On-line Appendix to the Paper "Automatic Composition of Transition-based Semantic Web Services with Messaging"

Daniela Berardi[1], Diego Calvanese[2], Giuseppe De Giacomo[1], Richard Hull[3], Massimo Mecella[1]

[1]*Università di Roma "La Sapienza"*
berardi@dis.uniroma1.it
degiacomo@dis.uniroma1.it
mecella@dis.uniroma1.it

[2]*Libera Università di Bolzano/Bozen*
calvanese@inf.unibz.it

[3]*Bell Labs, Lucent Technologies*
hull@lucent.com

This appendix includes some additional formal definitions and details for selected aspects of the Colombo framework and the $\text{Colombo}^{k,b}$ model, described in the paper D. Berardi, D. Calvanese, G. De Giacomo, R. Hull, and M. Mecella, "Automatic Composition of Transition-based Semantic Web Services with Messaging", Proc. of VLDB 2005.

1 Selected Formal Details of the Model

1.1 Atomic Processes

Remark 1.1: Unlike OWL-S atomic processes, we do not use a "pre-condition", or equivalently, we assume that the pre-condition is uniformly true. We do this to enable a more uniform treatment of atomic process executions: when a web service invokes an atomic process in Colombo, the invoking service will transition to a new state whether or not the atomic process "succeeds". Optionally, the designer of the atomic process can include an output boolean variable '*flag*', which is set to true if the execution "succeeded" and is set to false if the execution "failed". These are conveniences that simplifies book-keeping, with no real impact on expressive power. □

Definition: An *atomic process* is an object p which has a signature of form (I, O, CE) with the following properties.
Input Signature: I is a sequence $\langle u_1 : d_1, \ldots, u_n : d_n \rangle$ where the u_i's are distinct variables, and each $d_j \in \{Bool$, '$=$', '$\leq$'$\}$. For example, if $d_i =$ '$=$', then the value associated with u_i in an invocation of p should be an element of $Dom_=$ (or ω).
Output Signature: O is a sequence $\langle v_1 : d_1, \ldots, v_m : d_m \rangle$ where the v_j's are distinct variables and each $d_j \in \{Bool$, '$=$', '$\leq$'$\}$. For example, if d_j '$=$', then the value assigned to v_j by an invocation of p will be an element of $Dom_=$ (or ω).
Conditional Effects: CE is a set of pairs of form (c, E), where c is a (*atomic process*) *condition* and E is a finite non-empty set of (*atomic process*) *effect* (*specifications*). Condition c is a boolean expression over atoms over accessible terms over some family of constants and the variables $u_1, \ldots, u_n$.

An effect $e \in E$ is a pair (es, ev) where:
Effect on World State: es is a set of expressions having the forms

(i) $\mathit{insert}\ R(t_1, \ldots, t_k; s_1, \ldots, s_l)$

(ii) $\mathit{delete}\ R(t_1, \ldots, t_k)$

(iii) $\mathit{modify}\ R(t_1, \ldots, t_k; r_1, \ldots, r_l)$

where R ranges over relations in the world schema, R has key of length k and l additional columns, where the t_i's and s_j's are accessible terms over some set of constants and over the variables $u_1, \ldots, u_n$, and where each r_j is either an accessible term over some set of constants and $u_1, \ldots, u_n$ or the special symbol '$-$' (denoting that that position of the identified tuple in R should be unchanged).
Effect on Output Variables: ev is a set of expressions of the form

(iv) $v_j := t$, where $j \in [1..m]$ and t is an accessible term over some set of constants and over the variables $u_1, \ldots, u_n$,

(v) $v_j := \omega$, where $j \in [1..m]$

There must be exactly one expression for each v_j, $j \in [1..m]$.

We now describe the semantics associated with atomic process execution. An atomic process p with characteristics as specified above will be invoked in the context of (i) an assignment α over a set X of variables (which typically

Università di Roma "La Sapienza"
Technical Report 06-05, 2005

corresponds to the local store of a web service); (ii) a world state $\mathcal{I}$, and (iii) a family Σ of integrity constraints on the world schema.

A semantics is associated with the execution of p as follows. (This semantics is straightforward but intricate, so we include the detailed definition to avoid ambiguity.) Atomic process p is invoked in the context of variable set X using an expression having form $p(y_1, \ldots, y_n; z_1, \ldots, z_m)$ where the y_i's are distinct elements of X, and the z_j's are distinct elements of X. The result of executing this specification will depend on α, $\mathcal{I}$, and Σ, and will result in an assignment α' and world state $\mathcal{I}'$ (which may be identical to α and $\mathcal{I}$, respectively).

(a) If no conditions in CE are true in $\mathcal{I}$ under α then this execution of p has a "no-op" effect, i.e., α' is simply α and $\mathcal{I}'$ is simply $\mathcal{I}$. If two or more conditions in CE are both true under α then again this execution of p has no-op effect.

For the remainder, assume that (c, E) is the pair in CE where c is the unique condition in CE that is true in $\mathcal{I}$ under α. Assume further that (es, ev) is a (non-deterministically chosen) element of E.

(b) If in any of the $insert$, $delete$ and/or $modify$ expressions, as interpreted using α, there is an ω value occurring in a key field, then execution of p has no-op effect.

(c) If there is a "conflict" between any of the $insert$, $delete$ and/or $modify$ expressions in es, as interpreted using α (e.g., the expressions under α call for inserting two tuples with the same key, or inserting a tuple with a given key but also deleting a tuple with that key, etc.), then execution of p has no-op effect.

We now define the *potential effect* (on the world state) of executing es, assuming assignment α, world state $\mathcal{I}$, and constraint set Σ, and assuming that item (c) above does not apply. After defining the notion of potential effect, we describe the conditions under which it will actually be applied (namely, if applying it does not violate any constraints in Σ.)

(d) For each expression $insert\ R(t_1, \ldots, t_k; s_1, \ldots, s_l)$, in $\mathcal{I}'$ the tuple $\langle \alpha(t_1), \ldots, \alpha(t_k), \alpha(s_1), \ldots, \alpha(s_l) \rangle$ is in R. If there was different tuple $\langle \alpha(t_1), \ldots, \alpha(t_k), c_1, \ldots, c_l \rangle$ in R in $\mathcal{I}$, that tuple is not present in R in $\mathcal{I}'$.

(e) For each expression $delete\ R(t_1, \ldots, t_k)$, in $\mathcal{I}'$ there is no tuple in R with first k fields being $\langle \alpha(t_1), \ldots, \alpha(t_k) \rangle$.

(f) For each expression $modify\ R(t_1, \ldots, t_k; r_1, \ldots, r_l)$, there are two cases. If in $\mathcal{I}$ there is no tuple in R having key $\langle \alpha(t_1), \ldots, \alpha(t_k) \rangle$, then in $\mathcal{I}'$ there is no tuple in R having that key. If in $\mathcal{I}$ there is a tuple of form $\langle \alpha(t_1), \ldots, \alpha(t_k), c_1, \ldots, c_l \rangle$, then in $\mathcal{I}'$ there is a tuple having form $\langle \alpha(t_1), \ldots, \alpha(t_k), c'_1, \ldots, c'_l \rangle$, where for each $j \in [1..l]$, $c'_j = \alpha(r_j)$, if r_j is not '$-$', and c'_j is c_j otherwise.

(g) (Frame condition): The potential effect state $\mathcal{I}'$ is identical to $\mathcal{I}$ except as indicated in items (d), (e), and (f).

Finally, we describe the conditions under which the effect of executing (es, ev) should actually be applied to $\mathcal{I}$ and α, and describe the impact on both of those.

(h) Assume that item (b) does apply, item (c) does not apply, that (es, ev) is a non-deterministic choice from E, and that the potential effect of executing es is as described as in items (d), (e), (f), and (g). If $\mathcal{I}'$ satisfies all of the integrity constraints in Σ, then the world state becomes $\mathcal{I}'$ after this execution of (es, ev).

(i) Also, assume that the conditions of item (h) hold. Then the new assignment α' is constructed from α as follows. For each variable z_j, $j \in [1..m]$, if '$z_j := t$' occurs in es, then $\alpha'(z_j)$ is given the value of $\alpha(t)$ as interpreted over $\mathcal{I}'$; and is given the value ω otherwise. Assignment α' is identical to α on local variables of S not occurring among $z_1, \ldots, z_m$.

Remark 1.2: A broad variety of generalizations are possible in the `Colombo` framework, e.g., to move away from exclusively key-based look-ups; letting variables range over sets of tuples in addition to single tuples, etc. □

1.2 Linkages, Stores, Transmit, Read

Remark 1.3: The notion of linkage is closely related to the notion of linkage in BPEL, and is used implicitly in the "service schema" of the Conversation model. The notion of link is inspired by, and closely related to, channels as typical of process algebras, and as found in the emerging SWSL ontology. In `Colombo`^{k,b}we do not change the linkage at runtime, but in principle such dynamic changes can be supported in the `Colombo` framework. Other variations can be represented in `Colombo`, such as allowing multiple services to give input to a channel, or having multiple services read a message in a channel. □

Let S be a web service, which is not a client. The *local store* of S, typically denoted as `LStore` or $\texttt{LStore}_S$, is a finite set $\{v_1 : d_1, \ldots, v_n : d_n\}$ where the v_i's are distinct variables and the d_i's are types from $\{Bool, Eq, Leq\}$. For each incoming port $(m, \texttt{in})$ of S we assume that there is a distinguished boolean variable π_m in $\texttt{LStore}_S$, called the *message-present* flag or variable; intuitively, this will be set to true if a message has arrived into the queue of $(m, \texttt{in})$, and is set to false when that message is read by the service (see below).

In `Colombo` we assume that a message that has been transmitted is held in a queue associated to the incoming port of the receiving service. In general, the queues might be bounded or unbounded. For the current paper, in $\texttt{Colombo}^{\mathrm{k,b}}$ we assume that the queues are bounded and have length one.

In addition to the local store, each non-client service S has a *queue store*, typically denoted by $\texttt{QStore}$ or $\texttt{QStore}_S$. This store is used to hold the parameter values of incoming messages, which can be thought of as being held by a queue. Specifically, for each incoming port $(m, \texttt{in})$ of S, where m has signature $\langle d_1, \ldots, d_l \rangle$, we include l variables denoted as v_k^m, for $k \in [1..l]$.

We use $\texttt{Store}$ or $\texttt{Store}_S$ to denote the union $\texttt{LStore}_S \cup \texttt{QStore}_S$.

For passing messages between services we have two basic operations: *transmit* and *read*. The syntax of operator transmit, used in the process specification of the sending service S, is $!m(r_1, \ldots, r_l)$, where each r_k is either a constant or a variable in LS_S, of appropriate type. Let (S, m, S', n) be a link between services S and S', let α be a variable assignment for $\texttt{LStore}_S$ at some point during an enactment of a system involving S and S'. Execution of $!m(r_1, \ldots, r_l)$ at this point will succeed iff the queue of S' for $(n, \texttt{in})$ has room (in our case, if the queue is empty). In this case, the variable v_k^n of $\texttt{QStore}_{S'}$ will be assigned the value $\alpha(r_k)$, for $k \in [1..l]$, and π_n in $\texttt{QStore}_{S'}$ is set to true. As far as the processing of S', the receiving of a transmitted message is essentially an asynchronous event, and is not explicitly represented in the process model specification for S'.

What happens if the queue for $(n, \texttt{in})$ is full? Several options are available in the general $\texttt{Colombo}$ framework. A natural option, which makes the transmit operator similar to $\texttt{Colombo}$ atomic processes, is to assume that this operator is "executed", but that it has no impact on S', and that a flag is set in S (e.g., so that the transmit could be attempted again later on). However, in $\texttt{Colombo}^{\mathrm{k,b}}$, we assume that this operator *blocks* until the queue of S' has room.

We now turn to the read operation. The syntax of this operator, specified in the process specification of the service S' that receives messages transmitted over a link (S, m, S', n), is $?n(v_1, \ldots, v_l)$, where each v_k is a variable in LS_S (but not any of the distinguished message-present flags), of appropriate type. Let α be the assignment in effect for $\texttt{LStore}_{S'}$ and β the assignment for $\texttt{QStore}_{S'}$, and assume that $\alpha(\pi_n)$ is true. The effect of executing $?n(v_1, \ldots, v_l)$ is that α is modified to become α', where $\alpha'(v_k) = \beta(v_k^n)$ for $k \in [1..l]$, that $\alpha'(\pi_n)$ is set to false, and α' is identical to α elsewhere.

What happens if service S' with assignment α on $\texttt{LStore}_{S'}$ attempts to execute $?n(v_1, \ldots, v_l)$, but $\alpha(\pi_n)$ is false? As with transmit, several options are available in the general $\texttt{Colombo}$ framework. A natural option, which makes the read operator similar to $\texttt{Colombo}$ atomic processes, is to assume that this read operator is "executed", but that it has no impact on S', except perhaps for setting a flag. Alternatively, the designer of S' could include a test on π_n before attempting to execute the read. However, in $\texttt{Colombo}^{\mathrm{k,b}}$, we assume that the read operator $?n$ *blocks* until there is a message in the queue of $(n, \texttt{in})$.

Remark 1.4: The assumption that all queues have length one, along with a subsequent restriction on web services that they are "blocking" as just described, end up implying that all message transmissions are essentially synchronous, as typical of process algebras, in that a message send and the receiving/reading of the message must happen with no intervening activities (neither atomic process invocations nor other message sends). However, we maintain the formalism of queues in $\texttt{Colombo}^{\mathrm{k,b}}$, because we expect that the results obtained in the current paper can be generalized to support broader models for message passing as typically arise in the web service standards and research literature. In particular, it is easy to see how the notion of queue store can be extended to support queues with arbitrary bounded size. $\Box$

We now describe how message passing works with a client service. We assume that a client C has access to a finite set $\texttt{Constants}_C$ of elements from Dom, which are the constants available to C at any time. For client C we also maintain a unary relation, denoted $\texttt{HasSeen}$ or $\texttt{HasSeen}_C$, which holds elements of Dom. Intuitively, at a given time in an execution of C, $\texttt{HasSeen}_C$ will include all of $\texttt{Constants}_C$, and also all domain elements that occur in messages that have been transmitted to C.

What happens if a service S with assignment α executes a transmit operation $!m(r_1, \ldots, r_l)$ directed at C? In $\texttt{Colombo}^{\mathrm{k,b}}$ we assume that this always succeeds, and that $\texttt{HasSeen}_C$ is replaced with $\texttt{HasSeen}_C \cup \{\alpha(r_k) \mid k \in [1..l]\}$. Intuitively, then, when a message is transmitted to C is is also read by C immediately.

In $\texttt{Colombo}^{\mathrm{k,b}}$, we assume that a client C can transmit a message at the beginning of its enactment, but after that, it can transmit a message only after it has received a message. We assume that C acts non-deterministically, and that after receiving a message it can execute any transmit $!m(c_1, \ldots, c_l)$ where $(m, \texttt{out})$ is a port of C (that occurs in some link of the system) and $c_k \in \texttt{HasSeen}_C$ for each $k \in [1..l]$. Aside from these restrictions on transmit and $\texttt{HasSeen}_C$, we do not model in $\texttt{Colombo}^{\mathrm{k,b}}$ the internal workings of client C.

1.3 Internal process model, services, clients, systems

Let S be a service with signatures $\texttt{Port}(S), \texttt{GA}(S)$. An *instantaneous description*, or *id*, of S over world schema $\mathcal{W}$ and constraints Σ is a tuple (s, α, β) where s is a state of $\texttt{GA}(S)$, α is an assignment for $\texttt{LStore}_S$, β is an assignment for $\texttt{QStore}_S$. In general we consider pairs of the form $(\texttt{id}^S, \mathcal{I})$, where $\texttt{id}^S$ is an id over S and $\mathcal{I}$ is a world state over $\mathcal{W}$ that satisfies Σ. We sometimes use $\texttt{id}$ to denote $\texttt{id}^S$, if S is understood from the context.

An id $\texttt{id}^S = (s, \alpha, \beta)$ is *initial* if s is the start state of S, $\alpha(\pi_n)$ is false for each n where $(n, \texttt{in}) \in \texttt{Port}(S)$ and α is ω elsewhere, and β is uniformally set to ω.

We now define the "moves-to" relation and the "trace" for individual services. The *moves-to* relation $\vdash_S$ (or simply $\vdash$ if S is understood from the context) will hold between pairs of the form $(\mathtt{id}^S,\mathcal{I}),(\mathtt{id}^{S'},\mathcal{I}')$ under conditions presented below, and corresponds intuitively to cases where service S can move from one internal state to the next, and/or where the global store can change (e.g., if a message is received.) The definition is more-or-less standard, except that we build in the possibility that moves by other services might be interspersed (see items (b), (c), and (d) below). The *trace* of a pair $(\mathtt{id}^S,\mathcal{I}),(\mathtt{id}^{S'},\mathcal{I}')$ (where $(\mathtt{id}^S,\mathcal{I}) \vdash_S (\mathtt{id}^{S'},\mathcal{I}')$) will provide, intuitively, a grounded record or log of salient aspects of the transition from $(\mathtt{id}^S,\mathcal{I})$ to $(\mathtt{id}^{S'},\mathcal{I}')$, including, e.g., what parameter values were input/output from an atomic process invocation, or were received, read or sent. We define $\vdash$ and trace simultaneously. Let $(\mathtt{id}^S,\mathcal{I}),(\mathtt{id}^{S'},\mathcal{I}')$ satisfy $\mathtt{id}^S = (s,\alpha,\beta)$ and $\mathtt{id}^{S'} = (s',\alpha',\beta')$.

(a) *Atomic Process:* Suppose that $\mathtt{GA}(S)$ has a transition from s to s' labeled by $(g, p(r_1,\ldots,r_n;v_1,\ldots,v_m))$ where $g[\alpha]$ evaluates to true; if

$$((\alpha,\mathcal{I}), p(r_1,\ldots,r_n;v_1,\ldots,v_m)) \vdash (\alpha',\mathcal{I}');$$

and β' is β. Then $(\mathtt{id}^S,\mathcal{I}) \vdash (\mathtt{id}^{S'},\mathcal{I}')$. Also, the trace of pair $(\mathtt{id}^S,\mathcal{I}),(\mathtt{id}^{S'},\mathcal{I}')$, denoted $\mathtt{trace}((\mathtt{id}^S,\mathcal{I}),(\mathtt{id}^{S'},\mathcal{I}'))$, is $(p(c_1,\ldots,c_n;d_1,\ldots,d_m),\mathcal{I},\mathcal{I}')$, where $c_i = \alpha(r_i)$ for $i \in [1..n]$ and $d_j = \alpha'(v_j)$ for $j \in [1..m]$.

(b) *Receive Message:*[1] Suppose that $\mathtt{Port}(S)$ includes $(n,\mathtt{in})$ for some message type $n(r_1,\ldots,r_l)$ with arity l; $\alpha(\pi_n)$ is false (i.e., the queue for message n is empty), and also $\beta(v^n_k) = \omega$ for $k \in [1..l]$; $\alpha'(\pi_n)$ is true and α' is identical to α elsewhere; $\beta'(v^n_k)$ has arbitrary values (consistent with the signature of n) and β' is identical to β elsewhere. Then $(\mathtt{id}^S,\mathcal{I}) \vdash (\mathtt{id}^{S'},\mathcal{I}')$. In this case, $\mathtt{trace}((\mathtt{id}^S,\mathcal{I}),(\mathtt{id}^{S'},\mathcal{I}'))$ is $(?n(c_1,\ldots,c_k),\mathcal{I}')$, where $c_i = \alpha(r_i)$ for $i \in [1..l]$.

(Note in this case, and cases (c) and (d) below, there are no restrictions on $\mathcal{I}$ to $\mathcal{I}'$. Intuitively, this freedom is incorporated to reflect the possibility that in a system of services, S might move into $(\mathtt{id}^S,\mathcal{I})$ at some time t_1, then other services might make a variety of moves including some that change the world state to $\mathcal{I}'$ at time t_2, and finally a service might send a message of type n to S at time t_2. So as far as S is concerned, it was in $(\mathtt{id}^S,\mathcal{I})$ just after time t_1, and then at time t_2 it moves to $(\mathtt{id}^{S'},\mathcal{I}')$.)

(c) *Read Message:* Suppose that $\mathtt{GA}(S)$ has a transition from s to s' labeled by $(g, ?m(v_1,\ldots,v_l))$ where $g[\alpha]$ evaluates to true; $\alpha(\pi_m)$ is true; $\alpha'(v_k) = \beta(v^m_k)$ for $k \in [1..l]$, $\alpha'(\pi_m)$ is false, and α' equals α elsewhere; $\beta'(v^m_k) = \omega$ for $k \in [1..l]$ and β' equals β elsewhere. Then $(\mathtt{id}^S,\mathcal{I}) \vdash (\mathtt{id}^{S'},\mathcal{I}')$. In this case, $\mathtt{trace}((\mathtt{id}^S,\mathcal{I}),(\mathtt{id}^{S'},\mathcal{I}'))$ is $(?n(d_1,\ldots,d_l),\mathcal{I}')$, where $d_k = \alpha'(v_k)$ for $k \in [1..l]$.

(d) *Transmit Message:* Suppose that $\mathtt{GA}(S)$ has a transition from s to s' labeled by $(g, !m(r_1,\ldots,r_l))$ where $g[\alpha]$ evaluates to true; α' is identical to α; and β' is identical to β. Then $(\mathtt{id}^S,\mathcal{I}) \vdash (\mathtt{id}^{S'},\mathcal{I}')$. In this case, $\mathtt{trace}((\mathtt{id}^S,\mathcal{I}),(\mathtt{id}^{S'},\mathcal{I}'))$ is $(!m(c_1,\ldots,c_l),\mathcal{I}')$, where $c_k = \alpha(r_k)$ for $k \in [1..l]$.

An *enactment* of S is a finite sequence $\mathcal{E} = \langle(\mathtt{id}_1,\mathcal{I}_1),\ldots,(\mathtt{id}_q,\mathcal{I}_q)\rangle$, $q \geq 1$, where (a) $\mathtt{id}_1$ is an initial id for S, and (b) $(\mathtt{id}_p,\mathcal{I}_p) \vdash (\mathtt{id}_{p+1},\mathcal{I}_{p+1})$ for each $p \in [1..(q-1)]$. The enactment is *successful* if $\mathtt{id}_q$ is in a final state of $\mathtt{GA}(S)$.

The notion of *execution tree* for S is now defined. (This can be viewed as a stepping stone for defining execution tree for a system $\mathcal{S}$.) Intuitively, an execution tree is an infinitely branching tree T that records all possible enactments. The root is not labeled, and all other nodes are labeled by pairs of form $(\mathtt{id},\mathcal{I})$ where $\mathtt{id}$ is an id of S and $\mathcal{I}$ is a valid world state. For the children of the root, the id is the initial id of S and $\mathcal{I}$ is arbitrary. An edge $((\mathtt{id},\mathcal{I}),(\mathtt{id}',\mathcal{I}'))$ is included in the tree if $(\mathtt{id},\mathcal{I}) \vdash (\mathtt{id}',\mathcal{I}')$; in this case the edge is labeled by $\mathtt{trace}((\mathtt{id},\mathcal{I}),(\mathtt{id}',\mathcal{I}'))$. A node $(\mathtt{id},\mathcal{I})$ in the execution tree is *terminating* if $\mathtt{id}$ is in a final state of $\mathtt{GA}(S)$.

We now turn to clients; we model *clients* as a special kind of web service. Clients correspond intuitively to a human (or automated) agent which interacts with one or more web service to accomplish some goals. While abstract properties of the internal model of non-client web services is specified in considerable detail (using notions of local store, automata-based process model, etc.), the abstract properties of the client are described with only salient details.

At this point we are focused primarily on $\mathtt{Colombo}^{\mathrm{k,b}}$, but make our definitions slightly more general, so that they can be used with other studies in the broader Colombo framework. An *instantaneous description* (*id*) for a client C (in the case of $\mathtt{Colombo}^{\mathrm{k,b}}$) is a pair $(s,\mathtt{HasSeen})$, where $s \in \{\mathtt{ReadyToTransmit},\mathtt{ReadyToRead}\}$ and $\mathtt{HasSeen}$ is a unary relation over elements of *Dom* (holding, intuitively, all domain elements that C has "seen" up to this point in an execution). This id is *initial* if $s = \mathtt{ReadyToTransmit}$ and $\mathtt{HasSeen}$ is the set of constants present in the definition of C.

We now define the *moves-to* relation and notion of *trace* for clients, in the restricted case of $\mathtt{Colombo}^{\mathrm{k,b}}$. Let $(\mathtt{id}^C,\mathcal{I}),(\mathtt{id}^{C'},\mathcal{I}')$ satisfy $\mathtt{id}^C = (s,\mathtt{HasSeen})$ and $\mathtt{id}^{C'} = (s',\mathtt{HasSeen}')$. For clients in $\mathtt{Colombo}^{\mathrm{k,b}}$, we combine the read activity with the receive activity. (As with receive, read, and send for services, the values of $\mathcal{I},\mathcal{I}'$ are not restricted for receive. We insist that $\mathcal{I} = \mathcal{I}'$ for send,

[1]Cases (b), (c), and (d) are for the case of $\mathtt{Colombo}^{\mathrm{k,b}}$; variations of these will be appropriate for other variants of Colombo.

to capture the intuition that in $\texttt{Colombo}^{\mathrm{k,b}}$ nothing can happen in between the client reading a message and then transmitting another one.)

(a) *Receive Message (which includes Read):* Suppose that $\texttt{Port}(C)$ includes $(n, \texttt{in})$ for some message type n with arity l; $s = \texttt{ReadyToRead}$; $s' = \texttt{ReadyToTransmit}$; let $\langle d_1, \ldots, d_l\rangle$ be a sequence of l (not necessarily distinct) domain elements; and let $\texttt{HasSeen}' = \texttt{HasSeen}_{\cup}\{d_1, \ldots, d_l\}$ where the d_i's are a set of at most l (not necessarily distinct) domain elements. Then $(\texttt{id}^S, \mathcal{I}) \vdash (\texttt{id}^{S'}, \mathcal{I}')$. In this case, $\texttt{trace}((\texttt{id}^S, \mathcal{I}), (\texttt{id}^{S'}, \mathcal{I}'))$ is $(\texttt{receive}\ n(d_1, \ldots, d_l), \mathcal{I}')$.

(b) *Transmit Message:* Suppose that $\texttt{Port}(C)$ includes $(m, \texttt{out})$ for some message type m with arity l; $\texttt{HasSeen}' = \texttt{HasSeen}$; $\langle c_1, \ldots, c_l\rangle$ is a sequence of (not necessarily distinct) elements from $\texttt{HasSeen}$; $s = \texttt{ReadyToTransmit}$; $s' = \texttt{ReadyToRead}$; and $\mathcal{I} = \mathcal{I}'$. Then $(\texttt{id}^S, \mathcal{I}) \vdash (\texttt{id}^{S'}, \mathcal{I}')$. In this case, $\texttt{trace}((\texttt{id}^S, \mathcal{I}), (\texttt{id}^{S'}, \mathcal{I}'))$ is $(!m(c_1, \ldots, c_l), \mathcal{I}')$.

Note that in $\texttt{Colombo}^{\mathrm{k,b}}$ there are several forms of non-determinism in the execution of a client C. This includes which kind of message to send, and which elements from $\texttt{HasSeen}$ to send in that message. Other forms of non-determinism are present based on how C interacts with other services; these include that there are no restrictions on: $\mathcal{I}, \mathcal{I}'$ for receives, the timing of receives, and the parameter values of incoming messages.

The notion of (*successful*) *enactment* and *execution tree* for clients is defined analogously as for services.

The notion of *instantaneous description* (*id*) for system $\mathcal{S}$ is defined in a natural fashion, based on a generalization of id for individual services. Specifically, an id for $\mathcal{S}$ is a tuple $\texttt{id}^{\mathcal{S}} = (\texttt{id}^C, \{\texttt{id}^S \mid S \in \mathcal{F}\})$ where $\texttt{id}^C$ is an id of C and $\texttt{id}^S$ is an id of S for each $S \in \mathcal{F}$. This id is *initial* if the id's for C and the S's are initial.

Because of the blocking behaviors incorporated into $\texttt{Colombo}^{\mathrm{k,b}}$, it turns out that in an enactment at most one service (or the client) will be "executing" at any time (i.e., no concurrency).

2 Selected Formal Details of the PDL Encoding

2.1 Preliminaries on PDL

Propositional Dynamic Logic (PDL) is a well-known logic of programs developed to verify properties of program schemas [2]. PDL formulas are formed by starting from a set $\mathcal{P}$ of atomic propositions and a set $\mathcal{A}$ of atomic actions, according to the following abstract syntax:

$$\begin{array}{lcl} \phi & \longrightarrow & P \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \langle r\rangle\phi \mid [r]\phi \\ r & \longrightarrow & a \mid \phi? \mid r_1; r_2 \mid r_1 \cup r_2 \mid r^* \end{array}$$

where P is an atomic proposition in $\mathcal{P}$, a is an atomic action in $\mathcal{A}$. That is, PDL formulas are composed from atomic propositions by applying arbitrary propositional connectives, and modal operators $\langle r\rangle\phi$ and $[r]\phi$, where r is a *program* formed as a regular expression over the atomic actions in $\mathcal{A}$ and the *tests* $\phi?$.

Intuitively, the modal operators, $\langle r\rangle\phi$ expresses that there exists an execution of r reaching a state where ϕ holds, while $[r]\phi$ expresses that all *terminating* executions of r reach a state where ϕ holds (i.e., it express a partial correctness condition). As for programs, a means "execute action a"; $\phi?$ means "proceed only if ϕ is true"; $r_1 \cup r_2$ means "choose non deterministically between r_1 and r_2"; $r_1; r_2$ means "first execute r_1 then execute r_2"; r^* means "execute r a non deterministically chosen number of times (zero or more)".

A PDL interpretation is a Kripke structure of the form $\mathcal{M} = (\Delta^{\mathcal{M}}, \cdot^{\mathcal{M}})$, where $\Delta^{\mathcal{M}}$ is a non-empty set of states, and $\cdot^{\mathcal{M}}$ is an interpretation function which interprets atomic propositions $P^{\mathcal{M}} \subseteq \Delta^{\mathcal{M}}$ – denoting the states in $\Delta^{\mathcal{M}}$ were P is true – and atomic actions $a^{\mathcal{M}} \subseteq \Delta^{\mathcal{M}} \times \Delta^{\mathcal{M}}$ – denoting the state transition caused by the atomic action a. The interpretation function $\cdot^{\mathcal{M}}$ is extended to arbitrary formulas and programs as follows:

$$\begin{array}{rcl} P^{\mathcal{M}} & \subseteq & \Delta^{\mathcal{M}} \\ (\neg\phi)^{\mathcal{M}} & = & \Delta^{\mathcal{M}}/\phi^{\mathcal{M}} \\ (\phi_1 \wedge \phi_2)^{\mathcal{M}} & = & \phi_1^{\mathcal{M}} \cap \phi_2^{\mathcal{M}} \\ (\phi_1 \vee \phi_2)^{\mathcal{M}} & = & \phi_1^{\mathcal{M}} \cup \phi_2^{\mathcal{M}} \\ (\langle r\rangle\phi)^{\mathcal{M}} & = & \{s \mid \exists s'.(s, s') \in r^{\mathcal{M}} \wedge \phi^{\mathcal{M}}\} \\ ([r]\phi)^{\mathcal{M}} & = & \{s \mid \forall s'.(s, s') \in r^{\mathcal{M}} \rightarrow \phi^{\mathcal{M}}\} \\ a^{\mathcal{M}} & \subseteq & \Delta^{\mathcal{M}} \times \Delta^{\mathcal{M}} \\ (\phi?)^{\mathcal{M}} & = & \{(s, s) \mid s \in \phi^{\mathcal{M}}\} \\ (r_1 \cup r_2)^{\mathcal{M}} & = & r_1^{\mathcal{M}} \cup r_2^{\mathcal{M}} \\ (r_1; r_2)^{\mathcal{M}} & = & r_1^{\mathcal{M}}; r_2^{\mathcal{M}} \\ (r^*)^{\mathcal{M}} & = & (r^{\mathcal{M}})^* \end{array}$$

A PDL formula is satisfiable iff $\phi^{\mathcal{M}}$ is nonempty. Checking PDL satisfiability of a PDL formula is EXPTIME-complete in the size of the formula [1].

PDL enjoys two properties that are of particular interest for our aims. The first is the *tree model property*, which says that every model of a formula can be unwound to a (possibly infinite) tree-shaped model (considering states in $\Delta^{\mathcal{M}}$ as nodes and relations interpreting actions as edges). The second is the *small model property*, which says that every satisfiable formula admits a finite model whose number states $|\Delta^{\mathcal{M}}|$ is at most exponential in the size of the formula itself.

We use the standard abbreviations for booleans, (e.g., $true, false, \rightarrow$). We also use "$-$" as an abbreviation for program $(\cup_{a\in\mathcal{A}} a)$, which denotes the execution of the next action, "$-a$" to denote all the actions in $\mathcal{A}$ except a, and "u" as an abbreviation for the program $(\cup_{a\in\mathcal{A}} a)^*$. Notice that $[u]$ represents the *master modality*, which can be used to state universal assertions [2].

2.2 Encoding in PDL

Assume that all services are in $\texttt{Colombo}^{\mathrm{k,b}}$, and assume No External Modifications. Let $\mathcal{G} = (C, \{G\}, L)$ be a goal system and $\mathcal{U} = \{S_1, \ldots, S_n\}$ a finite family of available web services, all of which satisfy Blocking Behavior and Bounded Access. Let p be the number of states and q the size of the local store of the mediator to be synthesized.

We present selected parts of the PDL formula $\Phi^{\mathcal{G},\mathcal{U}}_{p,q}$ encoding the composition synthesis problem. The formula is formed by a conjunction including the following formulas.

General constraints. There are several bookkeeping constraints that need to be formulated in PDL. The most important are:

$$[*](exec_q \rightarrow \neg exec_p)$$

for $p \neq q$ and $p, q = 0, 1, \ldots, n$. This says that only one component service or the mediator can be in execution at each step (sometimes the goal also will be executing).

$$[*](st_i^p \wedge \neg exec_p \rightarrow [-]st_i^p)$$

which says that if S_p, $p = 0, 1, \ldots, n, g$, is not executing, it does not change state.

There are some requirements on final states.

$$[*](Final^g \rightarrow Final^1 \wedge \ldots \wedge Final^n \wedge FINAL^0)$$

that says that when the goal is in a final state then also the component services and the (to be synthesized) mediator is in a final state.

$Final$ is defined in the obvious way

$$[*](st^p \rightarrow Final^p)$$

for every final state st^p of the service S_p with $p = 1, \ldots, n, g$.

Constraints on guessing the Mediator behavior. The following constraints force the mediator to make exactly the same choices every time it finds itself in the same circumstances.

$$\begin{array}{l}\langle * \rangle(exec_0 \wedge st_i^0 \wedge \widehat{\widehat{\alpha_0}} \wedge \widehat{\widehat{\gamma}} \wedge DO(!m)) \rightarrow \\ \quad [*](exec_0 \wedge st_i^0 \wedge \widehat{\widehat{\alpha}} \wedge \widehat{\widehat{\gamma}} \rightarrow DO(!m))\end{array}$$

$$\begin{array}{l}\langle * \rangle(exec_0 \wedge st_i^0 \wedge \widehat{\widehat{\alpha_0}} \wedge \widehat{\widehat{\gamma}} \wedge NEXT(st_j^0)) \rightarrow \\ \quad [*](exec_0 \wedge st_i^0 \wedge \widehat{\widehat{\alpha}} \wedge \widehat{\widehat{\gamma}} \rightarrow NEXT(st_j^0))\end{array}$$

$$\begin{array}{l}\langle * \rangle(exec_0 \wedge st_i^0 \wedge \widehat{\widehat{\alpha_0}} \wedge \widehat{\widehat{\gamma}} \wedge MAP(\vec{q_m^0}, \vec{u})) \rightarrow \\ \quad [*](exec_0 \wedge st_i^0, \widehat{\widehat{\alpha}} \wedge \widehat{\widehat{\gamma}} \rightarrow MAP(\vec{q_m^0}, \vec{u}))\end{array}$$

$$\begin{array}{l}\langle * \rangle(exec_0 \wedge st_i^0 \wedge \widehat{\widehat{\alpha_0}} \wedge \widehat{\widehat{\gamma}} \wedge MAP(\vec{u}, \vec{q_m^i})) \rightarrow \\ \quad [*](exec_0 \wedge st_i^0, \widehat{\widehat{\alpha}} \wedge \widehat{\widehat{\gamma}} \rightarrow MAP(\vec{u}, \vec{q_m^i}))\end{array}$$

$$\langle * \rangle(st_i^0 \wedge FINAL^0) \rightarrow [*](st_i^0 \rightarrow FINAL^0)$$

As an example, the first formula states that, if the mediator is executing with current assignment $\widehat{\widehat{\alpha}}$ and it guessed to send the message $!m$ with parameters $\vec{u}$ to the service S_i, then next the assignment would be changed to $\widehat{\widehat{\alpha'}}$ that differs from $\widehat{\widehat{\alpha}}$ for the values assigned to the port for m in S_i. Also next the execution is left to S_i while the goal will not be (in fact will continue not to be) in execution.

Note that all guessed propositions are disjoint from guessed propositions in the same family. Note also that if S_0 is not executing then the values of the guessed proposition is irrelevant.

Mediator reads message from a service. These are the subformulas that characterize the mediator reading from a component service.

If the mediator is prescribed to do next $?m$ then it does it, and goes to a guessed state:

$$\begin{array}{l}[*](exec_0 \wedge DO(?m) \wedge NEXT(st_l^0 imp \\ \quad (\langle ?m \rangle \top \wedge [-?m]\bot \wedge \\ \quad [?m]st_j^0))\end{array}$$

In doing $?m$ it is possible to guess in which variable of its local store the mediator put the contents of the massage:

$$[*](exec_0 \wedge DO(?m) \wedge \widehat{\widehat{\alpha}} \wedge \textstyle\bigwedge_i MAP(\vec{q_m^0}, \vec{u}) \rightarrow [?m]\widehat{\widehat{\alpha'}})$$

where $\widehat{\widehat{\alpha'}}$ is the result of updating $\widehat{\widehat{\alpha}}$ by assigning the values in the queues $\vec{q_m^0}$ associated with the port for the message m to the variables $\vec{u}$ in the local store of S_0. Notice that in fact only α'_0 is in general different from α_0, while α'_i, and α'_g remain equal to α_i, and α_g, respectively.

If the mediator does $?m$ then next it will continue executing while the goal will not:

$$[*](exec_0 \wedge DO(?m) \rightarrow [?m](exec_0 \wedge \neg exec_g))$$

The svc and the world state instance remain unchanged:

$$\begin{array}{l}[*](exec_0 \wedge DO(?m) \wedge \widehat{\widehat{\gamma}} \rightarrow [?m]\widehat{\widehat{\gamma}}) \\ [*](exec_0 \wedge DO(?m) \wedge \widehat{\widehat{\mathcal{I}}} \rightarrow [?m]\widehat{\widehat{\mathcal{I}}})\end{array}$$

Mediator sends a message to a service. These are the subformulas that characterize the mediator sending a message to a component service.

If we guess that the mediator does $!m$ next then it does it, and goes to a guessed state:

$$\begin{array}{l}[*](exec_0 \wedge DO(!m) \wedge NEXT(st_j^0) \rightarrow \\ \quad (\langle !m \rangle \top \wedge [-!m]\bot) \wedge \\ \quad [!m]st_j^0)\end{array}$$

In doing $!m$ we guess in which variable of its local store the mediator puts in the contents of the massage; next the chosen service will be in execution while the goal will not:

$$\begin{array}{l}[*](exec_0 \wedge DO(!m) \wedge \widehat{\widehat{\alpha}} \wedge \bigwedge_i MAP(\vec{u}, q^{\vec{i}}_m) \rightarrow \\ \quad [!m](\widehat{\widehat{\alpha'}} \wedge exec_i \wedge \neg exec_g))\end{array}$$

where $\widehat{\widehat{\alpha'}}$ is the result of updating $\widehat{\widehat{\alpha}}$ by assigning the values in the variables $\vec{u}$ of the local store of S_0 to the the queue $q^{\vec{i}}_m$ associated with the port for the message m of the service S_i; the execution is given to service S_i. Notice that in fact only the part of α' that is different from α' is the part relative to the port variables for message m in service S_i.

The svc and the world state instance remain unchanged:

$$[*](exec_0 \wedge DO(!m) \wedge \widehat{\widehat{\gamma}} \rightarrow [!m]\widehat{\widehat{\gamma}})$$
$$[*](exec_0 \wedge DO(!m) \wedge \widehat{\widehat{\mathcal{I}}} \rightarrow [!m]\widehat{\widehat{\mathcal{I}}})$$

Service S_i receives a message. These are the subformulas that characterize a service receiving a message from the mediator.

Let S_i being in execution in a state st^i_h with current svc $\widehat{\widehat{\gamma}}$ and current assignment $\widehat{\widehat{\alpha}}$. Let also assume that S_i in st^i_h has a transition labeled by an guarded action $\phi/?m(\vec{x})$ getting to a state $st^i_{h'}$ and that ϕ evaluates true wrt $\widehat{\widehat{\alpha}}$ and $\widehat{\widehat{\gamma}}$. Then we have

$$\begin{array}{l}[*](exec_i \wedge st^i_h \wedge \widehat{\widehat{\gamma}} \wedge \widehat{\widehat{\alpha}} \rightarrow (\langle ?m \rangle \top \wedge [-?m]\bot \wedge \\ \qquad\qquad [?m]st^i_{h'} \wedge \\ \qquad\qquad [?m]\widehat{\widehat{\alpha'}}))\end{array}$$

where $\widehat{\widehat{\alpha'}}$ is obtained from $\widehat{\widehat{\alpha}}$ by coping the symbolic values from the port $q^{\vec{i}}_m$ to the variables $\vec{x}$. The above formula says that: (1) next the action $?m$ will be performed (and no other action are possible); (2) next state for S_i will be $st^i_{h'}$; (3) the assignment is changed to $\widehat{\widehat{\alpha'}}$.

The execution remains to service S_i, which now follows the goal service S_g:

$$[*](exec_i \wedge \langle ?m \rangle \top \rightarrow [?m](exec_i \wedge exec_g))$$

The svc and the world state instance remain unchanged:

$$[*](exec_i \wedge \langle ?m \rangle \top \wedge \widehat{\widehat{\gamma}} \rightarrow [?m]\widehat{\widehat{\gamma}})$$
$$[*](exec_i \wedge \langle ?m \rangle \top \wedge \widehat{\widehat{\mathcal{I}}} \rightarrow [?m]\widehat{\widehat{\mathcal{I}}})$$

Service S_i sends a message. These are the subformulas that characterize a service sending a message to the mediator.

Let S_i being in execution in a state st^i_h with current svc $\widehat{\widehat{\gamma}}$ and current assignment $\widehat{\widehat{\alpha}}$. Let also assume that S_i in st^i_h has a transition labeled by an guarded action $\phi/!m(\vec{x})$ getting to a state $st^i_{h'}$ and that ϕ evaluates true wrt $\widehat{\widehat{\alpha}}$ and $\widehat{\widehat{\gamma}}$. Then we have

$$\begin{array}{l}[*](exec_i \wedge st^i_h \wedge \widehat{\widehat{\gamma}} \wedge \widehat{\widehat{\alpha}} \rightarrow \\ \quad (\langle !m \rangle \top \wedge [-!m]\bot \wedge \\ \quad [!m]st^i_k \wedge \\ \quad [!m]\widehat{\widehat{\alpha'}}))\end{array}$$

where $\widehat{\widehat{\alpha'}}$ is obtained from $\widehat{\widehat{\alpha}}$ by coping the symbolic values from the variables $\vec{x}$ to the port $q^{\vec{0}}_m$. The above formula says that: (1) next the action $!m$ will be performed (and no other action are possible); (2) next state for S_i will be $st^i_{h'}$; (3) the assignment is changed to $\widehat{\widehat{\alpha'}}$.

The execution is given to the mediator S_0 and the goal execution is interrupted and the execution of a read $?m$ in the mediator is prescribed:

$$\begin{array}{l}[*](exec_i \wedge \langle !m \rangle \top \rightarrow \\ \quad [!m](exec_0 \wedge \neg exec_g \wedge DO(?m)))\end{array}$$

The svc and the world state instance remain unchanged:

$$[*](exec_i \wedge \langle !m \rangle \top \wedge \widehat{\widehat{\gamma}} \rightarrow [!m]\widehat{\widehat{\gamma}})$$
$$[*](exec_i \wedge \langle !m \rangle \top \wedge \widehat{\widehat{\mathcal{I}}} \rightarrow [!m]\widehat{\widehat{\mathcal{I}}})$$

For brevity, we do not report the characterization of the client and the interaction with the client. As for the execution of atomic process we refer to the main body of the paper.

References

[1] M. J. Fischer and R. E. Ladner. Propositional Dynamic Logic of Regular Programs. *J. of Computer and System Sciences*, 18:194–211, 1979.

[2] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. The MIT Press, 2000.