Semantics-based Refinement of Mandatory Behavior of Sequence Diagrams

Lunjin Lu and Dae-Kyoo Kim
Department of Computer Science and Engineering
Oakland University
Rochester, MI 48309

December 16, 2010
## Contents

1 Introduction  
   1.1 Motivation .......................... 2  
   1.2 Contributions .......................... 3

2 Related Work  

3 Abstract Syntax  

4 Semantics  
   4.1 Semantic Domain .......................... 8  
   4.2 Semantic function .......................... 10  
      4.2.1 Observable and unobservable events .......................... 10  
      4.2.2 Strict fragments .......................... 10  
      4.2.3 Critical fragments .......................... 10  
      4.2.4 Alt fragments .......................... 10  
      4.2.5 Opt fragments .......................... 11  
      4.2.6 Par fragments .......................... 11  
      4.2.7 Block fragments .......................... 11  
      4.2.8 Seq fragments .......................... 12  
      4.2.9 Loop fragments .......................... 13  
      4.2.10 Properties of semantics .......................... 13

5 Refinement  

6 Conclusion and Future Work  


Abstract

Sequence diagrams are a widely used design notation for describing software behavior. Many reusable software artifacts such as design patterns and design aspects make use of sequence diagrams to describe interaction behavior. When a pattern or an aspect is reused in an application, it is important to ensure that the sequence diagrams for the application correctly refines the corresponding sequence diagrams for the pattern or aspect. However, reasoning about refinement of sequence diagrams has not been addressed adequately. In this paper, we focus on refinement of required behavior specified by a UML sequence diagram. A novel trace semantics is given that captures precisely required behavior specified by a sequence diagram and a refinement relation between sequence diagrams is formalized based on the semantics. Properties of the trace semantics and the refinement relation are studied.
Figure 1: Sequence Diagrams for the Running Example

1 Introduction

UML sequence diagrams [51] and their predecessors Message sequence charts [55] are specification languages that are widely used during software development. A sequence diagram (SD) describes inter-object/inter-process behavior of a system in a graphical manner. It shows as parallel vertical lines different objects or processes that communicate with each other via messages that are shown as horizontal arrows. Each message has an associated sending event and an associated receiving event. Events are basic behavioral constructs of UML SDs. They can be combined to form larger behavioral constructs called fragments. A fragment is either an event or formed of an interaction operator, one or two operands which may be themselves fragments and an optional condition. It involves a collection of lifelines and is formed of events and smaller fragments. In this paper, we shall use the terms SD and fragment interchangeably.

Example 1.1 We shall use SDs in Figure 1 as a running example. In SD Login, the alt fragment is labelled a and the sending and receiving events for a message are labelled with two consecutive numbers. Let $e_i$ abbreviate the event labelled $i$. For instance, $e_1$ abbreviates !id the sending event of message id and $e_2$ abbreviates ?id the receiving event of message id omitting the sender and the receiver of the message. SD Login may be thought of as a pattern for a user to sign in to get a service from a server. The user provides to the server his user-id id and password pwd. The server checks if the user-id and password are correct using a system variable OK to indicate the result. If OK equals true then the user issues a command cmd to the server.
1.1 Motivation

Software design is an iterative process. Starting with an initial design model, a series of design models are obtained, each of which refines its predecessor. This process is applied to behavioral models as well as structural models. Each immediate model needs to be verified against its predecessor. The need for such verification also arises when software evolves. A fundamental issue arising from using SDs is whether one SD model refines its predecessor in that it possesses the behavior required by the predecessor and at the same time rejects the behavior prohibited by its predecessor.

Example 1.2 Consider SD Login again. Let \( t = e_1 e_2 e_3 e_4 e_5 e_6 \) and \( t' = e_1 e_3 e_2 e_4 e_5 e_6 \). Let \( r = t[OK = true] e_7 e_8 \) and \( r' = t'[OK = true] e_7 e_8 \). SD Login specifies two alternative obligations \( O = \{ r, t \} \) and \( O' = \{ r', t' \} \). A system satisfies SD Login if it fulfills one of them. A system fulfills \( O \) if it has runs that produce the trace \( r \) and runs that produce the trace \( t \). A system fulfilling \( O' \) can be described similarly. That guard conditions occur in traces shall be explained later.

Software development can greatly benefit from reusing existing artifacts including architectural patterns, design patterns, design aspects, software components and code. Many reusable artifacts make use of SDs to specify interaction behavior. In aspect-oriented software development, design models may be developed by composing aspects with primary models, which involves composing SDs from aspects [23]. It is necessary to guarantee that the composed SD conforms to the component SDs. In pattern-based development, an SD developed by the designer must conform to the SDs of a design pattern [24]. Conformance verification of one SD to another SD can be reduced to refinement verification by hiding the messages and lifelines introduced during reuse and undoing name changes performed during reuse.

An SD is partial in that it describes a number of alternative obligations that an implementation may choose to fulfill. For instance, the fragment operator \( \text{par} \) does not mandate that an implementation must be distributed, concurrent or multi-threaded. It rather indicates that the implementation can realize any interleaving of the behaviors of its operands. When an SD is refined, it is made more defined in that the number of alternatives is reduced.

Example 1.3 This example shows that SD refinement verification is essential in verifying correctness of SD reuses. The SD Login2 describes a sign-in interaction for a customer of a brokerage and can be obtained by reusing SD Login as follows. Firstly, the developer renames user to customer, server to brokerage, id to acc, pwd to pin, chk to chkP, OK to pOK, cmd to trade. He then eliminates non-determinism by requiring that \(?\text{acc}\) occurs before \(!\text{pin}\). He also introduces a new system variable \( k\text{OK} \) and two new messages key and chkK which produces output \( k\text{OK} \). The condition for the opt fragment has also been strengthened.

That SD Login2 is a correct reuse of SD Login can be informally checked as follows. SD Login3 may be obtained from SD Login2 by hiding messages key and chkK(replacing them by tau which stands for all unobservable messages), using
default value true for kOK and changing the names back. Moreover, SD Login3 is the same as SD Login except that in SD Login3, ?id must occur before !pwd while they can occur in any order in SD SD Login and that SD Login3 may send and receive unobservable messages. Formally, SD Login3 specifies one obligation which is \( O \) given in Example 1.2. Any system satisfying SD Login3 fulfills \( O \) - one of the two alternative obligations of SD Login. In other words, SD Login3 refines SD Login.

We have shown that verification of SD refinement is important in software development such as in model driven, pattern-based and aspect-based software development. Refinement verification requires a formalization of a refinement relation between SDs which in turn requires a formal trace semantics that captures precisely required behaviors of SDs.

1.2 Contributions

In existing behavioral semantics [3, 11, 14, 32, 49], an SD denotes a set of all possible traces that the specified system may produce and a set of proscribed traces that the specified system must not produce. They are useful as a semantic base for verifying SDs against safety properties. However, they are not useful as a semantic base for defining a refinement relation between SDs since they do not tell which possible traces are required in that the specified system must produce. Semantics of SDs has also been given in terms of translations to these state machines such as statecharts, Büchi automata, labelled transition systems and modal transition systems. Since a state machine accepts one set of positive traces and one set of negative traces in the case of a Büchi automata and a modal transition system, a semantics based on translation to state machines does not capture precisely alternative but incomparable minimal required behaviors specified by an SD. In addition, translations from SDs to these state machine models do not deal with critical regions and guard conditions. Thus, results on refinement of these state machine models do not carry over to SDs.

This paper gives a trace semantics that characterizes required behaviors specified in an SD and formalizes a refinement relationship between SDs. Refinement is defined in terms of a simulation relation between traces. The notion of one trace simulating another will be made clear later. Roughly speaking, a trace \( t_1 \) simulates another trace \( t_2 \) if all events in \( t_2 \) are simulated in \( t_1 \) in the order in which they occur and there are no observable events in \( t_1 \) other than those that simulate events in \( t_2 \). An SD \( D_1 \) refines another \( D_2 \) if an implementation of \( D_1 \) is also an implementation of \( D_2 \). In other words, \( D_1 \) preserves the required behaviors of \( D_2 \) but may specify more required behaviors. These concepts will be made clearer in Section 5. The main contributions of this work are as follows.

- A novel trace semantics is formulated for a subset of UML SDs. Unlike the trace semantics proposed in literature [11, 32, 19] that capture possible behaviors of SDs, our trace semantics captures precisely required behaviors of SDs and forms a basis for a semantics based refinement relation. While
those trace semantics for possible behaviors of SDs ignore guard conditions, our trace semantics encodes guard conditions in SDs as elements of traces. This is essential to ensure soundness of refinement as discussed in Section 2. The semantics possesses substitutivity which is not enjoyed by the trace semantics proposed in literature [11, 32, 49]. Substitutivity guarantees that a component of an SD can be replaced with a semantically equivalent component without changing the semantics of the SD.

- A refinement relation between SDs is defined based on the semantics. The refinement relation is transitive, implying that the correctness of a multi-step refinement can be checked by verifying the correctness of each individual refinement step. The refinement relation also possesses substitutivity, implying that an SD can be refined compositionally.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 presents an abstract syntax for SDs and Section 4 defines the trace semantics. Section 5 defines the refinement relation. Section 6 concludes. Proofs can be found in appendix.

2 Related Work

This section discusses work related to semantics and refinement of SDs. Since we are concerned with behavioral refinement reasoning for SDs, this section focuses on behavioral semantics and semantic-based refinement of SDs and their variants.

In [10, 11] and [49], the semantics of an SD is a pair consisting of a set of positive traces and a set of negative traces. Haugen et al. [32] define the semantics of an SD as a set of obligations all of which must be fulfilled. Each obligation is a pair consisting of a set of positive traces and a set of negative traces. Without the fragment operator \( xalt \) which they introduce to capture the mandatory non-determinism, the semantics of an SD contains a single obligation and is equivalent to that of [49]. Lund and Stølen provide an operational semantics for UML SDs [41] which is sound and complete with respect to the trace semantics of [32]. There is no discussion on refinement in [41]. Refinement in [11, 32, 49] is defined as eliminating positive traces and making them proscribed. Under this interpretation, the system is only required to have one of positive traces, which is problematic as shown below. For SD Login, the set of positive traces is \( \{te_7e_8, t' e_7e_8, t, t'\} \) where \( t \) and \( t' \) are given in Example 1.2. The set of positive traces does not capture precisely the required behavior of SD Login. As shown in Example 1.2, the specified system does not need to produce all positive traces in order to satisfy SD Login. It only has to produce \( t \) and \( te_7e_8 \) or \( t' \) and \( t' e_7e_8 \).

A logical semantics for basic SDs is presented in [14]. A basic SD \( D \) has only finite number of finite traces. The semantics of \( D \) is a temporal logic formulae with freeze quantifier [13]. The semantics captures a single set of possible traces and applies to a small subset of SDs. SDs are formalized in [4] as PVS theories.
that specify a set of possible traces for each object in the system. Refinement is not discussed in [3, 14].

The above semantics [3, 14, 11, 32, 49] associate an SD with a set of possible traces. They are useful for verification of SDs against safety properties of SDs such as dead-lock freedom [2]. However, they are inadequate for SD conformance reasoning in several aspects. Firstly, they do not distinguish required behaviors from optional behaviors as pointed out in [47]. Secondly, they ignore guard conditions, which compromises soundness of conformance reasoning. For instance, let $D_1 = alt(c, !m, !n)$ and $D_2 = !m$ then ignoring constraints would assign two traces $!m$ and $!n$ to $D_1$ and one trace $!m$ to $D_2$ and lead to a false conclusion that $D_1$ possesses all the required behaviors of $D_2$. In fact, $D_2$ requires the specified system to produce $!m$ in all runs whilst $D_1$ only requires the specified system to produce $!m$ in those runs that start with system states in which the condition $c$ holds. Thirdly, they do not deal with critical regions adequately. All but one [49] of above mentioned semantics are defined for SDs with critical regions. Semantics in [49] does not possess substitutivity in the presence of critical regions. Let $D_1$ be critical(\text{strict}(!a, !b)) and $D_2$ strict(\text{strict}(!a, !b)). Then $D_1$ and $D_2$ have the same meaning according to [49] but $\text{par}(D_1, !c)$ and $\text{par}(D_2, !c)$ do not.

The semantics in [40] captures the effect of a synchronous message specified by an SD on logical properties of the specified system. It abstracts away too much details of interactions and hence is not amiable to analysis of trace properties including behavioral conformance. The same applies to logical semantics for MSCs in [17].

SDs and their predecessor MSCs have been studied via translation to automata, process calculi and other formalisms. Mauw and Reniers [42] present a process-based semantics for basic MSCs (in short bMSCs) which are MSCs without fragment operators. A bMSC is translated to a process in ACP [5]. Chen et al. provide semantics for bMSCs with data [12] by translating a bMSC to a process in a variant of CCS [43].

Whittle and Schumann generate statecharts from a collection of UML SDs and a collection of OCL constraints [56]. Ziadi et al. translate a scenario specification in UML SDs into statecharts [55]. As noted in [55], such translations result in statecharts whose behaviors include all behaviors of the scenario but also include extra behaviors that are not required by the scenarios. Hammal defines the semantics of an SD as an automaton whose states are maps from objects to traces and whose edges are labelled with events [27]. To obtain a finite automaton, possible traces that contain the same set of events are identified. An SD has also been translated to a Petri net (e.g., [8, 17, 19]) with lifelines translated to processes, actions to transitions and messages to communication places and to abstract state machines (e.g., [9, 37, 57]). Refinement is not considered in [8, 9, 17, 19, 27, 37, 50, 57, 58].

Grosu and Smolka give safety and liveness semantics for SDs in terms of Büchi automata [26]. Refinement is defined using as set containment. Knapp et al. [39] translates SDs to automata for model checking using the SPIN model checker. Alur et al. translate MSCs to automata for checking against safety
properties such as dead-lock freedom [1, 2]. Refinement is not discussed in [1, 2, 36].

Uchitel et. al [53] synthesize a labelled transition system (LTS) from MSC scenarios and use it to detect scenarios that are implied by positive and negative scenarios [54]. An LTS is a finite state machine with each transition labelled with an action (event) or \( \tau \). In [52], modal transition systems are synthesized from properties in Fluent Linear Temporal Logic [25] and traces of scenarios. A modal transition system (MTS) [52] is a generalization of an LTS. An MTS has two transition relations, one describing possible transitions and the other required transitions. Possible transitions that are not required can be made required or proscribed in later phase of model development. Sibay et al. [48] translate existential LSCs to MTSs. Krka et al. synthesize MTSs from a set of basic SDs and OCL constraints [38] - one MTS for each component of the specified system. Refinement of MTSs has been studied in [20, 21, 22].

Defining semantics of SDs via translation allows us to leverage established results in other areas to analyze SDs. Bisimulation [20, 21, 22, 45], the must preorder [33, 34, 44] and the failures preorder [33] are close relatives [15, 18] and have been used to define refinement of automata and processes. Refinement in bisimulation, must and failures preorder semantics keeps required traces while decreasing non-determinism. However, the translation algorithms are limited to small subsets of SDs and ignore essential features of SDs. For instance, they all ignore critical regions and they all except [36] ignore guard conditions. Note that guard conditions cannot be disregarded for conformance reasoning as pointed out in the previous section. It is difficult to extend these translation algorithms to include critical regions.

There have been work on empowering SDs and MSCs with more language constructs. We now briefly discuss those extensions that are more influential. Live Sequence Charts (LSCs) [16, 31] are introduced to capture existential and universal modalities. LSCs have been subject to as much study as MSCs and SDs (see references in [6, 31, 30]) and have had synergistic impact on SDs. Modal Sequence Diagrams (MSD) [28, 29] are an extension of SDs to deal with challenges with negate and assert operators. In MSD, a modal stereotype is attached to an interaction fragment to specify whether it describes a hot (mandatory) or cold (possible) behavior. Triggered MSCs (TMSCs) [47] are introduced to catch conditional scenarios. Each instance in a TMSC has a trigger and an action. A system satisfies a TMSC if whenever it exhibits the behavior described by the trigger of an instance, its subsequent behavior is limited to the behavior described by the action of the instance. The meaning of a TMSC is an acceptance tree [33, 34] which maps a trace \( w \) to an acceptance set which is a measure of non-determinism of the system after exhibiting \( w \). The must preorder has been adapted for Triggered MSCs (TMSCs) [47]. TMSCs do not have a construct similar to critical or use guard conditions [47].

In summary, there does not exist a suitable semantic basis refinement of required behavior of SDs because direct style semantics do not precisely capture required behaviors of SDs and translations to other formalisms disregard essential features of SDs such as guard conditions and critical regions.
3 Abstract Syntax

An SD specifies runtime behaviors of a system in a graphical manner. It shows as parallel vertical lines different objects or processes that communicate with each other via messages that are shown as horizontal arrows. A simple diagram which does not have any combined fragment has been modelled as a partial order on event occurrences [8]. Intuitively, \( e_1 \preceq e_2 \) indicates that \( e_1 \) occurs no later than \( e_2 \). Since \( \preceq \) is asymmetric, there is unique irreflexive and non-transitive relation \( \rightarrow \) such that \( \sim = \rightarrow^* \) where \( * \) is the reflexive and transitive closure operator. The relation \( \rightarrow \) is the transitive and reflexive reduction of \( \sim \) and we call it a strict sequencing order.

Events are basic behavioral constructs of UML SDs. They can be combined to form larger behavioral constructs called fragments. A fragment is formed of an interaction operator, one or two operands which may be themselves fragments and an optional condition. It involves a collection of lifelines and is formed of events and smaller fragments. In this sense, an event is a primitive fragment.

In this work, we do not consider interaction operators ignore, consider, assert, neg or break. Since we are concerned with checking if the behaviors described by one model are found in another model, ignore and consider fragments play no role and thus can be removed. Despite prior effort in clarifying assert and neg operators [29, 50], no commonly accepted interpretation for these operators has been established. The UML 2.0 standard states that “a break fragment is a breaking scenario that is performed instead of the remainder of its enclosing fragment”. It is not clear whether the enclosing fragment means the innermost enclosing fragment or the innermost loop fragment. We assume that all references to SDs through interaction operator ref have been eliminated via syntactic unfolding since SDs are non-recursive.

Let \( \text{Name} \) be a denumerable set of names of messages, lifelines, system variables and values. An event \( e \in \text{Ev} \) has the following structure. An event sending a message with name \( N \in \text{Name} \), sender \( S \in \text{Name} \), receiver \( R \in \text{Name} \), parameter list \( P \subseteq \text{Name} \) is written as \( (!, N(P), S, R) \) or \( !N(S, R, P) \), and the corresponding receiving event \( (? , N(P), S, R) \) or \( ?N(S, R, P) \). We shall simply write \( !N(P) \) or \( ?N(P) \) when the sender and receiver are clear from context. We abstract from details of guard conditions \( c \in \text{Cond} \) and require that the collection of guard conditions is closed under classical logical negation (\( \overline{c} \)), conjunction (\( \land c \)) and disjunction (\( \lor c \)) operations. We write \( c_1 \models c_2 \) iff \( c_2 \) is true in all value assignments in which \( c_1 \) is true. Other primitive syntactic entities are labels \( \ell \in \text{Lab} \) and \( \tau \) representing unobservable events. The abstract syntax for SDs in \( \text{Sdl} \) is given below.

\[
D ::= \tau \mid e \mid \text{opt}(c, D_1) \mid \text{alt}(c, D_1, D_2) \mid \text{loop}(c, D_1) \mid \text{critical}(D_1) \mid \text{par}(D_1, D_2) \mid \text{strict}(D_1, D_2) \\
| \text{seq}(D_1, D_2) \mid \text{block}(L, \iota, \rightarrow)
\]

where the interaction operator block is introduced to structure operands of other interaction operators, \( L \) is a non-empty set of labels, \( \iota \) is a mapping from \( L \) to
Example 3.1 SD Login in Figure 7 is represented in the abstract syntax as Login = block( {1..6, a}, {i ↦ e_i | 1 ≤ i ≤ 6} ∪ {a ↦ D_a}, ¬o ) where ¬o = { ⟨1, 2⟩, ⟨1, 3⟩, ⟨3, 4⟩, ⟨2, 4⟩, ⟨4, 5⟩, ⟨5, 6⟩, ⟨6, a⟩ } and D_a = opt(OK = true, block( {7, 8}, {7 ↦ e_7, 8 ↦ e_8} )).

4 Semantics

This section presents the semantic domain and the semantic equations for the trace semantics. We first introduce auxiliary notations and operations used in the construction of the domain and the definition of the semantic equations.

The domain of a function f is denoted \( \text{dom}(f) \) and its image \( \text{image}(f) = \{ f(x) | x \in \text{dom}(f) \} \). Let \( \Sigma \) be an alphabet. \( \Sigma^* \) denotes the set of all strings over \( \Sigma \). A language \( L \) over \( \Sigma \) is a set of strings over \( \Sigma \). The Kleene closure of \( L \) is denoted \( L^* \). Let \( \omega \in \Sigma^* \). The length of \( \omega \) is denoted \( |\omega| \). The string \( \omega \) may be thought of as a function from \( \{0..|\omega| − 1\} \) to \( \Sigma \). The \( i \)-th element in \( \omega \) is written as \( \omega(i) \). The interleave of two strings is the set of strings obtained by interleaving the two strings in all possible ways. Let \( x, y \in \Sigma \) and \( \mu, \nu \in \Sigma^* \). This definition of the interleave operator is due to [49].

\[
\varepsilon \parallel \mu = \mu \parallel \varepsilon = \mu \\
x\mu \parallel y\nu = \{ x \} \cdot (\mu \parallel y\nu) \cup \{ y \} \cdot (x\mu \parallel v)
\]

where \( \cdot \) is the language concatenation operator.

Let \( \oplus \) be a binary operation on domain \( S \). Then \( \oplus^\sharp \) defined below is a binary operation on the power set \( \wp(S) \) of \( S \).

\[
X \oplus^\sharp Y = \{ x \oplus y | x \in X \land y \in Y \}
\]

For instance, \( \cap^\sharp, \cup^\sharp \) and \( \cdot^\sharp \) are respectively pairwise wise set intersection, set union and language concatenation.

A rewriting relation \( \Rightarrow \) on a set \( A \) is a binary relation on \( A \). An element \( a \in A \) is a normal form if there is no \( a' \in A \) such that \( a \Rightarrow a' \). A rewriting relation \( \Rightarrow \) is called finitely terminating iff it has no infinite descending chain \( a_0 \Rightarrow a_1 \Rightarrow a_2 \cdots \). It is called confluent if, for each \( x, u, w \in A \) such that \( x \Rightarrow^* u \) and \( x \Rightarrow^* w \), there is a \( z \) such that \( u \Rightarrow^* z \) and \( w \Rightarrow^* z \). \( \Rightarrow \) is called convergent if it is both confluent and finitely terminating. If \( \Rightarrow \) is convergent, then for each \( a \) there is a unique normal form denoted \( a_{\Rightarrow} \), such that \( a \Rightarrow^* a_{\Rightarrow} \) [4].

4.1 Semantic Domain

An SD is a partial specification of required and prohibited behaviors of an application. This paper is concerned only with required behaviors. Consider this simple SD.
An implementation that produces the trace !m?m!n?n satisfies this specification; another implementation that produces the trace !m!n?m?n also satisfies the specification. Thus, the SD specifies two alternative minimum obligations $O_1 = \{!m?m!n?n\}$ and $O_2 = \{!m?m!n\}$. Of course, an implementation that non-deterministically produces one of the two traces also satisfies the specification. However, the obligation $O_1 \cup O_2$ is redundant since it includes $O_1$ and $O_2$ as proper subsets and hence not minimum. An obligation may contain more than one traces. Once an obligation is chosen, all traces in the obligation are required in that for each trace $t$ in the obligation, there is an interaction that produces $t$. For instance, $alt(v = ok, !m, !n)$ has one obligation with two traces $\{(v = ok)!m, (v \neq ok)!n\}$. A condition such as $(v = ok)$ is a guard for the rest of the trace, meaning that the rest is exhibited only if the condition evaluates to true. The above obligation requires an implementation to produce !m if $(v = ok)$ is true when it runs and to produce !n if $(v \neq ok)$ is true.

A critical fragment requires that there isn’t any intervening event between two consecutive events in the region. For instance, a critical fragment in the specification of a telephone service may specify that after receiving a 911 call from a user, the operator must forward the call to the emergency service without any interruption. We wrap a sub-trace from a critical fragment and treat it as a single token. A trace is a sequence of tokens which are either events, guard conditions or critical segments $[\sigma]$ where $\sigma$ is a sequence of events and guard conditions. A critical segment $[\sigma]$ protects the sub-trace $\sigma$ from interference. Occurring in a trace, $[\sigma]$ will be treated as atomic when the trace is combined with other traces through interleaving and weak sequencing. The domains of tokens and traces are respectively

$$Tk = \text{Evt} \cup \text{End} \cup ([\text{Evt} \cup \text{End}])^*$$

$$Tr = Tk^*$$

Consider two required traces $cm$ and $\neg_c(c)m$. Then message $m$ is always sent since it is always the case that either $c$ or $\neg_c(c)$ holds. Occurring in an obligation, they represent an unnecessary decision point. Define $\Rightarrow$ by $O \cup \{ac^\beta, ac'^\beta\} \Rightarrow O \cup \{a^\beta\}$ if $c \vee c' \models true$. Then $\Rightarrow$ is a convergent rewriting relation on obligations. So, function $fold(O) = O_\Rightarrow$ is well defined. The function is lifted to sets of obligations as $fold(M) = \{fold(O) \mid O \in M\}$. Define

$$\downarrow M = \{O \in M \mid \neg \exists O' \in M.(O' \subset O)\}$$

The operation $\downarrow$ removes redundancy from its argument. The semantic domain is

$$\text{Sem} = \{M \in \varphi(\psi(Tr)) \mid M = \downarrow M \land fold(M) = M\}$$
4.2 Semantic function

The semantics of an SD $D$ is denoted $[D]$. It is defined as the least solution to a system of semantic equations.

4.2.1 Observable and unobservable events.

We are now ready for defining semantic equations. Observable and unobservable events have obvious semantics:

$$[e] = \{\{e\}\}$$
$$[r] = \{\{e\}\}$$

4.2.2 Strict fragments.

The concatenation of an obligation $O_1$ in $[D_1]$ and an obligation $O_2$ in $[D_2]$ gives rise to an obligation $O$ in $[\text{strict}(D_1, D_2)]$.

$$[\text{strict}(D_1, D_2)] = \downarrow \text{fold}([D_1] \circ [D_2])$$

where $\text{fold}$ is applied to eliminate unnecessary decision points and $\downarrow$ to remove redundant obligations. These functions are also applied in semantic functions for other kinds of fragment.

4.2.3 Critical fragments.

The semantics of critical($D$) is defined by unwrapping all critical segments in traces of $D$ and wrapping the result in $/llparenthesis \cdot /rrparenthesis$. Define $\lhd$ by $\alpha /llparenthesis \sigma /rrparenthesis \beta \lhd \alpha \sigma \beta$. A rewriting step with $\lhd$ exposes a sub-trace protected by a critical segment. Since $\lhd$ is convergent, $\text{unwrap}((\sigma)) = \sigma$ is a well defined function. Define $\text{wrap}(\sigma) = /llparenthesis \sigma /rrparenthesis$. The function $\text{unwrap}$ is lifted to sets of sets as $\text{unwrap}(M) = \{\text{unwrap}(\omega) \mid \omega \in \mathcal{O} \mid \mathcal{O} \in M\}$. Lift $\text{wrap}$ in the same way. The semantics of critical($D$) is defined

$$[\text{critical}(D)] = \text{wrap}(\downarrow \text{fold}(\text{unwrap}([D])))$$

For instance, $[\text{critical}(\text{strict}(e, f))] = \{\{\langle e, f \rangle\}\}$ where $e, f \in \text{Evt}$.

4.2.4 Alt fragments.

Alt, opt and loop fragments introduce guards to traces. Define $c \triangleright e = e$ and $c \triangleright \sigma = c \sigma$ where $c \in \text{Cnd}$ and $\sigma \in \text{Tr}$ such that $\sigma \neq e$. Let $c \triangleright M = \{c \triangleright \sigma \mid \sigma \in \mathcal{O} \mid \mathcal{O} \in M\}$. The semantics of $\text{alt}(c, D_1, D_2)$ is obtained by guarding traces of $D_1$ with $c$ and those of $D_2$ with $\neg_c(c)$:

$$[\text{alt}(c, D_1, D_2)] = \downarrow \text{fold}((c \triangleright [D_1]) \cup /llbracket (\neg_c(c) \triangleright [D_2]) /rrbracket)$$

Let $e, f, g \in \text{Evt}$ and $c \in \text{Cnd}$. Then $[\text{alt}(c, e, e)] = \{\{e\}\}$ and $[\text{alt}(c, \text{strict}(e, f), g)] = \{\{cef, c'g\}\}$ where $c' = \neg_c(c)$.
4.2.5 Opt fragments.

The semantics of \( \text{opt}(c, D) \) is obtained similarly:

\[ \text{[opt}(c, D)\text{]} = \downarrow \text{fold}((c \triangleright^\sharp [D]) \cup^\sharp \{\{\epsilon\}\}) \]

For instance, \( \text{[opt}(c, \tau)\text{]} = \{\{\epsilon\}\} \).

4.2.6 Par fragments.

Consider parallel interleave \( \text{par}(D_1, D_2) \) of two sub-diagrams. Let \( \mathcal{O}_1 \) be an obligation of \( D_1 \) and \( \mathcal{O}_2 \) of \( D_2 \). Parallel interleaving produces a set of alternative obligations from \( \mathcal{O}_1 \) and \( \mathcal{O}_2 \). Define

\[ \mathcal{O}_1 \parallel \mathcal{O}_2 = \{ \sigma \mid \forall \sigma_1 \in \mathcal{O}_1, \forall \sigma_2 \in \mathcal{O}_2. \exists \sigma \in \mathcal{O}. (\sigma \in \sigma_1 \parallel \sigma_2) \} \]

\( \mathcal{O}_1 \parallel \mathcal{O}_2 \) may have redundant obligations. Put \( \mathcal{O}_1 = \{ e_1 \} \) and \( \mathcal{O}_2 = \{ e_2 \} \). Then \( \mathcal{O}_1 \parallel \mathcal{O}_2 = \{ \{e_1, e_2\}, \{e_2, e_1\}, \{e_1, e_2, e_1\} \} \). The obligation \( \{e_1, e_2, e_1\} \) is redundant. The meaning of \( \text{par}(D_1, D_2) \) is defined

\[ \text{[par}(D_1, D_2)\text{]} = [D_1] \parallel^\ast [D_2] \]

where \( \mathcal{M}_1 \parallel^\ast \mathcal{M}_2 = \downarrow (\cup_{\mathcal{O}_1 \in \mathcal{M}_1, \mathcal{O}_2 \in \mathcal{M}_2} \mathcal{O}_1 \parallel \mathcal{O}_2) \).

**Example 4.1** Let \( c \in \text{Cond} \) and \( f, g, h \in \text{Event} \). Then

\[ \text{[par}(\text{alt}(c, f, g), h)\text{]} = \{\{cf, c'g\}\} \parallel^\ast \{\{h\}\} \]

\[ = \left\{ \begin{array}{l}
\{hcf, hc'g\}, \{hcf, c'gh\}, \\
\{hcf, h'cg\}, \{hcf, c'gh\}, \\
\{cfh, c'gh\}, \{cfh, c'gh\}, \{cfh, c'gh\} \end{array} \right\} \]

where \( c' = \neg c \).

4.2.7 Block fragments.

To enforce sequencing orders, we tag tokens in a trace generated from a fragment. Function \( \text{tag} \) labels each token in a trace by a given label: \( \text{tag}(\epsilon, \ell) = \epsilon \) and \( \text{tag}(t \cdot \sigma, \ell) = (t, \ell) \cdot \text{tag}(\sigma, \ell) \). Function \( \text{untag} \) does the opposite and is defined \( \text{untag}(\epsilon) = \epsilon \) and \( \text{untag}( (t, \ell) \cdot \sigma) = t \cdot \text{untag}(\sigma) \). \text{tag} and \text{untag} are extended to sets of sets in the same way as \( \text{unwrap} \). Function \( \text{lifelines} \) maps a token to the set of the lifelines associated with the token. A sending event is associated with the sender, a receiving event with the receiver and a critical segment with all the lifelines associated with the events in the critical segment. \( \text{lifelines} \) is defined by \( \text{lifelines}(!, N(P, S, R)) = \{S\} \), \( \text{lifelines}(?, N(P, S, R)) = \{R\} \), \( \text{lifelines}(\ell | \sigma) = \bigcup_{i \in \text{dom}(\sigma)} \text{lifelines}(\sigma(i)) \) and \( \text{lifelines}(c) = \emptyset \). Let \( \text{lb} \) be the function that returns the label of a tagged token. Then \( \text{lb}( (t, \ell) ) = \ell \). Relation \( \sim \) relates two tagged tokens iff they share lifelines: \( (t_1, \ell_1) \sim (t_2, \ell_2) \) iff \( \text{lifelines}(t_1) \cap \text{lifelines}(t_2) \neq \emptyset \). Let \( \mathbb{T} = (\mathbb{T}k \times \mathbb{La})^* \) be the set of tagged
traces. The set of traces of tagged tokens satisfying a strict sequencing order $\rightarrow$ is denoted $st(\rightarrow)$.

$$st(\rightarrow) = \left\{ \hat{\sigma} \in \mathbb{T_t} \Bigg| \forall 0 \leq i, j < |\hat{\sigma}|. \quad ((lb(\hat{\sigma}(i)) \rightarrow lb(\hat{\sigma}(j)) \Rightarrow (i \leq j)) \right\}$$

The semantics of block fragments is defined as

$$[\text{block}(L, \iota, \rightarrow)] = \text{untag}(\{st(\rightarrow)\} \cap^\# (\|_{\iota \in L} \text{tag}(\text{[L]}, \ell)))$$

Traces from immediate sub-fragments of $\text{block}(L, \iota, \rightarrow)$ are first interleaved in all possible ways and then those traces are removed that violate the strict sequencing order. The labels that are used to tag tokens do not occur in the resulting semantics; they are only used in enforcing the strict sequencing order $\rightarrow$.

**Example 4.2** Continue with Example 3.1. We have $[D_1] = \downarrow \text{fold}(\{\{\text{OK} = \text{true}\}\} \bullet [\text{block}(\{7, 8\}, \{7 \mapsto e_7, 8 \mapsto e_8, \{(7, 8)\})] \cup^\# \{\{\epsilon\}\}) = \{\{\text{OK} = \text{true}|e_7e_8, \epsilon\}\}$ and $[\text{Login}] = \{\{r, t\}, \{r', t'\}\}$ where $r, t, r', t'$ are given in Example 1.2.

### 4.2.8 Seq fragments.

The interaction operator $\text{seq}$ combines traces from component SDs via weak sequencing. The semantics of $\text{seq}(D_1, D_2)$ is obtained as follows. Every token in each trace in $[D_1]$ is tagged with 1 and every token in each trace of $[D_2]$ is tagged with 2. Tagged traces are then interleaved as in the semantics of $\text{par}(D_1, D_2)$. Then any tagged trace that violates weak sequencing order imposed by $\text{seq}$ is removed. The set of tagged traces that satisfy the weak sequencing order is

$$\mathbb{T_t}_{\text{seq}} = \left\{ \hat{\sigma} \in \mathbb{T_t} \Bigg| \forall 0 \leq i, j < |\hat{\sigma}|. \quad ((lb(\hat{\sigma}(i)) = 1 \land lb(\hat{\sigma}(j)) = 2 \land (\hat{\sigma}[i] \sim \hat{\sigma}[j]) \Rightarrow (i < j)) \right\}$$

The semantics of weak sequencing fragments is defined

$$[\text{seq}(D_1, D_2)] = [D_1] \text{merge}^\# [D_2]$$

where

$$\mathcal{M} \text{merge}^\# \mathcal{N} = \text{untag}(\{\mathbb{T_t}_{\text{seq}} \cap^\# (\text{tag}(\mathcal{M}, 1) \text{||} \text{tag}(\mathcal{N}, 2)))$$

**Example 4.3** Let $o_1, o_2$ be lifelines, $f_1 = (l, m, o_1, o_2)$, $f_2 = (?, m, o_1, o_2)$, $f_3 = (l, n, o_1, o_2)$ and $f_4 = (?, n, o_1, o_2)$. Put $D_1 = \text{strict}(f_1, f_2)$ and $D_2 = \text{strict}(f_3, f_4)$. Then

$$[D_1] = \{\{f_1f_2\}\}$$
$$[D_2] = \{\{f_3f_4\}\}$$
\[ \text{strict}(D_1, D_2) = \{\{f_1 f_2 f_3 f_4\}\} \]
\[ \text{seq}(D_1, D_2) = \{\{f_1 f_2 f_3 f_4\}, \{f_1 f_3 f_2 f_4\}\} \]
\[ \text{par}(D_1, D_2) = \{\{f_1 f_2 f_3 f_4\}, \{f_1 f_3 f_2 f_4\}, \{f_1 f_3 f_4 f_2\}, \{f_3 f_1 f_4 f_2\}, \{f_3 f_1 f_2 f_4\}\} \]

Let \( i(i) = D_i \) for \( i = 1, 2 \). \[ \text{block}([1, 2], \iota, \{\langle 1, 2 \rangle\}) = \text{strict}(D_1, D_2) \text{, and} \]
\[ \text{block}([1, 2], \iota, \emptyset) = \text{par}(D_1, D_2) \].

### 4.2.9 Loop fragments.

The UML standard stipulates that traces from consecutive runs of the loop body are combined via weak sequencing: \[ \text{loop}(c, D) \] is the limit of this series: \[ X_0 = \{\{\epsilon\}\} \text{ and } X_{i+1} = (c \triangleright^* ([D] \triangleright^* X_i)) \cup \{\{\epsilon\}\}. \]

### 4.2.10 Properties of semantics.

The abstract syntax requires that the \textit{block} fragment has at least one immediate sub-fragment. As a consequence, a sequence diagram specifies at least one obligation.

**Lemma 4.1** Let \( D \in \mathbb{Sd} \). Then \( \exists \mathcal{O} \in [D], (\mathcal{O} \neq \emptyset). \)

Let \( \mathbb{E}v \mathbb{t}(D) \) be the set of observable events in \( D \).

**Lemma 4.2** If \( \mathbb{E}v \mathbb{t}(D) = \emptyset \) then \( [D] = \{\{\epsilon\}\} = [\tau] \).

We adapt the concept of a context from term writing. A context is an SD with one of its fragments replaced by a special symbol \( z \). For instance, \( \text{seq}(z, e) \) with \( e \in \mathbb{E}v \mathbb{t} \) is a context. Let \( D \) be an SD and \( C \) a context. The embedding of \( D \) into \( C \), denoted \( C[D] \) is the SD obtained from replacing \( z \) with \( D \). Two SDs are called equivalent if they have the same meaning. The following proposition shows that the semantics possesses substitutivity. Substitutivity is a desirable property since it allows any fragment in an SD to be replaced with a semantically equivalent fragment.

**Proposition 4.1** Let \( C \) be a context and \( D_1, D_2 \in \mathbb{Sd} \). If \( [D_1] = [D_2] \) then \( [C[D_1]] = [C[D_2]] \).

### 5 Refinement

This section formalizes the notion of SD refinement. Non-determinism in an SD leads to multitude of alternative obligations. A system implementing an SD must fulfill at least one of the obligations. When the designer refines an SD by eliminating non-determinism and making it more defined, he must ensure that any implementation of the refined model is also an implementation of the original model. We first define a simulation relation between traces that take guard conditions into account.
Definition 5.1 Let \( c_1, c_2 \in \mathbb{Cnd}, \ e_1, e_2 \in \mathbb{Ext} \) and \( \alpha, \beta, \gamma \in \mathbb{Tr} \). The trace simulation relation \( \kappa \) is defined inductively as follows.

- \( c_1 \kappa c_2 \) if \( c_2 \models c_1 \).
- \( e_1 \kappa e_2 \) if \( e_1 = e_2 \).
- \( \langle \alpha \rangle \kappa \langle \gamma \rangle \) if \( \alpha \kappa \gamma \).
- \( \alpha \kappa \gamma \) if there are a trace \( \beta \) such that \( \alpha \monoarrow \beta \) and a strictly increasing function \( \eta: \text{dom}(\beta) \rightarrow \text{dom}(\gamma) \) such that
  - (1) for any \( i \in \text{dom}(\beta) \), \( \beta(i) \kappa \gamma(\eta(i)) \); and
  - (2) for \( j \in \text{dom}(\gamma) \), if \( j \not\in \text{image}(\eta) \) then \( \gamma(j) \in \mathbb{Cnd} \).

Some explanations are in order. A critical segment can only be simulated by a critical segment. The condition \( \alpha \monoarrow \beta \) allows events in protected sub-traces to be used to simulate events in \( \gamma \) by breaking up some occurrences of \( \{\} \). Note that \( \beta \) may be \( \alpha \) itself. The strict monotonicity of \( \eta \) ensures that different events in \( \gamma \) are simulated by different events in \( \beta \). The condition (2) ensures that if events in \( \gamma \) occur then events in \( \beta \) occur too. The condition (1) ensures that each event in \( \gamma \) is simulated by an event in \( \beta \).

Lemma 5.1 If \( \alpha_1 \kappa \alpha_2 \) and \( \alpha_2 \monoarrow \beta_2 \) then there is a \( \beta_1 \) such that \( \alpha_1 \monoarrow \beta_1 \) and \( \beta_1 \kappa \beta_2 \).

The following is the consequence of the reflexivity and transitivity of \( \models_c \).

Lemma 5.2 \( \kappa \) is reflexive and transitive.

Example 5.1 Let \( e_1, e_2, e_3 \) be different events and \( c \) a guard condition. Then \( e_1 \cdot e_3 \kappa e_1 \cdot e_3 \) and \( \{e_1 \cdot e_3\} \kappa \{e_1 \cdot c \cdot e_2\} \). But, \( \{e_1 \cdot e_2 \cdot e_3\} \kappa \{e_1 \cdot e_3\} \) does not hold since \( e_2 \) is an event and it is between \( e_1 \) and \( e_3 \). Nor does \( c \cdot e_1 \kappa e_1 \) hold since there is no guarantee that the constraint \( c \) is satisfied.

We now define the notion of one SD refines another SD. An SD specifies a number of alternative obligations and an implementation may choose to realize any of them. An SD \( D_1 \) refines another SD \( D_2 \) if any implementation of \( D_1 \) is also an implementation of \( D_2 \).

Definition 5.2 Let \( D_1, D_2 \in \mathbb{Sdl} \). \( D_1 \) is said to refine \( D_2 \), denoted \( D_1 \succeq D_2 \), if \( \forall O_1 \in [D_1], \exists O_2 \in [D_2], \forall t_1 \in O_1, \exists t_2 \in O_2, t_1 \in O_1, (t_1 \kappa t_2) \).

It follows from definition 5.2 and the definition of \( [] \) that \( \text{strict}(D_1, D_2) \succeq \text{seq}(D_1, D_2) \) and \( \text{seq}(D_1, D_2) \succeq \text{par}(D_1, D_2) \) for any SDs \( D_1 \) and \( D_2 \).

Theorem 5.1 The refinement relation \( \succeq \) is reflexive and transitive, i.e.,

1. \( D \succeq D \) for any \( D \in \mathbb{Sdl} \);
2. if $D_1 \succeq D_2$ and $D_2 \succeq D_3$ then $D_1 \succeq D_3$ for any $D_1, D_2, D_3 \in \mathbb{Sd}$

3. If $D_1 \succeq D_2$ then $C[D_1] \succeq C[D_2]$ for any $D_1, D_2 \in \mathbb{Sd}$ and any context $C$

Example 5.2 This example shows that SD Login3 refines SD Login. Let $e_i$ denote the event that is labelled $i$. Then Login3 = block({b, 5, 6, 11, 12, c}; {b → D_b, 5 → f_5, 6 → f_6, 11 → τ, 12 → τ, c → D_c}, {⟨b, 5⟩, ⟨5, 6⟩, ⟨6, 11⟩, ⟨11, 12⟩, ⟨(2, c)} with $D_b = strict(block({1, 2}, {1 → f_1, 2 → f_2}, {{1, 2}})$, $strict(block({3, 4}, {3 → f_3, 4 → f_4}, {{3, 4}}))$, $block({9, 10}, {9 \mapsto \tau, 10 \mapsto \tau}, {{9, 10}})$) and $D_c = opt(OK = true, block({7, 8}, {7 \mapsto f_7, 8 \mapsto f_8}, {{7, 8}})$) after the tautology true = true is removed from the guard condition. Then, by the definition of $[,]$

- $[block({\langle 9, 10 \rangle, {9 \mapsto \tau, 10 \mapsto \tau}, {{9, 10}}})] = \{\epsilon\}$
- $[block({1, 2}, {1 \mapsto e_1, 2 \mapsto e_2}, {{1, 2}})] = \{\{e_1e_2\}\}$
- $[block({3, 4}, {3 \mapsto e_3, 4 \mapsto e_4}, {{3, 4}})] = \{\{e_3e_4\}\}$

Thus, $[D_b] = \{\{e_1e_2e_3e_4\}\}$. We also have

- $[D_c] = \{\{[OK = true]e_7e_8, \epsilon\}\}$

Since the strict sequencing order in SD Login3 is total, traces from its components are combined using string concatenation and

$$[Login3] = \left\{ \begin{array}{l}
e_1e_2e_3e_4[OK = true]e_7e_8, \\
e_1e_2e_3e_4e_5e_6 
\end{array} \right\}$$

where $r$ and $t$ are given in Example 4.2. Recall from Example 4.2 $[Login] = \{\{r, t\}, \{r', t'\}\}$ where $r'$ and $t'$ are also given in Example 4.2. Finally, by Definition 5.3 Login3 $\succeq$ Login, that is, SD Login3 refines SD Login.

6 Conclusion and Future Work

Refinement of required behaviors of SDs is an important issue in software development process such as aspect-oriented and pattern-based software development. In this paper, we have presented a trace semantics for required behaviors of SDs and formalized a notion of refinement based on the semantics.

The semantics and the refinement relation developed in this paper can be used to underpin SD manipulation tools. An interesting topic in this line is how to represent an SD in normal form so as to simplify the task of such a tool. Another future work is to extend the semantics and the refinement relation to include the interaction operator neg. This requires one to take into account proscribed behaviors of SDs.
References


Proof of Lemma 4.1: By structural induction on $D$. The base cases where $D = \tau$ and $D = e$ are trivial. Assume that $D = \text{opt}(c, D_1)$, by induction hypothesis, $\exists O_1 \in [D_1]. (O_1 \neq \emptyset)$. Let $O = \{ct | t \in O_1\} \cup \{e\}$. We have that $O \neq \emptyset$ and $O \in [D]$. Other inductive cases are similar.

Proof of Lemma 4.2: Proof can be done by structural induction on $D$ in a similar way to Lemma 4.1 except that there is only one base case since $\text{Evrt}(e) \neq \emptyset$.

Proof of Proposition 4.1: The proof is done by structural induction on $C$. In the base case, $C = \infty$. We have $[C[D_1]] = [D_1] = [D_2] = [C[D_2]]$.

Now assume that $C = \text{opt}(c, C')$. By the induction hypothesis, we have $[C'[D_1]] = [C'[D_2]]$. Then

$$[C[D_1]] = [\text{opt}(c, C'[D_1])] = \downarrow \text{fold}([\{c\}] \bullet [C'[D_1]] \cup \{\{c\}\}) = \downarrow \text{fold}([\{c\}] \bullet [C'[D_2]] \cup \{\{c\}\}) = [\text{opt}(c, C'[D_2])] = [C[D_2]]$$

Other inductive cases are similar.

Proof of Lemma 5.1: Without loss of generality, we assume that $\alpha_1$ is a sequence of tokens, for otherwise the result follows immediately. Then by definition of $\kappa$, there are an $\alpha'$ such that $\alpha_1 \mathrel{\kappa} \alpha'$ and an $\eta : \text{dom}(\alpha') \mapsto \text{dom}(\alpha_2)$ such that

(a) For any $i \in \text{dom}(\alpha')$, $\alpha'(i) \prec \alpha_2(\eta(i))$; and

(b) For any $j \in \text{dom}(\alpha_2)$, if $j \not\in \text{image}(\eta)$ then $\alpha_2(j) \in \text{Cnld}$.

Since $\alpha_2 \mathrel{\kappa} \beta_2$, there are $\omega_1, \omega_2$ and $\omega_3$ such that $\alpha_2 = \omega_1(\omega_2)\omega_3$ and $\beta_2 = \omega_1\omega_2\omega_3$. Since $\eta$ is strictly increasing and $\alpha_2([\omega_1]) = [\omega_2] \notin \text{Cnld}$, there is a unique $\ell$ such that $\eta(\ell) = [\omega_1]$. There are also $u_1, u_2$ and $u_3$ such that $\|u_1\| = \ell$, $\alpha' = u_1(u_2)u_3$ and $u_2 \prec \omega_2$. Let $\alpha'' = u_1u_2u_3$. Then $\alpha \mathrel{\kappa} \alpha' \mathrel{\kappa} \alpha''$. Since $u_2$ and $\omega_2$ do not contain critical segment tokens and $u_2 \prec \omega_2$, there is an $\eta' : \text{dom}(u_2) \mapsto \text{dom}(\omega_2)$ such that

(c) For any $i' \in \text{dom}(u_2)$, $u_2(i') \prec \omega_2(\eta'(i'))$; and

(d) For any $j' \in \text{dom}(\omega_2)$, if $j' \not\in \text{image}(\eta')$ then $\omega_2(j') \in \text{Cnld}$.

Since $\alpha_1 \mathrel{\kappa} u_1(u_2)u_3$, there are $v_1$ and $v_2$ such that $\alpha_1 = v_1(u_2)v_3$ and $v_1 \prec \omega_1$ and $v_3 \prec \omega_3$. Let $\beta_1 = u_1u_2u_3$. Then $\alpha_1 \mathrel{\kappa} \beta_1 \mathrel{\kappa} u_1u_2u_3 = \alpha''$. Now define $\eta'' : \text{dom}(\alpha'') \mapsto \text{dom}(\beta_2)$ as follows.

$$\eta''(i'') = \begin{cases} 
\eta(i'') & i'' < \ell \\
\eta'(i'' - \ell) & \ell \leq i'' < \ell + \|\omega_2\| \\
\eta(i'' - \|\omega_2\| + 1) & i'' \geq \ell + \|\omega_2\|
\end{cases}$$
The following follows from (a)-(d).

- For any $i'' \in \text{dom}(\alpha'')$, $\alpha''(i'') \Vdash \beta_2(\eta''(i''))$; and
- For any $j'' \in \text{dom}(\beta_2)$, if $j'' \notin \text{image}(\eta'')$ then $\beta_2(j'') \in \text{Cndl}$. So, $\beta_1 \Vdash \beta_2$. □

**Proof of Lemma 5.2** Let $\gamma$ be an arbitrary trace. We prove $\alpha \Vdash \gamma$ by structural induction on $\gamma$. In the base case where $\gamma = e$, $\alpha \Vdash e$ by definition. In the inductive case where $\gamma = \{ \gamma' \}$, we have $\gamma' \Vdash \gamma'$ by the induction hypothesis, which implies $\alpha \Vdash \gamma$. Assume that $\gamma = t_1 \cdots t_n$. By the induction hypothesis, we have that $t_i \Vdash t_i$ for $0 \leq i < n$. Then $\gamma \Vdash \gamma$ by putting $\alpha = \beta = \gamma$ and $\eta(i) = i$ for any $0 \leq i < n$ in the definition of $\Vdash$.

Assume that $\alpha \Vdash \beta$ and $\beta \Vdash \gamma$. We prove $\alpha \Vdash \gamma$ by structural induction on $\alpha$.

**Case (a):** $\alpha = e$. Then $\beta$ must be of the form $\omega_1 \omega_2$ with $\omega_1, \omega_2 \in \text{Cndl}^*$. Since $\beta \Vdash \gamma$, there is a strictly increasing function $\eta' : \text{dom}(\beta) \rightarrow \text{dom}(\gamma)$ such that

1. For any $i \in \text{dom}(\beta)$, $\beta(i) \Vdash \gamma(\eta'(i))$;
2. For any $j \in \text{dom}(\gamma)$, if $j \notin \text{image}(\eta')$ then $\gamma(j) \in \text{Cndl}$.

Let $\ell$ be the unique position at which $e$ occurs in $\beta$. Then (2') implies that $\gamma(j) \in \text{Cndl}$ for all $j \notin \eta'(\ell)$; (1') implies that $\gamma(\eta'(\ell)) = e$. Thus, $\alpha \Vdash \gamma$.

**Case (b):** $\alpha = \{ \alpha' \}$. There are $\beta'$ and $\gamma'$ such that $\beta = \{ \beta' \}$, $\gamma = \{ \gamma' \}$, $\alpha' \Vdash \beta'$ and $\beta' \Vdash \gamma'$. By the induction hypothesis, we have $\alpha' \Vdash \gamma'$ which implies $\alpha \Vdash \gamma$.

**Case (c):** $\alpha = \{ \alpha' \}$. There are $\beta'$ and $\gamma'$ such that $\beta \Vdash \gamma$, there are $\beta'$ such that $\beta \rightarrow^* \beta'$ and $\eta_2 : \text{dom}(\beta') \rightarrow \text{dom}(\gamma)$ such that

1. For any $i \in \text{dom}(\beta')$, $\beta'(i) \Vdash \gamma(\eta_2(i))$;
2. For any $j \in \text{dom}(\gamma)$, if $j \notin \text{image}(\eta_2)$ then $\gamma(j) \in \text{Cndl}$;
3. For any $i \in \text{dom}(\alpha'')$, $\alpha''(i) \Vdash \beta'(\eta_1(i))$;
4. For any $j \in \text{dom}(\alpha'')$, if $j \notin \text{image}(\eta_1)$ then $\beta''(j) \in \text{Cndl}$;

Now define $\eta : \text{dom}(\alpha'') \rightarrow \text{dom}(\gamma)$ by $\eta = \eta_2 \circ \eta_1$. Then (i)-(iv) imply that

- For any $i \in \text{dom}(\alpha'')$, $\alpha''(i) \Vdash \gamma(\eta(i))$;
- For any $j \in \text{dom}(\gamma)$, if $j \notin \text{image}(\eta)$ then $\gamma(j) \in \text{Cndl}$;

This, together with $\alpha \rightarrow^* \alpha''$, implies that $\alpha \Vdash \gamma$.

**Proof of theorem 5.1** 1 and 2 follow from Lemma 5.2. The proof of 3 follows from a simple structural induction on $C$. □