

A Taxonomy of Web Services Using CSP

Lee Momtahan, Andrew Martin and A. W. Roscoe¹

Oxford University Computing Laboratory, Oxford OX1 3QD, England

Abstract

Terms such as *conversational* and *stateless* are widely used in the taxonomy of web services. We give formal definitions of these terms using the CSP process algebra. Within this framework we also define the notion of *Service-Oriented Architecture*. These definitions are then used to prove important scalability properties of stateless services. The use of formalism should allow recent debates, concerning how and whether web services provide standardized access to state, to progress more rigorously.

Keywords: distributed systems, formal methods, web services, communicating sequential processes (CSP)

1 Introduction

There is currently a debate within the web services and Grid communities over whether, and how, web services should allow for access to state. One view is “web services... have no notion of state” [6] while others have argued that the critical role that state plays in distributed systems requires that it be addressed within the web services architecture [2].

The debate is hindered by a lack of formality and clarity in its discourse. This paper contributes by defining some of the key terms used in the debate, within the Communicating Sequential Processes (CSP) [3] formalism. The motivation for this work is not primarily in the formal verification of web services (although this may be a possible consequence) but rather to gain a deeper understanding of the concepts. We hope more principled comparisons

¹ Email: {leemo, apm, awr}@comlab.ox.ac.uk

between different proposals to standardize access to state can be made in the light of these definitions.

1.1 Overview

In section 2 we quote the existing natural language definitions used in the taxonomy of web services. The following section gives a series of definitions in CSP culminating in a formal version of the same taxonomy. Section 4 discusses some of the implications of this formalized taxonomy. Section 5 concludes with a summary of the main findings. The first appendix presents our definitions in an alternative form that can be used with the CSP model checker, FDR. The second appendix presents proofs of theorems used in the paper.

2 A taxonomy of state and services

Our intention in this section is to persuade the reader that the use of natural language to define the terms *conversational* and *stateless* is inappropriate, and that the clarity of such definitions in the current literature is poor.

In [1] the following taxonomy of web services is given:

- A *stateless service* implements message exchanges with no access or use of information not contained in the input message. A simple example is a service that compresses and decompresses documents, where the documents are provided in the message exchanges with the service.
- A *conversational service* implements a series of operations such that the result of one operation depends on a prior operation and/or prepares for a subsequent operation. The service uses each message in a logical stream of messages to determine the processing behaviour of the service. The behaviour of a given operation is based on processing preceding messages in the logical sequence. Many interactive Web sites implement this pattern through use of HTTP sessions and cookies.
- A *service that acts upon stateful resources* provides access to, or manipulates a set of logical stateful resources (documents) based on messages it sends and receives.

[1] continues:

When we talk in the third model about a *service that acts upon stateful resources* we mean a service whose *implementation* executes against dynamic state, i.e., state for which the service is responsible between message exchanges with its requesters. A service that acts upon stateful resources may

be described stateless if it delegates responsibility for the management of the state to another component such as a database or file system.

Substantial modifications of the wording used in the definitions occurred between v1.0 and v1.1 of [1], perhaps indicating the difficulty of defining these concepts in natural language.

A related definition is that of a service in the context of Service-Oriented Architecture (SOA). [4] gives the following:

A service is a well-defined set of actions, it is self-contained, stateless, and does not depend on the state of other services. . .

Here, stateless means that each time a consumer interacts with a Web Service, an action is performed. After the results of the service invocation have been returned, the action is finished. There is no assumption that subsequent invocations are associated with prior ones.

3 Web services in CSP

In this section a series of definitions is given which builds our model of web services and their taxonomy. The reader unfamiliar with the CSP notation is referred to Appendix A for a brief introduction.

Definition 3.1

$$Stateless'(P) \Leftrightarrow \forall s : traces(P) \bullet P = P/s$$

Our first attempt to define the notion of statelessness of a process P says that after communicating any events, the process returns to its initial state.

This definition is satisfactory only so long as a typical request-response² operation is modelled as a single event. But we want to consider the interaction of the server with back-end stateful resources, which usually occurs between the request and response messages and therefore have to model the request and response as separate events.

3.1 Threads

We first define a CSP process W , which is willing to accept any events on the channels *response* and *request*, provided the events alternate between request and response and begin with request.

² Others type of operations e.g. solicit-response are not modelled in this paper, however we expect our work could be easily extended to deal with these other types.

Definition 3.2

$$W = request?x \rightarrow W'$$

$$W' = response?y \rightarrow W$$

We also define for convenience αW as all request and response events.

Definition 3.3

$$\alpha W = \{request, response\}$$

We now define a *thread*.

Definition 3.4

$$Thread(P) \Leftrightarrow P = P \parallel [\alpha W] \parallel W \wedge$$

$$initials(P) \subseteq \{request\}$$

Thus if a process P is a thread, it alternates between request and response events, and can do nothing until its first request is received. Other than that, events may occur at any time.

3.2 Stateless Threads

Our definition of a *stateless thread* is as follows, where $last(s)$ returns the last event in the trace s .

Definition 3.5

$$Stateless(P) \Leftrightarrow Thread(P) \wedge$$

$$\forall s : traces(P) \mid last(s) \in \{response\} \bullet P/s = P$$

Thus a thread P is stateless if it always returns to its initial state after communicating a response event.

3.3 Scalable Threads

We define a further property threads may exhibit we refer to as *scalability*.

Definition 3.6

$$Scalable(P) \Leftrightarrow Thread(P) \wedge$$

$$P = (P \parallel P) \parallel [\alpha W] \parallel W$$

This says in the presence of W , which has the effect of limiting the number of outstanding requests to one, two copies of P behave like a single one.

In corollary C.8 we show that it follows from this definition that $P = (P \parallel P \parallel P) \llbracket \alpha W \rrbracket W$. Indeed when an arbitrary number of P 's are interleaved in the presence of W the resulting combination is identical to P .

We also show in Theorem C.6 that $Stateless(P) \Rightarrow Scalable(P)$. Interestingly the converse does not hold. Consider:

$$P(n) = request.up \rightarrow response.ok \rightarrow P(n+1)$$

□

$$(n > 0) \& request.down \rightarrow response.ok \rightarrow P(n-1)$$

$P(0)$ is scalable but not stateless. We note that although $P(0) = (P(0) \parallel P(0)) \llbracket \alpha W \rrbracket P(0)$, a request to the interleaved combination must be forwarded to the right thread i.e. the one which can accept the event, and this feature of the interleaving operator seems hard to realize in practice. Of course if P is stateless, requests can be forwarded to either thread, since both always accept the same events.

3.4 Examples

The following defines a process P for which $Stateless(P)$ holds:

$$P = request.name \rightarrow \text{if } Cleared(name) \text{ then}$$

$$(store.name \rightarrow response.ok \rightarrow P)$$

□

$$(full \rightarrow response.failed \rightarrow P)$$

else

$$response.failed \rightarrow P$$

This process models a thread used in a very simple airline booking system. A booking request is made with the passenger's name, then a security check is made with the function *Cleared*. If the passenger clears security, an attempt is made to add them to the passenger list (an auxiliary process), via the event *store.name*, otherwise they are rejected. The passenger can still be rejected if the flight is full.

The following defines a process $P(0)$ for which $\text{Thread}(P(0))$ holds, but $\text{Stateless}(P(0))$ does not:

$$\begin{aligned}
 P(x) = & \text{request?}n \rightarrow \text{if } \text{Cleared}(\text{name}) \text{ then} \\
 & ((x < \text{MaxPassengers}) \& \text{response.ok} \rightarrow P(x+1)) \\
 & \square \\
 & ((x = \text{MaxPassengers}) \& \text{response.failed} \rightarrow P(x)) \\
 & \text{else} \\
 & \text{response.failed} \rightarrow P(x)
 \end{aligned}$$

This process models the same booking system, but with no need for an auxiliary process to keep track of bookings. This process is not scalable: a single copy of $P(0)$ permits only MaxPassengers successful bookings whereas two copies of $P(0)$ might permit more.

3.5 Services

We now build a model of services. We will assume that the set Session contains a number of session identifiers. Compared to a single thread, every request and response to a service contains the session identifier as an additional parameter. If P is a thread, we form a *service* made of threads with P 's behaviour by defining the function Service .

Definition 3.7

$$\text{Service}(P) = \parallel s : \text{Session} \bullet P[\text{request} \leftarrow \text{req}.s, \text{response} \leftarrow \text{resp}.s]$$

Our model is deliberately simplistic in that we do not show how clients acquire sessions. We also assume there is a thread available for each session, so that clients never have to wait for available threads which is possibly unrealistic. An extra layer, modelling how sessions are assigned and limits on the number of concurrently active sessions could easily be added but is beyond the scope of this paper.

If P is scalable, then an obvious consequence is $\text{Service}(P)$ is scalable in the sense that:

$$\text{Service}(P) = (\text{Service}(P) \parallel \text{Service}(P)) \parallel \{\{\text{req}, \text{resp}\}\} \parallel \text{Service}(W)$$

where $\text{Service}(W)$ models the fact that there is never more than one request outstanding per session.

In our example airline booking system (Sec. 3.4) we can see that building a service with the stateless thread version would be advantageous if the func-

tion *Cleared* takes considerable resources; the workload can be spread across multiple servers.

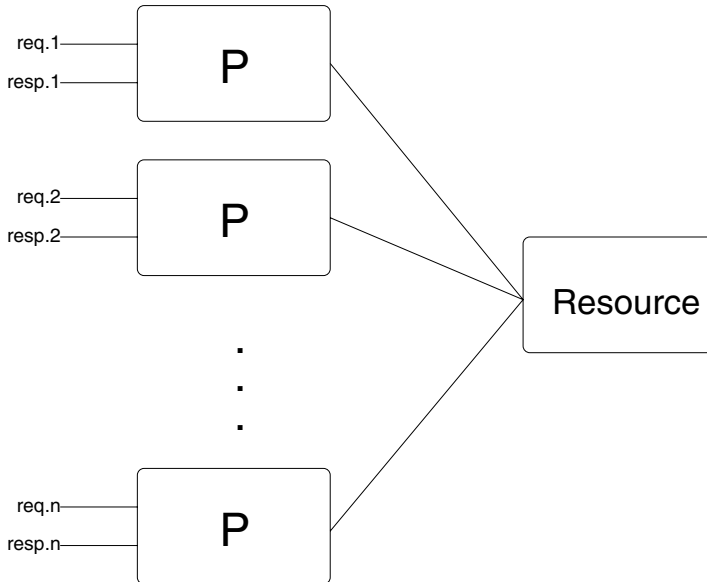
We model the *stateful resources* upon which a service may act simply as another process which is forbidden to communicate directly with clients. The following predicate determines if a process R is a resource.

Definition 3.8

$$Resource(R) \Leftrightarrow Chaos(\Sigma - \{req, resp\}) \sqsubseteq R$$

(In this definition, Σ denotes the set of all events, $Chaos(A)$ denotes a process which can always choose to communicate or reject any member of A . Thus, for R to be a resource, it must not communicate on the *req* and *resp* channels.)

The following diagram shows the communication channels and component processes in our general model of a service that acts upon stateful resources



3.6 Formalized taxonomy

Using our preceding definitions of thread, stateless thread, service, and stateful resource, we redefine the [1] taxonomy of web services formally.

- A *stateless service* is a service made of stateless threads.
- A *conversational service* is a service made of threads (stateless or not).
- A *stateless service that acts upon stateful resources* is a stateless service in parallel with one or more stateful resources.

- A *conversational service that acts upon stateful resources* is a conversational service in parallel with one or more stateful resources.

This taxonomy can be made disjoint, by defining each category to exclude the ones which precede it in the above list. e.g. a conversational service is a service made of threads which are not stateless.

We can also formalize the definition of service in the context of Service-Oriented Architecture given in [4], [7]. Such services correspond to stateless services and stateless services that act upon stateful resources (the first and third types in the above taxonomy).

4 Discussion

Key to our distinction between stateless services that act upon stateful resources and conversational services that act upon stateful resources, is that the threads from which they are composed are not aware of which session they serve. That is although, each client makes requests of the form *req.s.x*, where *s* identifies the session, only the *x* component of the event is passed to the receiving thread. Suppose alternatively that *s* is also passed to the thread, so that the session can be identified. The threads from which a conversational service is composed, can be modified to load their state (indexed by each session *s*) from back-end stateful resources immediately after receiving a request, and to save their state to stateful resources immediately before each response. Thus (under our extra assumption), for every conversational service that acts upon stateful resources there exists a stateless service that acts upon stateful resources with the same behaviour.

In fact this is a well-known technique for achieving what is in effect a conversational service in the context of Service-Oriented Architecture, known as *contextualization* [4]: every message passed between the service and its client contains a unique context identifier.

This being the case we may ask if the distinction we draw matters? As a ‘black box’ there is little between a stateless service that acts upon stateful resources and a conversational one. However, a stateless service has important ‘white box’ properties that the conversational service does not: the ability to replicate its stateless front end to achieve scalability. The use of stateless services may also improve the modularity of a design.

We note also, that statelessness is not preserved by refinement e.g.

$$P = request?x \rightarrow (response.1 \rightarrow P$$

□

$$response.2 \rightarrow P)$$

is a stateless thread, whilst:

$$P' = request?x \rightarrow response.1 \rightarrow request?x \rightarrow response.2 \rightarrow P'$$

is not, even though though P' refines P . We argue that one should be concerned only with the statelessness of specifications and not implementations. For suppose $SPEC$ is a stateless thread and $IMPL$ is a refinement of it, which is not stateless. Although $(IMPL \parallel IMPL) \llbracket \alpha W \rrbracket W \neq IMPL$ in general, it still holds (by monotonicity) that $SPEC \sqsubseteq (IMPL \parallel IMPL) \llbracket \alpha W \rrbracket W$. $IMPL$ still has the property that it can be replicated as required for scalability whilst satisfying its specification.

5 Conclusion

We have given formal definitions of stateless and conversational services and Service-Oriented Architecture, and explained their relationship. If contextualization is permitted, the distinction between stateless and conversational services, that act upon stateful resources cannot be determined by external behaviour; rather it is an internal property that can be used to achieve scalability. Finally we have explained how, in the presence of non-determinism, it is possible to have a stateful implementation of a stateless service specification, and thus it is only whether a service's specification is stateless that matters. We hope the ongoing debate into services and state is informed by these observations.

References

- [1] I. Foster, J. Frey, S. Graham, and S. Tuecke. Modelling stateful resources with web services. Computer Associates International, Fujitsu Limited, IBM, The Hewlett-Packard Development Company, The University of Chicago, 2003.
- [2] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration, 2002.
- [3] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
- [4] S. Parastatidis, J. Weber, P. Watson, and T. Rischbeck. A grid application framework based on web services specifications and practices. North East Regional e-Science Centre, School of Computing Science, University of Newcastle, UK, 2003.
- [5] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1998.

- [6] Werner Vogels. Web services are not distributed objects. *IEEE Internet Computing*, 7(6):59–66, 2003.
- [7] Web services architecture. W3C. <http://www.w3.org/TR/2003/WD-ws-arch-20030808/>.

A Communicating Sequential Processes

A.1 Events and processes

The language of CSP is a notation for describing the behaviour of concurrently-evolving objects, or *processes*, in terms of their interaction with their environment. This interaction is modelled in terms of *events*: abstract, instantaneous, synchronisations that may be shared between several processes.

We use compound events to represent communication. The event name $c.x$ may represent the communication of a value x on a *channel* named c . At the event level, no distinction is made between *input* and *output*. The willingness to engage in a variety of similar events—the readiness to accept input—is modelled at the process level. The same is true of output, which corresponds to an insistence upon a particular event from a range of possibilities. The notation $\{c, d\}$ denotes the set of all possible events from channels c and d .

A *process* describes the pattern of availability of certain events. The simplest process of all is *Stop*, which makes no events available at any time. The prefix process $a \rightarrow P$ is ready to engage in event a ; should this event occur, the subsequent behaviour is that of P , which must itself be a process.

An external choice of processes $P \sqcap Q$ is resolved through interaction with the environment: the first event to occur will determine the subsequent behaviour. If this event was possible for only one of the two alternatives, then the choice will go on to behave as that process. If it was possible for both, then the choice becomes internal.

An internal choice of processes $P \sqcap Q$ will behave either as P or as Q . The result of this choice can be neither influenced nor predicted by the environment. Both forms of choice exist in an indexed form: for example, $\bigsqcap i : I \bullet P(i)$ is an internal choice between processes $P(i)$, where i ranges over the (finite) indexing set I , while $\sqcap i : I \bullet P(i)$ is the corresponding external choice.

We may denote input in one of two ways. The process $c?x \rightarrow P$ is willing initially to accept any value (of the appropriate type) on channel c . On the other hand, if we wish to restrict the set of possible input values to a subset of the type associated with the channel c , then we may write $\square x : X \bullet c.x \rightarrow P$. Furthermore, the processes $c?x?y \rightarrow P$ and $\square x : X \bullet \square y : Y \bullet c.x.y \rightarrow P$ are equivalent (assuming that X and Y the appropriate types).

In the parallel combination $P \parallel Q$, the component processes cooperate upon shared events. To identify which events are shared and which may be performed independently, we associate each component with an interface, or *alphabet*. If αP is the alphabet of P , then P will share in every event from this set; conversely, no event from αP can take place without P 's participation. Indexed parallel composition, written $\bigsqcap i : I \bullet P(i)$ insists that the all processes of the form $P(i)$, where i is drawn from the indexing set I are performed in parallel, synchronising when necessary.

We will sometimes wish to take the other extreme and insist that no events are shared, whatever the specified alphabets. We write $P \parallel\!\!\!\parallel Q$ to represent the *interleaved* parallel combination of P and Q .

Finally, we may remove events from the interface of a process using the hiding operator. The process $P \setminus A$ behaves exactly as P except that events from the set A no longer require

the cooperation of the environment; they are no longer visible to other processes.

A.2 Refinement

The standard notion of refinement for CSP processes, which is defined in [5], is based upon the failures-divergences model of CSP. In this model, each process is associated with a set of behaviours: tuples of sequences and sets that record the occurrence and availability of events.

The *traces* of a process P , given by $traces(P)$, are finite sequences of events in which that process may participate *in that order*; the *failures* of P , given by $failures(P)$, are pairs of the form (tr, X) , such that tr is a trace of P and X is a set of events which may be refused by P after tr has occurred; and the *divergences* of P , given by $divergences(P)$, are those traces of P after which an infinite sequence of internal events may be performed.

(We also use the *after* operator P/s to be the behaviour of P after it has produced the trace s . $initials(P)$ denotes the initial events of all traces of P)

A process, Q , is said to be a *failures-divergences refinement* of another process, P , written $P \sqsubseteq Q$, if $failures(Q) \subseteq failures(P)$ and $divergences(Q) \subseteq divergences(P)$.

We may use this theory of refinement to investigate whether a potential design meets its specification. The refinement checker FDR does exactly this. A pleasing feature of FDR is that if a potential design fails to meet its specification, a counter-example is returned to indicate why this is so.

B Checking thread properties with FDR

We have used in our definition of threads and stateless threads properties which cannot be readily checked with the FDR model checker for CSP. We here give alternative definitions which can.

To check $initials(P) \subseteq \{\{request\}\}$ we can ask FDR:

$$request?x \rightarrow RUN_{\Sigma} \sqsubseteq_T P$$

(RUN_{Σ} denotes a process which is always ready communicate any possible event.)

We now consider how to check $\forall s : traces(P) \mid last(s) \in \{\{response\}\} \bullet P/s = P$. Without loss of generality we assume *tock* is an event in the alphabet which is not used by the process P . (We can always enlarge the alphabet with a spare event if required.) We define:

$$S = ?x : (\Sigma - \{\{response\}\}) \rightarrow S$$

□

$$?x : \{\{response\}\} \rightarrow tock \rightarrow STOP$$

Thus $Q = P \parallel [\Sigma - \{tock\}] S$ behaves like P up to and including the first response event, and then becomes $tock \rightarrow STOP$. So the process $(Q \parallel [\{tock\}] tock \rightarrow P) \setminus \{tock\}$ similarly behaves like P up to and including the first response event, and then behaves like P . Thus we check whether P is stateless by checking P is a thread, and then asking FDR if:

$$(Q \parallel [\{tock\}] tock \rightarrow P) \setminus \{tock\} = P$$

C Proofs of theorems

Throughout this section, let P denote a thread.

Definition C.1

$$\text{diverges}(X) \Leftrightarrow X = X \sqcap \text{div}$$

N.B. This predicate holds exactly when process X can diverge immediately.

Lemma C.2

$$\forall s : \text{traces}(P) \bullet \#s \upharpoonright \{\text{request}\} \geq \#s \upharpoonright \{\text{response}\}$$

Proof. This is an easy consequence of $\text{Thread}(P)$. □

Definition C.3

Write Q_s for P/s if $s : \text{traces}(P) \wedge \#s \upharpoonright \{\text{request}\} = \#s \upharpoonright \{\text{response}\}$

Write R_s for P/s if $s : \text{traces}(P) \wedge \#s \upharpoonright \{\text{request}\} > \#s \upharpoonright \{\text{response}\}$

Lemma C.4

$$\text{initials } Q_s \cap \{\text{response}\} = \emptyset$$

$$\text{initials } R_s \cap \{\text{request}\} = \emptyset$$

Proof. This is an easy consequence of $\text{Thread}(P)$. □

Convention C.5 *The interface of the parallel operator (\parallel) is αW unless stated otherwise. The interleaving operator ($\parallel\parallel$) binds more tightly than the parallel operator. e.g. $P \parallel\parallel P \parallel W$ stands for $(P \parallel\parallel P) \parallel \alpha W \parallel W$*

Theorem C.6

$$\text{Stateless}(P) \Rightarrow \text{Scalable}(P)$$

Proof. Suppose $\text{Stateless}(P)$. We show that under our assumptions, for every trace t of $P \parallel\parallel P \parallel W$ and every trace t of P :

$$(P \parallel\parallel P \parallel W)/t = (P/t) \parallel\parallel P \parallel W = P/t$$

$$\text{if } t = \langle \rangle \vee \text{last}(t) \in \{\text{response}\}$$

$$(P \parallel\parallel P \parallel W)/t = (P/t) \parallel\parallel P \parallel W' = P/t$$

$$\text{otherwise}$$

and hence $(P \parallel\parallel P) \parallel W = P$, i.e. $\text{Scalable}(P)$.

To prove the above equality, we show that the initials, refusals and initial divergences of the terms are equal, and that after each initial event the result states are also equal if we assume the above statements. This can be formally justified by reference to the theory of constructive recursions and unique fixeds points (UFPs) in CSP [5].

We note that due to $\text{Stateless}(P)$ if $x \in \{\text{response}\}$ then $Q_s \frown_{\langle x \rangle} = Q_{\langle \rangle}$.

Case (i) $P1 = Q_{\langle \rangle} \wedge P2 = Q_{\langle \rangle} \parallel Q_{\langle \rangle} \parallel W$.

$$\begin{aligned} \text{initials}(P1) &= \text{initials}(P2) \\ \text{refusals}(P1) &= \text{refusals}(P2) \\ \text{diverges}(P1) &\Leftrightarrow \text{diverges}(P2) \end{aligned}$$

$$\forall x : \text{initials}(P1) \bullet P1/\langle x \rangle = R_{\langle x \rangle}$$

$$\begin{aligned} \forall x : \text{initials}(P2) \bullet P2/\langle x \rangle \\ &= (R_{\langle x \rangle} \parallel Q_{\langle \rangle} \parallel W') \sqcap (Q_{\langle \rangle} \parallel R_{\langle x \rangle} \parallel W') \\ &= R_{\langle x \rangle} \parallel Q_{\langle \rangle} \parallel W' \end{aligned}$$

Case (ii) $P1 = R_s \wedge P2 = R_s \parallel Q_{\langle \rangle} \parallel W'$.

$$\begin{aligned} \text{initials}(P1) &= \text{initials}(P2) \\ \text{refusals}(P1) &= \text{refusals}(P2) \\ \text{diverges}(P1) &\Leftrightarrow \text{diverges}(P2) \end{aligned}$$

$$\begin{aligned} \forall x : \text{initials}(P1) \bullet P1/\langle x \rangle = \\ \text{if } x \in \{\text{response}\} \text{ then } Q_s \frown_{\langle x \rangle} \text{ else } R_s \frown_{\langle x \rangle} = \\ \text{if } x \in \{\text{response}\} \text{ then } Q_{\langle \rangle} \text{ else } R_s \frown_{\langle x \rangle} \end{aligned}$$

$$\begin{aligned} \forall x : \text{initials}(P2) \bullet P2/\langle x \rangle = \\ \text{if } x \in \{\text{response}\} \text{ then } Q_s \frown_{\langle x \rangle} \parallel Q_{\langle \rangle} \parallel W \\ \text{else } R_s \frown_{\langle x \rangle} \parallel Q_{\langle \rangle} \parallel W' = \\ \text{if } x \in \{\text{response}\} \text{ then } Q_{\langle \rangle} \parallel Q_{\langle \rangle} \parallel W \\ \text{else } R_s \frown_{\langle x \rangle} \parallel Q_{\langle \rangle} \parallel W' \end{aligned}$$

□

Theorem C.7

$$P \parallel P \parallel P \parallel W = P \parallel (P \parallel P \parallel W) \parallel W$$

Proof. The proof of this theorem is based on the a similar technique to the previous one. We cover all reachable states by showing the following equalities:

$$\begin{aligned} Q_s \parallel Q_t \parallel Q_v \parallel W &= Q_s \parallel (Q_t \parallel Q_v \parallel W) \parallel W \\ R_s \parallel Q_t \parallel Q_v \parallel W' &= R_s \parallel (Q_t \parallel Q_v \parallel W) \parallel W' \\ Q_s \parallel R_t \parallel Q_v \parallel W' &= Q_s \parallel (R_t \parallel Q_v \parallel W') \parallel W' \\ Q_s \parallel Q_t \parallel R_v \parallel W' &= Q_s \parallel (Q_t \parallel R_v \parallel W') \parallel W' \end{aligned}$$

That is, we show the initials, refusals and initial divergences of the terms are equal and that the result states are also equal if we assume the above statements.

Case (i) $P1 = Q_s \parallel Q_t \parallel Q_v \parallel W \wedge P2 = Q_s \parallel (Q_t \parallel Q_v \parallel W) \parallel W$.

initials $P1 = \text{initials } Q_s \cup \text{initials } Q_t \cup \text{initials } Q_v = \text{initials } P2$

refusals $P1 = \text{refusals } Q_s \cap \text{refusals } Q_t \cap \text{refusals } Q_v = \text{refusals } P2$

diverges $P1 \Leftrightarrow \text{diverges } Q_s \vee \text{diverges } Q_s \vee \text{diverges } Q_v \Leftrightarrow \text{diverges } P2$

$$\begin{aligned}
 \forall x : \text{initials } P1 \bullet P1 / \langle x \rangle = \\
 \quad \sqcap i : \{s, t, v\} \mid x \in \text{initials}(P/i) \bullet \\
 \quad \text{if } x \notin \{\text{request}\} \text{ then} \\
 \quad \quad (i = s) \& (Q_s \frown_{\langle x \rangle} \parallel Q_t \parallel Q_v \parallel W) \\
 \quad \quad \square \\
 \quad \quad (i = t) \& (Q_s \parallel Q_t \frown_{\langle x \rangle} \parallel Q_v \parallel W) \\
 \quad \quad \square \\
 \quad \quad (i = v) \& (Q_s \parallel Q_t \parallel Q_v \frown_{\langle x \rangle} \parallel W) \\
 \quad \text{else} \\
 \quad \quad (i = s) \& (R_s \frown_{\langle x \rangle} \parallel Q_t \parallel Q_v \parallel W') \\
 \quad \quad \square \\
 \quad \quad (i = t) \& (Q_s \parallel R_t \frown_{\langle x \rangle} \parallel Q_v \parallel W') \\
 \quad \quad \square \\
 \quad \quad (i = v) \& (Q_s \parallel Q_t \parallel R_v \frown_{\langle x \rangle} \parallel W')
 \end{aligned}$$

$$\begin{aligned}
 \forall x : \text{initials } P2 \bullet P2 / \langle x \rangle = \\
 \quad \sqcap i : \{s, t, v\} \mid x \in \text{initials}(P/i) \bullet \\
 \quad \text{if } x \notin \{\text{request}\} \text{ then} \\
 \quad \quad (i = s) \& (Q_s \frown_{\langle x \rangle} \parallel (Q_t \parallel Q_v \parallel W) \parallel W) \\
 \quad \quad \square \\
 \quad \quad (i = t) \& (Q_s \parallel (Q_t \frown_{\langle x \rangle} \parallel Q_v \parallel W) \parallel W) \\
 \quad \quad \square \\
 \quad \quad (i = v) \& (Q_s \parallel (Q_t \parallel Q_v \frown_{\langle x \rangle} \parallel W) \parallel W) \\
 \quad \text{else} \\
 \quad \quad (i = s) \& (R_s \frown_{\langle x \rangle} \parallel (Q_t \parallel Q_v \parallel W) \parallel W') \\
 \quad \quad \square \\
 \quad \quad (i = t) \& (Q_s \parallel (R_t \frown_{\langle x \rangle} \parallel Q_v \parallel W') \parallel W') \\
 \quad \quad \square \\
 \quad \quad (i = v) \& (Q_s \parallel (Q_t \parallel R_v \frown_{\langle x \rangle} \parallel W') \parallel W')
 \end{aligned}$$

Case (ii) $P1 = R_s \parallel Q_t \parallel Q_v \parallel W' \wedge P2 = R_s \parallel (Q_t \parallel Q_v \parallel W) \parallel W'$.

initials $P1 =$

$$(\text{initials } R_s \cup \text{initials } Q_t \cup \text{initials } Q_v) - \{\text{request}\} =$$

initials $P2$

refusals $P1 =$

$$\{r : \text{refusals } R_s \mid r - \{\text{request}\} \in \text{refusals } Q_t \cap \text{refusals } Q_v\} =$$

refusals $P2$

diverges $P1 \Leftrightarrow$

$$\text{diverges } R_s \vee \text{diverges } Q_t \vee \text{diverges } Q_v \Leftrightarrow$$

diverges $P2$

$$\forall x : \text{initials } P1 \bullet P1 / \langle x \rangle =$$

$$\sqcap i : \{s, t, v\} \mid x \in \text{initials}(P/i) \bullet$$

if $x \notin \{\text{response}\}$ then

$$(i = s) \& (R_s \frown_{\langle x \rangle} \parallel Q_t \parallel Q_v \parallel W')$$

□

$$(i = t) \& (R_s \parallel Q_t \frown_{\langle x \rangle} \parallel Q_v \parallel W')$$

□

$$(i = v) \& (R_s \parallel Q_t \parallel Q_v \frown_{\langle x \rangle} \parallel W')$$

else

$$(Q_s \frown_{\langle x \rangle} \parallel Q_t \parallel Q_v \parallel W)$$

$$\forall x : \text{initials } P2 \bullet P2 / \langle x \rangle =$$

$$\sqcap i : \{s, t, v\} \mid x \in \text{initials}(P/i) \bullet$$

if $x \notin \{\text{response}\}$ then

$$(i = s) \& (R_s \frown_{\langle x \rangle} \parallel (Q_t \parallel Q_v \parallel W) \parallel W')$$

□

$$(i = t) \& (R_s \parallel (Q_t \frown_{\langle x \rangle} \parallel Q_v \parallel W) \parallel W')$$

□

$$(i = v) \& (R_s \parallel (Q_t \parallel Q_v \frown_{\langle x \rangle} \parallel W) \parallel W')$$

else

$$(Q_s \frown_{\langle x \rangle} \parallel (Q_t \parallel Q_v \parallel W) \parallel W)$$

Case (iii) $P1 = Q_s \parallel R_t \parallel Q_v \parallel W' \wedge P2 = Q_s \parallel (R_t \parallel Q_v \parallel W') \parallel W'$.

initials $P1 =$

$$(initials\ Q_s \cup initials\ R_t \cup initials\ Q_v) - \{\text{request}\} = \\ initials\ P2$$

refusals $P1 =$

$$\{r : refusals\ R_t \mid r - \{\text{request}\} \in refusals\ Q_s \cap refusals\ Q_v\} = \\ refusals\ P2$$

diverges $P1 \Leftrightarrow$

$$diverges\ R_s \vee diverges\ Q_t \vee diverges\ Q_v \Leftrightarrow \\ diverges\ P2$$

$$\forall x : initials\ P1 \bullet P1/\langle x \rangle =$$

$$\sqcap i : \{s, t, v\} \mid x \in initials(P/i) \bullet$$

if $x \notin \{\text{response}\}$ then

$$(i = s) \& (Q_s \frown_{\langle x \rangle} \parallel R_t \parallel Q_v \parallel W')$$

□

$$(i = t) \& (Q_s \parallel R_t \frown_{\langle x \rangle} \parallel Q_v \parallel W')$$

□

$$(i = v) \& (Q_s \parallel R_t \parallel Q_v \frown_{\langle x \rangle} \parallel W')$$

else

$$(Q_s \parallel Q_t \frown_{\langle x \rangle} \parallel Q_v \parallel W)$$

$$\forall x : initials\ P2 \bullet P2/\langle x \rangle =$$

$$\sqcap i : \{s, t, v\} \mid x \in initials(P/i) \bullet$$

if $x \notin \{\text{response}\}$ then

$$(i = s) \& (Q_s \frown_{\langle x \rangle} \parallel (R_t \parallel Q_v \parallel W') \parallel W')$$

□

$$(i = t) \& (Q_s \parallel (R_t \frown_{\langle x \rangle} \parallel Q_v \parallel W') \parallel W')$$

□

$$(i = v) \& (Q_s \parallel (R_t \parallel Q_v \frown_{\langle x \rangle} \parallel W') \parallel W')$$

else

$$(Q_s \parallel (Q_t \frown_{\langle x \rangle} \parallel Q_v \parallel W) \parallel W)$$

Case (iv) $P1 = Q_s \parallel Q_t \parallel R_v \parallel W' \wedge P2 = Q_s \parallel (Q_t \parallel R_v \parallel W') \parallel W'$. Similarly. □

Corollary C.8

$$Scalable(P) \Rightarrow P = P \parallel P \parallel P \parallel W$$

Proof.

$$\text{Scalable}(P) \Rightarrow P = P \parallel P \parallel W = P \parallel (P \parallel P \parallel W) \parallel W$$

$$P \parallel (P \parallel P \parallel W) \parallel W = P \parallel P \parallel P \parallel W$$

$$\text{Scalable}(P) \Rightarrow P = P \parallel P \parallel P \parallel W$$

□