ASEHA: A Web Services Protocol Modelling Formalism

Pemadeep Ramsokul
Arcot Sowmya
School of Computer Science and Engineering
The University of New South Wales, Sydney 2052, Australia
and
National ICT Australia(NICTA)
Locked Bag 6016, NSW 1466, Australia
. Email: {pkramsok, sowmya}@cse.unsw.edu.au

UNSW-CSE-TR-0521
November 2005

THE UNIVERSITY OF
NEW SOUTH WALES

School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia

Abstract

Agents require standard and reliable protocols to interact with different service providers in order to provide high quality service to customers over the web. Many useful protocols are coming into the market, but are ambiguously specified by protocol designers and without being fully verified. These can lead to interoperabilty problems among implementations of same protocol and high software maintenance costs.

In this paper, we propose a hierarchical automata-based framework to model the necessary features of protocols to verify their correctness. Our experience shows that the graphical models produced, provide invaluable insights and can be used to complement specifications to drastically reduce, if not eliminate, ambiguities. We illustrate our formalism with a version of *WS-AtomicTransaction* protocol.

1 Introduction

Among their many goals, Web Services(WS) are designed to help agents to communicate, through a standard interface, with their partners regardless of platforms, thereby solving the interoperability problem.

Figure 1 illustrates a fictional travel arrangement facility, which is often used in the literature [10]. In this case, the agent and service providers all belong to the same company (Z) but are potentially in different locations. Customers wishing to go on a trip may want to book a ticket with Z Airlines, reserve a room with Z Hotel, and rent a car with Z Car Services. They may want the ticket only if the hotel and car are also available and the agent must find a way to make this possible.

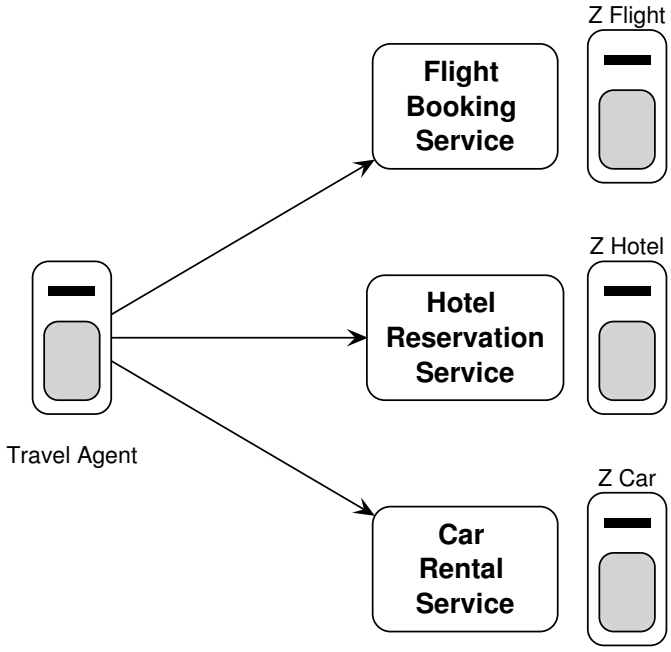

Figure 1: Travel arrangement facility

A standard protocol that allows collaboration among these partners would be highly useful in this situation. Many such protocols have been defined over the past few years, the well known ones being WS-AtomicTransaction (WS-AT) [8], WS-BusinessActivity [9] and Business Transaction Protocol [4].

Unfortunately, the use of non-formal language in the specifications re-introduces the interoperability problem among the implementations of the same protocol, which is supposedly addressed by WS. This is because the non-formal language

could be, and *often is*, interpreted differently by different service providers and furthermore, not all possible interaction scenarios may have been fully considered in the specifications. Consequently, these can potentially lead to financial loss due to costs of software maintenanace.

There have been research efforts towards modelling and automated verification[1, 3, 5, 6, 7, 16, 17, 20, 23], which work well for network protocols. Due to significant differences between network and WS protocols, these formalisms are not directly applicable to the latter. On the other hand, the recent formalisms for WS protocols [13, 14] have not shown any noticeable signs of adoption in the industry perhaps due to their complexity and intimidating syntax.

We present a formalism called *Asynchronous Extended Hierarchical Automata* (ASEHA) inspired by EHA[19] that is informed by the study of WS protocols. It has a sound mathematical foundation and aims to reconcile desirable features found in other formalisms while still maintaining syntactic and semantical simplicity.

We are using this formalism to model and verify commonly used WS protocols. Consequently, we find that the formalism provides invaluable insights and previously 'unthinkable' possible scenarios about the protocol in question. The modelling exercise also leads to concise, unambiguous and comprehensive documentation of the functional aspects of the protocols that can be used by both protocol designers and implementation teams.

1.1 Related Work and Overview

The area of protocol verification has been studied extensively over the past decades. The recent survey by Bhaduri and Ramesh [2] uncovers most of the recent advances. Most of the graphical models are based on statecharts[11] variants. While there are many different graphical models with different semantics, the verification is mainly handled by model checking using SMV[18] and SPIN[12].

Research in [1, 3, 5, 6, 7, 16, 17, 20, 23], including EHA[19], work well for network protocols but are not directly applicable to WS protocols. The main differences between network and WS protocols include heterogeneity of entities, communication mechanism and the complexity of the whole protocol interaction. Furthermore, some WS protocols operate using a two pipe model meaning that there is an application message exchange and a separate protocol message exchange. The application exchange introduce causal dependencies not apparent in the protocol message echange. These will become apparent in the later sections of this paper.

However, limitations in the existing formalisms include concise *graphical* representation of message exchange permutations and reliance on broadcasting as the main means of communication. We have instead adopted the shared buffer model which closer to what actually happens during WS protocol interactions. It also

abstracts away unnecessary communication details while allowing analysis of correctness of the protocols. We have also integrated desirable features from existing formalisms such as state hierarchy, inter-level transitions (supported indirectly), concurrency and variable support[2, 22].

The models used in [13, 14] have yet to enjoy great success in the industry, most likely due to their syntactic and semantic complexity. It is believed that the primary resistance for adopting formal methods is that designers are unfamiliar with specification processes, notations and strategies. Our formalism is very simple and intuitive to understand and use. We believe that a graphical approach to WS protocol specification is more likely to be successful while abstracting away the mathematics behind it.

1.2 Paper Contributions

In this paper, we describe an automata based framework for modelling the essential components of WS protocols in order to assess their correctness. This framework can be used by designers to make sure that their protocols work as intended before release into the market. We claim that the graphical model produced together with some additional information, should be adequate for engineers to understand and implement the protocols. Furthermore, the framework can also be used by an 'external reviewer' to model newly released protocols and determine their correctness.

Correctness is determined by automatic verification—the model is translated into the input language of a suitable tool such as SPIN. Verification, however, is not focused here.

We are currently using this framework to generate automata based models for some popular WS protocols, which can complement their specification and drastically reduce their size.

The paper is organised as follows: Section 2 presents a complete informal overview of the proposed framework, Section 3 contains the formalization of ASEHA while introducing its different components in a bottom-up fashion. We give the semantics of ASEHA in Section 4 and discuss its features in Section 5. A case study is given in Section 6 and finally we conclude in Section 7 with some hints on future work.

2 Overview of ASEHA

We illustrate our framework by a simple version of WS-AT. In Figure 2, the $WSAT$ automaton, called the *root automaton*, has one AND-state[11, 19] ($w0$), as indicated by the presence of dashed lines and three basic states namely co, ab and ex.

$w0$ is also the default state for $WSAT$ as identified by an incoming arrow and co, ab and ex are its final basic states as indicated by the double borders. $w0$ is mapped to three automata: $Participant$, $Initiator$ (Figure 3) and $Coordinator$ (Figure 4). A non-basic state is implicitly a final state, therefore, $w0$ is also a final state of $WSAT$.

Each entity of the protocol has an ID associated with it and a set of variables and is modelled by a single automaton or a set of automata. In this case, the $Participant$ and $Initiator$ each represent one entity with IDs P and I respectively. The $Coordinator$ automaton has three basic states ($c0$, $c1$, $c5$) and has three OR-states[11, 19] ($c2$, $c3$, $c4$) as indicated by the shade difference. The $Registration$ automaton, which *refines* $c2$ is shown on the right in Figure 4. $c3$ and $c4$ in turn are refined by $CommitTrans$ and $AbortTrans$ (Figure 5) respectively. Together, $Coordinator$, $Registration$, $CommitTrans$ and $AbortTrans$ form an entity with ID C and have access to the variables p, v and d.

At the beginning the default state of the root automaton($w0$) is active and consequently makes the states $p0, i0$ and $c0$ active. In ASEHA, only one transition can be taken at a time and in an asynchronous fashion. Each transition label has a *trigger*, an *action* and a *target determinator* component. For a transition to be enabled, the trigger must evaluate to true and the action must be unblocked.

In the label of the transition, from state $p0$ to $p1$, $in(i4)$ and $Register, volatile!C$ are the trigger and the action components respectively. The transition can only be taken if state $i4$ in the $Initiator$ automaton, is active (which is currently not the case).

Thus, the only possible transition, at this point, is $CCC!C$, which causes the message CCC to be *put* in the shared buffer and can only be removed by an automaton belonging to the entity C. After this transition is taken, $i0$ becomes inactive and $i1$ becomes active. The only next possible transition is $CCC?I$ found in the $Coordinator$ automaton. This causes the removal(reception) of the CCC message from the shared buffer by a *get* action. A get is blocking whereas a put is always non-blocking. This process continues until no more transitions are possible. Correctness is determined by checking if all active states are final states.

When $c2$ eventually becomes active, it also causes $r0$, the default state of $Initiator$, to become active simultaneously. If there are more than one enabled transition, one is taken non-deterministically. Later, if the transition $c2$ to $c5$ is taken, $c2$ becomes inactive and so does the active state within the $Registration$ automaton.

At some point, say the transition $c2$ to $c4$ is taken, $c4$ will become active. As indicated by the target determinator $\{a0\}$, $a0$ will become active (since $c4$ is refined by $AbortTrans$) regardless of $AbortTrans$'s default state. However, in this case, the target determinator is redundant.

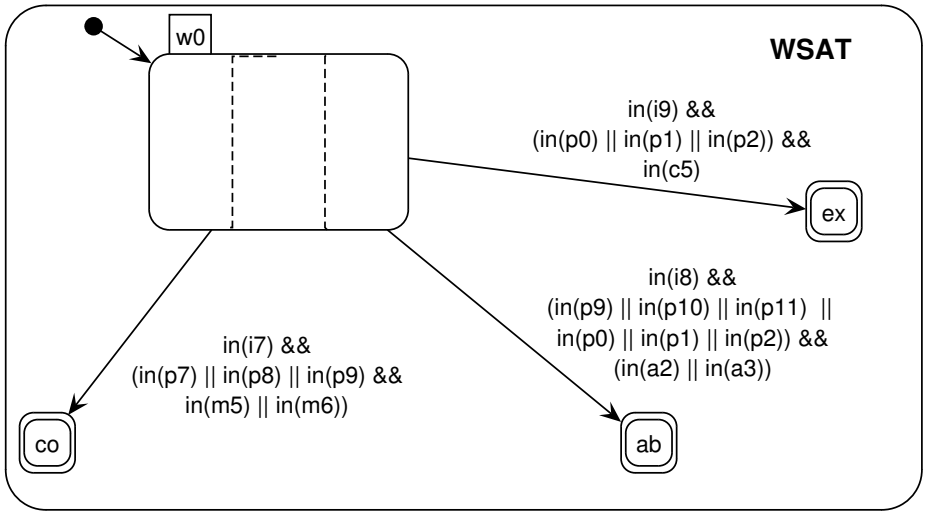

Figure 2: WSAT automaton

3 Formal Definitions

An ASEHA consists of many components and rules, which are discussed in the following subsections. We begin by defining the most basic component of an ASEHA—a sequential automaton.

3.1 Definition of Sequential Automaton

Let A be a sequential automaton, i.e. one in which at most one transition can be made at a time. A is defined by the tuple $(\Sigma, \iota, \Lambda, \Phi, \rightarrow)$ where Σ is the set of states, ι is the initial state, Λ is the set of transition labels (which we will define precisely later), Φ is the set of final states and $\rightarrow \subseteq \Sigma \times \Lambda \times \Sigma$ is the transition relation. When A is entered, the default state ι becomes active, unless specified otherwise by the incoming transitions's target determinator (discussed in Section 3.6.1).

Example 1 *For the sequential automaton* $WSAT$, $\Sigma = \{w0,\ co,\ ab,\ ex\}$, $\iota = w0$, $\Phi = \Sigma$, $\rightarrow = \{(w0,\ \ell_l,\ co),\ (w0,\ \ell_2,\ ab),\ (w0,\ \ell_3,\ ex)\}$, *where* $\ell_1, \ell_2, \ell_3 \in \Lambda$.

A sequential automaton can have its states mapped to one or more automata to form a hierarchy. This mapping is specified by the *refinement function* defined next.

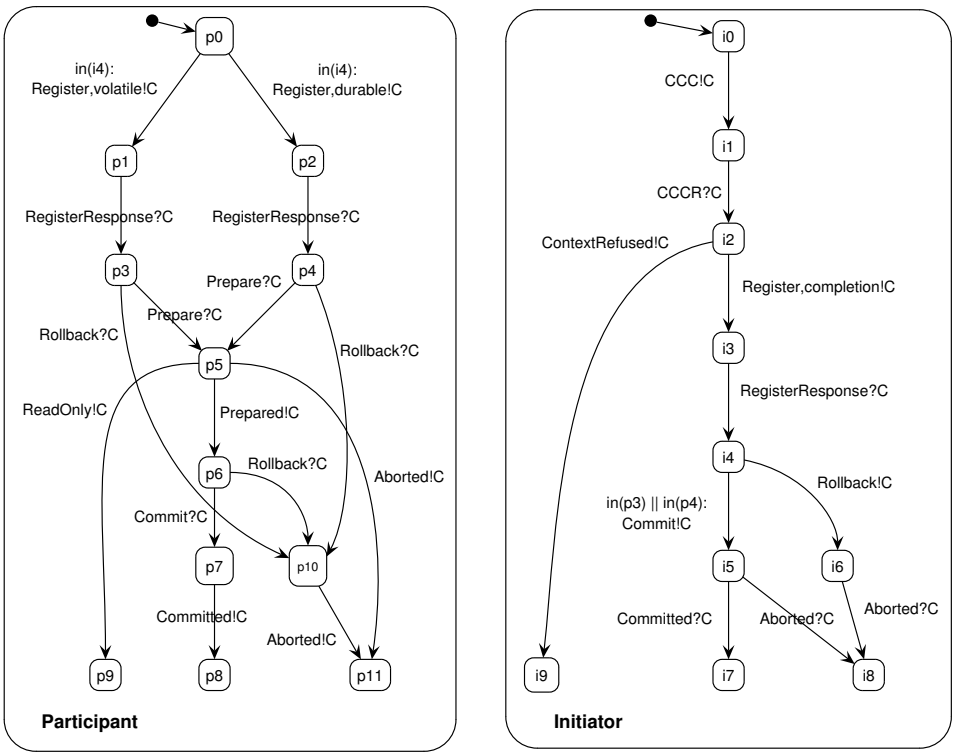

Figure 3: Participant and Initiator automata

3.2 Definition of Refinement Function

For a set of sequential automata $\mathcal{F} = \{A_1, ..., A_n\}$ with mutually disjoint state spaces, $\Sigma_{\mathcal{F}} = \bigcup_{A \in \mathcal{F}} \Sigma_A$ is the union of all states of all automata in $\mathcal{F}$. The refinement function $\gamma : \Sigma_{\mathcal{F}} \rightarrow 2^{\mathcal{F}}$ maps a state s of a sequential automaton to a subset of automata $G \subseteq \mathcal{F}$; s is said to be refined by G in such a case.

For any $s \in \Sigma_{\mathcal{F}}$, if $|\gamma(s)| = 0$, s is a basic state, else if $|\gamma(s)| = 1$, s is an OR-state else for $|\gamma(s)| > 1$, s is an AND-state. Moreover, if $\gamma(s) = \{A'_1, .., A'_n\}$, we say that s is non-basic, $A'_1, .., A'_n$ are its sub-automata, they refine s and s is their parent state. If $s \in \Sigma_A$, A is the parent automaton of $A', .., A'_n$.

Analogous to [19], for γ to be a valid function, the following conditions must hold:

- $\mathcal{F}$ can only have one root automaton (denoted by γ_{root}), i.e. $\exists\, A \in \mathcal{F} : A \notin range(\gamma)$. γ_{root} is the top-level automaton, which *does not* have a parent state. It glues the entities together.

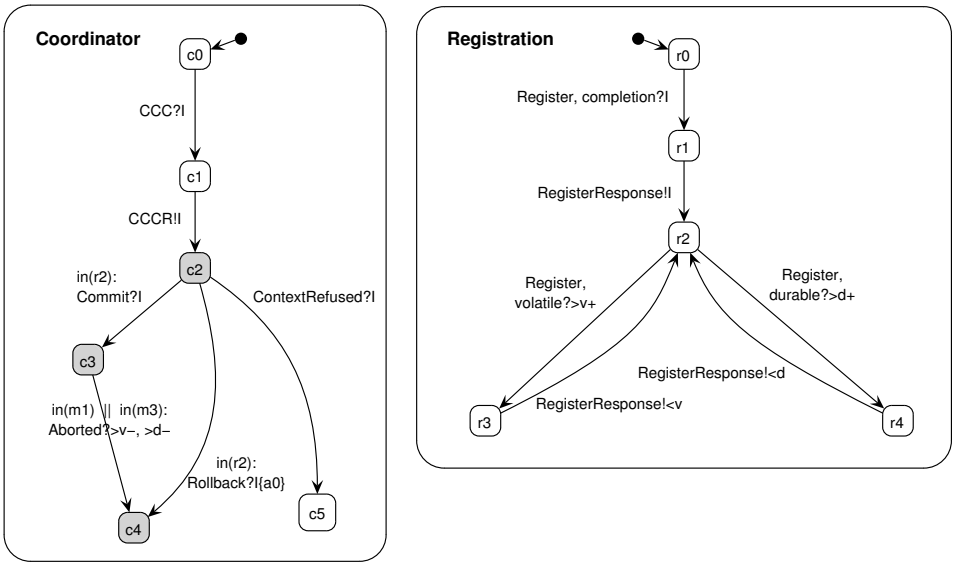

Figure 4: Coordinator and Registration automata

- Each automaton, except from the root γ_{root}, has exactly one parent state i.e. $range(\gamma) = \mathcal{F} \setminus \{\gamma_{root}\} \wedge \forall A \in \mathcal{F} \setminus \{\gamma_{root}\}\, \exists\, s, s' \in \Sigma_{\mathcal{F}}, A \in \gamma(s) \wedge\ A \in \gamma(s') \Rightarrow s = s'$.
- All non-basic states of the automata of $\mathcal{F}$ are final states, i.e. $\forall A \in \mathcal{F} : (s \in A \wedge \gamma(s) \neq \phi) \Rightarrow s \in \Phi_A$.

These conditions imply that γ does not contain any cycle.

Example 2 *The refinement function on* $\mathcal{F} = \{WSAT,\ Participant,\ Initiator,\ Coordinator, Registration,\ CommitTrans,\ AbortTrans\}$ *with* $\gamma_{root} = WSAT$ *is:*

$$\begin{aligned}\gamma \;=\; & \{(w0,\ \{Coordinator,\ Participant,\ Initiator\}), (c2,\ \{Registration\}),\\ & (c3,\ \{CommitTrans\}),\ (c4,\ \{AbortTrans\})\} \cup\\ & \{(s,\ \phi) \mid s \in \Sigma_{\mathcal{F}} \setminus \{w0,\ c2,\ c3,\ c4\}\}\end{aligned}$$

γ induces a successor function $\theta : \mathcal{F} \rightarrow 2^{\mathcal{F}}$. It returns all the automata whose parent automaton is A:

$$\theta(A) = \{A' \,|\exists\, s \in \Sigma_A : A' \in \gamma(s)\}$$

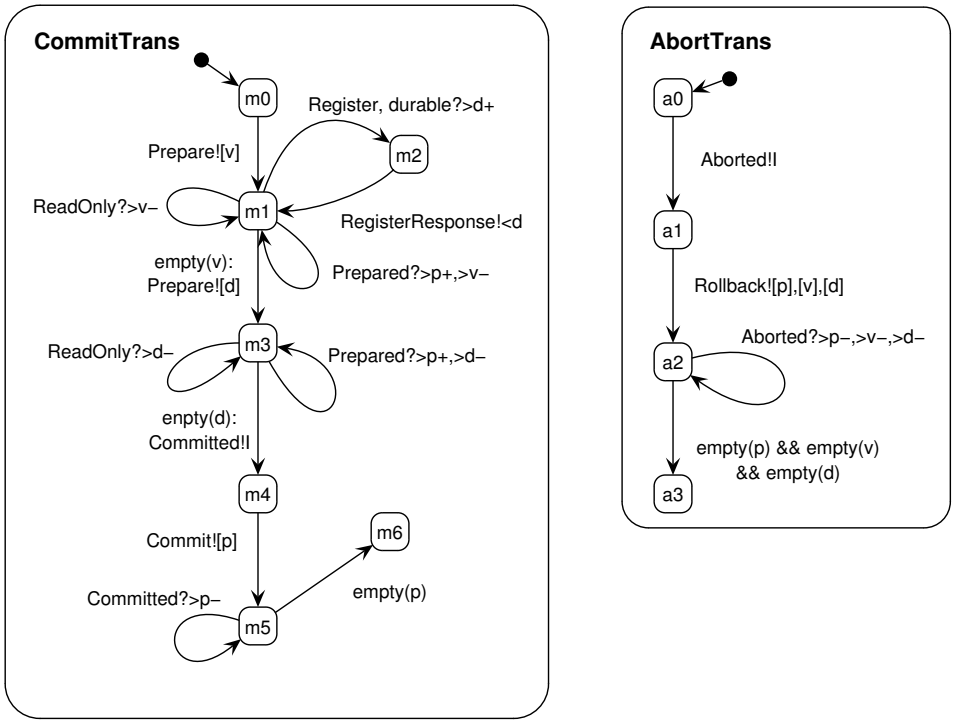

Figure 5: CommitTrans and AbortTrans automata

The irreflexive and reflexive transitive closures of θ are denoted by θ^i and θ^r respectively, i.e. $\forall\, A \in \mathcal{F}, \mathcal{A} \subset \mathcal{F} : (A, \mathcal{A}) \in \theta^i \Rightarrow A \notin \mathcal{A}$ and $\forall\, A \in \mathcal{F}, \exists\, \mathcal{A} \subseteq \mathcal{F} : (A, \mathcal{A}) \in \theta^r \wedge A \in \mathcal{A}$.

Example 3 *Using the definition of the refinement function in Example 2, we have:*

$\theta = \{(WSAT,\, \{Participant,\, Initiator,\, Coordinator\}),\, (Coordinator,\, \{Registration,\, CommitTrans, AbortTrans\})\} \cup \{(A,\, \phi) \mid A \in \mathcal{F} \backslash \{WSAT,\, Coordinator\}\}$

$\theta^i = \{(WSAT, \mathcal{F} \setminus \{WSAT\}), (Coordinator,\, \{Registration,\, CommitTrans,\, AbortTrans\})\} \cup \{(A,\, \phi) \mid A \in \mathcal{F} \setminus \{WSAT,\, Coordinator\}\}$

$\theta^r = \{(WSAT, \mathcal{F}),\, (Coordinator,\, \{Coordinator,\, Registration,\, CommitTrans,\, AbortTrans\})\} \cup \{(A,\, \{A\}) \mid A \in \mathcal{F} \setminus \{WSAT,\, Coordinator\}\}$

3.3 Entity Model

An entity that participates in a protocol is modelled by a sequential automaton or by a set of automata. Each entity has an ID and a set of variables associated with it. An ID can be thought of as a shorter form of the entity's name for keeping the labels of transitions short. The mapping of an automaton to its entity ID is specified by the *EID lookup function.*

Entities communicate with each other by exchanging messages through the shared buffer using put and get actions, as shown in Figure 6. Furthermore, only entities in ASEHA can access the buffer i.e. ASEHA is a closed system. To cope

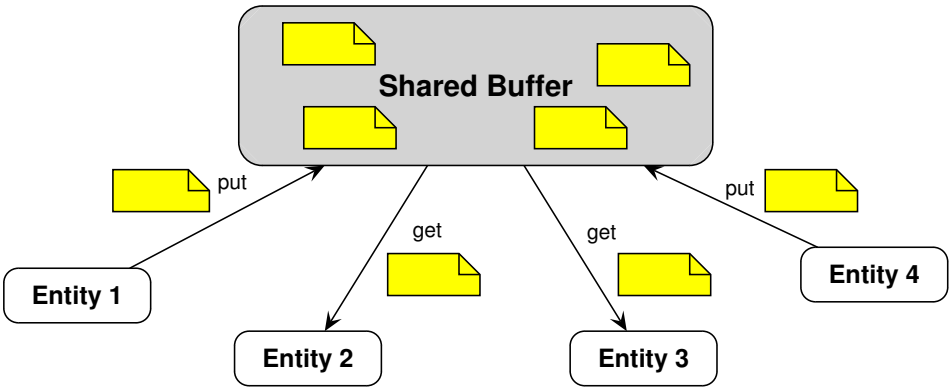

Figure 6: Interaction of entities

with the complexity of dealing with many entities, variables are of great help. They are mainly used for remembering certain responses of a group of entities and to handle message exchange permutations. The *evaluation function* keeps track of variables and their values.

3.3.1 Definition of EID Lookup Function

For a set of entity IDs $\mathcal{D} = \{d_1, .., d_m\}$, where $m \leq |\mathcal{F}|$, an EID lookup function, $\eta : \mathcal{F} \rightarrow \mathcal{D}$, maps an automaton to its entity ID. The η function indicates the entity that an automaton is part of—the root automaton is strictly not part of any entity as it represents the whole protocol interaction, but is given an entity ID to simplify the model. η is valid iff:

- All automata that refine the states of the root automaton γ_{root} have different EIDs, i.e. $\forall\, A, A' \in \theta(\gamma_{root}) : (\eta(A) = d \wedge \eta(A') = d) \Rightarrow A = A'$, where $d \in \mathcal{D}$.

- All automata except the root γ_{root} and the automata which refine its states, have the same EID as their parent automaton, i.e. $\forall A \in \theta(\gamma_{root}), A' \in \theta^i(A) \Rightarrow \eta(A) = \eta(A')$.

Hereafter, we will refer to η as 'an EID function'.

3.3.2 Definition of Evaluation Function

Each entity with ID d, has a set of variables $\mathcal{V}_d$ associated with it. A variable can only hold EIDs for values. An evaluation function $\xi : \mathcal{V} \rightarrow \mathcal{D} \times 2^{\mathcal{D}}$, returns the *current* value and the set of buffered (remembered) values of a variable. A variable $v \in \mathcal{V}$ is initialized to (d, ϕ) unless specified otherwise. We define selector functions as follows: if $\xi(v) = (d, R)$, then $val(v) = d$ and $buffer(v) = R$.

Example 4 *Continuing from Example 2, for $\mathcal{D} = \{W, P, I, C\}$, we have $\eta = \{(WSAT, W), (Coordinator, C), (Registration, C), (CommitTrans, C), (AbortTrans, C), (Participant, P), (Initiator, I)\}$, and each entity's set of variables is given by: $\mathcal{V}_W = \phi$, $\mathcal{V}_C = \{p, v, d\}$, $\mathcal{V}_P = \phi$ and $\mathcal{V}_I = \phi$.*

This means that the $Coordinator$, $Registration$, $CommitTrans$ and $AbortTrans$ automata belong to the entity with ID = C and only these automata have access to the variables p, v, and d. Also, $WSAT$, $Participant$ and $Initiator$ belong to the entity with ID = W, ID = P and ID = I respectively but have no variables.

3.4 Definition of ASEHA

An ASEHA $\mathcal{H}$ is defined by $\mathcal{H} = (\mathcal{F}, \mu, \mathcal{V}, \gamma, \eta)$, where $\mathcal{F} = \{A_1, ..., A_n\}$ is a set of sequential automata with mutually disjoint state spaces, μ is a set of messages, $\mathcal{V} = \biguplus_{d \in \mathcal{D}} \mathcal{V}_d$ is the set of variables (disjoint union of the variables of all entities), γ is a refinement function on $\mathcal{F}$, and η is the EID function on $(\mathcal{F}, \mathcal{D}, \gamma)$, where $\mathcal{D}$ is a set of EIDs.

Furthermore, $\mathcal{B} \subseteq \mathcal{D} \times \mu \times \mathcal{D}$ represents the contents of the shared buffer used for communication among the entities. Intuitively, it stores messages that have been sent but not yet received.

Moreover, the visited list $\mathcal{T} \subseteq \bigcup_{A \in \mathcal{F}} \Sigma_A$, stores the list of all states visited so far. A state is s said to be visited when it becomes active. In section 4.2, we will explain how and when $\mathcal{T}$ is updated.

We next formalize the notion of configuration before defining a transition label in ASEHA.

3.5 Definition of Configuration

A configuration is a set of states in $\mathcal{H}$ that can be active *simultaneously*. Analogous to [19], a set $C \subseteq \bigcup_{A \in \mathcal{F}} \Sigma_A$ is a configuration iff:

- Any automaton contributes at most one state to the configuration i.e. $A \in \mathcal{F} \Rightarrow |\Sigma_A \cap C| \leq 1$
- Downward closure: if a state s is in C then its descendants (states lower that itself in the hierarchy) are in C, i.e. $s \in C \wedge A \in \gamma(s) \Rightarrow \exists\, s' \in \Sigma_A, s' \in C$.
- Upward closure: if a state s is in C, then its ancestors (states higher that itself in the hierarchy) are in C, i.e. $\{s\} = \Sigma_A \cap C \wedge (\exists\, s' \in \Sigma_{A'} : A \in \gamma(s')) \Rightarrow s' \in C$.

The set of all configurations is given by $Conf(\gamma)$ and the initial configuration $C_{\gamma_{root}}$ is the set containing only default states, i.e. $\{C_{\gamma_{root}}\} = \{C \,|C \in Conf(\gamma) \wedge \forall\, s \in C\, \exists A \in \mathcal{F} : s = \iota_A\}$

Example 5 *If we define $\mathcal{H}$ using the information from the previous examples, we have the initial configuration $C_{\gamma_{root}} = \{w0, p0, i0, c0\}$.*

3.6 Definition of Transition Label

Interlevel transitions[11, 19] are not allowed but are simulated using triggers and target determinators. The motivations behind disallowing interlevel transitions are mainly that they introduce a lot of semantic complexity and make compositionality very hard, if not impossible[19].

Recall that an automaton $A \in \mathcal{F}$ is defined by the tuple $(\Sigma_A, \iota_A, \Lambda_A, \Phi_A, \rightarrow_A)$ and $\rightarrow_A = \Sigma_A \times \Lambda_A \times \Sigma_A$. A label $\ell \in \Lambda_A$ of a transition can be further defined as a 3-tuple (tr, ac, td), where tr is the trigger, ac is the action taken and $td \subseteq \Sigma_{\mathcal{F}}$ is the target determinator, where $\Sigma_{\mathcal{F}} = \bigcup_{A' \in \mathcal{F}} \Sigma_{A'}$.

The syntax of the labels used in the examples is:

$$trigger : action\{target\ determinator\}$$

If the trigger and the colon(:) are omitted, the trigger is assumed to be $true$, if the action is omitted, it is assumed to be τ and if the target determinator, enclosed in braces ({}), is omitted, it is assumed to be empty.

3.6.1 Target Determinator

For the transition $(s, (tr, ac, td), s') \in \rightarrow_A$, if s' is a non-basic state, the default states of the sub-automata are recursively entered unless the target determinator td has been specified.

If C is the current configuration then, td is valid iff

$$td \neq \phi \Rightarrow [head \cup \{s'\} \cup td] \in Conf(\gamma)$$

where $head = \left(C \setminus \left(\bigcup_{A' \in \theta^r(A)} \Sigma_{A'}\right)\right)$.

Intuitively, if td is specified (non-empty), it has to specify the set of states to enter such that the resulting set($head \cup \{s'\} \cup td$) is a valid configuration. $\left(\bigcup_{A' \in \theta^r(A)} \Sigma_{A'}\right)$ is the set of states that will be exited and $\{s'\} \cup td$ is the new set of states that will be entered. Although td can possibly be specified by fewer states, we stick to the above definition since it simplifies the semantics.

Example 6 *Consider the transition between $c2$ and $c4$ in the $Coordinator$ automaton in Figure 4. If the current configuration is $C = \{w0, p4, i6, c2, r2\}$, then if the transition is taken, $\{$c2, r2$\}$ will be exited, $\{$c4, a0$\}$ will be entered while $\{$w0, p4, i6$\}$ will still be in the new configuration i.e. $C' = \{w0, p4, i6, c4, a0\}$. In this case, the target determinator $td = \{a0\}$ could have been omitted since the default state of $AbortTrans$ is $a0$.*

3.6.2 Action

As mentioned earlier, an action ac can be a put, get or $internal$. There are two versions of the get and put actions—one based on a specific entity ID and the other based on the evaluation of the *local operation sequence* (LOS) function, which we define later.

Given the current value of the message buffer $\mathcal{B}$ is B and the following actions appear in a transition $t \in \rightarrow_A$:

$put(msg, r)$ Puts the message $msg \in \mu$ for the entity with ID $r \in \mathcal{D}$. Effectively, after the put action, the new value of $\mathcal{B}$ will be $B \cup \{(\eta(A), msg, r)\}$.

$put(msg, f_t)$ Similar to the function above, except that the tuple containing the message $msg \in \mu$ is put based on the LOS function f_t to be explained in the next section.

$get(msg, d)$ Returns the retrieved tuple from the message buffer $\mathcal{B}$. Effectively, a get action finds the tuple $(d, msg, \eta(A))$ in $\mathcal{B}$, where $d \in \mathcal{D}$ is the sender's EID, $msg \in \mu$ is the message and $\eta(A)$ is A's EID. If such a tuple is

found, the receive action in A is unblocked else the receive action is blocked until such a tuple is found. When the transition is taken, the tuple is removed from $\mathcal{B}$. After the receive action, the new value of $\mathcal{B}$ becomes $B \setminus \{(d, msg, \eta(A))\}$.

$get(msg, f_t)$ Similar to the function above, except that the tuple containing the message $msg \in \mu$ is retrieved based on the LOS function f_t to be explained in the next section.

τ is an internal action, i.e. one which is not visible to the environment.

Syntactically, an action may be specified in BNF as:

$$
\begin{array}{rcl}
action & ::= & snd\_rcv \mid \text{`'} \\
snd\_rcv & ::= & message \text{ `!' } address \mid \\
 & & message \text{ `?' } address \\
address & ::= & EID \mid mem\_op \text{ [`,'} mem\_op \text{]} * \\
mem\_op & ::= & \text{`>'} var \mid \text{`<'} var \mid var \text{`+'} \mid var \text{`-'} \mid \\
 & & \text{`['} var \text{`]'} \mid \text{`>'} var \text{`+'} \mid \text{`>'} var \text{`-'} \mid \\
 & & \text{`<'} var \text{`+'} \mid \text{`<'} var \text{`-'}
\end{array}
$$

Some messages usually have parameters—in that case, they are separated by commas(,). Recall that *?* means *get* and *!* means *put* and empty action refers to τ. var represents a variable name. The syntax for local operations on variables is given in the next subsection.

The automaton performing the *get*/*put* action on the shared buffer $\mathcal{B}$ has atomic access to the latter since a transition is deemed complete only after the action has been *fully* carried out.

3.6.3 Definition of LOS function

For a transition $t = (src, (tr, ac, td), dest) \in \rightarrow_A$, the LOS function $f_t : \{1, ..., N\} \rightarrow MemAct$, returns an ordered sequence of operations that must be performed on variables while the transition t is taken—N is the number of operations to perform. As the name suggests, a LOS function defines operations on local variables only.

For any variable $v \in \mathcal{V}_{\eta(A)}$ with value $\xi(v) = (d, R)$, the set $MemAct$ consists of the following operations:

$fetch(v)$ Fetches the value of v and puts it in a set, i.e. $\{val(v)\}$, which is $\{d\}$. Syntax: $< v$.

$enlist(v)$ Fetches the set of all remembered values, i.e. $buffer(v)$, which is R. Syntax: $[v]$.

$update(v)$ Requests the update of the variable v based on the sender of a message. Syntax: $>v$.

$remember(v)$ Adds the value of v to the set of remembered values, i.e. after $remember(v)$, v becomes $(d, R \cup \{d\})$. Syntax: v+.

$forget(v)$ Removes the value of v from the set of remembered values, i.e. after $forget(v)$, v becomes $(d, R \setminus \{d\})$. Syntax: v–.

Hence, we have $MemAct = \{op \,|\exists v \in \mathcal{V}_{\eta(A)} : op = fetch(v) \vee op = enlist(v) \vee op = update(v) \vee op = remember(v) \vee op = forget(v)\}$.

The operations are executed sequentially *while* the transition t is taken, i.e. $f_t(1)$ is executed, followed by $f_t(2)$ and so on. Only certain sequence of operations are allowed based on the action ac; f_t is a valid function iff:

- If ac is a put action, the f_t function can only contain fetch and/or enlist, i.e. $\exists\, msg \in \mu : ac = put(msg, f_t) \Rightarrow \forall\, i \in \{1, 2, .., |f_t|\}, \exists\, v \in \mathcal{V} : f_t(i) = enlist(v) \vee f_t(i) = fetch(v)$.
- If ac is a get action, there must be at least one update but no fetch or atmost one fetch, which must also be first, i.e. $\exists\, msg \in \mu : ac = get(msg, f_t) \Rightarrow (\exists\, i \in \{1, 2, .., |f_t|\}, v \in \mathcal{V} : f_t(i) = update(v) \wedge \forall v \in \mathcal{V}, i \in \{1, 2, .., |f_t|\} : f_t(i) \neq fetch(v) \vee (\forall i \in \{1, 2, .., |f_t|\}\, \exists\, v \in \mathcal{V} : f_t(i) = fetch(v) \Rightarrow i = 1))$.

Some of the operations can be combined syntactically, e.g. $>v+$ stands for $\{(1, update(v)), (2, remember(v))\}$.

3.6.4 Trigger

For the transition $(src, (tr, ac, td), dest) \in \rightarrow_A$, the trigger tr is a boolean combination of predicates defined on variables or states. Predicates are side-effect free and can be connected by $\wedge$ and/or $\vee$. The unary boolean operator $\neg$ is also supported. Specifically, given the current configuration C, $v \in \mathcal{V}$ and $s \in \bigcup_{A' \in \mathcal{F}} \Sigma_{A'}$, tr is a monomial over the following propositions:

$empty(v)$ evaluates to true iff $|enlist(v)| = 0$.

$nempty(v)$ evaluates to true iff $|enlist(v)| > 0$.

$in(s)$ Evaluates to true iff $s \in C$.

$visited(s)$ Evaluates to true iff s has been visited at least once, i.e. $s \in \mathcal{T}$.

Note that the $in(s)$ predicate restricts the enabledness of a transition based on the state s and thus can be used together with td to simulate inter-level transitions. It can also be used to introduce causal dependencies between entities for synchronization.

Example 7 *The definition of μ used in the examples is: $\mu = \{CCC,\ CCCR,\ ContextRefused,\ RegisterResponse,\ Prepare,\ Prepared,\ ReadOnly, Commit,\ Committed,\ Rollback,\ Aborted\} \times \{\epsilon\} \cup \{Register\} \times \{completion,\ volatile,\ durable\}$. Due to space constraints, we will only give the set of labels Λ for the $Coordinator$ automaton.*

$$\begin{aligned}\Lambda_{Coordinator} = \ & \{(true,\ get((CCC,\ \epsilon),\ I), \phi),\\ & (true,\ put((CCCR,\ \epsilon),\ I), \phi),\\ & (in(r2), get((Commit,\ \epsilon),\ I),\ \phi),\\ & (in(r2),\ get((Rollback,\ \epsilon),\ I), \{a0\}),\\ & (in(m1) \vee in(m3),\ get((Aborted,\ \epsilon),\ f_t),\ \phi)\}\end{aligned}$$

where $f_t = \{(1, update(v)),\ (2, forget(v)),\ (3, update(d)),\ (4, forget(d))\}$.

4 Semantics of ASEHA

We define the semantics of ASEHA using a Kripke structure, $\mathcal{K}$, used by many model checkers for verification. Once we translate ASEHA into Promela (the input language of SPIN), we will have show that the Kripke structure, $\mathcal{K}'$, used by SPIN is bisimilar to $\mathcal{K}$ to make sure our translation is correct. However, this will not be further addressed in this paper.

To define $\mathcal{K}$, we first introduce the notion of enabled transitions.

4.1 Definition of Enabled Transitions

Given an ASEHA $\mathcal{H} = (\mathcal{F}, \mu, \mathcal{V}, \gamma, \eta)$, to determine which transitions are enabled at any time, we need to know the configuration C of $\mathcal{H}$, the message buffer $\mathcal{B}$, the visited list $\mathcal{T}$, and the evaluation function ξ on $(\mathcal{V}, \mathcal{D})$. Collectively, we refer to $(C, \mathcal{B}, \mathcal{T}, \xi)$ as a *status* in the rest of this paper.

A transition $t = (s, (tr, ac, td), s') \in \rightarrow_A$ is *enabled* in the status $(C, \mathcal{B}, \mathcal{T}, \xi)$, denoted by $enabled_{(C,\mathcal{B},\mathcal{T},\xi)}(t)$, iff:

- s is active, i.e. $s \in C$.

- tr evaluates to true, i.e. $(C, \mathcal{B}, \mathcal{T}, \xi) \models tr$.
- ac is unblocked, i.e. for some $d \in \mathcal{D}$, $v \in \mathcal{V}$ and LOS function f_t:
  - $ac = get(msg, d) \Rightarrow (d, msg, \eta(A)) \in \mathcal{B}$
  - $(ac = get(msg, f_t) \wedge f_t(1) = fetch(v)) \Rightarrow (val(v), msg, \eta(A)) \in \mathcal{B}$
  - $(ac = get(msg, f_t) \wedge f_t(1) \neq fetch(v)) \Rightarrow (d, msg, \eta(A)) \in \mathcal{B}$

The set of enabled transitions in $(C, \mathcal{B}, \mathcal{T}, \xi)$ is given by:

$$ET_{(C,\mathcal{B},\mathcal{T},\xi)} = \left\{ t \,|\, t \in \bigcup_{A \in \mathcal{F}} \rightarrow_A \wedge enabled_{(C,\mathcal{B},\mathcal{T},\xi)}(t) \right\}$$

Example 8 *Consider the status* $(C, \mathcal{B}, \mathcal{T}, \xi) = (\{w0,\ c2, r2, i4, p1\},\ \{(C, (RegisterResponse, \epsilon), P)\},\ \{w0,\ p0, p1,\ i0,\ i1,\ i2,\ i3,\ i4,\ c0,\ c1,\ c2,\ r0,\ r1,\ r2,\ r3\}, \{(v, (P, \{P\}), (d, (C, \phi)), (p, (C, \phi)\})$.

This means that currently: the states $w0, c2, r2, i4, p1$ *are active, the shared buffer contains the tuple* $(C, (RegisterResponse, \epsilon), P)$*, the states* $w0,\ p0,\ p1,\ i0,\ i1,\ i2,\ i3,\ i4, c0,\ c1,\ c2,\ r0,\ r1,\ r2,\ r3$ *have been visited and the variable* v *has the value* P *and has remembered* P.

In this status, the set of enabled transitions is

$$\begin{aligned} ET_{(C,\mathcal{B},\mathcal{T},\xi)} \quad = \quad & \{(p1,\ (true,\ get((RegisterResponse,\ \epsilon),\ C),\ \phi),\ p3), \\ & (r2, (true, put((Rollback, \epsilon), C), \phi), r4)\} \end{aligned}$$

4.2 Definition of Kripke Structure

The semantics of $\mathcal{H} = (\mathcal{F}, \mu, \mathcal{V}, \gamma, \eta)$ is a Kripke structure $\mathcal{K} = (S, s_0, F, \overset{STEP}{\longrightarrow})$, where:

- $S \subseteq Conf(\gamma) \times 2^{\mathcal{D} \times \mu \times \mathcal{D}} \times 2^{\Sigma_{\mathcal{F}}} \times 2^{\mathcal{V} \times (\mathcal{D} \times 2^{\mathcal{D}})}$ is the set of all possible statuses of $\mathcal{H}$, which is a subset of the cross-product of all configurations, contents of the message buffer, visited states and variables with their corresponding possible values.
- $s_0 \in S$ is the *initial status*, given by:

$$s_0 = (Conf_{\gamma_{root}}, \phi, Conf_{\gamma_{root}}, \{(v, (d, \phi)) |\, d \in \mathcal{D} \wedge v \in \mathcal{V}\})$$

- $F \subseteq S$ is the set of final (accepting) states derived as follows:

$$(C, \mathcal{B}, \mathcal{T}, \xi) \in (F \cap S) \Leftrightarrow \forall\, s \in C\, \exists\, A \in \mathcal{F} : s \in \Phi_A$$

This means that a state in $\mathcal{K}$ is accepting iff all states in the configuration are final states of automata in $\mathcal{H}$.

- $\overset{STEP}{\longrightarrow} \subseteq S \times S$ is the transition relation of $\mathcal{K}$, where $(C, \mathcal{B}, \mathcal{T}, \xi) \overset{STEP}{\longrightarrow} (C', \mathcal{B}', \mathcal{T}', \xi')$ iff there exists a transition $\{t\} = \{(s, (tr, ac, td), s')\} \in (ET_{(C,\mathcal{B},\mathcal{T},\xi)} \cap \rightarrow_A)$ for some $A \in \mathcal{F}$ such that:

$$C' = \begin{cases} head \cup \{s'\} \cup td & \textit{if } td \neq \phi \\ \mathcal{C} & \textit{otherwise} \end{cases}$$

where $head = C \setminus \left(\bigcup_{A' \in \theta^r(A)} \Sigma_{A'}\right)$ and $\{\mathcal{C}\} = \{G \,|\, G \in Conf(\gamma)) \wedge (head \cup \{s'\} \subseteq G) \wedge \forall\, x \in G \setminus (head \cup \{s'\}), \exists A' \in \mathcal{F} : x = \iota_{A'}\}$

The new configuration C' is obtained by:

  - exiting the state s and all the states below it, i.e. $\left(\bigcup_{A' \in \theta^r(A)} \Sigma_{A'}\right)$
  - entering s' i.e. $\{s'\}$
  - entering td if it is specified or entering the default states of the automata below s', if td is unspecified.

To get the contents of the new shared buffer $\mathcal{B}'$, the action ac is the carried out using the definitions in Sections 3.6.2 and 3.6.3.

$$\mathcal{B}' = \begin{cases} \mathcal{B} \cup \{(\eta(A), msg, d)\} & \textit{if } ac = put(msg, d) \\ \mathcal{B} \setminus \{(d, msg, \eta(A))\} & \textit{if } ac = get(msg, d) \\ \mathcal{B}^p & \textit{if } ac = put(msg, f_t) \\ \mathcal{B}^g & \textit{if } ac = get(msg, f_t) \\ \mathcal{B} & \textit{otherwise} \end{cases}$$

for some $msg \in \mu$, $d \in \mathcal{D}$ and LOS function f_t. $\mathcal{B}^p$ and $\mathcal{B}^g$ are obtained by sequentially 'processing' the operations in the LOS function f_t.

Let $\mathcal{B}^p_{0..|f_t|}$ be an array of set of tuples, with $\mathcal{B}^p_0 = \mathcal{B}$ and $\mathcal{B}^p_{1..|f_t|}$ be defined by:

$$\mathcal{B}^p_i = \mathcal{B}^p_{i-1} \cup \begin{cases} \{(\eta(A), msg, val(v))\} & \textit{if } f_t(i) = fetch(v) \\ \{(\eta(A), msg, x) \,|x \in buffer(v)\} & \textit{if } f_t(i) = enlist(v) \\ \phi & \textit{otherwise} \end{cases}$$

for some $v \in \mathcal{V}$. $\mathcal{B}^p$ is simply given by $\mathcal{B}^p = \mathcal{B}^p_{|f_t|}$.

$$\mathcal{B}^g = \mathcal{B} \setminus \begin{cases} \{(val(v), msg, \eta(A))\} & \textit{if } f_t(1) = fetch(v) \\ \{(addr, msg, \eta(A))\} & \textit{otherwise} \end{cases}$$

where $addr$ is an arbitrary value such that $addr \in \mathcal{D}$ and $(addr, msg, \eta(A)) \in \mathcal{B}$.

The new set of visited states $\mathcal{T}'$ is obtained by adding the states in the new configuration C' to the old set of visited states $\mathcal{T}$.

$$\mathcal{T}' = \mathcal{T} \cup C'$$

Let $\xi_{0..|f_t|}$ be an array of evaluation functions, with $\xi_0 = \xi$ and $\xi_{1..|f_t|}$ defined by:

$$\xi_i = \xi_{i-1} \setminus \{(v, (d, R))\} \cup \begin{cases} \{(v, (addr, R)\} & \textit{if } f_t(i) = update(v) \\ \{(v, (d, R \cup \{d\})\} & \textit{if } f_t(i) = remember(v) \\ \{(v, (d, R \setminus \{d\})\} & \textit{if } f_t(i) = forget(v) \\ \{(v, (d, R))\} & \textit{otherwise} \end{cases}$$

where $v \in \mathcal{V}, (d, R) = \xi_{i-1}(v)$ and $addr$ obtained from the definition of $\mathcal{B}^g$ (above). Thus, ξ' is simply given as $\xi' = \xi_{|f_t|}$ after all operations have been performed on the variables.

The states of $\mathcal{K}$ are thus all the possible statuses of $\mathcal{H}$.

Example 9 *Continuing from previous examples, the first few states of the resulting Kripke structure is shown in Figure 7.*

5 Features of ASEHA

ASEHA has many desirable features which include:

Communication One of the distuiguishing aspects about ASEHA compared to other existing formalisms, is that the communication mechanism is based on shared buffer model using interleaving semantics for step execution. This model is captures all the key information without going into unnecessary details.

Simplicity We believe that simplicity is the most important factor for a formalism to be accepted and used by people who do not have an in depth knowledge of formal methods. The diagrams generated by our tool are very simple and intuitive for any programmer or protocol designer to pick up and understand within minutes. Furthermore, the semantics is very simple: one step in ASEHA is just one transition.

Reusability Automata defined in ASEHA are easily reusable; one just has to link a state to an automaton or group of automata for hierarchical composition.

Hierarchy Due to the hierarchical nature of ASEHA, different levels of abstraction are provided by zoom in and zoom out mechanisms. This is based on the tried and tested ideas of David Harel [11].

Permutation Handling As the number of entities increases, the number of possible message exchange sequences can increase exponentially. To be able to represent exchanges concisely, we have variable support.

Verification ASEHA's simple semantics results into almost straightforward generation of verification code while leaving room for clever optimizations.

Last but not least, ASEHA's formalism can easily be extended by adding new predicates (for triggers) or data-structures(for LOS function) and can be easily adjusted for use in other domains.

6 Case Study: WS-AT

We have successfully modelled and verified WS-AT. Its purpose is to ensure that a transaction, done across several trusted WS providers, is atomic[8, 14].

We used our own XML-based language called ASEHAX to encode the ASEHA for the WS-AT protocol. In the model, we have assumed unexpected messages (messages arriving out of order or non-protocol messages) are discarded for simplicity. The parser, written in the Java language for ASEHAX, reads the input and automatically generates diagrams for each automaton using *dotty*[15]. The resulting diagrams have been shown earlier (Figures 2, 3, 4, 5), which have been slightly modified for clarity.

We varied the number of participants from 1 to 4. During this process, we also had to adjust the causal dependencies between the initiator and participant and the triggers for entering the final states of $WSAT$ automaton. The initiator can only request transaction commitment, by sending the $Commit$ message ($i4$ to $i5$), after the desired participants have successfully registered with the coordinator. We had

to introduce more causal dependencies to allow participants, which registered for volatile 2PC, to allow other participants to register for durable 2PC before sending the $Prepared$ message ($p5$ to $p6$). For more information, consult [8, 14].

Verification code in Promela was generated for each case and then input to SPIN. Correctness was determined by observing that one of the states co, ab and ex was entered (which represent the whether a transaction is committed, aborted, or exited prematurely, respectively) and that all these states were reachable.

7 Conclusions and Future Work

In this paper we have presented ASEHA, the formalism for modelling WS protocols, which integrates many desirable features while still having a simple syntax and semantics. We modelled the essential components of protocols and described how their correctness can be verified using Kripke structures.

This formalism has been used to model the WS-AT protocol and aid in its formal specification thereby partially solving the interoperability problem.

In the next paper of this sequel, we will demonstrate the translation of ASEHA to Promela and the verification process in details. Some work in progress include protocol compatibility checking (i.e. making sure that implementations actually correctly implement the protocols and that they can interact correctly) and graphical support for end-users.

Acknowledgments

National ICT Australia is funded through the Australian Government's initiative, in part through the Australian Research Council. We also thank David Langworthy of Microsoft, Tim Bourke and Piyush Maheshwari of School of Computer Science and Engineering, University of New South Wales, for their valuable comments and feedback.

References

[1] R. Alur, S. Kannan, and M. Yannakakis. Communicating hierarchical state machines. *Lecture Notes in Computer Science*, 1644:169–178, 1999.

[2] P. Bhaduri and S. Ramesh. Model checking of statechart models: Survey and research directions. *CoRR*, cs.SE/0407038, 2004.

[3] T. Bienmuller, W. Damm, and H. Wittke. The statemate verification environment - making it real. In *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*, pages 561–567, London, UK, 2000. Springer-Verlag.

[4] A. Ceponkus, S. Dalal, T. Fletcher, P. Furniss, A. Green, and B. Pope. Business Transaction Protocol Specification. http://www.oasis-open.org/committees/download.php, May 2004.

[5] W. Chan, R. Anderson, P. Beame, S. Burns, F. Modugno, D.Notkin, and J. Reese. Model checking large software specifications, 1998.

[6] N. Chandra, S. Sonalkar, and N. Korade. Programming environment for communicating reactive state machines. *Technical Report*, TR-02-11, Feb 2002.

[7] E. Clarke and W. Heinle. Modular translation of statecharts to smv, 2000.

[8] D. Langworthy (editor). Web Services Atomic Transaction Specification. ftp://www6.software.ibm.com/software/developer/library/WS-AtomicTransaction.pdf, Aug 2005.

[9] D. Langworthy (editor). Web Services Business Activity Framework. ftp://www6.software.ibm.com/software/developer/library/WS-Coordination.pdf, Aug 2005.

[10] J. Gray. The Transaction Concept: Virtues and Limitations. In *IEEE Proceedings of the Seventh International Conference on Very Large Data Bases*, Sep 1981.

[11] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, Jun 1987.

[12] G. J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[13] J. E. Johnson. Formal Specification of a Web Services Protocol. *First Int'l Workshop of Web Services and Formal Methods (WS-FM)*, 2004.

[14] J. E. Johnson, D. Langworthy, L. Lamport, and F. Vogt. Specification of the Web Services Atomic Transaction Protocol. http://research.microsoft.com/users/lamport/tla/ws-at.html, 2004.

[15] E. Koutsofios and S. C. North. Editing graphs with dotty. http://www.graphviz.org/Documentation/dottyguide.pdf.

[16] D. Latella, I. Majzik, and M. Massink. Automatic verification of a behavioural subset of uml statechart diagrams using the spin model-checker. *The International Journal of Formal Methods*, 11(6):637–664, 1999.

[17] J. Lilius and I. P. Paltor. vUML: A Tool for Verifying UML Models. In *ASE '99: Proceedings of the 14th IEEE International Conference on Automated Software Engineering*, page 255, Washington, DC, USA, 1999. IEEE Computer Society.

[18] K. L. McMillan. *Symbolic model checking: an approach to the state explosion problem*. PhD thesis, Pittsburgh, PA, USA, 1992.

[19] E. Mikk, Y. Lakhnech, and M. Siegel. Hierarchical Automata as Model for Statecharts. In *Proceedings of Asian Computing Science Conference (ASIAN '97)*, volume 1345, pages 181–196. Lecture Notes in Computer Science, Springer Verlag, Dec 1997.

[20] E. Mikk, Y. Lakhnech, M. Siegel, and G. J. Holzmann. Implementing statecharts in promela/spin, 1997.

[21] P. Ramsokul and A. Sowmya. ASEHA: A Web Services Protocol Modelling Formalism. *Technical Report*, UNSW–CSE–TR–0521, Nov 2005.

[22] M. von der Beek. A comparison of statechart variants. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863, pages 128–148. Lecture Notes in Computer Science, Springer Verlag, 1994.

[23] R. Walters. Automating checking of models built using a graphically based formal modelling language. In *Proceedings of The Twenty-Seventh Annual International Computer Software and Applications Conference*, pages 98–104, Dallas, Texas, 2003.

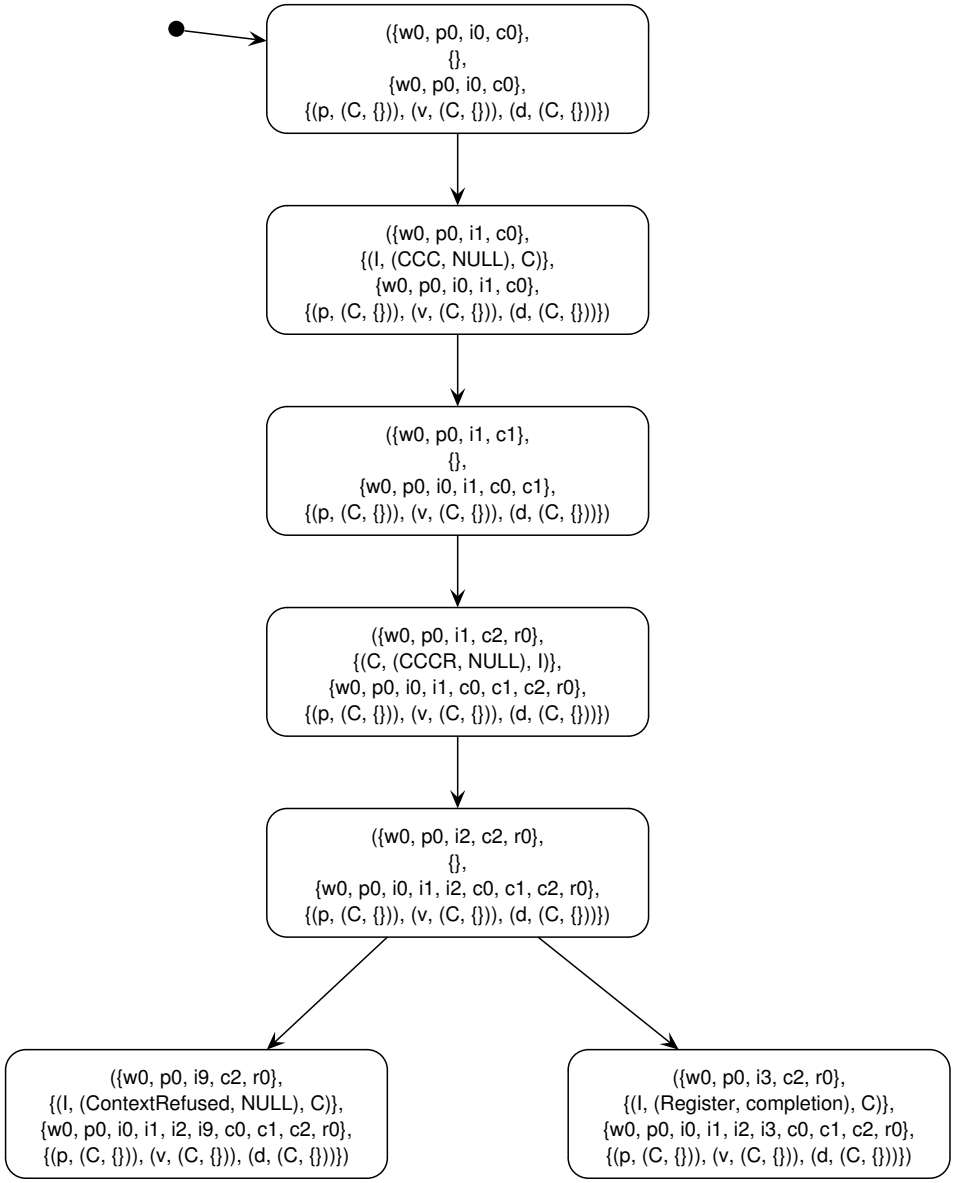

Figure 7: First few states of the resulting Kripke structure