 (PO-to-NFA translation) Consider the SD of Figure 2(a). The partial order it induces is depicted in Figure 2(b), where sending and receiving events are represented as integers. Using the PO-to-NFA translation, we obtain the NFA of Figure 2(c). For simplicity, transitions are labeled by events e instead of $l(e)$. □

We shall henceforth assume that basic SDs can be effectively parsed to basic NFAs over an appropriate alphabet. Moreover, since nesting in high-level SDs is non-recursive, we shall similarly assume that high-level bounded SDs can be effectively parsed to high-level NFAs.

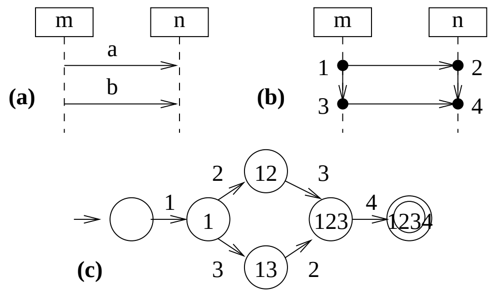

Figure 2. (a) Basic SD. (b) Associated PO. (c) Corresponding NFA.

4. Safety-Liveness Semantics of UML 2.0 SDs

The main difficulty of the classical, closed-world semantics for SDs is that it is not compositional. The standard notion of refinement via simulation or language inclusion requires that each observable behavior of an implementation is also an observable behavior of the specification. An SD, however, may be composed with other SDs via sequential composition, alternative composition, parallel composition and star+ composition (looping), and a direct translation of SDs to partial orders (or automata) leads to implementations with a larger number of behaviors than their corresponding specifications. In this section, we present a solution to this problem by defining a safety-liveness semantics for SDs.

4.1. From NFAs to Büchi Automata

UML 2.0 SDs are intended to capture the behavior of *reactive systems*: those whose role is to maintain an ongoing interaction with their environment rather than produce some final value upon termination. NFAs are therefore not an appropriate formal model for reactive systems, since termination should be viewed as an error rather than a desirable outcome. A more suitable formal model for SDs are ω-automata: automata over infinite words. In particular, *Büchi automata* are ω-automata requiring an accepting state to be visited infinitely often for every accepted input sequence. In the rest of this section, we show how to construct a Büchi automaton for a given SD.

A standard way of avoiding (premature) termination within a finite automaton is to allow the automaton to loop forever in each accepting state. Accordingly, we extend the NFA construction of Section 3 by adding a self-loop labeled by Σ to each accepting state. This extension can be understood as allowing looping *regardless of the input*. A state of this nature is often referred to in process theory as *Chaos*, and so it shall be here.

The closed-world semantics of Section 3 captures the asynchronous communication paradigm inherent in SDs

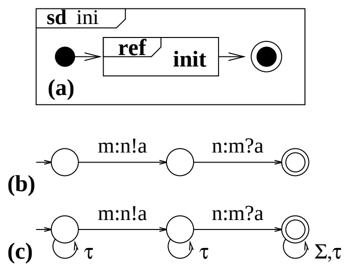

Figure 3. (a) High-level SD ini. (b) Associated NFA. (c) Corresponding Büchi automaton.

by imposing a partial order among sending and receiving events. To be more specific about the fact that processes may operate at different speeds, but not too specific, we add a τ self-loop to every state. Such a loop models the possibility of *stuttering* between events. In what follows, let $\Sigma_\tau = \Sigma \cup \{\tau\}$.

Example 2 (Büchi automaton) The finite automaton of Figure 3(b) corresponding to a closed-world view of SD `ini` in Figure 3(a), becomes the Büchi automaton of Figure 3(c) if we modify its acceptance condition, add a stuttering (self-loop) τ transition to each state, and add a Σ self-loop (Chaos) to its accepting state. The language accepted by this automaton is $\tau^* \, m{:}n!a \, \tau^* \, n{:}m?a \; \Sigma_\tau^\omega$ □

4.2. Positive and Negative Automata

As discussed in Section 2, a UML 2.0 SD may contain arbitrarily nested interaction fragments, corresponding to basic SDs embedded within a high-level SD. Interaction fragments (and, thus, SDs) can be positive or negative, with the latter indicated by the interaction operator `neg`. Positive and negative SDs, however, are intended to capture orthogonal properties, namely liveness and safety. For analysis purposes, it is therefore convenient to separate the positive from the negative parts.

To achieve this separation, we extract both a negative high-level NFA and a positive high-level NFA from the high-level NFA associated with a high-level SD, where each node in the high-level NFA corresponds to either a positive or negative basic SD. The negative NFA is obtained by turning all negative nodes into *accepting sink nodes* (without outgoing transitions), and all other nodes non-accepting. The positive NFA is obtained by *removing* all negative nodes and all their associated transitions.

Example 3 (Positive/negative NFAs) Consider the high-level NFA of Figure 4(a), corresponding to the high-level SD `iod` of Figure 1(d). Making only its negative nodes accepting leads to the negative NFA of Figure 4(b). Deleting the negative nodes and their associated transitions leads to the positive NFA of Figure 4(c). □

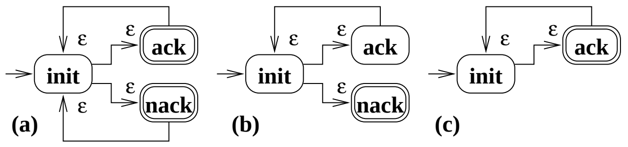

Figure 4. (a) High-level NFA. (b) Negative high-level NFA. (c) Positive high-level NFA.

4.3. From High-Level to Basic NFAs

Basic negative and positive NFAs (and their associated Büchi automata) can be derived from their high-level counterparts via *flattening*. A high-level NFA can be viewed as a (non-recursive) hierarchy of NFAs with basic NFAs at the leaf nodes. During the flattening process, only the accepting states of basic NFAs that are descendants of accepting nodes are preserved as accepting in the flattened NFA. All other states are non-accepting. For a given high-level SD S, we denote the resulting negative Büchi automaton derived from S as $\texttt{neg}(S)$ and the resulting positive Büchi automaton as $\texttt{pos}(S)$.

Sequencing or concatenation of NFAs is an implicit operation in high-level NFAs. To flatten a high-level NFA, we consider two forms of NFA concatenation derived from their high-level SD counterparts. In the *synchronous concatenation* of two SDs S_1 and S_2, officially known as *strict sequencing* in UML 2.0, all the events in SD S_1 finish before any event in SD S_2 occurs. Consequently, the language $L(S_1\, S_2)$ of the concatenation of S_1 and S_2 is the concatenation $L(S_1)\, L(S_2)$ of the languages of the component SDs.

In the *asynchronous concatenation* of two SDs S_1 and S_2, officially known as *weak sequencing* in UML 2.0, events are concatenated on a process-by-process (lifeline-by-lifeline) basis. As such, events in S_2 may occur before the last event in S_1 completes. For example, the NFA of Figure 2(b) corresponds to the asynchronous concatenation of S_1 and S_2, where S_1 and S_2 are the basic SDs corresponding to messages a and b, respectively.

Under asynchronous concatenation, the language of a high-level SD is not necessarily regular [9]. For example, in the SD `unb` of Figures 5(a) and 5(b), process m can send arbitrarily many messages a to process n before any message is actually received by process n.

To avoid such pathological cases, Alur and Yannakakis introduce in [7] a restriction of high-level SDs, called *bounded high-level SDs*, which are shown to accept regular languages by providing a translation to basic NFAs. The details of the translation can be found in [7], but we note that we can directly apply it to positive and negative high-level NFAs to obtain the desired positive and negative basic NFAs.

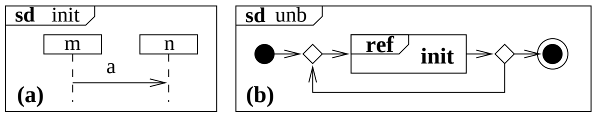

Figure 5. (a) Basic SD. (b) Unbounded high-level SD.

4.4. Safety Automaton

According to the UML 2.0 standard [24], a negative fragment describes traces *that are invalid* and should therefore *never happen* in an execution of the system. In other words, a negative fragment is intended to capture a *safety property*.

Let N be a flattened, negative Büchi automaton constructed using the procedure described in Sections 4.1, 4.2 and 4.3, and let $B = \neg N$ be the Büchi automaton associated with the complement of N. B is not necessarily a safety automaton (property), according to the definition given in [1], because it is not limit-closed. An automaton is *limit-closed* if all its infinite behaviors are completely characterized by its finite behaviors. We now present a transformation that will render B limit-closed. A Büchi automaton is *reduced* if it does not contain a state from which no accepting state is reachable. Note that any automaton can be transformed to an equivalent reduced automaton.

Definition 3 (Safety property [1]) Consider a safety property P that stipulates that a "bad thing" b never happens. If b does occur within an infinite sequence α, then it must occur after a finite prefix of α, and no matter how α is extended after the occurrence of b, α is still a violation of P. Taking the contrapositive:

$$\forall \alpha \in \Sigma_\tau^\omega .\ \alpha \models P \Leftrightarrow \forall i > 0.\ \exists \beta \in \Sigma_\tau^\omega .\ \alpha[1..i]\, \beta \models P$$

we get the formal definition of safety. □

Definition 3 can be seen as a formal justification for our construction of Section 4.2, where the outgoing transitions from the accepting states of the the negative high-level NFA are deleted.

Definition 4 (Safety transformation [2]) Given a reduced Büchi automaton $B = \neg N$, define $\mathtt{Safe}(B)$ to be the version of B in which every state has been transformed into an accepting state. □

Example 4 (Safety automaton) The transformation of the SD of Figure 1(d) to the safety automaton in Figure 6(b) can be summarized as follows: (i) Construct the associated negative NFA as shown in shown in Figure 4(b). (ii) Flatten this to obtain the NFA in Figure 6(a). (iii) Close this with respect to stuttering and Chaos. (iv) Complement and reduce it. (v) Make all states accepting, as shown in Figure

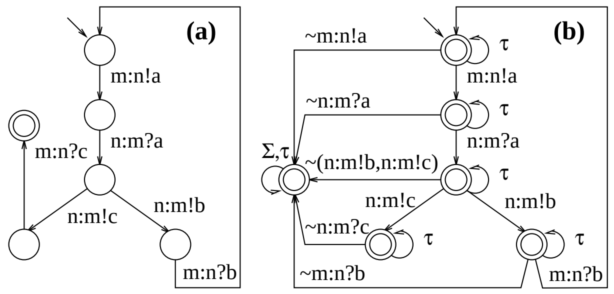

Figure 6. (a) Negative automaton. (b) Corresponding safety automaton.

6(b) where $\sim\mathtt{a} = \Sigma - \{\mathtt{a}\}$. Interpret the resulting NFA as a Büchi automaton. This automaton rejects any omega-trace in $\tau^*\, \mathtt{m:n!a}\, \tau^*\, \mathtt{n:m?a}\, \tau^*\, \mathtt{n:m!c}\, \tau^*\, \mathtt{m:n?c}\ \Sigma_\tau^\omega$. □

The above construction is effective and transforms any high-level NFA associated with an SD to a safety Büchi automaton. Since all states of the safety automaton are accepting, $\mathtt{Safe}(A)$ never rejects an input by failing to enter an accepting state (lack of "good thing").

Theorem 1 (Safety [2]) The Büchi automaton $\mathtt{Safe}(A)$ specifies a safety property.

Proof sketch The proof proceeds by first observing that $\mathtt{Safe}(\mathtt{Safe}(A)) = \mathtt{Safe}(A)$ and then by showing that a reduced Büchi automaton B specifies a safety property if and only if $L(B) = L(\mathtt{Safe}(B))$.

4.5. Liveness Automaton

According to the UML 2.0 standard [24], the positive fragments describe traces *that are valid and should be possible*. As a consequence, each finite execution should be extendible to an execution where the positive trace *eventually happens*. In other words, the positive fragments are intended to capture *liveness properties*.

Closing the positive automaton constructed in Section 4.2 with stuttering and Chaos, does not necessarily lead to a liveness Büchi automaton (property), as defined in [1]. The former still represents a closed-world safety constraint. We now present a transformation that leads to an automaton that is open to new behaviors, but rejects the ones preventing positive traces from happening, by infinitely stuttering in a non-accepting state.

Definition 5 (Liveness property [1]) Consider a liveness property P that stipulates that a "good thing" g eventually happens. If g does not occur within a finite sequence α, then α can be extended to an infinite sequence in which g eventually occurs.

$$\forall \alpha \in \Sigma_\tau^* .\ \exists \beta \in \Sigma_\tau^\omega .\ \alpha\, \beta \models P \qquad \square$$

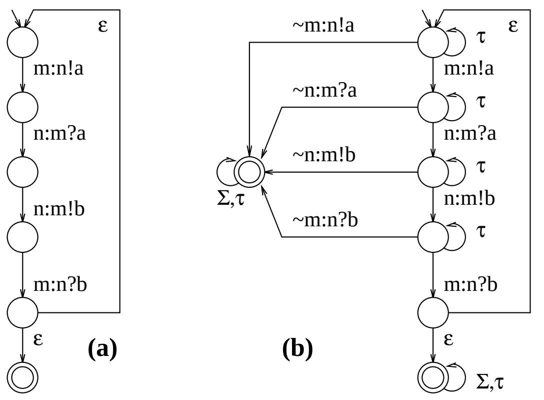

Figure 7. (a) Positive automaton. (b) Corresponding liveness automaton.

Definition 6 (Liveness transformation [2]) Given a reduced Büchi automaton A representing the positive high-level NFA P, define $\texttt{Live}(A)$ to be the Büchi automaton derived from A as follows: (i) Construct the automaton $\texttt{Uns}(A){=}\neg\texttt{Safe}(A)$. (ii) Take the union $A \cup \texttt{Uns}(A)$. □

All steps in Definition 6 are effective and $\texttt{Live}(A)$ never rejects an input that leads to an undefined transition in A (presence of a "bad thing").

Example 5 (Liveness automaton) The derivation of the liveness automaton of Figure 7(b) from to the SD of Figure 1(d) can be summarized as follows: (i) Construct the associated positive NFA as shown in shown in Figure 4(c). (ii) Flatten this to obtain the NFA in Figure 7(a). (iii) Close it with respect to stuttering and Chaos. (iv) Use the liveness construction (in this case this simplifies to adding a trap state) to obtain the desired liveness Büchi automaton of Figure 7(b). This automaton rejects any omega-trace that may prevent the positive behavior to complete: τ^ω, $\tau^*\,\texttt{m:n!a}\,\tau^\omega$, $\tau^*\,\texttt{m:n!a}\,\tau^*\,\texttt{n:m?a}\,\tau^\omega$, and $\tau^*\,\texttt{m:n!a}\,\tau^*\,\texttt{n:m?a}\,\tau^*\,\texttt{n:m!b}\,\tau^\omega$. □

Theorem 2 (Liveness [2]) The Büchi automaton $\texttt{Live}(A)$ specifies a liveness property. Moreover $L(\texttt{Live}(A)) = L(A) \cup (\Sigma_\tau^\omega - L(\texttt{Safe}(A))$.

Proof sketch The proof proceeds by first showing that $L(\texttt{Safe}(\texttt{Live}(A))) = \Sigma_\tau^\omega$, and then showing that a reduced Büchi automaton B specifies a liveness property if and only if $L(\texttt{Safe}(B)) = \Sigma_\tau^\omega$.

4.6. Safety-Liveness Semantics

We are now ready to give the safety-liveness semantics for bounded UML 2.0 SDs.

Definition 7 (Safety-liveness semantics) For a bounded SD S, let $S_l = \texttt{Live}(\texttt{pos}(S))$ and $S_s = \texttt{Safe}(\neg\texttt{neg}(S))$ be the corresponding Büchi automata constructed as described above. Then $L(S)$ is given by $L(S_l \times S_s)$. □

Example 6 (Safety-liveness automaton) Taking the product (conjunction) of the safety automaton of Figure 6(b) and the liveness automaton of Figure 7(b), one obtains the desired automaton for the SD of Figure 1(b). □

Theorem 3 (Safety-liveness semantics) $L(S)$ is the set $L(S_l) \cap L(S_s)$.

Proof sketch The language of the product of two Büchi automata is the intersection of the languages of the component automata.

The safety-liveness semantics of an SD S rejects any invalid trace, i.e., a trace not in $L(\texttt{Safe}(\neg\texttt{neg}(S)))$, and any trace that may prevent a positive behavior from happening, i.e., a trace not in $L(\texttt{Live}(\texttt{pos}(P)))$.

5. Refinement of UML 2.0 SDs

In the early phases of software development, it is rarely the case that one knows in advance all the positive and negative traces of the target system. Instead, these sets grow gradually as the developer's understanding of the system increases. Refinement is a notion that captures this incremental activity.

Definition 8 (Refinement) Let S_1 and S_2 be two (high-level) SDs. We say that S_1 refines S_2, written $S_1 \preceq S_2$, if $L(S_1) \subseteq L(S_2)$. □

We show below in Theorem 4 that it is possible to express the notion of one SD refining another in terms of the languages of their positive and negative components. To do so, we first introduce the operators $\texttt{Safe}$ and $\texttt{Rej}$ over ω-languages. Let A be a Büchi automaton. Then $\texttt{Safe}(L(A)) = L(\texttt{Safe}(A))$ and $\texttt{Rej}(L(A)) = \texttt{Safe}(L(A)) - L(A)$. Intuitively, the ω-traces in $\texttt{Rej}(L(A))$ are those that could prevent the completion of an ω-trace in $L(A)$. Furthermore, it follows from Theorem 2 that $L(\texttt{Live}(A)) = \neg\texttt{Rej}(L(A))$.

Given a high-level SD S, let $P = \texttt{pos}(S)$ and $N = \texttt{neg}(S)$. Rewriting $L(S)$ in terms of $\texttt{Safe}$ and $\texttt{Rej}$ we obtain:

$$L(S) = \quad \texttt{Safe}(\neg L(N)) \cap \neg\,\texttt{Rej}(L(P))$$

Since $\cap$ and $\texttt{Safe}$ are monotonic operations and since $L(N)$ appears complemented, increasing the set $L(N)$ of negative traces of S makes $L(S)$ smaller, thereby preserving refinement. Similarly, if $\texttt{Rej}$ is monotonic, increasing $L(P)$ preserves refinement. Unfortunately, this

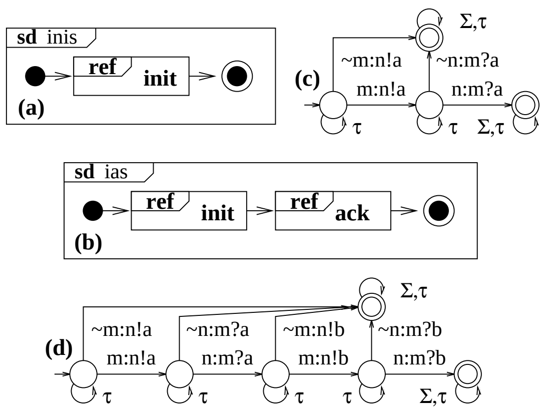

Figure 8. Refinement by sequential composition with strict sequencing

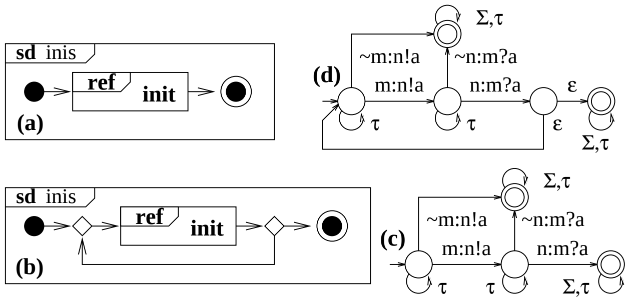

Figure 9. Refinement by star composition with strict sequencing.

is not always the case: if $L(P_1) \subseteq L(P_2)$ and $L(P_1) \cap \texttt{Rej}(L(P_2)) \neq \emptyset$ then $\texttt{Rej}(L(P_1)) \not\subseteq \texttt{Rej}(L(P_2))$. As a consequence, to make sure that refinement for positive traces holds, we have to make sure that $L(P_1) \cap \texttt{Rej}(L(P_2)) = \emptyset$.

Theorem 4 (Refinement) Let S_1 and S_2 be bounded SDs, and for $i = 1, 2$, $P_i = \texttt{pos}(S_i)$ and $N_i = \texttt{neg}(S_i)$. Then $S_1 \preceq S_2$ if the following hold: (i) $L(N_2) \subseteq L(N_1)$, (ii) $\texttt{Rej}(L(P_2)) \subseteq \texttt{Rej}(L(P_1))$.

Proof sketch Follows from the monotonicity of $\cap$ and `Safe`, and that `Rej` occurs complemented.

Corollary 1 (Ref1) Let S, T, U be bounded SDs and assume that S^+ is bounded, too and that $L(\texttt{pos}(S)) \cap \texttt{Rej}(L(\texttt{pos}(T))) = L(\texttt{pos}(T)) \cap \texttt{Rej}(L(\texttt{pos}(S))) = \emptyset$ Then: (1) $S\,U \preceq S$; (2) $S^+ \preceq S$; (3) $S{+}T \preceq S$ and $S{+}T \preceq T$; (4) $S\|T \preceq S$ and $S\|T \preceq T$.

Proof sketch Follows immediately from Theorem 4 and Definition 7. $S\|T$ translates to interleaving composition of corresponding automata.

Example 7 (Refinement by sequential composition) In Figures 8(a) and 8(b) we illustrate SD refinement by sequential composition with strict sequencing. Figures 8(c) and 8(d) show the corresponding automata. Obviously the automaton of Figure 8(d) refines the one of Figure 8(c). □

Example 8 (Refinement by star composition) In Figures 9(a) and 9(b) we illustrate SD refinement by star composition with strict sequencing. In Figures 9(c) and 9(d) we show the corresponding automata. Obviously the automaton of Figure 9(d) refines the one of Figure 9(c). □

Example 9 (Refinement by plus composition) In Figures 10(a) and 10(b) we illustrate SD refinement by plus composition. In Figures 10(c) and 10(d) we show the corresponding automata. Obviously the automaton of Figure 10(d) refines the one of Figure 10(c). □

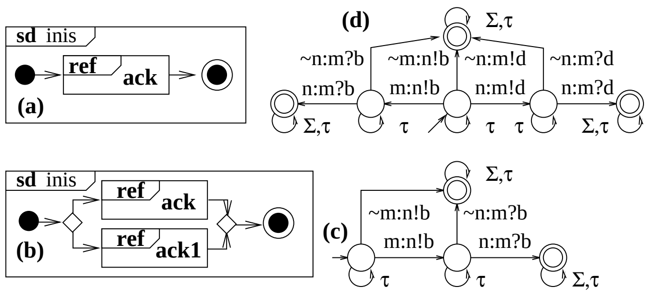

Figure 10. Refinement by plus composition.

Corollary 2 (Ref2) Let S, T and U be three bounded SDs and assume that SDs $(S)^*$ and $(T)^*$ are bounded, too. Then: (1) if $T \preceq U$ then $S\,T \preceq S\,U$; (2) if $S \preceq T$ then $(S)^* \preceq (T)^*$; (3) if $T \preceq U$ then $S{+}T \preceq S{+}U$ and $T{+}S \preceq U{+}S$ (4) if $T \preceq U$ then $S\|T \preceq S\|U$ and $T\|S \preceq U\|S$

Proof sketch Follows from Theorem 4 and Definition 7.

Example 10 (Sequential composition within star) In Figures 11(a) and 11(b) we illustrate SD refinement by sequential composition within star with strict sequencing. In Figures 10(c) and 11(d) we show the corresponding automata. Obviously the automaton of Figure 10(d) refines the one of Figure 10(c). □

6 Related Work

The process algebraic [20] and partial-order [5, 17, 22, 7, 3, 21, 14] semantics are to our knowledge not compositional with respect to refinement and take a closed-world approach; i.e., given a set of SDs, no new SDs may be added to it. However, the main interest of these semantics was in proving properties of distributed implementations and not in refinement.

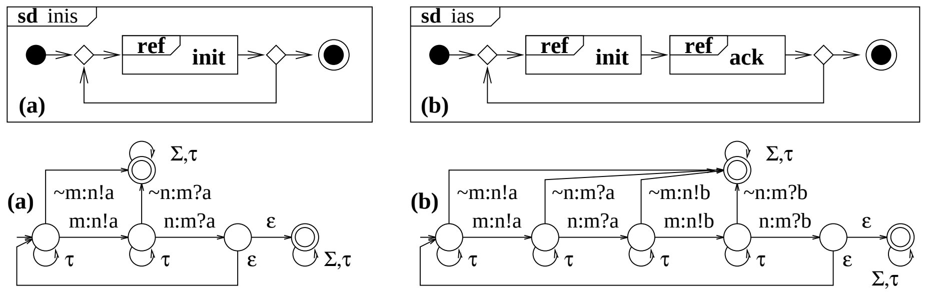

Figure 11. Refinement by sequential composition within star.

Live sequence charts [12, 15] are an elegant, alternative automata-theoretic solution to the refinement question, which relies on the optional/mandatory classification of SDs to define a compositional semantics. This classification, however, is also a departure from the good/bad classification of the UML standard.

Triggered message sequence charts [26, 27, 28] seem to be closest in spirit to our approach, as they classify SDs into prescriptive ("do this") and constraint-based ("don't do that"). However, their formal semantics and refinement notion is based on DeNicola and Hennessy's must preorder [23].

An interesting trace-based approach is advocated in STAIRS [16], where SD behaviors are classified into positive, negative and inconclusive. While intuitively appealing, the formal treatment is essentially restricted to a closed-world view of the positive behaviors.

The approach presented in [13] considers a closed-world semantics for MSCs. However, in order to define an assume-guarantee rule for determining whether a network of processes satisfies an MSC, a safety-liveness construction similar to our own is used to synthesize, for each process, a cooperating environment. Cooperation is understood as not violating the expectations of the associated processes, nor preventing them from making progress.

Other approaches, e.g. [19, 11], also approach the refinement problem by appealing to a semantics that departs from the closed-world one. None of these, in our opinion, achieves the simplicity and conceptual clarity of our safety-liveness semantics for SDs.

7. Conclusions

We have presented an automata-theoretic semantics for scenario-based descriptions (SDs) of reactive systems that solves in a simple and elegant way one of the main open questions about the UML 2.0 standard: how can one assign a precise meaning to a set of SDs without compromising refinement? Our semantics relies on the observation that bad (or negative) and good (or positive) SDs in the UML standard should be regarded as safety and liveness properties, respectively.

More formally, given a set of UML SDs, for each SD in the set, we construct two Büchi automata, one expressing safety and one expressing liveness, and take their product as the SD's semantics. Our approach has several salient properties. First, it targets all key aspects of the UML standard and is thus close to engineering practice. Secondly, it allows us to interpret SD refinement, e.g. by SD extension or addition of new SDs, in a canonical way as language inclusion. Finally, it allows us to regard a set of SDs as a pair of temporal properties to be satisfied by future implementations.

Our refinement results for SDs, as characterized by Corollary 2, provide a direct technique for checking SD refinement in a compositional setting: given two bounded SDs S and T such that S refines T, S continues to refine T in the contexts allowed by Corollary 2. When such compositional reasoning is not possible, our results support the development of a general-purpose model checker for property and refinement verification of SDs. The translation of high-level NFAs to positive and negative high-level NFAs can be performed in linear time. The translation of the negative/positive NFAs to their corresponding safety/liveness automata, as presented in Section 3, is exponential due to flattening [7]. An interesting question is whether flattening can be avoided in the case of synchronous concatenation. Intuitively, one could apply stuttering, chaos, safety and liveness closure to each node of the high-level SD; for each node, the closure is linear in the size of the associated NFA. Reachability analysis of the high-level safety/liveness product automaton is PSPACE-complete [6].

As future work, we plan to investigate how the semantic notions of mandatory-optional (from Live sequence charts [12, 15]), prescriptive-constraint-based (from Triggered message sequence charts [26, 27, 28]), and safety-liveness SDs are interrelated, and if it is advantageous to mix these classifications.

Acknowledgments: We thank Ingolf Krüger and Alexander Knapp for useful comments on a draft version of this paper.

References

[1] B. Alpern and F.B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[2] B. Alpern and F.B. Schneider. Recognizing safety and liveness. Technical Report TR 86-727, Cornell University, 1986.

[3] R. Alur, K. Etessami, and M. Yannakakis. Inference of message sequence charts. In *Proc. of 22nd Int. Conf. on Software Engineering*, 2000.

[4] R. Alur and R. Grosu. Modular refinement of hierarchic reactive machines. In *Proceedings of the 27th Annual ACM Symposium on Principles of Programming Languages*, pages 390–402, 2000.

[5] R. Alur, G.J. Holzmann, and D. Peled. An analyzer for message sequence charts. *Software Concepts and Tools*, 17(2):70–77, 1996.

[6] R. Alur, S. Kannan, and M. Yannakakis. Communicating hierarchical state machines. In *Automata, Languages and Programming, 26th International Colloquium*, pages 169–178. Springer, 1999.

[7] R. Alur and M. Yannakakis. Model checking of message sequence charts. In *CONCUR'99: Concurrency Theory, Tenth International Conference*, LNCS 1664, pages 114–129. Springer-Verlag, 1999.

[8] F. Belina, D. Hogrefe, and A. Sarma. *SDL with Applications from Protocol Specification*. Prentice Hall, 1991.

[9] H. Ben-Abdallah and S. Leue. Syntactic detection of process divergence and non-local choice in message sequence charts. In *Proceedings of the Second International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, 1997.

[10] G. Booch, I. Jacobson, and J. Rumbaugh. *Unified Modeling Language User Guide*. Addison Wesley, 1997.

[11] M.V. Cengarle and A. Knapp. UML 2.0 Interactions: Semantics and refinement. In *Proc. of the 3rd Int. Workshop on Critical Systems Development with UML (CSDUML'04)*, pages 85–99, 2004.

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

[13] B. Finkbeiner and I. Krüger. Using message sequence charts for component-based formal verification. In *Specification and Verification of Component-Based Systems (SAVCBS). Workshop at OOPSLA*, 2001.

[14] E. Gunter, A. Muscholl, and D. Peled. Compositional message sequence charts. In *Proc. of TACAS'01*, pages 496–511. Lecture Notes in Computer Science 2031, 2001.

[15] D. Harel and H. Kugler. Synthesizing state-based object systems from LSC specifications. *Lecture Notes in Computer Science*, 2088, 2001.

[16] Ø. Haugen and K. Stølen. Stairs - steps to analyze interactions with refinement semantics. In *Proceedings of the Sixth Intenational Conference on the Unified Modelin Language (UML)*, 2003.

[17] G.J. Holzmann, D.A. Peled, and M.H. Redberg. Design tools for requirements engineering. *Lucent Bell Labs Technical Journal*, 2(1):86–95, 1997.

[18] ITU-TS. Recommendation Z.120: Message Sequence Chart (MSC). Geneva. 1996.

[19] I. Krüger. *Distributed System Design with Message Sequence Charts*. PhD thesis, Technical Univ. of Munich, 2000.

[20] S. Mauw and M.A. Reniers. An algebraic semantics of basic message sequence charts. *Computer Journal*, 37, 1994.

[21] A. Muscholl and D. Peled. Analyzing message sequence charts. In *Proc. of SAM'00, Grenoble*, 2000.

[22] A. Muscholl, D. Peled, and Z. Su. Deciding properties of message sequence charts. In *Foundations of Software Science and Computation Structures*, 1998.

[23] R. De Nicola and M.C.B. Hennessy. Testing equivalence for processes. *Theoretical Computers Science*, 34:83–133, 1984.

[24] OMG-PTC/03-08-02. UML 2.0 Superstructure Specification. 2003.

[25] B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley and Sons, Inc., 1994.

[26] B. Sengupta and R. Cleaveland. Refinement-based requirements modeling using triggered message sequence charts. In *Proc. of the 11th IEEE International Requirement Engineering Conference*. IEEE Computer Society Press, 2003.

[27] B. Sengupta and R. Cleaveland. Towards formal but flexible scenarios. In *Proc. of the 2nd International Workshop on Scenarios and State Machines: Models, Algorithms and Tools*, 2003.

[28] B. Sengupta and R. Cleaveland. Trim: A tool for triggered message sequence charts. In *Proc. of the Computer-Aided Verification Conference*, pages 106–109. Lecture Notes in Computer Science, volume 2725, 2003.