

Separation Logic and the Mashup Isolation Problem

Ankur Taly

Computer Science Department,
Stanford University

Abstract. This work was done as part of my PhD qualifier exam. My qualifier exam problem was to perform an in depth study of separation logic. In this paper, I survey two variants of separation and then show how they can be applied to solve the isolation problem for mashups. I rigorously define the inter-component isolation property for a basic class of mashups, called *variable separated mashups*. I then derive two procedures for verifying isolation of variable separated mashups using separation logic and its variants. Finally, I provide a rigorous proof of correctness for both the procedures.

1 Introduction

Most contemporary websites, combine content from multiple sources thereby generating high value applications, also known as *mashups*. Some prominent examples are OpenSocial [5] platforms, iGoogle [6], Facebook [19], and the Yahoo! Application Platform [21], which allow third parties (users of the site) to build applications that get served to other users. The most common language for writing content for such applications is Javascript (intertwined with HTML). The individual contents being mixed are known as *components* of the mashup and content providers as *principals*. In most cases, the principals participating in the mashup are mutually untrusting and want to protect the functionality of their components from potentially malicious behavior of other components. Thus mashups must be appropriately verified to ensure that every possible execution of the mashup satisfies the security requirements of all the principals involved, including the hosting page and the end-user. It is helpful to split the security goals for a mashup into: (1) Isolation of the hosting page (2) Isolation of one component from another. Significant progress [10, 11] has been made in designing sandboxing techniques for isolating third-party components from security critical resources of the hosting page. However not many theoretical techniques have been developed for tackling the problem of isolating one component from another. As illustrated by some of the attacks on Yahoo! AdSafe [4] and Facebook FBJS [20], described in [9], isolating security-critical elements of a hosting page from applications, does not always isolate applications from each other.

In this paper we consider the problem of verifying complete inter-component isolation for a restricted class of mashups called *variable-separated mashups*. These mashups are very similar to the “basic mashups” defined in [9]. All components of such a mashup are programs from a sequential programming language. The mashup is obtained by renaming the variables appearing in each program and then sequentially composing them in some order. This is a reasonable model for a web page consisting of multiple non-interacting advertisements or a Facebook ([20]) like page consisting of multiple non-interacting Facebook applications. The isolation property that we want to verify can be informally stated as follows: Isolation holds for a mashup iff the execution of each component within the mashup is *similar* to the independent execution of that component alone in the same initial execution environment. Informally, executions are considered similar if they involve the exact same sequence of read/write memory actions. This isolation property is semantically similar¹ to the inter-component isolation property considered in [9]. However that focus of the work in [9] is primarily on designing techniques for enforcing isolation rather than verifying isolation for a given mashup. In this paper, we design techniques for verifying isolation properties of mashups using separation logic. A simple imperative language with references and records has been used in this work, to create a theoretical foundation for exploring the applications of separation logic in verifying isolation for mashups. Although the model used may not be able to incorporate the vast features and functionalities of a language like JavaScript used in actual mashups, it is a good starting point for working out some of the theoretical ideas.

¹ The isolation property in [9] is stated differently as: actions performed by any two components are non-influencing

Separation logic is an extension of Hoare logic designed specifically to deal with programs that support aliased storage locations and share mutable data structures. Its foundations are laid on the key observation by Burstall in 1972 [3] that separate program texts which work on separated sections of the store can be reasoned about independently. An intuitionistic logic based on this observation was developed independently by Reynolds [18] and Ishtiaq and O’Hearn [8]. Ishtiaq and O’Hearn, in their paper also presented a classical version of the logic [8] [13], for LISP like programs without support for address arithmetic. This logic was then generalized by Reynolds to arbitrary address arithmetic. In this paper we will use the classical version of separation logic without address arithmetic. Following the success of classical separation logic in analyzing programs with pointers and mutable data structures, several variants of the logic have been developed. One such variant that we consider in this paper is “separation logic with permissions” [1]. This variant was developed by Bornat et al by incorporating fractional permissions [2] in separation logic.

In the remaining part of this section, we will briefly describe separation logic and then explain the motivation behind using separation logic for solving the mashup isolation problem. We focus on describing the intuitions behind the main constructs of the logic. A more formal treatment of the logic is present in the later sections.

1.1 Separation Logic

In this section, we present the main motivations behind the design of separation logic. Let *while-programs* be a basic imperative programming language with just while loops, if statements and simple assignments of the form $x := E$. Programs in this language are evaluated with respect to a store which is a mapping from variables to values.

Hoare logic. In 1969, Hoare [7] introduced an axiomatic method for proving properties of while-programs, which has since then been the foundation of several techniques in verification of imperative programs. Hoare logic consists of proving ‘pre condition/post condition assertions’ on programs, which are also called *Hoare triples*. The general form of a triple is $\{P\}C\{Q\}$ where C is the program and P and Q are first order assertions on the contents of variables. $\{P\}C\{Q\}$ means that if assertion P holds initially, if the execution of C halts², then Q holds on the final state of the variables. For example $\{x = n\}x = x + 1\{x = n + 1\}$ is a valid Hoare triple. The logic consists of axioms and inference rules for deriving specifications of a large program from corresponding specifications of smaller sub-programs. An attractive feature of Hoare logic is that, the axiom for assignment statements is remarkably simple - $\{P[E/x]\}x := E\{P\}$, where $P[E/x]$ is the assertion obtained by substituting x with E . The underlying simplicity is because program variables are used as both logical variables and assignable storage variables.

Handling pointers and references. We now add records and references to the language of while programs. We call the new language \mathcal{L} . Programs in \mathcal{L} are evaluated with respect to a heap and a store. The heap is essentially a description of memory and is specified by a mapping from locations to records. Records are mappings from property names to values which may either be natural numbers or locations. The language \mathcal{L} consists of a new form of assignment - $x.p := E$, which corresponds to assigning the value of expression E to the property p of the object at location l of the heap. Notice that the statement $x := E$ leads to an assignment to the variables x , whereas We will use the term location-property as a shorthand for “property of the object stored at the location”. Two locations properties l_1, p_1 are different if either $l_1 \neq l_2$ or $p_1 \neq p_2$.

The substitution based assignment axiom does not work for assignments of the form $x.p := E$. [12] presents a version of Hoare logic for while programs with pointers. In order to adapt that logic to the language \mathcal{L} , we need to extend the assertion language with assertions of the form $cont\ x.p = M$ (also written as $x \mapsto y$), which means heap location x has a property p containing value of expression M . Thus assertions now express conditions on both the contents of variables and contents of the heap.

Consider the assignment $x.p := E$. This assignment not only changes the assertion $cont\ x.p = M$ to $cont\ x.p = E$, but also any assertion $cont\ y.p = M$ if x and y are *aliases*, that is, contain the same location name. Therefore the corresponding Hoare triple now becomes $\{P[cont\ x.p := E / cont]\}x := E\{P\}$ where $cont\ x.p = E = \lambda l, q : \text{If } (x == l) \text{ AND } (p = q) \text{ then } E \text{ else } (cont\ l.q)$. This Hoare triple is clearly more complex and less intuitive than the one for simple assignments. This triple

² This is also called partial-correctness. In this paper we will not consider total correctness

essentially says that any assertion $\text{cont } y.q = M$ is preserved under the assignment $x.p := E$ if $p \neq q$ OR x and y are not aliases. Notice that this statement in english is quite intuitive. Thus if we could somehow explicitly write out all the “x does not alias y” like conditions in the initial assertion, then expressing the post-condition under the assignment $x.p := E$ would be straightforward. This is exactly the intuition that is formally captured by the separating conjunction $*$ in separation logic.

Basic separation logic. As mentioned above, the simplicity of separation logic axioms lies in making all alias information explicit in the assertions. Separation logic extends the assertion language of Hoare logic with assertions of the form $P * Q$. Instead of the assertion $\text{cont } x.p = M$, we will now use $x.p \mapsto M$ which is more standard in the separation logic literature. An assertion of the form $x.p \mapsto M * y.p \mapsto N$ asserts that (1) $x.p$ contains value M (2) $y.p$ contains value N and (3) $x.p$ and $y.p$ are different location-properties. Thus if the assertion holds initially then it follows that $y.p$ will be unchanged in an assignment to $x.p$. Therefore the triple $\{x.p \mapsto M * y.p \mapsto N\} x := E \{x.p \mapsto E * y.p \mapsto N\}$ holds in separation logic.

But what about invariance of other assertions like $z.p \mapsto N'$ (when $z \neq x, y$)? Does this mean that we need to make alias information explicit for all pairs of variables? The answer to this is NO in separation logic. This is because of the *frame rule*. The frame rule is an inference rule, formally stated as $\frac{\{P\}C\{Q\}}{\{P * R\}C\{Q * R\}}$ ³. The meaning of this rule is that if $\{P\}C\{Q\}$ holds, then any assertion R which is *separate* from P will be preserved during the reduction of C . Separate means that the set of location-properties mentioned by R are completely disjoint from the set mentioned by P . Thus the assignment axiom can be simply be written as $\{x.p \mapsto _ \} x.p = E \{x.p \mapsto E\}$. This local specification can be appropriately extended with assertions of the form $z.p \mapsto N'$ using the frame rule.

A valid separation logic triple $\{P\}C\{Q\}$ also has the additional property that any heap that satisfies the pre-condition P is such that it includes all locations and corresponding property names that ever get accessed during a reduction of the program C . This means that executing the program C with respect to a heap and store that only contain the locations and variables mentioned by P respectively, will never lead to a run-time error. This property has its roots in the slogan by Milner (as paraphrased by O’Hearn [8]): *well-specified programs dont go wrong*. An important consequence of this is that “any” specification for a program in separation logic gives us an over-approximation of the set of heap locations and property names that get accessed. This property will be the key to solving the mashup isolation problem

Separation logic with permissions. Just like adding the separating conjunction to Hoare logic assertions eases the analysis of pointer programs, adding permissions to points-to relations helps in proving more non-interference properties of programs. The intuition lies in pronouncing the points-to assertion $x.p \mapsto M$ as “the program has the permission to read and write property p of location x ”. Therefore the points-to relation $x.p \mapsto M$ is now replaced with relations of the form $x.p \overset{a}{\mapsto} M$ where $a \subseteq \{r, w\}$. Given a specification of the form $\{x.p \overset{r}{\mapsto} M\}C\{Q\}$ we now have the additional information that property p of location x is only read, when C is executed on a heap satisfying the pre-condition. The assignment axiom in this logic is therefore $\{x.p \overset{w}{\mapsto} _ \} x.p = E \{x.p \overset{w}{\mapsto} E\}$. The semantics of the separating conjunction are modified so that $P_1 * P_2$ allows P_1 and P_2 to have aliases to the same location-properties, provided they are all read-only. The intuition behind this becomes clear if we analyze the frame rule under the modified separating conjunction. Given any specification $\{P\}C\{Q\}$, any assertion R is preserved during the reduction of the program C , if any location-property mentioned in R is either not mentioned in P or is mentioned as read-only. In both cases, it is clear that the contents of the location-property would be preserved and hence the frame rule will be sound.

1.2 Solving the Mashup Isolation Problem

We now give the main intuition behind using separation logic for solving the mashup isolation problem. It is evident from the inter-component isolation property for mashups that verifying it would involve the following:

1. Approximating the portion of heap that would be accessed by each component.

³ This rule also has a side conditions on the set of variables modified by C , however we skip them here as they don’t add to the intuition

2. Showing that the portion of heap stays invariant under the set of actions performed by other components.

As mentioned in the previous section, any separation logic specification $\{P\}C\{Q\}$ of a component C carries enough information about the set of locations and property names that will be accessed during a reduction of C . Further, showing that a portion of the heap stays invariant under the actions performed by a component can be done by showing using the frame rule. Thus basic separation logic seems perfect for carrying out proofs of isolation properties of mashups. A more rigorous connection between the isolation problem and the applicability of separation logic is established in section 3.1. The procedure that we derive for verifying isolation can be informally stated as follows: Given a mashup with components C_1, \dots, C_n and an initial heap H :

1. Infer specifications $\{P_1\}C_1\{Q_1\}, \dots, \{P_n\}C_n\{Q_n\}$ for each of the components using the axioms and inference rules of separation logic.
2. Show that H satisfies the assertion $P_1 * \dots * P_n$

The above procedure is good for carrying out hand proofs of isolation properties of mashup (see [17][13] for some examples of proof by hand using separation logic). At this point, however the procedure is far from being fully automatic and decidable. These concerns are beyond the scope of this work and we plan to address them in future.

The above procedure might seem too restrictive as it requires components to access completely disjoint portions of the heap. Isolation can still hold if two components access a common property of a location but do not write to it. This is where separation logic with permissions becomes useful. In section 5, we present a procedure for verifying isolation using separation logic with permissions. Remarkably, the two procedures are syntactically the same, almost verbatim! The only difference is that in the procedure using separation logic with permissions, we infer the specifications using the logic with permissions. All the other differences get hidden inside the modified interpretation of the separating conjunction in separation logic with permissions.

1.3 Organization

We summarize the main contributions of this work as follows:

1. Rigorous definition of the mashup isolation problem, in the context of a simple imperative language with references and records.
2. Two different procedures for verifying isolation using two variants of separation logic.
3. Rigorous proofs of correctness for both the procedures.

Rest of this paper is organized as follows: section 2 describes the syntax, semantics and certain semantic properties of the programming language, section 3 formally describes the isolation problem for mashups, section 4 describes basic separation logic and presents a solution to the mashup isolation problem using it, section 5 describes separation logic with permissions and presents a solution to the mashup isolation problem using it, section 6 presents some of the ongoing work followed by a conclusion in section 7.

2 The Programming Language \mathcal{L}

In this section we describe the syntax and semantics of the programming language \mathcal{L} which will be considered throughout this paper.

2.1 Syntax

The language \mathcal{L} is a simple imperative language with records (restricted objects), references, while loops and conditionals. Reading and writing to record fields is done in the standard way using the 'dot' notation. The language does not allow direct pointer manipulation. The complete grammars for basic expression, boolean expressions and commands are given in figure 2.1. We use Exp , $Bexp$ and \mathcal{C} to denote the set of all possible basic expressions, boolean expressions and commands that are derivable from the respective grammars.

$E ::=$	x $E + E$ $E * E$ n	<i>BasicExpressions (Exp):</i> variable plus multiply Natural numbers
$B ::=$	$isLoc?(E)$ $isNat?(E)$ $E = E$ $B \implies B$ $true \mid false$	<i>BoolExpressions (Bexp):</i> location check number check equality check implication true/false
$C ::=$	$x := E$ $x := E.p$ $E.p = E$ $x := \{\tilde{p}_i : \tilde{E}_i\}_{i \in \{1, \dots, n\}}$ $if B then C else C$ $while B then C$ $C; C$ $normal$ $abort$	<i>Commands (Comm):</i> assignment lookup (p is any string) update object creation if while sequential composition Normal terminal state Erroneous terminal state

Fig. 1. Syntax

2.2 Semantics

We give a small-step style structural operational semantics [16] to the language. We formalize program states in the standard way, as triples consisting of a heap, store and the command being executed.

Heaps and Stores. Heaps are partial functions from a set Loc of locations to a set $HeapValues$. Heap values are records, which are partial functions from a set of property names \mathbb{P} to a set $StoreValues = Nat \cup Loc$ (where Nat is the set of all possible natural numbers). Stores are partial functions from a set of variable names $Vars$ to $StoreValues$. In this paper we will use H, K to denote heaps and A, B to denote stores. We use $Heaps$ to denote the domain of all possible heaps and $Stores$ to denote the domain of all possible stores. Further, we use $Heaps - Stores$ to denote the set $Heaps \times Stores$.

Given a heap H , we define $dom(H)$ as the set of all location-property pairs l, p where $(H(l))(p)$ is defined. For a store A , we define $dom(A)$ as the set of all variables x where $A(x)$ is defined. Given two heaps H and K , we say that $H \# K$ iff $dom(H) \cap dom(K) = \emptyset$ and $H \subseteq K$ iff $dom(H) \subseteq dom(K) \wedge \forall (l, p) \in dom(H) : H(l).p = K(l).p$

For any two heaps H and K , if $H \subseteq K$ then $K - H$ is the 'set-minus' of the partial functions H and K . Further if $H \# K$ holds then $H.K$ is the heap obtained by taking union of the partial function H and K . These relations are extended to stores in a similar fashion.

For any location l and property name p , we use $H(l).p$ as a shorthand to denote the value of property p of object $H(l)$, provided $(l, p) \in dom(H)$. Further we use $H[l.p \rightarrow v]$ to denote the heap obtained by updating property p of location l to value v . Similarly, for any variable x , we use $A[x \rightarrow v]$ to denote the store obtained by updating the variable x to value v , in the store A . Given a set of variable names \mathcal{X} and a store A , we define $Proj(A, \mathcal{X})$ as the store A' such that $dom(A') = dom(A) \cap \mathcal{X}$ and for all $x \in \mathcal{X} \cap dom(A)$, $A'(x) = A(x)$.

Semantics of expressions. We use the semantic functions

$\llbracket _ \rrbracket_{Exp} : Exp \rightarrow Stores \rightarrow StoreValues \cup \{error\}$ and

$\llbracket _ \rrbracket_{Bexp} : Bexp \rightarrow Stores \rightarrow \{true, false, error\}$

to express the semantics of basic expression and boolean expressions. Thus the semantics of expressions depend only on the store and not the heap. The definitions of these functions are completely standard and

are mentioned in the appendix 8.1. The semantics of a variable expression: $\llbracket x \rrbracket_{Exp} A$ is *Error* if $x \notin dom(A)$ and $A(x)$ otherwise. The semantics of $E_1 op E_2$ where $op \in \{+, -, <\}$, is *Error* whenever the semantics of E_1 or E_2 is not a natural number. In other words, semantics of 'ill-typed' expressions is *Error*. Boolean expressions consist of equality checks over basic expressions and two predicates $isLoc?(x)$ and $isNat?(x)$, whose semantics is defined as $\llbracket isLoc?(x) \rrbracket_{Bexp} A$ is true iff $A(x) \in Loc$ and $\llbracket isNat?(x) \rrbracket_{Bexp} A$ is true iff $A(x) \in Nat$. The predicate $isLoc?(x) \vee isNat?(x)$ is therefore true for a store A iff $x \in dom(A)$. We use $present(x)$ as a short hand for this predicate. It is easy to see that given a basic expression E , using a boolean combination of the predicates $isLoc?$ and $isNat?$ applied over the free variables of E , we can write a predicate $Wt(E)$ which is true iff E is well-typed. Since the set of boolean expressions $Bexp$ is closed under arbitrary boolean combinations, we have $\forall E \in Exp : Wt(E) \in Bexp$. Therefore for all stores A we have, $\llbracket Wt(E) \rrbracket_{Bexp} A \implies \llbracket E \rrbracket_{Exp} A \neq Error$.

Reduction rules for commands. The semantics of commands are described using small-step transition rules. The general form of a transition rule is $\frac{\langle premise \rangle}{H, A, C \rightarrow K, B, D}$ where $\langle premise \rangle$ denotes the conditions that must be true for the rule to apply. There are two kinds of transition rules

1. Axioms: the premise of these rules do not involve the transition relation \longrightarrow . These include the expression-context rules.
2. Command-context rules: these rules involve the reduction of a sub-command in their premise.

The semantics includes two kinds of terminal states - one where the command is *normal*, which denotes normal termination and the other where the command is *abort*, which denotes a run-time error. Run-time errors could either be a type error or memory error (accessing a location not present in the heap or variable not present in the store). All the reduction rules are deterministic except the one for object creation, where the location assigned to the object created could be chosen non-deterministically from the set of unallocated locations. The complete list of reduction rules is mentioned in appendix 8.1.

2.3 Semantic Properties

In this section we describe some of the notations, definitions and semantic properties that will be used to describe the results stated in the rest of the paper.

2.4 Notation and Definitions

Throughout this paper we use S, T to denote program states. We use *ProgStates* to denote the domain of all possible program states. Given a state S , we use $\mathcal{H}(S)$, $\mathcal{S}(S)$ and $\mathcal{C}(S)$ to denote the heap, store and command parts of a state respectively. Given any states S and T , we say that $S \rightsquigarrow T$ iff S reduces to T in zero or more reduction steps (zero case corresponds to $S = T$). Further for any state S , we say that $S \uparrow$ holds iff $\neg \exists S' : S \rightsquigarrow S' \not\rightarrow$. A reduction trace of a state S is defined as any sequence (finite or infinite) of states S, S_1, \dots such that $S \longrightarrow S_1 \longrightarrow \dots$. For any state S , we define $Traces(S)$ as the set of all possible reduction traces of S (multiple traces are possible as one of the reduction rules is nondeterministic). For any trace τ , we define $First(\tau)$ as the first state of the trace and $Final(\tau)$ as the final state of the trace, if the trace is finite.

A useful predicate on states is *Safe* which characterizes the set of states whose reduction never involves a run-time error. This is formally defined as follows.

Definition 1 (Run-time Safety). A state H, A, C is said to be safe, denoted by $Safe(H, A, C)$ iff

$$\neg \exists K, B : H, A, C \rightsquigarrow K, B, abort$$

Actions. We now define the set of 'actions' performed when a state S reduces to a state T in many steps. Actions are of two kinds: heap actions and store actions. A heap action is denoted by a triple (l, p, a, v) where $l \in Loc$, $p \in \mathbb{P}$, $a \in \{r, w\}$ and $v \in StoreValues$. which refers to reading value v (writing value v) from (to) property p of location l if $a = r$ (if $a = w$). Similarly, a store action is denoted by a pair (x, a, v) where $x \in Vars$, $a \in \{r, w\}$ and $v \in StoreValues$, which refers to reading value v (writing value v) from

(to) variable x of the store if $a = r$ (if $a = w$). Formally, we denote $HeapActions = Loc \times \mathbb{P} \times \{r, w\}$ as the domain of heap actions, $StoreActions = Vars \times \{r, w\}$ as the domain of store actions and $Actions = HeapActions \cup StoreActions$ as the domain of all actions. We use $act, act_1 \dots$ to denote actions. In our analysis, we shall use the notion of action sequences, which is an ordered list of actions (includes both store and heap actions). We use $\gamma, \gamma_1 \dots$ to denote action sequences.

Definition 2 (Access). Given states S and T such that $S \rightsquigarrow T$, $Acc(S, T)$ is defined as the precise sequence of actions performed during the reduction of S to T . The Acc function is extended to traces and $Acc(\tau)$ corresponds to the sequence of actions performed in any trace τ .

During our analysis we will often use the subsequences $Acc_{read}^{store}(S, T)$, $Acc_{write}^{store}(S, T)$, $Acc_{read}^{heap}(S, T)$ and $Acc_{write}^{heap}(S, T)$ which denote the store read, write and heap read, write action sequences respectively.

Given any command C , it is possible to statically upperbound the set of variables names that will be read or written during the execution of the command with respect to any heap store. These upperbounds are given by the sets $free(C)$ and $modifies(C)$ which can be defined inductively over the structure of the commands. The reason such an upperbound exists and is always finite is because variables names cannot be "computed" in the language \mathcal{L} (whereas they can be computed in a language like JavaScript). For example, $free(x := E) = \{x\} \cup free(E)$ and $modifies(x := E) = \{x\}$. The complete list of free and modifies sets for all commands is given in appendix 8.1.

We now state some basic properties that are obeyed by the reduction traces of terms. The first property is about the local behavior of commands and forms the semantic basis for the soundness of the frame rule in separation logic. The property essentially says that if the execution of a command is run-time safe and terminating for a heap-store H, A , then it will also be memory safe and terminating for any heap K, B such that $H \subseteq K$, $A \subseteq B$. Further the execution of C on K, B will only involve access to the sub-heap and sub-store H and A , and will therefore preserve the remaining portion of the heap and the store as it is.

Proposition 1. Consider any heap-store pairs H, A and K, B such that $H \subseteq K$ and $A \subseteq B$. For all commands $C \in Comm$ for the language \mathcal{L} , the following must hold:

1. Safe Monotonicity. $Safe(H, A, C) \implies Safe(K, B, C)$
2. Termination Monotonicity. $\neg H, A, C \uparrow \wedge Safe(H, A, C) \implies \neg K, B, c \uparrow Safe \wedge (K, B, C)$
3. Frame Property. If $Safe(H, A, C)$ holds then for all K', B', C' such that $K, B, C \rightsquigarrow K', B', C'$ there exists H'', A'' such that $H, A, C \rightsquigarrow H'', A'', C'$ AND $(K' - H'', B' - A'') = (K - H, B - A)$ AND $Acc((H, A, C), (H'', A'', C')) = Acc((K, B, C), (K', B', C'))$

State simulation. We now define a simulation relation on states which characterizes when the set of behaviors of one state can be simulated by the other. The simulation relation forms the crux of the isolation property of mashups, which is defined in the next section. Informally, a state S simulates T if for every reduction trace $\tau_T \in Traces(T)$, there is a reduction trace $\tau_S \in Traces(S)$ for which the action sequence is *similar* to the action sequence of τ_T . The notion of action sequence similarity is defined with respect to a variable renaming map $ren : Vars \rightarrow Vars$, which is a bijective map on the set of variables to itself. We extend any renaming map ren so that for a store action (x, a, v) we define $ren(x, a, v)$ as the action $ren(x), a, v$. For any heap action act , we define $ren(act) = act$.

Definition 3 (Access Similarity). Given action sequences γ_1, γ_2 and a variable renaming map $ren : Vars \rightarrow Vars$, we say that γ_1 is similar to γ_2 upto renaming ren , denoted by $\gamma_1 \sim_{ren} \gamma_2$ iff $length(\gamma_1) = length(\gamma_2)$ and

$$\forall i, 1 \leq i \leq length(\gamma_1) : ren(\gamma_1[i]) = \gamma_2[i]$$

where for an action sequence γ , $\gamma[i]$ denotes the i^{th} action in the sequence.

We are now ready to formally define the simulation relation on program states.

Definition 4 (Simulation Relation). A state S simulates T , denoted by $S \sim T$ iff there exists a variable renaming ren such that:

1. Safe Monotonicity. $Safe(S) \implies Safe(T)$
2. Termination Monotonicity. $\neg S \uparrow \wedge Safe(S) \implies \neg T \uparrow \wedge Safe(T)$

3. Access Similarity If $Safe(S)$ holds then for all $\tau_T \in Traces(T)$, there exists $\tau_S \in Traces(S)$ such that $Acc(\tau_S) \sim_{ren} Acc(\tau_T)$

From proposition 1, $H, A, C \sim K, B, D$ whenever $H \subseteq K$, $A \subseteq B$. We next state a proposition which provides an “if and only if” condition for when $S \sim T$, under the restriction that S, T are safe, $\mathcal{C}(S) = \mathcal{C}(T)$ and $Proj(\mathcal{S}(S), free(\mathcal{C}(S))) = Proj(\mathcal{S}(T), free(\mathcal{C}(T)))$. The main result is that the heaps of state S and T must have the same value for all location-property pairs that get read during the first time they are accessed in the reduction of S . In order to formalize this result, we define $Read(S)$ as the set of location-property name pairs that get *read* during the first time they are accessed in any reduction trace of S . Similarly $Write(S)$ is defined as the set of location-property name pairs l, p that get written-to during the first time that they are accessed.

Proposition 2. For all safe states S, T such that $\mathcal{C}(S) = \mathcal{C}(T)$ and $Proj(\mathcal{S}(S), free(\mathcal{C}(S))) = Proj(\mathcal{S}(T), free(\mathcal{C}(T)))$ holds, $S \sim T$ holds IFF:

$$\forall l, p : l, p \in Read(S) \cap dom(\mathcal{H}(S)) \implies \mathcal{H}(S)(l).p = \mathcal{H}(T)(l).p$$

3 The Mashup Isolation Problem

In this work, we consider a basic form of mashup based on sequential composition. Each component of the mashup is a program from the language \mathcal{L} . The mashup is obtained by prefixing each variable in a component by a principle-specific prefix and then sequentially composing the resulting components. The initial execution environment is specified by a heap and a store. The variable renaming is done in order to ensure that different components execute in different namespaces. This is mainly done to simplify the technical analysis and does not simplify the problem too much. This is because the components can still influence each other via the heap. Variable renaming only cuts out all forms of influence via the store. The isolation property considered in this paper, requires that the execution of any component in a mashup must be *oblivious* of all the other components and must be *similar* to executing the component independently on the starting heap-store of the mashup. We now move on to the formal definitions. We start with the definition of mashups.

Before giving the formal definition of variable-separated mashups we define a map Rn such that for any command C and string a , $Rn(C, a)$ is defined as the command obtained by replacing every variable x in C by the variable $a : x$, where $:$ denotes string concatenation. We assume that $\forall x : x \in Vars \implies a : x \in Vars$. Further for a store A , we define $Rn(A, a)$ as the store obtained by replacing each map $x \rightarrow v$ with $id_i : x \rightarrow v$.

Definition 5 (Variable-Separated Mashup). Given a heap-store H, A and commands $C_1, \dots, C_n \in Comm$, a variable-separated mashup $\mathcal{M}(H, A, (C_1, id_1), \dots, (C_n, id_n))$ is defined as the state $H_{mash}, A_{mash}, C_{mash}$ where

- $H_{mash} := H$.
- $A_{mash} := Rn(A, id_1) \dots Rn(A, id_n)$.
- $C_{mash} := Rn(C_1, id_1); \dots ; Rn(C_n, id_n)$.

Given a mashup $\mathcal{M}(H, A, (C_1, id_1), \dots, (C_n, id_n)) = (H_{mash}, A_{mash}, D_1; \dots ; D_n)$, consider any trace $\tau \in Traces(H_{mash}, A_{mash}, D_1; \dots ; D_n)$. It is straightforward from the semantics of sequential composition that τ has the following form:

$$\tau := H_{mash}, A_{mash}, CCon_1[D_1] \rightsquigarrow H_2, A_2, CCon_2[D_2] \rightsquigarrow \dots \rightsquigarrow H_k, A_k, CCon_k[D_k] \rightsquigarrow$$

Thus the trace can be split into sub-traces such that the i^{th} subtrace corresponds to the reduction of command C_i inside a reduction context. We denote the sub-trace of τ corresponding to the command D_i by $States(\tau, id_i)$. If the subtrace corresponding to command D_i does not terminate or results in a run-time error then $States(\tau, id_j) = \emptyset$ for all $j \geq i + 1$.

We are now ready to define isolated mashups. Informally a mashup $\mathcal{M}(H, A, (C_1, id_1), \dots, (C_n, id_n))$ is isolated for a heap-store H, A if for all i and all $\tau \in Traces(\mathcal{M}((C_1, id_1), \dots, (C_n, id_n)))$, the state H_{mash}, A_{mash}, C_i simulates the state H_i, A_i, C_i where H_i, A_i is the starting heap-store of the sub trace $States(\tau, id_i)$ (See section 2.3 for a definition of the simulates relation). Informally, this means that for any command, the behavior obtained by executing it as part of a mashup can be simulated by executing it independently. .

Definition 6 (Isolation property). Given a heap-store H, A and commands C_1, \dots, C_n from principles id_1, \dots, id_n respectively, the mashup $\mathcal{M}(H, A, (C_1, id_1), \dots, (C_n, id_n))$ is isolated iff for all traces $\tau \in \text{Traces}(\mathcal{M}((C_1, id_1), \dots, (C_n, id_n)))$, we have

$$\forall i : \text{States}(\tau, id_i) \neq \emptyset \implies (H, A, C_i) \sim H_i, A_i, \text{Rn}(C_i, id_i)$$

where $H_i, A_i = \mathcal{HS}(\text{First}(\text{States}(\tau, id_i)))$ The isolation property is formally denoted by the predicate $\text{Isolation}(\mathcal{M}(H, A, (C_1, id_1), \dots, (C_n, id_n)))$.

The above mashup isolation property is a special case of the one defined in [9]. The one in [9] also has the "isolation of hosting page" requirement, which informally means that each C_i does not perform certain forbidden read and write actions on the heap and store. As explained in [9], significant progress has been made in verifying and enforcing the "isolation of hosting page" property. In this work, we therefore focus on the inter-component isolation problem. We now state the mashup isolation problem that we attempt to solve in this paper.

Isolation Problem. Given a heap-store H, A and commands C_1, \dots, C_n from principles id_1, \dots, id_n respectively, verify that the mashup $\mathcal{M}(H, A, (C_1, id_1), \dots, (C_n, id_n))$ is isolated.

3.1 Analyzing the Isolation Problem

We now present a brief analysis of the isolation problem, in order to motivate our approach for solving the problem. We sketch out a high level reduction of the isolation problem to the problem of verifying that all commands preserve certain invariants on the heap.

Consider a heap-store H, A and components (C_1, id_1) and (C_2, id_2) . Consider the mashup $\mathcal{M}(H, A, (C_1, id_1), (C_2, id_2)) = (H, B_1.B_2, D_1; D_2)$ where, D_1, D_2 are the renamed versions of commands C_1, C_2 with respect to id_1 and id_2 respectively and B_1, B_2 are the renamed versions of the store A with respect to id_1 and id_2 respectively. According to the definition of the isolation property, this mashup is isolated IFF: (1) The state H, A, C_1 simulates $H, B_1.B_2, D_1$ (2) The state H, A, C_2 simulates H_2, A_2, D_2 , where H_2, A_2 is the heap-store obtained when the execution of command C_1 terminates. Using a bisimulation relation, we can show that condition 1 always holds (see lemma 5 in the appendix). Similarly we can show that H, A, C_2 simulates $H, B_1.B_2, D_2$ and therefore condition 2 holds iff $H, B_1.B_2, D_2$ simulates H_2, A_2, D_2 . We can prove that proposition 2 applies to the states $H, B_1.B_2, D_2$ and H_2, A_2, D_2 . Therefore $H, B_1.B_2, D_2$ simulates H_2, A_2, D_2 IFF:

$$\forall l, p : l, p \in \text{Read}(H, B_1.B_2, D_2) \cap \text{dom}(H) \implies H(l).p = H_2(l).p$$

Now H_2 is the final heap obtained by executing D_1 with respect to $H, B_1.B_2$. Thus the above condition and hence isolation holds for the mashup IFF: one of the following holds-

- A. No location-property name pairs that are read during the reduction of D_2 on H, A , are accessed during the reduction of D_1 on H, A .
- B. All location-property name pairs that are read during the reduction of D_2 on H, A , can at-most be read during the reduction of D_1 on H, A .
- C. If any location-property name pair l, p that is read during the reduction of D_2 on H, A , is written-to during the reduction of D_1 on H, A , then the original value $H(l).p$ must be restored before the reduction terminates.

A proof for ANY of the above cases would serve as a proof of the isolation property for the mashup. Further, if no proof exists for ALL of the above cases then we can claim that the isolation property does not hold for the mashup. In other-words complete procedures for verifying all of the above cases would give us a complete procedure for verifying the isolation property. The above analysis can easily be generalized to mashups with n components (in that case we will obtain the above three cases for all distinct pairs of components).

Case A. A sufficient condition for proving this case is that the commands D_1 and D_2 access disjoint portions of the heap, when executed with respect to the heap store $H, B_1.B_2$. Basic Separation logic [17] is an elegant logic for carrying out a proof of the above property (if it holds). The key idea is to use the

separating conjunction to express the disjointness of the portions of the heaps accessed by D_1 and D_2 . This idea is elaborated in section 4.

Case B. A sufficient condition for proving this case is that id any location-property l, p is accessed during the reduction of D_1 and D_2 then it must only be 'read' during the reduction traces of each of them. Separation logic with permissions [1] fits this case very well. The key idea here is to make the separating conjunction "aware" of what location-property names are *read-only* in the two portions of the heap. This idea is elaborated in section 5.

Case C. This case is the most difficult to formally verify. It is difficult to deduce a pre condition-post condition specification $\{P\}C\{Q\}$ for a command C , which can express that certain heap-locations always carry the same value in all initial and final reduction states. We present preliminary ideas for tackling this case in section 6.

4 Basic Separation Logic (SL_1)

We now present the complete assertion language, satisfaction relation and inference rules for classical separation logic applied to the programming language \mathcal{L} . The logic is essentially the same as that considered in [8], with minor differences which we will explain as and when we come across them. We call this logic SL_1 .

4.1 Assertion language

We call the assertion language of basic separation logic as \mathcal{A}_1 . The complete syntax for the assertion language is present in figure.4.1. The assertions are predicates on heap-store pairs. We now present the meaning of each assertion at a high level.

- *Boolean Expressions* - B : These assertions are standard from Hoare logic and express predicates on store variables.
- *Empty Heap* - emp : Asserts that the heap is empty.
- *Points-to* - $E_1.p \xrightarrow{E}_2$: Asserts that property p of location corresponding to expression E_1 contains the value of expression E_2 .
- *Separating Conjunction* - $P_1 * P_2$: Asserts that P_1 and P_2 hold on separate⁴ portions of the heap.
- *Separating Implication* - $P_1 \multimap P_2$: Asserts that P_2 holds on the heap obtained by concatenating the given heap and any heap satisfying P_1 .

The assertion language is closed under first order quantifications over variables. The language is also closed under boolean connectives (\neg, \vee, \wedge), which can be simulated using \implies and *false*. We define the shorthands $E.p \mapsto _$ for assertion $\exists x : E.p \mapsto x$ x not free in e and $E : \{p_i \mapsto E_{i \in \{1, \dots, n\}}\}$ for $E.p_1 \mapsto E_1 * \dots * E.p_n \mapsto E_n$. Further, for any assertion P , we define $free(P)$ as the set of variable names that appear free in P . It is straightforward to define the *free* map inductively on the structure of P .

The points-to relation is a slight variant of the one described in [8] and [17]. The one considered here is at the granularity of individual location-property names pairs and not just locations. As we will see in the next section this also slightly alters the meaning of the separating conjunction and separating implication assertions. This is because, we consider two heaps as separate if they mention the same locations, but with different sets of property names.

4.2 Satisfaction Judgments

Satisfaction judgments formally state the conditions under which an assertion is true for a heap-store. The general form of the satisfaction judgment is $H, A \models_{\mathcal{A}_1} P$. The judgment is defined inductively over the structure of the assertion in figure 4.2. We explain the satisfaction judgment for some of the important assertions here.

⁴ In original separation logic, separate was same as completely disjoint, however in subsequent variants of the logic, researchers have defined more flexible notions of when heaps are separate

Language (\mathcal{A}_1).

$P ::=$	
B	boolean expressions
emp	empty heap
$E.p \mapsto E$	$E \in Exp$, singleton heap
$P * P$	separating conjunction
$P \multimap P$	separating implication
$P \implies P$	classical implication
$\exists x.P$	universal quantification

Syntactic sugar:

$$\begin{array}{l}
 E.p \mapsto - \quad \triangleq \quad \exists x : E.p \mapsto x \quad x \text{ not free in } e \\
 E : \{p_i \mapsto E_{i \in \{1, \dots, n\}}\} \triangleq \quad E.p_1 \mapsto E_1 * \dots * E.p_n \mapsto E_n
 \end{array}$$

Fig. 2. Assertion language \mathcal{A}_1

- *Boolean Expressions*: Since the meaning of expressions is independent of the heap, the satisfaction judgment for boolean expressions depends only the store. The boolean expression $present(x)$ holds for a store IFF x is present in the domain of the store. This assertion will play an important role in the inference rules and axioms in the next section.
- *Points-to*: The meaning of the "points-to" relation is exact. Therefore $H, A \models_{\mathcal{A}_1} E_1.p \mapsto E_2$ IFF: H is a singleton heap with exactly⁵ one location $\llbracket E_1 \rrbracket_{Exp} A$ and one property name p which contains $\llbracket E_2 \rrbracket_{Exp} A$.
- *Separating Conjunction*: The separating conjunction $P_1 * P_2$ holds for a heap-store H, A if H can be divided into heaps H_1 and H_2 with disjoint domains such that $H_1, A \models_{\mathcal{A}_1} P_1$ and $H_2, A \models_{\mathcal{A}_1} P_2$. Notice that:
 1. Only the heap is split and not the store. This allows P_1 and P_2 to share information by using store variables. In other words global variables are allowed but global heap assertions are not.
 2. Since we formalize domains as pairs of heaps and property names, H_1 and H_2 can share heap locations as long as they contain different property names. We differ here from the satisfaction relation present in [8] which requires H_1 and H_2 to have disjoint sets of locations.
- *Separating Implication*: The separating implication allows us to write very powerful assertions, using which we can write complete specification of all heap manipulating commands. For example the assertion $(x.p \mapsto 3) * (x.p \mapsto 5 \multimap P)$ is true for a heap store H, A if the resulting heap obtained by updating $x.p$ to 5 satisfies P . Thus intuitively, we see that \multimap can elegantly be used for writing weakest pre-conditions.
- An assertion can be satisfied at a heap-store H, A even if the heap contains dangling references. For example, the heap $H = \{l_1 : \{p : l_2\}\}$ and store $A = \{x : l_1, y : l_2\}$ satisfies the assertion $x.p \mapsto y$, even though the location l_2 is dangling.

Given any assertion P , we define the set $\llbracket P \rrbracket_{\mathcal{A}_1}$ as $\{H, A \mid H, A \models_{\mathcal{A}_1} P\}$ An assertion P is said to be \mathcal{A}_1 -valid iff for all heap-stores H, A , $H, A \in \llbracket P \rrbracket_{\mathcal{A}_1}$.

4.3 Specifications and Inference rules

We now present inference rules for deducing partial correctness specifications for commands. In this paper we will only be concerned with partial correctness and not total correctness and therefore a specification for a command will always refer to a partial correctness specification. The general form of a specification is the same as that in Hoare logic- $\{P, C, Q\}\{.\}$ The crucial difference between a Hoare logic specification and a separation logic specification is in the definition of semantic-validity. A separation logic specification $\{P, C, Q\}\{i\}$ s semantically valid IFF: the execution of C on any heap store H, A such that $H, A \in \llbracket P \rrbracket_{\mathcal{A}_1}$ must never lead to a run-time error and if the execution terminates then the final heap-store must satisfy the assertion Q . This is formally stated as follows.

⁵ The exactness of the pointsto assertion is the key difference between the classical and intuitionistic versions of separation logic

$H, A \models_{\mathcal{A}_1} P$	$H, A \models_{\mathcal{A}_1} B$	iff	$\llbracket B \rrbracket_{Exp} A = true$
	$H, A \models_{\mathcal{A}_1} emp$	iff	$dom(H) = \emptyset$
	$H, A \models_{\mathcal{A}_1} E_1.p \mapsto E_2$	iff	$dom(H) = \{(\llbracket E_1 \rrbracket_{Exp} A, p)\}$ $H(\llbracket E_1 \rrbracket_{Exp} A).p = \llbracket E_2 \rrbracket_{Exp} A$
	$H, A \models_{\mathcal{A}_1} P_1 * P_2$	iff	$dom(H_1) \cap dom(H_2) = \emptyset \wedge$ $\exists H_1, H_2 : H_1.H_2 = H \wedge$ $H_1, A \models_{\mathcal{A}_1} P_1 \wedge H_2, A \models_{\mathcal{A}_1} P_2$
	$H, A \models_{\mathcal{A}_1} P_1 -* P_2$	iff	$dom(H_1) \cap dom(H) = \emptyset \wedge$ $\forall H_1, H_2 : H.H_1 = H_2 \wedge$ $H, A \models_{\mathcal{A}_1} P_1 \wedge H_2, A \models_{\mathcal{A}_1} P_2$
	$H, A \models_{\mathcal{A}_1} P_1 \implies P_2$	iff	$H, A \models_{\mathcal{A}_1} P_1 \implies H, A \models_{\mathcal{A}_1} P_2$
	$H, A \models_{\mathcal{A}_1} \exists x : P$	iff	$\exists v \in StoreValues : H, A[x \rightarrow v] \models_{\mathcal{A}_1} P$

Fig. 3. Satisfaction of assertions in \mathcal{A}_1

Definition 7 (Validity \models_{SL_1}). A specification $\{P\}C\{Q\}$ is SL_1 -valid, denoted by $\models_{SL_1} \{P\}C\{Q\}$, iff for all heap-stores $H, A \in \llbracket P \rrbracket_{\mathcal{A}_1}$, the following holds

1. $Safe(H, A, C)$
2. For all heap-stores $K, B, H, A, C \rightsquigarrow K, B$, normal $\implies K, B \in \llbracket Q \rrbracket_{\mathcal{A}_1}$

The runtime safety requirement has its roots in the slogan by Milner (as paraphrased by O’Hearn ([8]): *well-specified programs dont go wrong*. We point out two import implications of this requirement:

1. Evert specification $\{P, C, Q\}\{m\}$ ust be such that the pre-condition characterizes only those heaps which have all the location-property names that can potentially get accessed during the execution of the command. Thus the specification $\{true\}x\{. \}p := 5true$, which is semantically valid in Hoare logic, is not valid in separation logic because the empty heap $\{\}$ and store $\{x : l\}$ are present in $\llbracket true \rrbracket_{\mathcal{A}_1}$, but the reduction of $\{\}, \{x : l\}, x.p := 5$ leads to a runtime error.
2. Given any heap-store H, A and a specification $\{P, C, Q\}\{\cdot\}$ if there exists a sub-heap $H' \subseteq H$ is such that $H', A \in \llbracket P \rrbracket_{\mathcal{A}_1}$ then the heap $H - H'$ will be preserved as it is during the execution of the command C on H, A . Thus in some sense, we have a free source of invariants that would be preserved during the reduction of C on the heap H, A . This is the basis of the frame property of separation logic and it greatly helps in scaling up local specifications of program fragments to global specifications.

We now move to the axioms and inference rules for deducing separation logic specification of programs in C . The complete list of axioms and rules is given in figure 4.3. The assignment axiom **S-assignment** is the same as that in Hoare logic because all expressions in l are heap-independent. Further the rules **S-while**, **S-if**, **S-conseq** and **S-seq** are the same as that in Hoare logic. The new axioms are the ones corresponding to the heap manipulating commands in the language \mathcal{L} and the frame rule. We explain the axioms for the specification of the update command here. Consider the local version first.

$$\{E_1.p \mapsto - \wedge Wt(E_2)\}E_1.p = E_2\{E_1.p \mapsto E_2\}[\text{S-UPDATE-LOCAL}]$$

This specification says that for a heap store H, A IF: H contains exactly the location $\llbracket E_1 \rrbracket_{Exp} A$ with property p containing the value $\llbracket E_2 \rrbracket_{Exp} A$ and the store is such that the expression E_2 is well-typed THEN: the evaluation of the command $E_1.p = E_2$ will be run-time safe and the final heap-store K, B would be such that K contains “exactly” the location $\llbracket E_1 \rrbracket_{Exp} B$ with property p containing the value $\llbracket E_2 \rrbracket_{Exp} B$. The assertion $Wt(E_2)$ says that the expression E_2 is well-typed. This is needed for run-time safety. Since no variables are assigned during the evaluation of this command, the store $B = A$.

Using the separating implication, it is also possible to write a backwards specification for the update command.

$$\{(E_1.p \mapsto - \wedge \text{Wt}(E_2)) * ((E_1.p \mapsto E_2) \multimap P)\} E_1.p = E_2\{P\}[\text{S-UPDATE-BACKWARDS}]$$

The precondition of P is written using the separating implication. The main assertion in the pre-condition is $((E_1.p \mapsto E_2) \multimap P)$ which is true for a heap store H', A if whenever H' is concatenated with a heap with location $\llbracket E_1 \rrbracket_{Exp} A$ and property p containing $\llbracket E_2 \rrbracket_{Exp} A$ then P holds for the new heap obtained. We now explain the frame rule.

$$\frac{\{P\}C\{Q\} \quad \text{modifies}(C) \cap \text{free}(R) = \emptyset}{\{P * R\}C\{Q * R\}}[\text{S-FRAME}]$$

The intuition behind the frame rule was explained in section 1. Since the precondition P characterizes all location-properties that can potentially get accessed, all location-properties which are separate from those characterized by P will stay intact. The side condition $\text{modifies}(C) \cap \text{free}(R) = \emptyset$ arises because of the fact that the separating conjunction only splits the heap and not the store. Therefore it is possible that some of the free variables mentioned in R are modified by C , in which case the meaning of R changes and therefore the frame rule does not apply. The side condition $\text{modifies}(C) \cap \text{free}(R) = \emptyset$ ensures that this does not happen. Recently Bornat et al [15] have proposed variants of separation logic where these side conditions do not arise. The main idea there is to enrich the assertion language so that assertion can also express the variables that can potentially get modified and then cleverly change the meaning of the separating conjunction. The treatment is analogous to that of permissions which will be presented in the next section. Avoiding checking for these complex side conditions was part of the motivation for considering “variable-separated” mashups. The variable separation ensures that we only invoke the frame rule when the side condition holds.

We refer the reader to [17] for an explanation of the other inference rules. An important result proven by O’Hearn and Ishtiaq in [8] is that the weakest pre-condition for each heap manipulating command is expressed by the corresponding backwards specifications .⁶

We now formally state the soundness theorem for the logic. For any specification $\{P\}C\{Q\}$, we say that holds, iff $\{P\}C\{Q\}$ can be derived by using \mathcal{A}_1 -valid assertions and the axioms and rules of separation logic. This is also known as ‘provability’.

Theorem 1 (Soundness). $\vdash_{SL_1} \{P\}C\{Q\} \implies \models_{SL_1} \{P\}C\{Q\}$

4.4 Solving the Mashup Isolation Problem

We now present a solution to the mashup isolation problem using SL_1 . Consider a heap-store H, A and components $(C_1, id_1), \dots, (C_n, id_n)$. Let $\mathcal{M}(H, A, (C_1, id_1), \dots, (C_n, id_n)) = H_{mash}, A_{mash}, D_1; \dots; D_n$. By definition, D_i is the renamed version of command C_i , $H_{mash} = H$ and A_{mash} is a store containing multiple renamed copies of the store A . Recall the analysis of mashup isolation problem described at the end of section 3. In this section we present a sufficient condition for case A of the analysis, which in turn will be a sufficient condition for verifying isolation. In accordance with case A , we prove isolation by showing that for any two distinct commands D_i and D_j , the sets of location-properties accessed during their reductions on the starting heap-store H_{mash}, A_{mash} are completely disjoint. Remarkably, basic separation logic seems almost designed to prove properties of this sort. Here are the two main observations based on which we derive our soundness check:

- Any specification $\{P_i\}D_i\{Q_i\}$ completely characterizes the set of location-properties that can potentially get accessed during the reduction of D_i on an heap-store that satisfies the pre-condition.
- The satisfaction judgment for the separating conjunction guarantees that for an assertion $P_i * P_j$, any heap-store H, A , we have $H, A \in \llbracket P_i * P_j \rrbracket_{\mathcal{A}_1}$, IFF: the heap can be split into *disjoint* heaps H_i and H_j such that $H_i, A \in \llbracket P_i \rrbracket_{\mathcal{A}_1}$ and $H_j, A \in \llbracket P_j \rrbracket_{\mathcal{A}_1}$.

⁶ Although this result is proven for a slightly different pointsto relation, we conjecture that the result holds for our assertion language and specification axioms as well

Axioms (SL_1).

$$\begin{array}{c}
\frac{}{\{P[E/x] \wedge \text{present}(x) \wedge \text{Wt}(E)\}x := E\{P\}} \quad \text{[S-ASSIGNMENT]} \\
\frac{\text{y not free in } E}{\{\text{present}(x) \wedge \exists y : E.p \mapsto y\}x = E.p\{\exists y : E[y/x].p \mapsto x\}} \quad \text{[S-LOOKUP-LOCAL]} \\
\frac{\text{y not free in } E}{\{\exists y : (\text{present}(x) \wedge E.p \mapsto y) * ((E.p \mapsto y) \rightarrow P[y/x])\}x = E.p\{P\}} \quad \text{[S-LOOKUP-BACKWARDS]} \\
\frac{\{E_1.p \mapsto _ \wedge \text{Wt}(E_2)\}E_1.p = E_2\{E_1.p \mapsto E_2\} \wedge \text{Wt}(E_2)}{\{(E_1.p \mapsto _ \wedge \text{Wt}(E_2)) * ((E_1.p \mapsto E_2) \rightarrow P)\}E_1.p = E_2\{P\}} \quad \text{[S-UPDATE-LOCAL]} \\
\frac{\{(E_1.p \mapsto _ \wedge \text{Wt}(E_2)) * ((E_1.p \mapsto E_2) \rightarrow P)\}E_1.p = E_2\{P\}}{\text{y not free in any } E_i} \quad \text{[S-UPDATE-BACKWARDS]} \\
\frac{\text{y not free in any } E_i}{\{\text{present}(x) \wedge \text{Wt}(\tilde{E}_i)_{i \in \{1, \dots, n\}} \wedge \text{emp}\}x := \{\tilde{p}_i : \tilde{E}_i\}_{i \in \{1, \dots, n\}}\{\exists y : (x : \{p_i \mapsto E_i[y/x]\}_{i \in \{1, \dots, n\}})\}} \quad \text{[S-CREATION-LOCAL]} \\
\frac{\text{y not free in any } E_i}{\{(\text{present}(x) \wedge \text{Wt}(\tilde{E}_i)_{i \in \{1, \dots, n\}} \wedge \text{emp}) * ((\forall y : (y : \{p_i \mapsto E_i\}_{i \in \{1, \dots, n\}}) \rightarrow P[y/x]))\}x := \{\tilde{p}_i : \tilde{E}_i\}_{i \in \{1, \dots, n\}}\{P\}} \quad \text{[S-CREATION-BACKWARDS]} \\
\frac{}{\{P\} \text{normal} \{P\}} \quad \text{[S-NORMAL]}
\end{array}$$

Rules (SL_1).

$$\begin{array}{c}
\frac{\{P \wedge B\}C_1\{Q\} \quad \{P \wedge \neg B\}C_2\{Q\}}{\{P\} \text{if } B \text{ then } C_1 \text{ else } C_2\{Q\}} \quad \text{[S-IF]} \\
\frac{\{P \wedge B\}C\{P\}}{\{P\} \text{while } B \text{ then } C\{P \wedge \neg B\}} \quad \text{[S-WHILE]} \\
\frac{\{P\}C_1\{P'\} \quad \{P'\}C_2\{Q\}}{\{P\}C_1; C_2\{Q\}} \quad \text{[S-SEQ]} \\
\frac{\models P \implies P' \quad \{P'\}C\{Q'\} \quad \models Q' \implies Q}{\{P\}C\{Q\}} \quad \text{[S-CONSEQ]} \\
\frac{\{P\}C\{Q\} \quad \text{modifies}(C) \cap \text{free}(R) = \emptyset}{\{P * R\}C\{Q * R\}} \quad \text{[S-FRAME]} \\
\frac{\{P\}C\{Q\} \quad x \notin \text{free}(C)}{\{\exists x.P\}C\{Q\}} \quad \text{[S-AUXVARELIMINATION]} \\
\frac{\{P\}C\{Q\} \quad y \notin \text{free}(P) \cup \text{free}(C) \cup \text{free}(Q)}{\{P\}C\{Q\}[y/x]} \quad \text{[S-SUBSTITUTION]}
\end{array}$$

Fig. 4. Inference axioms and rules for SL_1

Based on the above observations, isolation will hold for a mashup if there exist provable specifications (soundness of the logic guarantees semantic-validity of such specifications) $\{P_1\}D_1\{Q_1\}, \dots, \{P_n\}D_n\{Q_n\}$ such that $H_{mash}, A_{mash} \in \llbracket P_1 * \dots * P_n * true \rrbracket_{\mathcal{A}_1}$. Moreover since D_i is just a variable renaming of the command C_i and A_{mash} contains the corresponding variable-renamed version of the store A , it is possible to derive a sound isolation check from the specification of the original commands C_i and heap-store H, A . Our isolation check can be described as follows:

$Isolation(\mathcal{M}(H, A, (C_1, id_1), \dots, (C_n, id_n)))$

1. Deduce partial correctness assertions $\{P_1\}C_1\{Q_1\}, \dots, \{P_n\}C_n\{Q_n\}$ using the inference rules of SL_1 and \mathcal{A}_1 -valid assertions.
2. Show that $H, A \in \llbracket P_1 * \dots * P_n * true \rrbracket_{\mathcal{A}_1}$ holds.

The above check demonstrates how separation logic aids in combining *local analysis* of the components to prove global properties about mashups. Although the check is good for carrying out proofs by hand, we don't know of any decidable procedure for verifying the check. The correctness of the check is guaranteed by the following theorem.

Theorem 2. *Given a heap-store H, A and commands C_1, \dots, C_n from principles id_1, \dots, id_n respectively, the mashup $\mathcal{M}(H, A, (C_1, id_1), \dots, (C_n, id_n))$ is isolated IF: there exist partial correctness assertions $\{P_1\}C_1\{Q_1\}, \dots, \{P_n\}C_n\{Q_n\}$ such that*

$$H, A \in \llbracket P_1 * \dots * P_n * true \rrbracket_{\mathcal{A}_1} \bigwedge \forall i \vdash_{SL_1} \{P_i\}C_i\{Q_i\}$$

A rigorous proof of the above theorem is described in the appendix 8.2

5 Separation logic with permissions

We now present a variant of separation logic which incorporates access permissions on heap locations. This variant is a simplified version of the one present by Bornat et al in [1]. We only consider three kinds of permissions $\{r\}, \{w\}$ and $\{r, w\}$ where the one in [1] a range of fractional permissions [2], which are useful in specifying properties of mutexes, semaphores and other concurrency primitives. We call our logic SL_2 . We now move on to description of the assertion language for this logic.

5.1 Assertions

The only difference between the assertion languages for SL_1 and SL_2 is in the points-to assertion. The new points-to assertion is written as $E_1.p \stackrel{a}{\mapsto} E_2$, where $a \in \{r, w\}$ and is pronounced as: “program has the permission to access property p of location corresponding to expression E_1 . We call the assertion language of separation logic with permissions as \mathcal{A}_2 .”

5.2 Satisfaction of assertions

Given an assertion P from the language \mathcal{A}_2 , we define satisfaction of an assertion at a heap, store with respect to a permission map. A permission map is a partial functions of type $Loc \rightarrow prop \rightarrow \{r, w\}$. Permission maps are required in the satisfaction relation because assertions in \mathcal{A}_2 not only express conditions on the heap and store but also the set of location-properties that will get read or written during the reduction of a command with respect to a heap store satisfying the precondition. Permission maps specify what locations can be read or written (or both). We use $\Sigma_1, \Sigma_2, \dots$ to denote permission maps. We extend the dom function to permission maps. Further given a location-property pair $l, p \in dom(\Sigma)$, we use $\Sigma(l).p$ as a shorthand to denote the permission on property p of location l .

The general form of the satisfaction judgment is $H, A, \Sigma \models_{\mathcal{A}_2} P$. For any assertion P , $H, A, \Sigma \models_{\mathcal{A}_2} P$ holds only if $dom(H) = dom(\Sigma)$. The relation is inductively defined on the structure of the assertion P in figure 5.2. We explain the satisfaction judgment for the points-to relation and separating conjunction here.

- *Points to:* $H, A, \Sigma \models_{\mathcal{A}_1} E_1.p \xrightarrow{a} E_2$ holds IFF: H and Σ are singleton with exactly one location $\llbracket E_1 \rrbracket_{Exp} A$ and one property name p containing $\llbracket E_2 \rrbracket_{Exp} A$ and a respectively.
- *Separating Conjunction:* The separating conjunction $P_1 * P_2$ holds for H, A, Σ IFF: there exist subheaps H_1 and H_2 of H and sub-permission maps Σ_1 and Σ_2 of Σ such that $dom(H_1) = dom(\Sigma_1)$, $dom(H_2) = dom(\Sigma_2)$ and for all l, p in $dom(\Sigma_1) \cap dom(\Sigma_2)$, $\Sigma_1(l).p = \Sigma_2(l).p = \{r\}$. Thus the separating conjunction allows the P_1 and P_2 to overlap on location-properties which are read-only.

Any assertion P is \mathcal{A}_2 – *valid* iff $H, A, \Sigma \models_{\mathcal{A}_2} P$ for all H, A, Σ . Given any assertion P , we define the set $\llbracket P \rrbracket_{\mathcal{A}_2}$ as $\{H, A \mid \exists \Sigma : H, A, \Sigma \models_{\mathcal{A}_2} P\}$.

$H, A, \Sigma \models_{\mathcal{A}_2} P$

$$\begin{array}{ll}
H, A, \Sigma \models_{\mathcal{A}_2} B & \text{iff } \llbracket B \rrbracket_{Bexp} A = true \wedge dom(H) = dom(\Sigma) \\
H, A, \Sigma \models_{\mathcal{A}_2} emp & \text{iff } dom(H) = \emptyset \wedge dom(H) = dom(\Sigma) \\
H, A, \Sigma \models_{\mathcal{A}_2} E_1.p \xrightarrow{a} E_2 & \text{iff } \begin{array}{l} dom(H) = \{\llbracket E_1 \rrbracket_{Exp} A, p\} = dom(\Sigma) \\ H(\llbracket E_1 \rrbracket_{Exp} A).p = \llbracket E_2 \rrbracket_{Exp} A \wedge \\ a = \Sigma(\llbracket E_1 \rrbracket_{Exp} A).p \end{array} \\
H, A, \Sigma \models_{\mathcal{A}_2} P_1 * P_2 & \text{iff } \begin{array}{l} H_1, A, \Sigma_1 \models_{\mathcal{A}_2} P_1 \wedge \\ H_2, A, \Sigma_2 \models_{\mathcal{A}_2} P_2 \wedge \\ \exists H_1, \Sigma_1, H_2, \Sigma_2 : dom(H_1), \Sigma_1 \bowtie dom(H_2), \Sigma_2 \wedge \\ H_1 \cup H_2 = H \wedge \\ \Sigma_1 \cup \Sigma_2 = \Sigma \end{array} \\
H, A, \Sigma \models_{\mathcal{A}_2} P \multimap P_2 & \text{iff } \begin{array}{l} H_1, A, \Sigma_1 \models_{\mathcal{A}_2} P_1 \wedge \\ H_2, A, \Sigma_2 \models_{\mathcal{A}_2} P_2 \wedge \\ \forall H_1, \Sigma_1, H_2, \Sigma_2 : dom(H), \Sigma \bowtie dom(H_1), \Sigma_1 \wedge \\ H_1 \cup H_2 = H \wedge \\ \Sigma \cup \Sigma_1 = \Sigma_2 \end{array} \\
H, A, \Sigma \models_{\mathcal{A}_2} P_1 \implies P_2 & \text{iff } H, A, \Sigma \models_{\mathcal{A}_2} P_1 \implies H, A, \Sigma \models_{\mathcal{A}_2} P_2 \\
H, A, \Sigma \models_{\mathcal{A}_2} \exists x : P & \text{iff } \exists v \in StoreValues : H, A[x \rightarrow v], \Sigma \models_{\mathcal{A}_2} P
\end{array}$$

Notation:

- A heap H and permission map Σ are compatible iff $dom(H) = dom(\Sigma)$.
- For heaps H_1, H_2 and compatible permission maps Σ_1, Σ_2 respectively, $H_1, \Sigma_1 \bowtie H_2, \Sigma_2$ holds iff

$$\forall (l, p) \in dom(H_1) \wedge dom(H_2) : H_1(l).p = H_2(l).p \wedge \Sigma_1(l).p = \Sigma_2(l).p = \{r\}$$

Whenever $H_1, \Sigma_1 \bowtie H_2, \Sigma_2$, the union of the partial functions H_1, H_2 and Σ_1, Σ_2 exists and is denoted by $H_1 \cup H_2$ and $\Sigma_1 \cup \Sigma_2$ respectively.

Fig. 5. Satisfaction of assertions in \mathcal{A}_2

5.3 Specifications and Inference rules

We now present the axioms and inference rules for deducing partial correctness triples. All the rules are listed in figure 5.3. The rules are essentially similar to the ones for SL_1 , with the only difference being that the points-to relation now appropriately mentions a permission set from $\{r, w\}$, depending on whether the corresponding location-property was read or written. As a result the notion of semantic-validity of partial correctness triple also includes the condition that the set of location-properties accessed must be in accordance with the corresponding permission map satisfying the precondition. A formal definition is as follows.

Definition 8 (\models_{SL_2}). A specification $\{P\}C\{Q\}$ is SL_2 -valid, denoted by $\models_{SL_2} \{P\}C\{Q\}$, iff for all heap-stores H, A and permission maps Σ such that $H, A, \Sigma \models_{\mathcal{A}_2} P$, the following holds

1. $\text{Safe}(H, A, C)$
2. For all heap-stores K, B, D such that $H, A, C \rightsquigarrow K, B, D$,
 - a. $K, B \in \llbracket Q \rrbracket_{\mathcal{A}_2}$ if $D = \text{normal}$.
 - b. $\forall l, p, a, v : (l, p, a, v) \in \text{Acc}^{\text{heap}}((H, A, C), (K, B, D)) \wedge l, p \in \text{dom}(H) \implies a \in \Sigma(l).p$

Observe that, there every SL_1 triple can be encoded in SL_2 by replacing all points-to assertions $E_{1.p} \xrightarrow{E}_2$ by $E_{1.p} \xrightarrow{\{r,w\}} E_2$. We don't go into the details here but the encoding is straightforward to derive. The main point is that the SL_2 is more powerful than SL_1 and therefore the isolation verification procedure that we derive using SL_2 , will subsume the procedure derived using SL_1 .

We now formally state the soundness theorem for the logic. For any specification $\{P\}C\{Q\}$, we say that $\vdash_{SL_2} \{P\}C\{Q\}$ holds, iff $\{P\}C\{Q\}$ can be derived by using \mathcal{A}_2 -valid assertions and the axioms and rules of separation logic.

Theorem 3 (Soundness SL_2). $\vdash_{SL_2} \{P\}C\{Q\} \implies \models_{SL_2} \{P\}C\{Q\}$

Axioms (SL_2).

$$\begin{array}{c}
\frac{\{P[E/x] \wedge \text{present}(x) \wedge \text{Wt}(E)\}x := E\{P\}}{y \text{ not free in } E} \quad \text{[S-ASSIGNMENT]} \\
\frac{\{ \text{present}(x) \wedge \exists y : E.p \xrightarrow{r} y \}x = E.p \{ \exists y : E[y/x].p \xrightarrow{r} x \}}{\{ \text{present}(x) \wedge \exists y : E.p \xrightarrow{r} y \} * ((E.p \xrightarrow{r} y) \multimap P[y/x])\}x = E.p\{P\}} \quad \text{[S-LOOKUP-LOCAL]} \\
\frac{y, z \text{ not free in } E}{\{ \exists y : (\text{present}(x) \wedge E.p \xrightarrow{r} y) * ((E.p \xrightarrow{r} y) \multimap P[y/x])\}x = E.p\{P\}} \quad \text{[S-LOOKUP-GENERAL]} \\
\frac{\{E_{1.p} \xrightarrow{w} _ \wedge \text{Wt}(E_2)\}E_{1.p} = E_2\{E_{1.p} \xrightarrow{w} E_2\} \wedge \text{Wt}(E_2)}{\{ (E_{1.p} \xrightarrow{w} _ \wedge \text{Wt}(E_2)) * ((E_{1.p} \xrightarrow{w} E_2) \multimap P) \}E_{1.p} = E_2\{P\}} \quad \text{[S-UPDATE-LOCAL]} \\
\frac{\{ (E_{1.p} \xrightarrow{w} _ \wedge \text{Wt}(E_2)) * ((E_{1.p} \xrightarrow{w} E_2) \multimap P) \}E_{1.p} = E_2\{P\}}{y \text{ not free in any } E_i} \quad \text{[S-UPDATE-GENERAL]} \\
\frac{\{ \text{present}(x) \wedge \text{Wt}(\tilde{E}_i)_{i \in \{1, \dots, n\}} \wedge \text{emp} \}x := \{ \tilde{p}_i : \tilde{E}_i \}_{i \in \{1, \dots, n\}} \{ \exists y : (x : \{ p_i \xrightarrow{r,w} E_i[y/x] \}_{i \in \{1, \dots, n\}}) \}}{y \text{ not free in any } E_i} \quad \text{[S-CREATION-LOCAL]} \\
\frac{\{ \text{present}(x) \wedge \text{Wt}(\tilde{E}_i)_{i \in \{1, \dots, n\}} \wedge \text{emp} \} * (\forall y : (y : \{ p_i \xrightarrow{r,w} E_i \}_{i \in \{1, \dots, n\}}) \multimap P[y/x])\}x := \{ \tilde{p}_i : \tilde{E}_i \}_{i \in \{1, \dots, n\}} \{P\}}{\{P\} \text{normal}\{P\}} \quad \text{[S-CREATION-GENERAL]} \\
\quad \quad \quad \text{[S-NORMAL]}
\end{array}$$

Rules (SL_2).

$$\begin{array}{c}
\frac{\{P \wedge B\}C_1\{Q\} \quad \{P \wedge \neg B\}C_2\{Q\}}{\{P\} \text{if } B \text{ then } C_1 \text{ else } C_2\{Q\}} \quad \text{[S-IF]} \\
\frac{\{P \wedge B\}C\{P\}}{\{P\} \text{while } B \text{ then } C\{P \wedge \neg B\}} \quad \text{[S-WHILE]} \\
\frac{\{P\}C_1\{P'\} \quad \{P'\}C_2\{Q\}}{\{P\}C_1; C_2\{Q\}} \quad \text{[S-SEQ]} \\
\frac{\models P \implies P' \quad \{P'\}C_1\{Q'\} \quad \models Q' \implies Q}{\{P\}C_1\{Q\}} \quad \text{[S-CONSEQ]} \\
\frac{\{P\}C\{Q\} \quad \text{modifies}(C) \cap \text{free}(R) = \emptyset}{\{P * R\}C\{Q * R\}} \quad \text{[S-FRAME]} \\
\frac{\{P\}C\{Q\} \quad x \notin \text{free}(C)}{\{\exists x.P\}C\{Q\}} \quad \text{[S-AUXVARELIMINATION]}
\end{array}$$

Fig. 6. Inference axioms and rules for SL_2

5.4 Solving the Mashup Isolation Problem

We now present a solution to the mashup isolation problem using SL_1 . Consider a heap-store H, A and components $(C_1, id_1), \dots, (C_n, id_n)$. Let $\mathcal{M}(H, A, (C_1, id_1), \dots, (C_n, id_n)) = H_{\text{mash}}, A_{\text{mash}}, D_1; \dots; D_n$. In

this section we present a sufficient condition for case B of the analysis present in section 3.1. In accordance with case B , a sufficient check for isolation would be to show that the common location-property names accessed during the reductions of any D_i and D_j starting from the heap store H_{mash}, A_{mash} are only read and are never written to. From the semantic validity of specifications in SL_2 and the satisfaction judgement for \mathcal{A}_2 , it follows that this check goes through if we can find specifications for D_i and D_j whose pre-conditions can be “star-ed” together. Since D_i is just a renaming of command C_i , this condition can be translated to an equivalent condition on the specification of C_i .

$Isolation(\mathcal{M}(H, A, (C_1, id_1), \dots, (C_n, id_n)))$

1. Deduce partial correctness assertions $\{P_1\}C_1\{Q_1\}, \dots, \{P_n\}C_n\{Q_n\}$ using the inference rules of SL_2 and \mathcal{A}_2 -valid assertions.
2. Show that $H, A \in \llbracket P_1 * \dots * P_n * true \rrbracket_{\mathcal{A}_1}$ holds.

Notice the similarity between the above isolation check and the one derived using SL_1 . All conditions about sharing read-only locations are cleverly embedded by the separating conjunction. However, since SL_2 is stronger than SL_1 , the above procedure can verify more mashups than the one derived using SL_1 . The correctness of the above check is guaranteed by the following theorem.

Theorem 4. *Given a heap-store H, A and commands C_1, \dots, C_n from principles id_1, \dots, id_n respectively, the mashup $\mathcal{M}(H, A, (C_1, id_1), \dots, (C_n, id_n))$ is isolated IF: there exist partial correctness assertions $\{P_1\}C_1\{Q_1\}, \dots, \{P_n\}C_n\{Q_n\}$ such that*

$$H, A \in \llbracket P_1 * \dots * P_n * true \rrbracket_{\mathcal{A}_2} \bigwedge \forall i \vdash_{SL_2} \{P_i\}C_i\{Q_i\}$$

A rigorous proof of the above theorem is described in the appendix 8.2.

6 Handling Case C - write and restore

We will now analyze Case C of the analysis of the isolation problem described in section ???. Consider a heap-store H, A and components $(C_1, id_1), \dots, (C_n, id_n)$. Let $\mathcal{M}(H, A, (C_1, id_1), \dots, (C_n, id_n)) = H_{mash}, A_{mash}, D_1; \dots; D_n$. A sufficient check for case C , is that for any programs D_i and D_j , all common locations-properties that get written-to during the execution of these programs starting from H_{mash}, A_{mash} must be restored back to their original value. Of course, the procedures derived using SL_1 and SL_2 are sufficient conditions for this check, however they are both very restrictive and do not allow writing to common heap location-properties. We would like to derive a check which would allow common locations to be written to, provided their original value is restored in the end. This is not straightforward because the “write and restore” property requires tracking the entire trace rather than a simple restriction on the pre-condition like in the case of read-only and write-only properties. We now present a naive check that we derive for this property.

Our check is based on the following observation: For any component D_i , which write restores back the values on a certain portion of the heap, there exists a valid specification of the form $\{P * R\}C\{Q * R\}$, where R is an assertion on the portion of the heap that is written to and restored back. Note that this specification is not derived by applying the frame rule on $\{P\}C\{Q\}$. This is because if the portion of the heap described by R is written to then $\{P\}C\{Q\}$ will not be a semantically-valid specification (run-time safety breaks). In order to establish that the exact same values are restored back on the heap described by assertion R , we force assertion R to an *exact* assertion. These are formally defined as follows.

Definition 9 (Exact Assertions). *An assertion R is SL_1 -exact iff for all stores A and heaps H_1, H_2 if $H_1, A \in \llbracket R \rrbracket_{\mathcal{A}_1}$ and $H_2, A \in \llbracket R \rrbracket_{\mathcal{A}_1}$ then $H_1 = H_2$*

Thus we derive the following procedure

$Isolation(\mathcal{M}(H, A, (C_1, id_1), \dots, (C_n, id_n)))$

1. Deduce partial correctness assertions of the form $\{P_1 * R\}C_1\{Q_1 * R\}, \dots, \{P_n * R\}C_n\{Q_n * R\}$ using the inference rules of SL_1 and \mathcal{A}_1 -valid assertions.

2. Show that predicate R is exact.
3. Show that $H, A \in \llbracket P_1 * \dots * P_n * R * true \rrbracket_{\mathcal{A}_1}$ holds.

Although the above check subsumes the previous check derived using SL_1 (take R as the *emp*), it is not clear whether it will be more powerful in practice. This is because it is not clear if peculiar assertions of the form $\{P_i * R\}C_i\{Q_i * R\}$ can be easily derived from the inference system of SL_2 , which is at best Recursively Enumerable even if we work in a decidable fragment of \mathcal{A}_2 . The correctness of the above check is guaranteed by the following theorem.

Theorem 5. *Given a heap-store H, A and commands C_1, \dots, C_n from principles id_1, \dots, id_n respectively, the mashup $\mathcal{M}(H, A, (C_1, id_1), \dots, (C_n, id_n))$ is isolated IF: there exist partial correctness assertions $\{P_1 * R\}C_1\{Q_1 * R\}, \dots, \{P_n * R\}C_n\{Q_n\}$ such that the assertion R is exact and*

$$H, A \in \llbracket P_1 * \dots * P_n * R * true \rrbracket_{\mathcal{A}_1} \bigwedge \forall i \vdash_{SL_1} \{P_i * R\}C_i\{Q_i * R\}$$

7 Conclusion and Future Work

Conclusions. This paper is a preliminary draft of ongoing work on solving the isolation problem for mashups using techniques from separation logic. In this work, we rigorously formulated the inter-component isolation property for a restricted class of mashups, known as “variable-separated mashup”. We considered the problem of verifying isolation for variable separated mashups and studied separation logic to solve this problem. We derive two procedures (the second one relatively more complete than the first) for verifying isolation based on the two different variants of separation logic and rigorously proved the correctness of these procedures.

Ongoing and Future Work. As described in the previous section, we still do not have a neat procedure for handling case C of the isolation problem. We are currently working on techniques for checking “write and restore” properties by using some of the information hiding ideas from [14].

Another direction that we are currently working on is understanding the connection between the procedure mentioned in this work with the one in [9]. The procedure in [9], can be informally stated as “Authority-Isolation \implies Isolation”. Authority of a program with respect to a heap-store is defined as an over-approximation of the set of heap actions performed during its reduction starting from that heap-store. Authority isolation means that the authority sets of any two components are “non-influencing”, meaning that there is no location-property to which one component writes to and the other reads from. At a high level, there seems to be a close connection between our procedure and the one using authorities. This is because any provable separation logic specification $\{P\}C\{Q\}$ for a program C is such that the precondition P can completely describe the authority of the program for those heap-stores that satisfy the precondition. The second condition in our procedure - $H, A \in \llbracket P_1 * \dots * P_n * true \rrbracket_{\mathcal{A}_1}$ can then be seen as a “non-influence” check. Thus it seems like our procedure is no stronger than the one using authorities. However one key difference between the two procedures is in the way authorities are computed and the way specifications are computed. Authorities are usually computed as worst case reachable portions of the heap. Specifications on the other hand are more precise and correspond more to the actual portion of the heap that is accessed. In future, we would like to precisely understand the relation between the two procedures.

References

1. Richard Bornat, Cristiano Calcagno, Peter O’Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *POPL ’05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 259–270, New York, NY, USA, 2005. ACM.
2. John Boyland. Checking interference with fractional permissions. In *SAS*, pages 55–72, 2003.
3. R. M. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972.
4. D. Crockford. ADsafe: Making JavaScript safe for advertising. <http://www.adsafe.org/>, 2008.
5. OpenSocial Foundation. OpenSocial. <http://www.opensocial.org/>.

6. Google. iGoogle. <http://www.google.com/ig>.
7. C. A. R. Hoare. An axiomatic basis for computer programming. *COMMUNICATIONS OF THE ACM*, 12(10):576–580, 1969.
8. Samin Ishtiaq and Peter W. O’Hearn. Bi as an assertion language for mutable data structures. In *IN POPL*, pages 14–26, 2001.
9. S. Maffeis, J. C. Mitchell, and A. Taly. Object capabilities and isolation of web applications. IEEE, 2010. To appear in Proceedings of 31st IEEE Symposium on Research in Security and Privacy.
10. S. Maffeis, J.C. Mitchell, and A. Taly. An operational semantics for JavaScript. In *Proc. of APLAS’08*, volume 5356 of *LNCS*, pages 307–325. Springer Verlag, 2008.
11. S. Maffeis, J.C. Mitchell, and A. Taly. Isolating JavaScript with filters, rewriting, and wrappers. In *Proc. of ESORICS’09*. Springer Verlag, 2009. See also: Dep. of Computing, Imperial College London, Technical Report DTR09-6.
12. John C. Mitchell. *Foundations for Programming Languages*. MIT Press, Cambridge, MA, USA, 1996.
13. Peter O’Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures, 2001.
14. Peter W. O’Hearn, Hongseok Yang, John C. Reynolds, Peter W. O’Hearn, and Queen Mary. Separation and information hiding, 2004.
15. Matthew Parkinson, Richard Bornat, and Cristiano Calcagno. Variables as resource in hoare logics. In *IN LICS06: LOGIC IN COMPUTER SCIENCE*, pages 137–146. IEEE Press, 2006.
16. Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.
17. John Reynolds. Separation logic: A logic for shared mutable data structures. pages 55–74. IEEE Computer Society, 2002.
18. John C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millennial Perspectives in Computer Science*, pages 303–321. Palgrave, 2000.
19. The Facebook Team. Facebook. <http://www.facebook.com/>.
20. The Facebook Team. FBJS. <http://wiki.developers.facebook.com/index.php/FBJS>.
21. Yahoo! Inc. Yahoo! Application Platform. <http://developer.yahoo.com/yap/>.

8 Appendix

8.1 Operational Semantics of \mathcal{L}

Semantics of Expressions.

Semantics of Basic Expressions $\llbracket E \in Exp \rrbracket_{Exp} : Stores \rightarrow StoreValues \cup \{Error\}$

$$\begin{aligned} \llbracket x \rrbracket_{Exp} A &:= \left\{ \begin{array}{ll} A(x) & \text{iff } \in \text{dom}(A) \\ Error & \text{iff } \text{otherwise} \end{array} \right\} \\ \llbracket E_1 op E_2 \rrbracket_{Exp} A &:= \left\{ \begin{array}{ll} \llbracket E_1 \rrbracket_{Exp} A op \llbracket E_2 \rrbracket_{Exp} A & \text{iff } \llbracket E_1 \rrbracket_{Exp} A \in Nat \wedge \llbracket E_2 \rrbracket_{Exp} A \in Nat \\ Error & \text{iff } \text{otherwise} \end{array} \right\} \\ \llbracket n \rrbracket_{Exp} A &:= n \end{aligned}$$

where $op = \{+, *\}$.

Semantics of Boolean Expressions $\llbracket B \in Bexp \rrbracket_{Bexp} : Stores \rightarrow Bool \cup \{Error\}$

$$\begin{aligned} \llbracket isLoc?(E) \rrbracket_{Bexp} A &:= A(x) \in Loc \\ \llbracket E_1 = E_2 \rrbracket_{Bexp} A &:= \llbracket E_1 \rrbracket_{Exp} A = \llbracket E_2 \rrbracket_{Exp} A \\ \llbracket B_1 \implies B_2 \rrbracket_{Bexp} A &:= \left\{ \begin{array}{ll} \llbracket B_1 \rrbracket_{Bexp} A \implies \llbracket B_2 \rrbracket_{Bexp} A & \text{iff } \llbracket E_1 \rrbracket_{Exp} A \in Nat \wedge \llbracket E_2 \rrbracket_{Exp} A \in Nat \\ Error & \text{iff } \text{otherwise} \end{array} \right\} \\ \llbracket true \rrbracket_{Bexp} A &:= true \\ \llbracket false \rrbracket_{Bexp} A &:= false \end{aligned}$$

Semantics of Commands.

States: $Heaps \times Stores \times Comm \cup \{abort, normal\}$

Reduction rules: $States \rightarrow States$

$$\begin{aligned} &\frac{v \neq Error}{H, A, x := v \longrightarrow H, [A|x : \llbracket E \rrbracket_{Exp} A], normal} \quad [C-ASSIGNNORMAL] \\ &\frac{v = Error}{H, A, x := v \longrightarrow H, A, abort} \quad [C-ASSIGNABORT] \\ &\frac{v \in \text{dom}(H) \text{ AND } x \in \text{dom}(A)}{H, A, x := v.p \longrightarrow H, A[x \rightarrow H(v.p)], normal} \quad [C-LOOKUPNORMAL] \\ &\frac{v \notin \text{dom}(H) \text{ OR } x \notin \text{dom}(A)}{H, A, x := v.p \longrightarrow H, A, abort} \quad [C-LOOKUPABORT] \\ &\frac{v_1 \in \text{dom}(H) \text{ AND } v_2 \neq Error}{H, A, v_1 := v_2 \longrightarrow H[v_1.p \rightarrow v_2], A, normal} \quad [C-UPDATENORMAL] \\ &\frac{v_1 \notin \text{dom}(H) \text{ OR } v_2 = Error}{H, A, v_1.p := v_2 \longrightarrow H, A, abort} \quad [C-UPDATEABORT] \end{aligned}$$

$\frac{l \in \text{Loc AND } \neg \exists p : (l, p) \in \text{dom}(H) \quad \text{forall } i \in \{1, \dots, n\}, v_i \neq \text{Error}}{H, A, x := \{\tilde{p}_i : \tilde{v}_i\}_{i \in \{1, \dots, n\}} \longrightarrow H[l \rightarrow \{\tilde{p}_i : \tilde{v}_i\}_{i \in \{1, \dots, n\}}], A[x \rightarrow l], \text{normal}}$	[C-OBJECTCREATIONNORMAL]
$\frac{\text{Exists } i \in \{1, \dots, n\}, v_i = \text{Error}}{H, A, x := \{\tilde{p}_i : \tilde{v}_i\}_{i \in \{1, \dots, n\}} \longrightarrow H, A, \text{abort}}$	[C-OBJECTCREATIONABORT]
$H, A, \text{if true then } C_1 \text{ else } C_2 \longrightarrow H, A, C_1$	[C-IFTRUE]
$H, A, \text{if false then } C_1 \text{ else } C_2 \longrightarrow H, A, C_2$	[C-IFFALSE]
$H, A, \text{if Error then } C_1 \text{ else } C_2 \longrightarrow H, A, \text{abort}$	[C-IFABORT]
$H, A, \text{while } B \text{ then } C \longrightarrow H, A, \text{if } B \text{ then } C_1; \text{while } B \text{ then } C_1 \text{ else normal}$	[C-WHILE]
$H, A, \text{normal}; C_2 \longrightarrow K, B, C_2$	[C-SEQUENCENORMAL]
$H, A, \text{abort}; C_2 \longrightarrow K, B, \text{abort}$	[C-SEQUENCEABORT]
$H, A, \text{abort} \not\longrightarrow$	[C-ABORT]
$H, A, \text{normal} \not\longrightarrow$	[C-NORMAL]
$\frac{\llbracket E \rrbracket_{Exp} A = v}{H, A, ECon[E] \longrightarrow H, A, v}$	[E-CONTEXT]
$\frac{\llbracket B \rrbracket_{BExp} A = b}{H, A, BCon[B] \longrightarrow H, A, b}$	[B-CONTEXT]
$\frac{H, A, C \longrightarrow K, B, D}{H, A, CCon[C] \longrightarrow K, B, CCon[D]}$	[C-CONTEXT]

where,

$$ECon := x := _ | x := _ . p | _ . p = E | v . p = _ | x := \{(\tilde{p}_i : \tilde{v}_i)_{i \in \{1, \dots, n\}}, p_k : _, (\tilde{p}_i : \tilde{v}_i)_{i \in \{k+1, \dots, n\}}\}$$

$$BCon := \text{while } _ \text{ then } C | \text{if_then } C_1 \text{ else } C_2$$

$$CCon := _ ; C$$

Free and Modifies sets.

Commands C	free(C)	modifies(C)
$x := E$	$\{x\} \cup \text{free}(E)$	$\{x\}$
$x := E.p$	$\{x\} \cup \text{free}(E)$	$\{x\}$
$E_1.p = E_2$	$\text{free}(E_1) \cup \text{free}(E_2)$	\emptyset
$x := \{\tilde{p}_i : \tilde{E}_i\}_{i \in \{1, \dots, n\}}$	$\{x\} \cup \text{free}(E_1) \cup \dots \cup \text{free}(E_n)$	$\{x\}$
$\text{if } B \text{ then } C_1 \text{ else } C_2$	$\text{free}(C_1) \cup \text{free}(C_2) \cup \text{free}(B)$	$\text{modifies}(C_1) \cup \text{modifies}(C_2)$
$\text{while } B \text{ then } C$	$\text{free}(C) \cup \text{free}(B)$	$\text{modifies}(C)$
$C_1; C_2$	$\text{free}(C_1) \cup \text{free}(C_2)$	$\text{modifies}(C_1) \cup \text{modifies}(C_2)$
normal	\emptyset	\emptyset
abort	\emptyset	\emptyset

8.2 Proofs of theorems 2, 4

In order to prove theorem 2, we state and prove a few supporting lemmas and propositions that will be used in the proof. We use the following shorthands in the proof:

- For states S_1, S_2 , $S_1 <: S_2$ is equivalent to $\mathcal{H}(S_1) \subseteq \mathcal{H}(S_2) \wedge \mathcal{S}(S_1) <: \mathcal{S}(S_2) \wedge \mathcal{T}(S_1) = \mathcal{T}(S_2)$.
- For heaps H_1, H_2 such that $H_1 \subseteq H_2$, we use $H_2 - H_1$ to denote the partial function obtained by setting all $l, p \in \text{dom}(H_1)$ to undefined in the partial function of H_2 .

- For heaps H_1, H_2 , we say that H_1 and H_2 are compatible iff $\forall l, p \in \text{dom}(H_1) \cap \text{dom}(H_2) : H_1(l).p = H_2(l).p$. We use $H_1 \diamond H_2$ to denote compatibility of two heaps. For any store S_1, S_2 , $S_1 \diamond S_2$ is similarly defined.
- For an assertion P and string a , we define $Rn(P, a)$ as the assertion obtained by replacing every variable x in P with $a : x$ (assuming that $\forall x : x \in \text{Vars} \implies a : x \in \text{Vars}$).
- Given a state S , $Rn(S, a)$ is the state $(\mathcal{H}(S), Rn(\mathcal{S}(S), a), Rn(\mathcal{C}(S), a))$.

Before stating the lemmas, for any state T , we define a relation $R(T)$ between program states.

$$R(T) := \left\{ (S_1, S_2) \mid \left(T <: S_1 \wedge T <: S_2 \wedge \begin{array}{l} \text{dom}(\mathcal{H}(S_1)) <: \text{dom}(\mathcal{H}(S_2)) \end{array} \right) \right\}$$

Lemma 1. *Given a heap-store H, A and a mashup*

$$\mathcal{M}(H, A, (C_1, id_1), \dots, (C_n, id_n)) = (H_{mash}, A_{mash}, D_1; \dots; D_n),$$

for all traces $\tau \in \text{Traces}(H_{mash}, A_{mash}, D_1; \dots; D_n)$ we have, for all $i \geq 1$, if $\text{States}(\tau, id_{i+1}) \neq \emptyset$ then one of the following holds

1. $\mathcal{HS}(\text{First}(\text{States}(\tau, id_i))), C_i \rightsquigarrow \mathcal{HS}(\text{First}(\text{States}(\tau, id_{i+1}))), \text{normal}$
2. $\mathcal{HS}(\text{First}(\text{States}(\tau, id_i))), C_i \uparrow \wedge \forall j > i : \text{States}(\tau, id_{j+1}) = \emptyset$

Proof. Follows the semantics of sequential composition and command-context rules. \square

Lemma 2. *For all strings a such that $\forall x : x \in \text{Vars} \implies a : x \in \text{Vars}$ and for all expression E , $\llbracket E \rrbracket_{Exp} A = \llbracket Rn(E, a) \rrbracket_{Exp} Rn(A, a)$.*

Proof. By Induction on the structure of E . \square

Lemma 3. *For all assertions P and heap stores H, A , if $H, A \in \llbracket P \rrbracket_{A_1}$ then the following holds*

1. *For all stores B such that $\text{Proj}(A, \text{free}(P)) \subseteq B$, $H, B \in \llbracket P \rrbracket_{A_1}$.*
2. *For all strings a such that $\forall x : x \in \text{Vars} \implies a : x \in \text{Vars}$, $H, Rn(A, a) \in \llbracket Rn(P, a) \rrbracket_{A_1}$*

Proof. Both parts can be proven by an induction on the structure of P . In the proof of part 2, the base case when P is a boolean expression will make use of lemma 2. \square

Lemma 4. *For all states S, T if $S \rightsquigarrow T$ then $\text{dom}(\mathcal{H}(S)) \subseteq \text{dom}(\mathcal{H}(T))$ and $\text{dom}(\mathcal{S}(S)) = \text{dom}(\mathcal{S}(T))$*

Proof. By induction on the set of reduction rules. \square

Lemma 5. *For all states S and all strings a such that $\forall x : x \in \text{Vars} \implies a : x \in \text{Vars}$, the following holds*

$$S \sim Rn(S, a)$$

Proof. We prove by an induction over the set of reduction rules that: for all rules $\frac{\langle \text{premise} \rangle}{S \longrightarrow T}$, we have

1. If the premise of the rule holds for S then it holds for $Rn(S, a)$ as well.
2. If under the reduction rule $Rn(S, a) \longrightarrow T'$ then there exists T such that $S \longrightarrow T$ under the same rule and $T' = Rn(a)$ and $\text{Acc}(Rn(S, a), T') = \text{Acc}(S, T)$.

Let this property be called property A . We now show that safe monotonicity, termination monotonicity and the access properties hold.

- (Access Property) From the property A it follows that for all traces of $Rn(S, a)$, there is a trace of S which has the same action sequence. Hence the access property holds.
- (Safe Monotonicity) Suppose $\text{Safe}(S)$ holds then for all $S', S \rightsquigarrow S' \implies \mathcal{C}(S') \neq \text{abort}$. Now for any state S , if $\mathcal{C}(Rn(S, a)) = \text{abort}$ then $\mathcal{C}(S) = \text{abort}$. Therefore if $\text{Safe}(Rn(S, a))$ does not hold then there exists S'' such that $Rn(S, a) \rightsquigarrow S'' \wedge \mathcal{C}(S'') = \text{abort}$. From property A , it follows that there exist S' such that $S \rightsquigarrow S' \wedge \mathcal{C}(S') = \text{abort}$. This is contradiction. Therefore $\text{Safe}(Rn(S, a))$ holds and therefore safe monotonicity holds.

- (Termination Monotonicity) From the above property it follows that if there is a non terminating trace of $Rn(S, a)$ then there must a non terminating trace of S . Hence if all traces of S terminate normally then it must be the case that all traces of $Rn(S, a)$ terminate normally (safe monotonicity guarantees normal termination). Therefore termination monotonicity holds.

□

Lemma 6. *For all states T such that $Safe(T)$ holds,*

$$\forall S_1, S_2 \in R(T) \implies S_1 \sim S_2$$

Proof. Consider any state pair $(S_1, S_2) \in R(T)$. We have the following facts

$$dom(\mathcal{H}(S_1)) \subseteq dom(\mathcal{H}(S_2)) \quad (1)$$

$$T <: S_1 \quad (2)$$

$$T <: S_2 \quad (3)$$

We now prove the Safe Monotonicity, Termination Monotonicity and Access properties to show that $S_1 \sim S_2$.

- *Safe Monotonicity.* Since $Safe(T)$ holds, by conditions 2 and 3, we have that $Safe(S_1)$ and $Safe(S_2)$ hold. Therefore safe monotonicity holds.
- *Termination Monotonicity.* Since $T <: S_1$ holds and $Safe(T_1)$ holds, by the frame property of proposition 1, we have that if $\exists S'_1 : S_1 \rightsquigarrow S'_1$ then $\exists T' : T \rightsquigarrow T'$. In other words, $\neg S_1 \uparrow \implies \neg T \uparrow$. Since $T <: S_2$, by proposition 1 we get that $\neg T \uparrow \implies \neg S_2 \uparrow$. This gives us $\neg S_1 \uparrow \implies \neg S_2 \uparrow$. Therefore termination monotonicity holds.
- *Access Property* By induction over the set of reduction rules. We define the relation R_0 on states such that $(S_1, S_2) \in R_0$ iff $\exists T : SafeT \wedge (S_1, S_2) \in R(T)$. For each rule $\frac{\langle \text{premise} \rangle}{S \longrightarrow T}$, we show that for all S_2 such that $(S_1, S_2) \in R_0$,
 1. If the premise of the rule holds for S_1 then it holds for S_2 as well.
 2. If $S_2 \longrightarrow T_2$ then there exists T_1 such that $S_1 \longrightarrow T_1$ and $Acc(S_1, T_1) = Acc(S_2, T_2)$

The transition axioms and expression context rules form the base cases and then command context rule forms the inductive case.

□

Lemma 7. *Consider states T_1 and T_2 such that*

$Safe(T_1), Safe(T_2), dom(\mathcal{S}(T_1)) \cap dom(\mathcal{S}(T_2)) = \emptyset, \mathcal{H}(T_1) \diamond \mathcal{H}(T_2)$, and

$\forall K, B : T_1 \rightsquigarrow K, B, normal \implies \forall l, p : (l, p) \in dom(\mathcal{H}(T_2)) \cap dom(\mathcal{H}(T_1)) \implies K(l).p = \mathcal{H}(T_2)(l).p$ holds.

For any heap-store pairs H, A and H_1, A_1 such that $((H, A, \mathcal{C}(T_1)), (H_1, A_1, \mathcal{C}(T_1))) \in R(T_1)$ and

$((H, A, \mathcal{C}(T_2)), (H_2, A_2, \mathcal{C}(T_2))) \in R(T_2)$, the following is true:

$$\forall H_2, A_2 : H_1, A_1, \mathcal{C}(T_1) \rightsquigarrow H_2, A_2, normal \implies ((H, A, \mathcal{C}(T_2)), (H_2, A_2, \mathcal{C}(T_2))) \in R(T_2)$$

Proof. Since $((H, A, \mathcal{C}(T_1)), (H_1, A_1, \mathcal{C}(T_1))) \in R(T_1)$ and $((H, A, \mathcal{C}(T_2)), (H_2, A_2, \mathcal{C}(T_2))) \in R(T_2)$, the following holds:

$$dom(H) <: dom(H_1) \quad (4)$$

$$T_1 <: (H, A, \mathcal{C}(T_1)) \quad (5)$$

$$T_1 <: (H_1, A_1, \mathcal{C}(T_1)) \quad (6)$$

$$T_2 <: (H, A, \mathcal{C}(T_2)) \quad (7)$$

$$T_2 <: (H_1, A_1, \mathcal{C}(T_2)) \quad (8)$$

$$(9)$$

Consider any H_2, A_2 such that $H_1, A_1, \mathcal{C}(T_1) \rightsquigarrow H_2, A_2, normal$. Using condition 6, the fact that $Safe(T)$ holds and the frame property from proposition 1 we have that there exists K, B such that $T_1 \rightsquigarrow K, B, normal$ and

- A. $Acc(T_1, (K, B, normal)) = Acc((H_1, A_1, \mathcal{C}(T_1)), (H_2, A_2, normal))$.
- B. $K \subseteq H_2$
- C. $H_1 - \mathcal{H}(T_1) = H_2 - K$.
- D. $B \subseteq A_2$
- E. $A_1 - \mathcal{S}(T_1) = A_2 - B$.

From the premise of the theorem an condition B we have:

$$\forall (l, p) \in dom(\mathcal{H}(T_2)) \cap dom(\mathcal{H}(T_1)) : H_2(l).p = \mathcal{H}(T_2)(l).p \quad (10)$$

From condition 8, we have

$$\forall (l, p) \in dom(\mathcal{H}(T_2)) \cap (dom(H_1) \setminus dom(T_1)) : H_1(l).p = \mathcal{H}(T_2)(l).p \quad (11)$$

From condition 7 and C , we have:

$$\forall (l, p) \in dom(\mathcal{H}(T_2)) \cap (dom(H_1) \setminus dom(T_1)) : H_2(l).p = \mathcal{H}(T_2)(l).p \quad (12)$$

Combining conditions 10, 12 and 7, we have

$$\forall (l, p) \in dom(\mathcal{H}(T_2)) : H_2(l).p = \mathcal{H}(T_2)(l).p.$$

This means that

$$\mathcal{H}(T_2) \subseteq H_2 \quad (13)$$

From condition E , we have $\forall x \in dom(A_1) \setminus dom(\mathcal{S}(T_1)) : A_2(x) = A_1(x)$. Since $dom(\mathcal{S}(T_1)) \cap dom(\mathcal{S}(T_2)) = \emptyset$, from condition 8, we have

$$\mathcal{S}(T_2) \subseteq A_2 \quad (14)$$

From lemma 4, we get that $dom(H_1) \subseteq dom(H_2)$. Combining this with condition 4, we get

$$dom(H) \subseteq dom(H_2) \quad (15)$$

From conditions 13, 15, 14 and 7, it follows that $((H, A, \mathcal{C}(T_2)), (H_2, A_2, \mathcal{C}(T_2))) \in R(T_2)$.

Since we started with an arbitrary H_2, A_2 such that $H_1, A_1, \mathcal{C}(T_1) \rightsquigarrow H_2, A_2, normal$, we have

$$\forall H_2, A_2 : H_1, A_1, \mathcal{C}(T_1) \rightsquigarrow H_2, A_2, normal \implies ((H, A, \mathcal{C}(T_2)), (H_2, A_2, \mathcal{C}(T_2))) \in R(T_2)$$

□

Proof of theorem 2. Let $\mathcal{M}(H, A, (C_1, id_1), \dots, (C_n, id_n)) = (H_{mash}, A_{mash}, D_1; \dots; D_n)$. By definition of \mathcal{M} ,

For each i , $D_i = Rn(C_i, id_i)$.

$H_{mash} = H$

$A_{mash} := Rn(A_1, id_1) \dots Rn(A_n, id_n)$

Consider any trace $\tau \in Traces(H_{mash}, A_{mash}, C_{mash})$. Consider the sequence $States(\tau, id_1), \dots, States(\tau, id_n)$ and let k ($k \in \{1, \dots, n-1\}$) be the smallest index such that $States(\tau, id_{k+1}) = \emptyset$. If no such k exists, then let $k = n$. By definition of $States$,

$$(\forall 1 \leq i \leq k : States(\tau, id_i) \neq \emptyset) \wedge (\forall k+1 \leq i \leq n : States(\tau, id_i) = \emptyset) \quad (16)$$

For any $1 \leq i \leq k$, let $H_i, A_i, D_i = First(States(\tau, id_i))$.

Since for each i , $\vdash_{SL_1} \{P_i\}C_i\{Q_i\}$ holds, By repeated applications of the substitution rule, we have that $\vdash_{SL_1} \{Rn(P_i, id_i)\}D_i\{Rn(Q_i, id_i)\}$ holds.

From lemma 3 and definitions of H_{mash} and A_{mash} , we can show that since $H, A \models_{\mathcal{A}_1} P_1 * \dots * P_n * true$, we have

$H_{mash}, A_{mash} \models_{\mathcal{A}_1} (Rn(P_1, id_1)) * \dots * (Rn(P_n, id_n)) * true$
Thus for $i \in \{1, \dots, n\}$, there exist heap-stores H_{P_i}, A_{P_i} such that

$$\forall i, j, i \neq j : H_{P_i} \# H_{P_j} \quad (17)$$

$$\forall i : H_{P_i} \subseteq H_{mash} \quad (18)$$

$$\forall i : A_{P_i} = Proj(A_{mash}, free(Rn(P_i, id_i))) \quad (19)$$

$$\forall i : H_{P_i}, A_{P_i} \models_{\mathcal{A}_1} Rn(P_i, id_i) \quad (20)$$

For each i , we define the states $T_{P_i} = H_{P_i}, A_{P_i}, D_i$. Since for each i , $\vdash_{SL_1} \{Rn(P_i, id_i)\} D_i \{Rn(Q_i, id_i)\}$ holds, by the soundness theorem we have that for each i , $\models_{SL_1} \{Rn(P_i, id_i)\} D_i \{Rn(Q_i, id_i)\}$ holds. Therefore we have,

$$\forall i, 1 \leq i \leq k : Safe(T_{P_i}) \quad (21)$$

$$\forall i, j, i \neq j : dom(\mathcal{S}(T_{P_i})) \cap dom(\mathcal{S}(T_{P_j})) = \emptyset \quad (22)$$

$$\forall i, j, i \neq j : \mathcal{H}(T_{P_i}) \diamond \mathcal{H}(T_{P_j}) \quad (23)$$

We now prove the following property:

Property 1: For all i such that $1 \leq i \leq k$ we have,

$$\forall j, i \leq j \leq k : ((H_{mash}, A_{mash}, D_j), (H_i, A_i, D_j)) \in R(T_{P_j})$$

We prove the above property by induction over i .

Base Case ($i = 1$): By definition, $H_1, A_1 = \mathcal{HS}(First(States(\tau, id_1))) = H_{mash}, A_{mash}$. Since $S_{P_j} <: (H_{mash}, A_{mash}, D_j)$, we have that $\forall j, j \geq 1 : ((H_{mash}, A_{mash}, D_j), (H_1, A_1, D_j)) \in R(T_{P_j})$

Induction Hypothesis: Assume for $i = m (< k)$.

Inductive Case: Consider $i = m + 1 (\leq k)$. From the induction hypothesis we have,

$$\forall j, m \leq j \leq k : ((H_{mash}, A_{mash}, D_j), (H_m, A_m, D_j)) \in R(T_{D_j}) \quad (24)$$

From lemma 1 and condition 16, we have

$$H_m, A_m, D_m \rightsquigarrow H_{m+1}, A_{m+1}, normal \quad (25)$$

From condition 24, we have $((H_{mash}, A_{mash}, D_m), (H_m, A_m, D_m)) \in R(T_{P_m})$. Combining this with conditions 21, 22, 23 and using lemma 7 we have:

$$\forall j, m + 1 \leq j \leq k : ((H_{mash}, A_{mash}), (H_{m+1}, A_{m+1})) \in R(P_j)$$

Thus the inductive case is true. Hence property 1 is true. From property 1, we have:

$$\forall 1 \leq i \leq k : ((H_{mash}, A_{mash}, D_i), (H_i, A_i, D_i)) \in R(T_{P_i}) \quad (26)$$

From lemma 6, it follows that for each i , $(H_{mash}, A_{mash}, D_i)$ simulates (H_i, A_i, D_i) . Since $Rn(A_i, id_i) \subseteq A_{mash}$ and $(H_{mash}, Rn(A_i, id_i), D_i)$ holds, from proposition 1 it follows that $(H_{mash}, Rn(A_i, id_i), D_i)$ simulates $(H_{mash}, A_{mash}, D_i)$. By 5, H, A_i, C_i simulates $(H_{mash}, Rn(A_i, id_i), D_i)$. Since the simulates relation is transitive, we have

$$\forall 1 \leq i \leq k : ((H, A, C_i), (H_i, A_i, D_i)) \in R(T_{P_i})$$

From the above condition and condition 16, we have that the mashup $\mathcal{M}(H, A, (C_1, id_1), \dots, (C_n, id_n))$ is isolated and we are done. \square

Proof of theorem 4. Let $\mathcal{M}(H, A, (C_1, id_1), \dots, (C_n, id_n)) = (H_{mash}, A_{mash}, D_1; \dots; D_n)$. By definition of \mathcal{M} ,

For each i , $D_i = Rn(C_i, id_i)$.

$$H_{mash} = H$$

$$A_{mash} := Rn(A_1, id_1) \dots Rn(A_n, id_n)$$

Consider any trace $\tau \in \text{Traces}(H_{\text{mash}}, A_{\text{mash}}, C_{\text{mash}})$. Consider the sequence $\text{States}(\tau, id_1), \dots, \text{States}(\tau, id_n)$ and let k ($k \in \{1, \dots, n-1\}$) be the smallest index such that $\text{States}(\tau, id_{k+1}) = \emptyset$. If no such k exists, then let $k = n$. By definition of States ,

$$(\forall 1 \leq i \leq k : \text{States}(\tau, id_i) \neq \emptyset) \wedge (\forall k+1 \leq i \leq n : \text{States}(\tau, id_i) = \emptyset) \quad (27)$$

For any $1 \leq i \leq k$, let $H_i, A_i, D_i = \text{First}(\text{States}(\tau, id_i))$.

Since for each i , $\vdash_{SL_2} \{P_i\} C_i \{Q_i\}$ holds, By repeated applications of the substitution rule, we have that $\vdash_{SL_2} \{Rn(P_i, id_i)\} D_i \{Rn(Q_i, id_i)\}$ holds.

From lemma 3 and definitions of H_{mash} and A_{mash} , we can show that since $H, A \in \llbracket P_1 * \dots * P_n * \text{true} \rrbracket_{A_2}$, we have

$$H_{\text{mash}}, A_{\text{mash}} \in \llbracket (Rn(P_1, id_1)) * \dots * (Rn(P_n, id_n)) * \text{true} \rrbracket_{A_2}$$

Thus for $i \in \{1, \dots, n\}$, there exist heap-stores H_{P_i}, A_{P_i} and permission map Σ_{P_i} such that

$$\forall i, j, i \neq j : (H_{P_i}, \Sigma_{P_i}) \bowtie (H_{P_j}, \Sigma_{P_j}) \quad (28)$$

$$\forall i : H_{P_i} \subseteq H_{\text{mash}} \quad (29)$$

$$\forall i : A_{P_i} = \text{Proj}(A_{\text{mash}}, \text{free}(P_i)) \quad (30)$$

$$\forall i : H_{P_i}, A_{P_i}, \Sigma_{P_i} \models_{A_2} P_i \quad (31)$$

$$\forall i : \text{dom}(H_{P_i}) = \text{dom}(\Sigma_{P_i}) \quad (32)$$

Since for each i , $\vdash_{SL_2} \{Rn(P_i, id_i)\} D_i \{Rn(Q_i, id_i)\}$ holds, by the soundness theorem we have that for each i , $\models_{SL_2} \{Rn(P_i, id_i)\} D_i \{Rn(Q_i, id_i)\}$ holds. Therefore we have,

$$\forall i, 1 \leq i \leq k : \text{Safe}(T_{P_i}) \quad (33)$$

$$\forall i, j, i \neq j : \text{dom}(\mathcal{S}(T_{P_i})) \cap \text{dom}(\mathcal{S}(T_{P_j})) = \emptyset \quad (34)$$

From condition 29, we have that

$$\forall i, j, i \neq j : \mathcal{H}(T_{P_i}) \diamond \mathcal{H}(T_{P_j}) \quad (35)$$

Further, from condition 31, we have that for all i and for all heap-stores K_i, B_i , if $T_{P_i} \rightsquigarrow K_i, B_i, \text{normal}$ then

$$\forall (l, p, a, v) \in \text{Acc}^{\text{heap}}(T_{P_i}, (K_i, B_i, \text{normal})) : a \in \Sigma_{P_i}(l).p$$

Combining this with conditions 32 and 28, we have that for all i, j such that $i \neq j$ and for all heap-stores K_i, B_i , if $T_{P_i} \rightsquigarrow K_i, B_i, \text{normal}$ then

$$\forall i, j, i \neq j : \forall (l, p, v) : (l, p, v) \in \text{Acc}_{\text{write}}^{\text{heap}}(T_{P_i}, (K_i, B_i, \text{normal})) \implies (l, p) \notin \text{dom}(\mathcal{H}(T_{P_j})) \quad (36)$$

From lemma 4, $\text{dom}(\mathcal{H}(T_{P_i})) \subseteq \text{dom}(K_i)$. This gives us the following condition

$$\forall i, j, i \neq j : \forall l, p : (l, p) \in \text{dom}(\mathcal{H}(T_{P_j})) \cap \text{dom}(\mathcal{H}(T_{P_i})) \implies K_i(l).p = \mathcal{H}(T_j)(l).p \quad (37)$$

We now prove the following property:

Property 1: For all i such that $1 \leq i \leq k$ we have,

$$\forall j, i \leq j \leq k : ((H_{\text{mash}}, A_{\text{mash}}, D_j), (H_i, A_i, D_j)) \in R(T_{P_j})$$

We prove the above property by induction over i .

Base Case ($i = 1$): By definition, $H_1, A_1 = \mathcal{HS}(\text{First}(\text{States}(\tau, id_1))) = H_{\text{mash}}, A_{\text{mash}}$. Since $T_{P_j} <: (H_{\text{mash}}, A_{\text{mash}}, D_j)$, we have that $\forall j, j \geq 1 : ((H_{\text{mash}}, A_{\text{mash}}, D_j), (H_1, A_1, D_j)) \in R(T_{P_j})$

Induction Hypothesis: Assume for $i = m (< k)$.

Inductive Case: Consider $i = m+1 (\leq k)$. From the induction hypothesis we have,

$$\forall j, m \leq j \leq k : ((H_{\text{mash}}, A_{\text{mash}}, D_j), (H_m, A_m, D_j)) \in R(T_{P_j}) \quad (38)$$

From lemma 1 and condition 27, we have

$$H_m, A_m, C_m \rightsquigarrow H_{m+1}, A_{m+1}, \text{normal} \quad (39)$$

From condition 38, we have $((H_{mash}, A_{mash}, D_m), (H_m, A_m, D_m)) \in R(T_{P_m})$. Combining this with conditions 33, 34, 35, 37 and using lemma 7 we have:

$$\forall j, m+1 \leq j \leq k : ((H_{mash}, A_{mash}), (H_{m+1}, A_{m+1})) \in R(P_j)$$

Thus the inductive case is true. Hence property 1 is true. From property 1, we have:

$$\forall 1 \leq i \leq k : ((H_{mash}, A_{mash}, D_i), (H_i, A_i, D_i)) \in R(T_{P_i}) \quad (40)$$

From lemma 6, it follows that for each i , $(H_{mash}, A_{mash}, D_i)$ simulates (H_i, A_i, D_i) . Since $Rn(A_i, id_i) \subseteq A_{mash}$ and $(H_{mash}, Rn(A_i, id_i), D_i)$ holds, from proposition 1 it follows that $(H_{mash}, Rn(A_i, id_i), D_i)$ simulates $(H_{mash}, A_{mash}, D_i)$. By 5, H, A_i, C_i simulates $(H_{mash}, Rn(A_i, id_i), D_i)$. Since the simulates relation is transitive, we have

$$\forall 1 \leq i \leq k : ((H, A, C_i), (H_i, A_i, D_i)) \in R(T_{P_i})$$

From the above condition and condition 16, we have that the mashup $\mathcal{M}(H, A, (C_1, id_1), \dots, (C_n, id_n))$ is isolated and we are done. \square