On Extending the Sweep-Line for Language Equivalence Checking

Guy Edward Gallasch

Computer Systems Engineering Centre
University of South Australia
Mawson Lakes Campus, SA, 5095, AUSTRALIA
Email: guy.gallasch@unisa.edu.au

Abstract. A vital step in formal protocol verification is the comparison of the externally visible behaviour exhibited by a protocol (the *protocol language*) with that allowed by its service (the *service language*). Sometimes, it is sufficient to check for *language inclusion*, which verifies that the protocol language is contained within the service language, i.e. the protocol does not exhibit behaviour that is not in its service. This is a specific instance of model checking a safety property, where the service language becomes the safety property to check. However, it is important to know also if all behaviour allowed by the service is exhibited by the protocol. Thus, the problem becomes one of checking *language equivalence*, which cannot be formulated as safety property. The Sweep-line method, designed to alleviate the problem of state explosion, was recently extended to allow on-the-fly checking of language inclusion (safety properties). This paper presents the genesis of a further extension to allow on-the-fly verification of language equivalence.

Keywords: Language Equivalence, Sweep-line Method, State Space Explosion.

1 Introduction

This paper is motivated by the area of *protocol verification* [3]. Protocols should satisfy a set of properties that encapsulate their desired behaviour, including the service that the protocol should provide to its users. This is known as a service specification. Protocol verification involves proving that a protocol does satisfy its specification. [3] provides a protocol verification methodology that uses Coloured Petri Nets [16] for the specification of protocol behaviour. The 'verification' part of protocol verification can be broken into two main activities. One involves checking that a number of general properties hold for the protocol, such as verifying the absence of undesired terminal states, dead transitions and livelocks. The other, which is of interest to us here, involves checking that all sequences of user-observable actions of the protocol conform to those given in the service specification. Here, the user-observable actions are called *service primitives*, the set of allowable sequences of service primitives as given in the service specification is called the *service language*, $\mathcal{L}_S$, and the set of sequences of service primitives exhibited by the protocol is called the *protocol language*, $\mathcal{L}_P$.

Comparing the protocol and service languages implies a notion of *language equivalence*. As an aside, as pointed out in [3], many notions of equivalence exist (van Glabbeek [13] pointed out 155 in 1993), such as Valmari's Chaos-Free Failures Divergences (CFFD) equivalence [24], that take into account branching behaviour, which allow deadlock and livelock properties to be preserved. However, these properties can be determined from the protocol's Reachability Graph, without reference to another specification. This is normally done first, as they are fundamental to good protocol design. Once these basic properties are proved, then we consider the more specific property of correct sequencing of service primitives. Hence language equivalence is sufficient, as it is precisely the property of interest.

We would like to ensure that both languages are equivalent, i.e. that they capture the same set of sequences of user-observable events. This implies that all user-observable behaviour exhibited by the protocol is allowed by the service ($\mathcal{L}_P \subseteq \mathcal{L}_S$) and that all behaviour defined by the service is actually implemented in the protocol ($\mathcal{L}_S \subseteq \mathcal{L}_P$). There are situations in which the weaker notion of *Language Inclusion* (that $\mathcal{L}_P \subseteq \mathcal{L}_S$) is sufficient for protocol verification, e.g. the Internet Open Trading Protocol [4] implemented an acceptable subset of its service [22, 23], however the key in that situation was knowing what sequences in the service language were missing from the protocol language, and determining what constituted an acceptable subset of behaviour for the protocol.

Languages can be represented by automata. A common approach (e.g. [14,22]) for checking $\mathcal{L}_P = \mathcal{L}_S$ has involved representing the protocol and service languages as deterministic Finite State Automata (FSA) [1] and using language comparison tools such as the FSM Library of AT&T [8]. The protocol language FSA and service language FSA must both be ϵ-free and deterministic in order to use the `fsmequiv` tool of [8]. We denote these $DFSA_P$ and $DFSA_S$ respectively. $DFSA_P$ can be extracted from the Reachability Graph (RG) of the CPN model of the protocol, by firstly defining a mapping from binding elements (arc labels) to service primitives or epsilon (ϵ, the empty move), defining an initial state (usually the initial state of the CPN model), and defining halt states (usually the terminal markings, as a minimum) [3]. ϵ removal and determinisation procedures [1,15] are then used to obtain a deterministic, ϵ-free representation of the protocol language. Unless the service specification is exceedingly simple, a CPN is often constructed in order to produce $DFSA_S$ in the same way.

The downside of this approach is that it requires the complete protocol RG to be generated and in memory at one time. This can be problematic due to the well-known *state explosion* problem [24]. Fortunately, it is usually possible to perform determinisation of automata on-the-fly, e.g. [17, 18] and is certainly possible for Finite State Automata.

The Sweep-line method [6, 20] is a state space exploration technique that uses a notion of *progress* within the system being modelled. The successful application of this method usually involves defining a sensible *progress mapping*, $\psi : M \rightarrow V$, a mapping from states to *progress values*, V, that reflects the notion of progress within the system being modelled. The Sweep-line's algorithm explores states in a least-progress-first manner. The exploration algorithm uses the progress mapping to determine when it is likely that particular states (those with smaller progress values) will not be visited again in the future, and hence can usually be safely deleted from memory. However, it is possible that the notion of progress captured by the progress mapping may be violated, i.e. an arc in the reachability graph moves the system from a state with a higher progress value to one with a lower progress value. This phenomenon is known as *regress*, and the offending edge as a *regress edge.* The generalised algorithm given in [20] is able to cope with this situation, at the cost of potentially exploring parts of the state space more than once.

In [12] we extended the Sweep-line method with the ability to check safety properties on-the-fly by performing an on-the-fly parallel composition of $DFSA_S$ (known a priori) and the protocol RG, by mapping the RG to a FSA on-the-fly. (This was originally presented in [11] in the context of protocol verification, for on-the-fly language inclusion checking.) In this paper we present some preliminary steps toward extending the Sweep-line method for language equivalence checking. Primarily, the contribution of this paper is to show how to combine the Sweep-line method with on-the-fly determinisation, which may also have significance generally in the model checking world beyond our intial application of language equivalence checking. Section 2 gives some further motivation and background details of the extension of Sweep-line

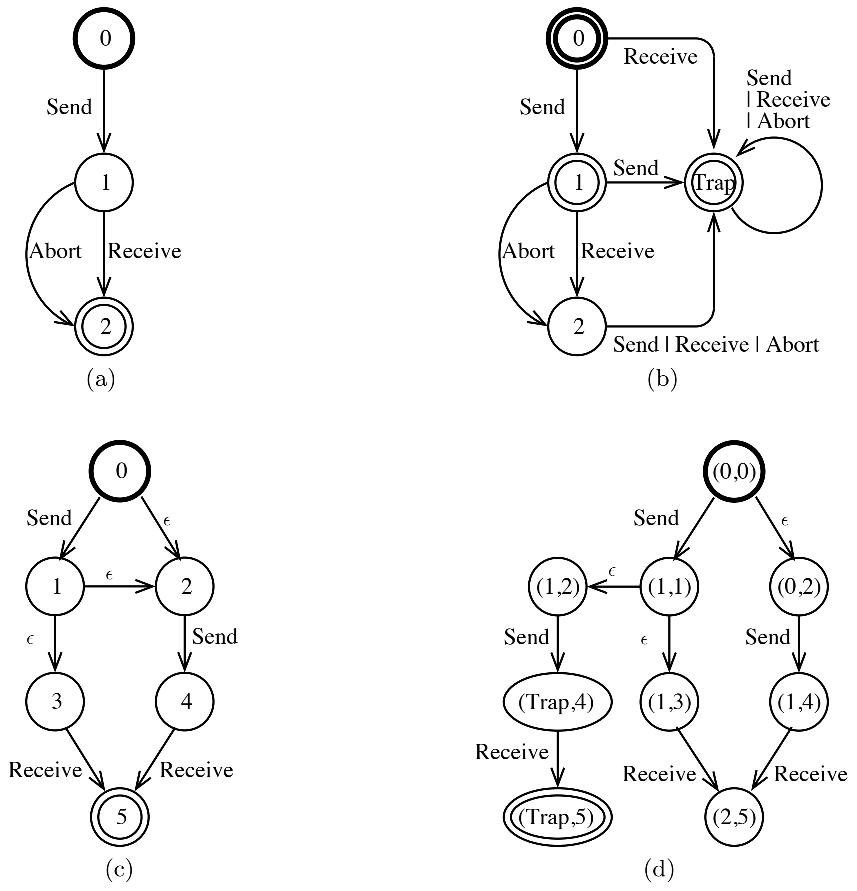

Fig. 1. Finite State Automata depicting (a) $DFSA_S$, representing the service language, $\mathcal{L}_S$; (b) $\overline{DFSA_S}$, representing the complement of $\mathcal{L}_S$; (c) FSA_P, representing the corresponding protocol language, $\mathcal{L}_P$; and (d) the parallel composition of $\overline{DFSA_S}$ and FSA_P.

to checking safety properties on-the-fly. Section 3 states our preliminary ideas for extending this to language equivalence. Finally, Section 4 outlines areas of future work.

Due to length restrictions, we assume that the reader is familiar with the workings of the Sweep-line method. Details can be found in [6, 20].

2 Language Inclusion Checking with the Sweep-line Method

Model checking of safety properties is well known, e.g. [2,7,21,25]. All of these techniques use automata to represent both the system and the safety property to check. Safety properties deal only with finite behaviours [24], hence it was reasoned in [12] that Finite State Automata (FSAs) [1] were adequate for this purpose. To verify a safety property on-the-fly with the Sweep-line method, the method adopted in [12] was to compose [5,19] the FSA representations of the system and the complement of the safety property.

For the sake of continuity with the ideas presented in Section 3, we shall use language inclusion checking as an example of on-the-fly checking of a safety property. We give such an example in Figure 1. Figure 1 (a) presents a simple service language comprising two sequences:

$\mathcal{L}_S = \{\mathsf{Send\ Receive}, \mathsf{Send\ Abort}\}$. The deterministic FSA in Figure 1 (a) is $DFSA_S$ and the set of service primitives is Send, Receive and Abort. The initial state is indicated by a bold circle and the halt state by a double circle. Figure 1 (c) presents a FSA produced by applying an appropriate mapping to the binding elements of the RG of an (erroneous) protocol, and identifying the initial state (state 0) and halt states (in this case state 5). We denote this FSA FSA_P, representing the protocol language $\mathcal{L}_P$. Note that FSA_P does not need to be deterministic for language inclusion checking.

To check that language inclusion holds, we can check that no sequences in the protocol language are contained in the complement of the service language, i.e. $\overline{\mathcal{L}_S} \cap \mathcal{L}_P = \emptyset$. Hence, we derive the the FSA representation of the complement of the service language, $\overline{DFSA_S}$, as shown in Figure 1 (b). The process of complementation relies on $DFSA_S$ being deterministic. This complement is then composed (parallel composition) with FSA_P (Figure 1 (c)) with the result shown in Figure 1 (d), to obtain $\overline{\mathcal{L}_S} \cap \mathcal{L}_P$. Essentially, the Sweep-line in [12] sweeps the FSA in Figure 1 (d), so it is possible (likely) that not all of Figure 1 (d) will be in memory at one time.

The parallel composition has an accepting state containing the Trap state of the service, indicating erroneous behaviour on the part of the protocol. Hence $\overline{\mathcal{L}_S} \cap \mathcal{L}_P \neq \emptyset$ and we have detected that our protocol is erroneous. However, we have only checked $\mathcal{L}_P \subseteq \mathcal{L}_S$. Note that the protocol does not exhibit the behaviour corresponding to the sequence Send Abort, so the protocol may be erroneous in this sense also. This cannot be detected by checking language inclusion only.

3 Toward On-the-Fly Language Equivalence Checking with the Sweep-line Method

One solution to performing the additional check of $\mathcal{L}_S \subseteq \mathcal{L}_P$ to verify language equivalence is to compose the service language FSA with the complement of the deterministic protocol language FSA. However, obtaining $\overline{DFSA_P}$ from the RG of the protocol is not straightforward, as often the RG is prohibitively large (hence our desire to apply the Sweep-line method). We can overcome this by extending the Sweep-line method with on-the-fly determinisation and complementation capabilities.

As a first step, we consider the situation in which progress increases monotonically, i.e. there are no regress edges. The ability to cope with regress is discussed in Section 3.3. The approach we propose below is based on the Sweep-line exploration algorithm having explored all states with a given (minimum) progress value and this fragment of RG having been mapped to a FSA (easily done) before the process of determinisation, complementation and parallel composition are undertaken for that fragment of FSA. Mapping the fragment of RG to a FSA, complementation and parallel composition are not discussed below, as they are straightforward processes that can be applied on a state-by-state and arc-by-arc basis once we know that the outgoing arcs of the states involved are deterministic. We hence limit our discussion to the most critical step - on-the-fly determinisation. [1] advocates a two-step method for determinisation: removing empty cycles and empty moves, then eliminating the remaining nondeterminism. This technique is much more suited to on-the-fly determinisation than the subset (power set) construction technique of [15], which requires power sets of states to be generated - a procedure that is counterproductive to state space reduction. However, we discuss the possibility of applying subset construction with *lazy subset evaluation* in Section 4. We illustrate our proposal with a series of RG snapshots.

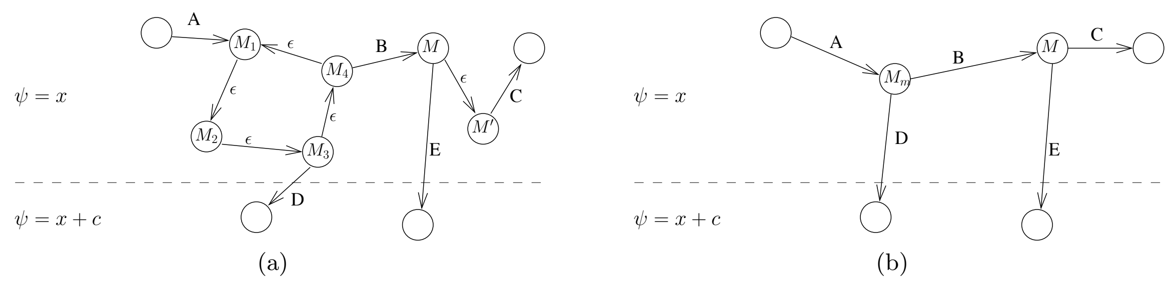

Fig. 2. (a) Epsilon cycles and empty moves within one progress level; and (b) after epsilon removal.

3.1 Removing Empty Moves

Figure 2 (a) presents two scenarios that are straightforwardly dealt with by the algorithms in [1]. On the left is an empty cycle (states M_1, M_2, M_3 and M_4) and on the right is an empty move from M to M' that is not part of a cycle. As is shown in Figure 2 (b), the states forming the cycle are merged into M_m and the other empty move is eliminated by adding appropriately labelled arcs from M to the successors of M' and deleting M'. Although not explicitly pictured in Figure 2, the algorithms in [1] cater for chains of ϵ-moves of arbitrary length, and for situations in which a merged state has predecessors that were not directly involved in the ϵ-removal process. The remaining examples in this paper can be extended in a similar way.

Figure 3 (a) illustrates two situations in which empty moves involve states in two 'progress levels'. On the left is a situation in which the removal of the empty move originating at state M_1 results in a new arc being added that spans two progress values. Given that the successors of M_1' would have already been calculated, M_1'' will be in memory, and hence removal of this ϵ-move causes no problems. On the right is a less benign situation: the empty move from state M_2 corresponds to an increase in progress. In the normal course of events, the Sweep-line method should delete all states with progress value x (once all such states have been explored). However, this is not possible, as at this stage we cannot yet eliminate this ϵ move as we do not know the successors of M_2'. We would like to retain M_2 in memory until the successors of M_2' have been calculated. To do this, we introduce the notion of *transient* states: states that should not be deleted immediately. Unlike the notion of a *persistent* state introduced in [20] to cope with regress edges, a transient state need only be retained in memory until the ϵ-move originating from it can be eliminated - it can be safely deleted at some point in the future. The result of both of these situations is shown in Figure 3 (b), where the transient state is indicated by a shaded circle. Transient states can be used for multiple successive ϵ-moves in a chain spanning any number of progress levels.

3.2 Determinisation of Non-Empty Moves

When all ϵ-moves have been eliminated, with the exception of situations such as that depicted at the right of Figure 3 (a), determinisation of the remaining non-ϵ moves can proceed. Following the procedures of [1], this is done by the merging of states. Figure 4 (a) shows two situations that are handled straightforwardly by these procedures. On the left is the situation where two states, M_1' and M_1'', are to be merged within the same progress level as the originator, M_1. On the right is the situation where two states, M_2' and M_2'', are to be

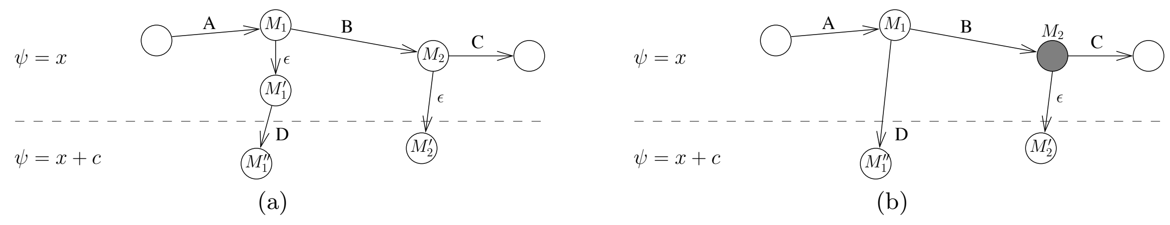

Fig. 3. (a) Epsilon moves involving two progress levels; and (b) removal of the epsilon move from M_1 to M_1', and marking M_2 as *transient*.

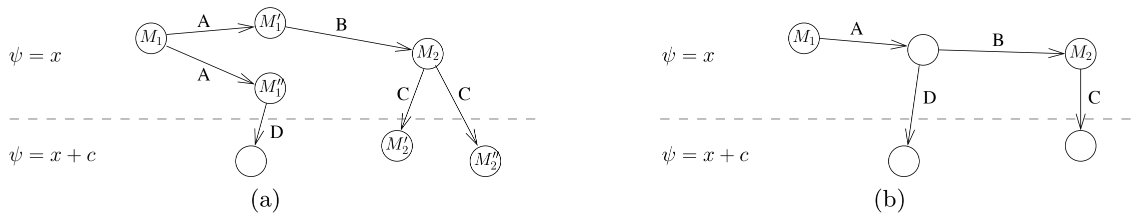

Fig. 4. (a) Nondeterminism involving non-ϵ moves; and (b) merging of states to remove this nondeterminism.

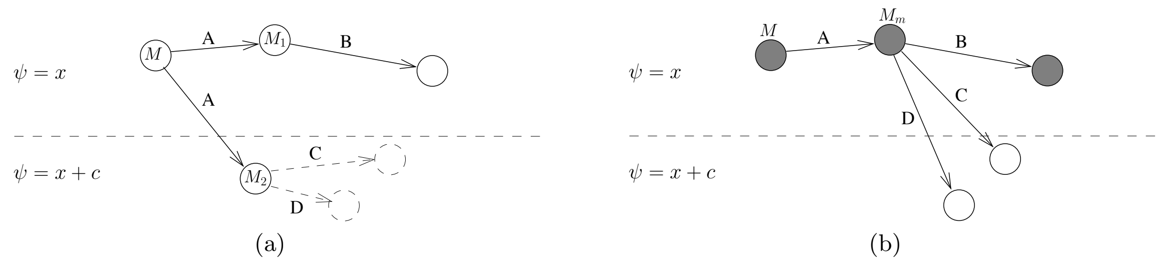

Fig. 5. (a) Nondeterminism of non-ϵ actions across two progress levels; and (b) marking states as *transient* to allow this nondeterminism to be removed once successors of M_2 are known.

merged in the same progress level, but which have a higher progress value than the originator, M_2. The result is shown in Figure 4 (b).

Figure 5 depicts a situation where merging of states occurs across two progress levels. In Figure 5 (a) $\psi(M_1) = x$ but $\psi(M_2) = x + c$. In this situation, it is not possible to merge M_1 and M_2 yet, as M_2 has yet to be explored (the dashed arcs and nodes in Figure 5 (a)). Our proposed solution is to mark M, M_1 and all successors of M_1 as transient, until the successors of M_2 have been calculated. At this point, the merging can occur, and we propose to give the merged state the progress value $min(\psi(M_1), \psi(M_2)) = x$, as illustrated in Figure 5 (b). The reason for this is simple: we wish to avoid regress edges in order to minimise the potential for re-exploration of parts of the state space. This can be generalised to the merging of any number of states in a straightforward manner.

3.3 Coping with Regress

Regress edges complicate matters in two ways. The first is that empty cycles may now exist that span multiple progress levels, as shown at the left of Figure 6 (a), and that ϵ-moves may

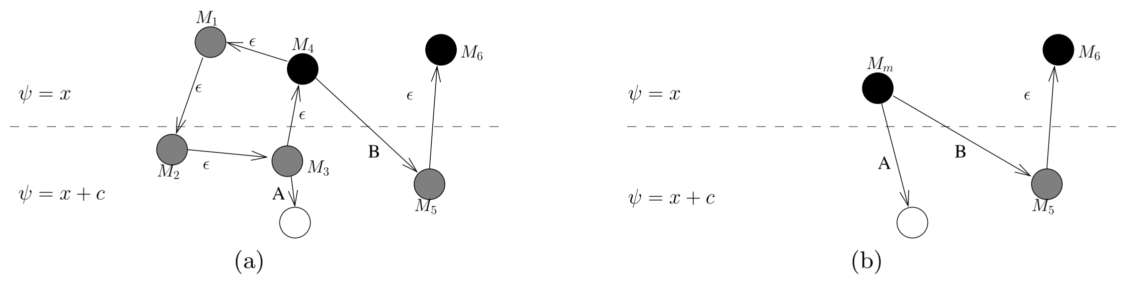

Fig. 6. (a) Empty cycles spanning two progress levels, and an ϵ-move outside a cycle corresponding to a regress edge; and (b) merging of the states in the empty cycle.

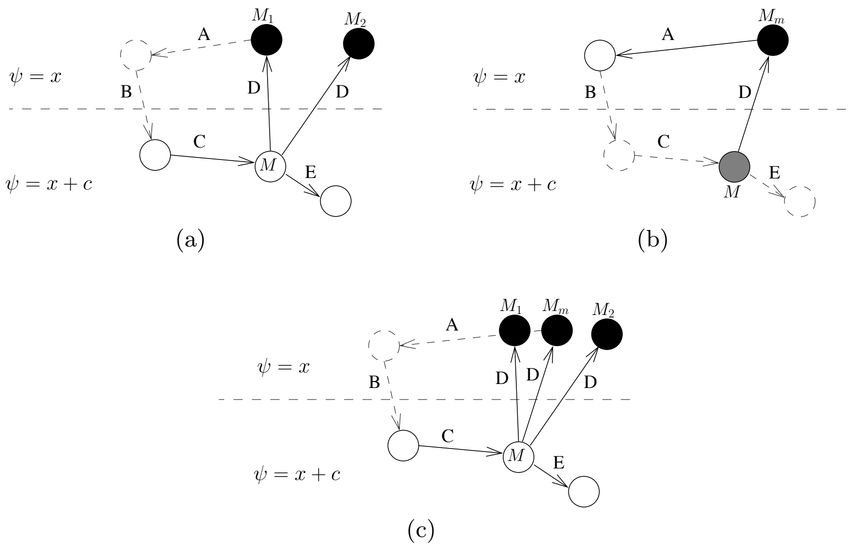

Fig. 7. (a) Two persistent states, M_1 and M_2, to be merged; (b) merging of M_1 and M_2 to M_m on a subsequent sweep; and (c) rediscovery of M_1 and M_2.

correspond to regress edges, as shown at the right of Figure 6 (a). The Sweep-line has marked M_4 and M_6 as persistent (solid circles). Our proposed method for ϵ-removal from Section 3.1 has marked M_1, M_2, M_3 and M_5 as transient. The second complication is that determinisation may require a persistent state to be deleted. For example, the existence of an empty cycle across multiple progress levels implies that at least one state in that cycle has been marked as persistent (in this case M_4). Determinisation will also require M_6 to be deleted (following the algorithm in [1]) as it is the destination of an ϵ-move and has no successors. We propose to treat these two complications by merging the empty cycle as before, but to assign the merged state the minimum progress value over all the states being merged (holding to the heuristic of minimising the number of regress edges) and to mark the merged state as persistent, as shown in Figure 6 (b), in the same way that merged states become halt states if any of their component states are halt states [1]. To eliminate the ϵ-move at the right of Figure 6 (a) it is a matter of exploring the successors of M_6. The only additional step in our proposal is to mark the successors of M_6 as persistent because M_6 was persistent, in the same way that the successors of M_6 would become halt states if M_6 were a halt state [1]. The same procedure should be applied when merging/deleting persistent states as a result of determinisation of non-empty moves.

As a consequence of our method of coping with regress and the deletion of persistent states, we must consider what will happen if a merged/deleted persistent state is re-explored. Consider Figure 7 (a), in which two regress edges from M induce two persistent states, M_1 and M_2. They cannot be merged until the successors of both are known. We propose that M be marked as transient until such time as M_1 and M_2 can be merged into M_m, as shown in Figure 7 (b). Then, re-exploration from the successors of M_m once again discovers the two persistent states from Figure 7 (a), as shown in Figure 7 (c). Because M_m was marked as persistent, we need to explore M_1 and M_2 again only as far as is necessary to merge them and rediscover M_m. An implementation that would produce this effect would be to mark the immediate successors (once ϵ-removal is complete) of the merged states as persistent, hence re-exploration is limited only to the successor states (exploration is truncated at persistent states visited for the second time).

4 Conclusions and Future Work

In this paper we have presented one possible genesis for an extension of the Sweep-line method to handle language equivalence checking, an extension of importance to protocol verification. However, substantial effort is required to bring this proposed extension to fruition. This work must be formalised to produce an extended Sweep-line algorithm, implemented and evaluated using a case study. There are also many avenues that remain to be investigated.

The discussion of determinisation presented in this paper is based on all states with the minimum progress value having been explored before determinisation/complementation/parallel composition begins. However, it should be possible to interleave the exploration, determinisation, complementation and parallel composition procedures. This may provide a significant saving in the time taken to detect violations of the language equivalence property, particularly if there are many states with identical progress values.

One aspect of this proposal that has not yet been discussed is the choice of progress mapping. Although progress mappings may be arbitrary, previous experience (e.g. [9,10]) has indicated that the choice of progress mapping plays a pivotal role in the performance of the Sweep-line method. The author believes that there is no inherent problem with a progress mapping that results in the merging of states with different progress values, as the progress mapping may depend on internal protocol actions and changes of state within protocol entities that are not part of the protocol's user-observable behaviour.

Although we have used the two-step process of [1] for determinisation, it may be possible to use the method of subset construction with lazy subset evaluation [15] for both epsilon removal and determinisation in a single step. For each new state encountered, its transitive closure comprising only empty moves is calculated. This subset of states becomes the state of the deterministic FSA. However, such a subset may contain states of many different progress values, hence the order of state exploration may violate the least-progress-first policy of the Sweep-line method. This may be able to be overcome by delaying completion of the construction of the subset using transient states in a similar way to the process described in this paper. However, this will reduce the effectiveness of the Sweep-line method as a state space reduction technique, as more transient states will remain in memory for longer.

In terms of simply detecting a violation of language equivalence, it may be possible to take a more direct approach under some conditions. If the parallel composition of $DFSA_S$

and $DFSA_P$ (not its complement) is calculated on-the-fly, it may be possible to examine each composed state to see what actions are allowed by the corresponding states in $DFSA_S$ and $DFSA_P$. If they don't match, we have a violation of language equivalence. This approach relies on knowing that livelocks do not exist in either the protocol or service specifications, but would not require the complement of the protocol FSA to be calculated on-the-fly.

Acknowledgments

The author would like to express his sincere thanks to Prof. Lars Kristensen for productive discussions in 2005 and for originally proposing the idea of transient states, and to Prof. Jonathan Billington for providing valuable comments on preliminary versions of this paper.

References

1. W.A. Barrett, R. M. Bates, D. A. Gustafson, and J.D. Couch. *Compiler Construction: Theory and Practice.* Science Research Associates, 2nd edition, 1986.
2. B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and Ph. Schnoebelen. *Systems and Software Verification - Model-Checking Techniques and Tools.* Springer, 2001.
3. J. Billington, G. E. Gallasch, and B. Han. A Coloured Petri Net Approach to Protocol Verification. In *Lectures on Concurrency and Petri Nets, Advances in Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 210–290. Springer-Verlag, 2004.
4. D. Burdett. Internet Open Trading Protocol - IOTP Version 1.0. RFC 2801, IETF, April 2000.
5. C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems.* Kluwer Academic Publishers, 1999.
6. S. Christensen, L. M. Kristensen, and T. Mailund. A Sweep-Line Method for State Space Exploration. In *Proceedings of TACAS 2001*, volume 2031 of *Lecture Notes in Computer Science*, pages 450–464. Springer-Verlag, 2001.
7. E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* The MIT Press, 2000.
8. FSM Library, AT&T Research Labs. `http://www.research.att.com/~fsmtools/fsm/`.
9. G. E. Gallasch, B. Han, and J. Billington. Sweep-line Analysis of TCP Connection Management. In *Proceedings of the International Conference on Formal Engineering Methods (ICFEM'05)*, volume 3785 of *Lecture Notes in Computer Science*, pages 156–172. Springer-Verlag, 2005.
10. G. E. Gallasch, C. Ouyang, J. Billington, and L. M. Kristensen. Experimenting with Progress Mappings for the Application of the Sweep-Line Analysis of the Internet Open Trading Protocol. In *Fifth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, pages 19–38. Department of Computer Science, University of Aarhus, 2004. Available via `http://www.daimi.au.dk/CPnets/workshop04/cpn/papers/`.
11. G. E. Gallasch, S. Vanit-Anunchai, J. Billington, and L. M. Kristensen. Checking Language Inclusion On-The-Fly with the Sweep-line Method. In *Proceedings of CPN'05*, Department of Computer Science Technical Report, DAIMI PB 576, pages 1–20. University of Aarhus, 2005.
12. G. E. Gallasch, S. Vanit-Anunchai, J. Billington, and L. M. Kristensen. Checking Safety Properties On-The-Fly with the Sweep-line Method. *International Journal on Software Tools for Technology Transfer*, 9(3-4):371–392, 2007.
13. R. J. van Glabbeek. The Linear Time - Branching Time Spectrum II: The semantics of sequential systems with silent moves (extended abstract). In *Proceedings of the 4th International Conference on Concurrency Theory (CONCUR'93)*, volume 715 of *Lecture Notes in Computer Science*, pages 66–81. Springer, 1993.
14. B. Han. *Formal Specification of the TCP Service and Verification of TCP Connection Management.* PhD thesis, Computer Systems Engineering Centre, UniSA, Adelaide, Australia, December 2004.
15. J. E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 2nd edition, 2001.
16. K. Jensen and L.M. Kristensen. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems.* Springer, 2009.
17. T. Jéron, H. Marchand, and V. Rusu. Symbolic Determinisation of Extended Automata. In *Proceedings of 4th IFIP International Conference on Theoretical Computer Science (TCS 2006)*, volume 209 of *IFIP International Federation for Information Processing*, pages 197–212. Springer Boston, 2006.

18. T. Jussila, K. Heljanko, and I. Niemelä. BMC via On-the-Fly Determinisation. *International Journal on Software Tools for Technology Transfer*, 7(2):89–101, 2005.
19. D. C. Kozen. *Automata and Computability.* Springer-Verlag, 1997.
20. L. M. Kristensen and T. Mailund. A Generalised Sweep-Line Method for Safety Properties. In *Proceedings of FME'02*, volume 2391 of *Lecture Notes in Computer Science*, pages 549–567. Springer-Verlag, 2002.
21. O. Kupferman and M. Y. Vardi. Model Checking of Safety Properties. *Formal Methods in System Design*, 19(3):291–314, 2001.
22. C. Ouyang. *Formal Specification and Verification of the Internet Open Trading Protocol using Coloured Petri Nets.* PhD thesis, Computer Systems Engineering Centre, UniSA, Adelaide, Australia, June 2004.
23. C. Ouyang and J. Billington. On Verifying the Internet Open Trading Protocol. In *Proceedings of 4th International Conference on E-Commerce and Web Technologies (EC-Web)*, volume 2738 of *Lecture Notes in Computer Science*, pages 292–302. Springer, 2003.
24. A. Valmari. The State Explosion Problem. In *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer-Verlag, 1998.
25. M. Y. Vardi and P. Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *Proceedings of 1st Symposium on Logic in Computer Science, Cambridge, USA*, pages 332–344. IEEE Computer Society Press, 1986.