A Modal Logic for Mobile Resources

Morten Dahl, Claus Thrane, Uffe Sørensen and Martin Clemmensen
Department of Computer Science, Aalborg University

June 2006

Abstract: We introduce a logic $\mathcal{L}_{\equiv}$ for a subset of the *Mobile Resources* (MR) calculus [6] where name restriction is removed. The calculus is tailored for expressing and analyzing the properties of systems containing mobile, possibly nested, computing devices that may again have internal resources which are not copyable nor modifiable per se. We provide the semantics for $\mathcal{L}_{\equiv}$ and compare the equivalence $=_{\mathcal{L}_{\equiv}}$ on MR processes (induced by $\mathcal{L}_{\equiv}$) with the MR structural congruence in order to show the logic's power to describe internal structures of processes. We also present a logic $\mathcal{L}_{\sim}$ based on the transition rules of the calculus which is similar to HML [8] and closely releated to bisimulation. We show that $\mathcal{L}_{\equiv}$ is strong enough to simulate $\mathcal{L}_{\sim}$ through an encoding function φ. Finally, we present a sound and complete model checking algorithm for our subset of the MR calculus, without replication and $\mathcal{L}_{\equiv}$ without composition adjunct.

1 Introduction

Mobile computing resources abound in today's society. Examples are smart cards used in SIM-cards or credit cards. These resources move from card issuers to card holders and in and out of mobile phones or ATMs. The ability to reason about correctness of the behaviour of systems containing such resources has been the inspiration to the development of a calculus of mobile resources (MR). The purpose of the calculus is to be able to design and analyze systems containing nested, mobile computing resources residing in named locations. Furthermore, a formal framework to express and prove properties that may depend on the assumption that such resources are neither copyable nor arbitrarily modifiable has been devised. These assumptions are crucial for the security of systems based on smart cards as trusted computing resources.

Two strategies for determining correctness of a model are common: behavioural equivalence of an implementation and a specification using a bisimulation or model checking a process against a set of properties described in a logic. Each has certain benefits. Since the MR calculus models mobility through spatial configuration, a logic has been designed to describe the precise spatial structure of a process expression. With our logic $\mathcal{L}_{\equiv}$ it is possible to express desirable properties and assert that these properties hold for a given model expressed in MR. Properties such as: "two parallel processes can move a subprocess from one process to the other" may be expressed and asserted. Furthermore, we show that $\mathcal{L}_{\equiv}$ which is based on a structural congruence may indeed also express behavioural properties in form of bisimulation.

Contents

2 The Mobile Resources Calculus

In the following Section we introduce the syntax and semantics of a reduced version of the Mobile Resources calculus; in the rest of the article we use MR to refer to this reduced version. The Mobile Resources calculus is inspired by the *Mobile Ambient* calculus [2] and has a simular concept of slots, resulting in the syntax shown in Figure 1.

As shown in Section 3 we have chosen not to include name restriction in the logic and hence MR is modified in the same way. Notice that since we do not add new or modify existing elements, MR processes are a subset of processes of the original Mobile Resources calculus. In Section 8 we briefly discuss how name restriction could be included in the logic and what would be affected.

2.1 The Syntax

We define an infinite countable set of *names* $\mathcal{N}$ and a set of *co-names* $\overline{\mathcal{N}} = \{\overline{n} | n \in \mathcal{N}\}$ to use as names in actions and directions. From $\mathcal{N}$ we use n and m to denote single elements and $\tilde{n}$ and $\tilde{m}$ to denote subsets. The letter γ denotes a sequence of names from $\mathcal{N}^*$, with ε being the empty sequence and δ a sequence with at least one element ($\mathcal{N}^+$). γ and δ are called *directions.* Let α range over the set of *actions* $\mathcal{A} = \mathcal{N} \cup \overline{\mathcal{N}}$. An action α synchronizes with the corresponding co-action $\overline{\alpha}$ like in CCS [11]. Also, let $\mathcal{P}$ be the set of *process expressions* ranged over by P and Q.

$$\begin{array}{rll@{\qquad}rll@{\qquad}rll}
P, Q ::= & \mathbf{0} & \textit{(nil)} & & & & & & \\
\mid & \lambda.P & \textit{(prefix)} & \lambda ::= & \gamma\alpha & \textit{(action)} & & & \\
\mid & P \parallel Q & \textit{(parallel composition)} & \mid & \delta \triangleright \overline{\delta'} & \textit{(move)} & r ::= & \bullet & \textit{(empty slot)} \\
\mid & !P & \textit{(replication)} & \mid & \natural m & \textit{(deletion)} & \mid & P & \textit{(occupied slot)} \\
\mid & \tilde{n}\lfloor r \rfloor_m & \textit{(slot)} & & & & & &
\end{array}$$

Figure 1: MR syntax

The constructs *nil*, *prefix* and *parallel composition* are inherited from CCS-like calculi and are used at usual. The remaining constructs are added to express the mobile behaviour of processes. The prefix *action* captures directed actions ($\gamma \neq \epsilon$) and actions ($\gamma = \epsilon$). Actions can synchronize with actions in the same slot or with directed actions from outer slots. The prefix *move* is a directed move of resources residing at any accessible sub-location δ relative to the move action. A *slot* is identified by a set of names $\tilde{n}$ and an additional deletion name m. A slot may be deleted by performing the $\natural m$ action from a process parallel to the slot. Processes within slots are referred to as resources. The *replication* action provides as many parallel instances of the process as required and adds to the calculus the power of recursive definitions.

Since slots may contain other slots we denote the spatial tree structure formed as the *slot hierarchy* of a process expression. We say that nested slots are lower in the hierarchy than the slot in which they are contained.

2.1.1 Contexts

The context constructs of the calculus are as defined in Figure 2. Contexts $\mathscr{C}$ are, as usual, process terms with a single hole $(-)$. We write $\mathscr{C}(P)$ for the insertion of a process P in the hole of context $\mathscr{C}$, yielding a new process.

$$\begin{array}{rl@{\qquad}l}
\mathscr{C}_\varepsilon ::= & (-) & \\
\mathscr{C}_{n\gamma} ::= & \tilde{n}\lfloor \mathscr{C}_\gamma \| P \rfloor_m & (\textit{where } n \in \tilde{n}) \\
\mathscr{D}_{\gamma n} ::= & \mathscr{C}_\gamma(\tilde{n}\lfloor(-)\rfloor_m) & (\textit{where } n \in \tilde{n}) \\
\mathscr{E} ::= & (-) \mid \tilde{n}\lfloor \mathscr{E} \rfloor_m \mid \mathscr{E} \| P &
\end{array}$$

Figure 2: Context syntax

The context $\mathscr{C}_\varepsilon$ is simply a flat context with a hole whereas $\mathscr{C}_{n\gamma}$ is a context with a hole which resides under path γ in the slot n. This context may have parallel processes at all levels. $\mathscr{D}_{\gamma n}$ is used for the special case where the hole is the only content of the slot n located under a path γ. Evaluation contexts $\mathscr{E}$ are contexts whose hole does not appear under prefix or replication.

2.1.2 Structural Congruence

An equivalence relation $\mathcal{S}$ on $\mathcal{P}$ is a *congruence* if it is preserved by all contexts, i.e. $\mathcal{S}$ is a congruence if $P\ \mathcal{S}\ Q$ implies $\mathscr{C}(P)\ \mathcal{S}\ \mathscr{C}(Q)$ for all contexts $\mathscr{C}$. The structural congruence relation $\equiv$ is defined as the least congruence on $\mathcal{P}$ which satisfies the rules in Figure 3. Figure 4 lists derived properties caused by $\equiv$ being a congruence.

$$\boxed{\begin{array}{ll} E_1)\ P \parallel \mathbf{0} \equiv P & E_2)\ !P \equiv P \parallel !P \\ E_3)\ P \parallel Q \equiv Q \parallel P & E_4)\ (P \parallel P') \parallel P'' \equiv P \parallel (P' \parallel P'') \end{array}}$$

Figure 3: Structural equivalence

Since we have no name restriction and hence no alpha conversion, structural congruence can informally be said to hold for two processes if one can be derived from the other just by simple rearrangement of parts without any computational difference. By definition we have E_5 from which E_6 and E_7 follows. E_8 and E_9 are easily shown by induction on the depth of the inference of $P \equiv Q$.

$$\boxed{\begin{array}{ll} E_5)\ \mathscr{E}(P) \equiv \mathscr{E}(Q),\text{ if } P \equiv Q & E_6)\ P \equiv Q \implies \forall R : P \parallel R \equiv Q \parallel R \\ E_7)\ P \equiv Q \iff \tilde{n}\lfloor P \rfloor_m \equiv \tilde{n}\lfloor Q \rfloor_m & E_8)\ P \equiv Q \land P \equiv \lambda.P' \implies \exists Q' : Q \equiv \lambda.Q' \land P' \equiv Q' \\ \multicolumn{2}{l}{E_9)\ P \equiv Q \land P \equiv P_1 \parallel P_2 \implies \exists Q_1, Q_2 : Q \equiv Q_1 \parallel Q_2 \land P_1 \equiv Q_1 \land P_2 \equiv Q_2} \end{array}}$$

Figure 4: Derived properties of structural congruence

2.2 The Semantics

Two semantics for MR are provided in [6]. One based on reduction rules and one based on SOS transition rules and a labelled transition system. As shown in [6] these semantics coincide for τ transitions. The reduction rules only express how a process can evolve by internal actions, whereas the SOS transition rules describe how the process may interact with its context. An interesting difference between MR and CCS is the fact that a three-way synchronization is achieved among the source, the destination and the issuing process when moving a resource.

2.2.1 Reduction Semantics

The relation $\searrow$ is defined as the least binary relation on $\mathcal{P}$ satisfying the rules of Figure 5 closed under $\equiv$, as defined in Figure 3, and under all evaluation contexts $\mathscr{E}$.

$$\boxed{\begin{array}{l} \gamma\alpha.P \parallel \mathscr{C}_\gamma(\overline{\alpha}.Q) \searrow P \parallel \mathscr{C}_\gamma(Q) \\ \gamma\delta_1 \triangleright \overline{\gamma\delta_2}.P \parallel \mathscr{C}_\gamma(\mathscr{D}_{\delta_1}(Q) \parallel \mathscr{D}_{\delta_2}(\bullet)) \searrow P \parallel \mathscr{C}_\gamma(\mathscr{D}_{\delta_1}(\bullet) \parallel \mathscr{D}_{\delta_2}(Q)) \\ \natural m.P \parallel \tilde{n}\lfloor r \rfloor_m \searrow P \end{array}}$$

Figure 5: Reduction rules

The first rule describes standard CCS synchronization of action α and co-action $\overline{\alpha}$. Additionally the context $\mathscr{C}_\gamma$ express that $\gamma\alpha$ synchronizes with an $\overline{\alpha}$ action found under path γ. The second rule shows how resources may be moved under contexts. Moves are done by a third party process and not by a resource itself, contrary to the approach in the Mobile Ambient calculus. Moves must be performed by a process located higher in the slot hierarchy than the process being moved. The third rule defines deletion of slots.

2.2.2 Transition Semantics

The semantics of MR is defined in terms of SOS transition rules and labelled transition semantics given in Figures 7 and 8 respectively. Figure 6 summarizes the observable labels of the labelled semantics with some of them requiring an explaination: in the *exit* (respectively *enter*) label the δ is the path under which the slot is located and P the process leaving (respectively entering) the slot. The co-action for *exit* (respectively *enter*) is *take* (respectively *give*) where δ is the path of the slot where the process P is leaving from (respectively entering to).

$$\begin{array}{llll} \pi ::= \beta & & \beta ::= \tau \mid \lambda \mid \overline{\delta}\alpha & \\ \quad \mid \ \overline{\delta} \triangleright \langle \mathrm{P} \rangle & \textit{(exit)} & \quad \mid \ \overline{\delta} \triangleright \delta' & \textit{(comove)} \\ \quad \mid \ \langle \mathrm{P} \rangle \triangleright \overline{\delta} & \textit{(give)} & \quad \mid \ (P) \triangleright \delta & \textit{(enter)} \\ & & \quad \mid \ \delta \triangleright (P) & \textit{(take)} \end{array}$$

Figure 6: Observable labels in the semantics

In Figure 9 we provide the different ways three-way synchronization can be constructed and the example in Section A.6 shows a complete derivation of such a synchronization. The order of synchronization is nondeterministic, thus any of the derivations from Figure 9 may occur when evaluating a process. Any two processes may initiate the synchronization, resulting in a co-action matching the third process and finally evaluating to a τ *(sync)* action. The remaining rules are evaluated as usual.

$$(\textit{prefix})\ \frac{}{\lambda.P \xrightarrow{\lambda} P} \qquad (\textit{par})\ \frac{P \xrightarrow{\pi} P'}{P \parallel Q \xrightarrow{\pi} P' \parallel Q}$$

$$(\textit{rep})\ \frac{P \parallel !P \xrightarrow{\pi} P'}{!P \xrightarrow{\pi} P'} \qquad (\textit{sym})\ \frac{P \parallel Q \xrightarrow{\pi} P' \parallel Q}{Q \parallel P \xrightarrow{\pi} Q \parallel P'}$$

$$(\textit{sync})\ \frac{P_1 \xrightarrow{\overline{\pi}} P_1' \quad P_2 \xrightarrow{\pi} P_2'}{P_1 \parallel P_2 \xrightarrow{\tau} P_1' \parallel P_2'}$$

Figure 7: Standard transition rules

$$(\textit{exit})\ \frac{}{\tilde{n}\lfloor P \rfloor_m \xrightarrow{\overline{n} \triangleright \langle \mathrm{P} \rangle} \tilde{n}\lfloor \bullet \rfloor_m}\ n \in \tilde{n} \qquad (\textit{comove})\ \frac{P_1 \xrightarrow{\overline{\delta_1} \triangleright \langle \mathrm{Q} \rangle} P_1' \quad P_2 \xrightarrow{(Q) \triangleright \delta_2} P_2'}{P_1 \parallel P_2 \xrightarrow{\overline{\delta_1} \triangleright \delta_2} (P_1' \parallel P_2')}$$

$$(\textit{enter})\ \frac{}{\tilde{n}\lfloor \bullet \rfloor_m \xrightarrow{(P) \triangleright n} \tilde{n}\lfloor P \rfloor_m}\ n \in \tilde{n} \qquad (\textit{nesting})\ \frac{P \xrightarrow{\pi} P'}{\tilde{n}\lfloor P \rfloor_m \xrightarrow{n \cdot (\pi)} \tilde{n}\lfloor P' \rfloor_m}\ n \in \tilde{n}$$

$$(\textit{give})\ \frac{P_1 \xrightarrow{\overline{\delta_1} \triangleright \langle \mathrm{Q} \rangle} P_1' \quad P_2 \xrightarrow{\delta_1 \triangleright \overline{\delta_2}} P_2'}{P_1 \parallel P_2 \xrightarrow{\langle \mathrm{Q} \rangle \triangleright \overline{\delta_2}} P_1' \parallel P_2'} \qquad (\textit{delete})\ \frac{}{\tilde{n}\lfloor r \rfloor_m \xrightarrow{\overline{\natural m}} \mathbf{0}}$$

$$(\textit{take})\ \frac{P_2 \xrightarrow{\delta_1 \triangleright \overline{\delta_2}} P_2' \quad P_1 \xrightarrow{(Q) \triangleright \delta_2} P_1'}{P_1 \parallel P_2 \xrightarrow{\delta_1 \triangleright (Q)} P_1' \parallel P_2'}$$

Figure 8: Transition rules for resources and mobility

An important detail lies within the $n \cdot (\_)$ operation used in the *nesting* rule. The operation dictates how actions happening inside a slot is viewed by the outside world and is defined by the following rules:

$$\begin{array}{lll} n \cdot (\tau) = \tau & n \cdot (\overline{\gamma}\alpha) = \overline{n\gamma}\alpha & n \cdot (\overline{\delta_1} \triangleright \delta) = \overline{n\delta_1} \triangleright n\delta_2 \\ n \cdot ((P) \triangleright \delta) = (P) \triangleright n\delta & n \cdot ((\tilde{n})\overline{\delta} \triangleright \langle P \rangle) = (\tilde{n})\overline{n\delta} \triangleright \langle P \rangle & \end{array}$$

By not being defined for all labels from π the operation restricts some actions from being visible from outside a slot, e.g. a directed action inside a slot is not visible from the outside since $n \cdot (\delta\overline{\alpha})$ is not defined.

Another detail is the use of $\overline{\pi}$ in the *sync* rule. Depending on the label from π we have that $\overline{\pi}$ is:

$$\begin{array}{rcl|lcl} \textit{action} & \overline{\overline{\delta}\alpha} = \delta\overline{\alpha} & \textit{coaction} & \textit{move} & \overline{\delta_1 \triangleright \overline{\delta_2}} = \overline{\delta_1} \triangleright \delta_2 & \textit{comove} \\ \textit{exit} & \overline{\overline{\delta} \triangleright \langle P \rangle} = \delta \triangleright (P) & \textit{take} & \textit{enter} & \overline{(P) \triangleright \delta} = \langle P \rangle \triangleright \overline{\delta} & \textit{give} \end{array}$$

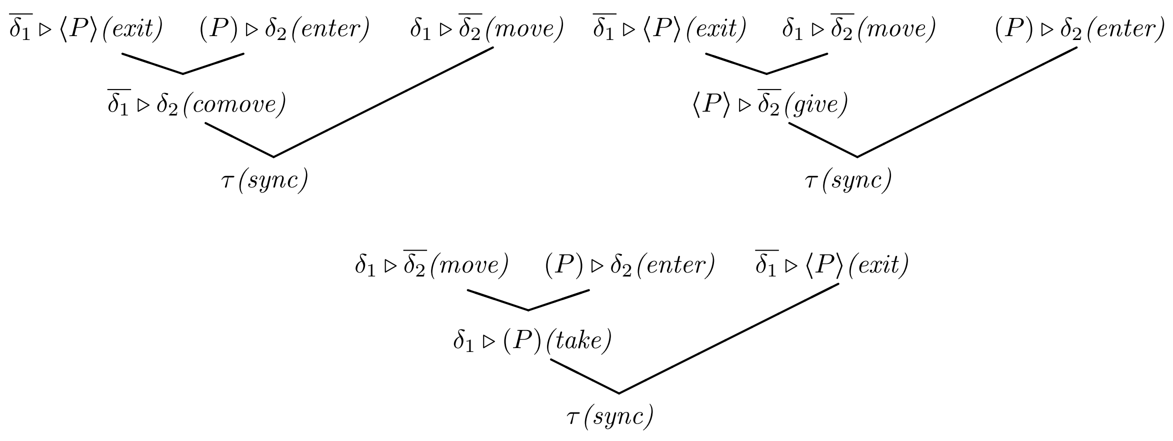
$\overline{\delta_1} \triangleright \langle P \rangle$ (exit)
$(P) \triangleright \delta_2$ (enter)
$\delta_1 \triangleright \overline{\delta_2}$ (move)
$\overline{\delta_1} \triangleright \delta_2$ (comove)
τ (sync)
$\overline{\delta_1} \triangleright \langle P \rangle$ (exit)
$\delta_1 \triangleright \overline{\delta_2}$ (move)
$(P) \triangleright \delta_2$ (enter)
$\langle P \rangle \triangleright \overline{\delta_2}$ (give)
τ (sync)
$\delta_1 \triangleright \overline{\delta_2}$ (move)
$(P) \triangleright \delta_2$ (enter)
$\overline{\delta_1} \triangleright \langle P \rangle$ (exit)
$\delta_1 \triangleright (P)$ (take)
τ (sync)

Figure 9: All possible three way synchronizations of MR processes

So for example for two processes $n\lfloor a.P \rfloor \xrightarrow{\overline{n}a} n\lfloor P \rfloor$ and $n\overline{a}.Q \xrightarrow{n\overline{a}} Q$ when put together the *sync* rule gives that $n\lfloor a.P \rfloor \parallel n\overline{a}.Q \xrightarrow{\tau} n\lfloor P \rfloor \parallel Q$.

2.2.3 Strong Bisimulation

If we use the labels from Figure 6 for the bisimulation, we end up with a simulation that is too strong. As an example for processes P and Q:

$$P\colon\ n\lfloor a.a \rfloor \xrightarrow{\overline{n}\triangleright\langle a.a\rangle} n\lfloor \bullet \rfloor$$
$$Q\colon\ n\lfloor a \parallel a \rfloor \xrightarrow{\overline{n}\triangleright\langle a\ \parallel\ a\rangle} n\lfloor \bullet \rfloor$$

Although P and Q should be bisimular, they are not since P can do a transition that Q cannot (and Q can do a transition that P cannot). For this reason the *exit* and *give* actions are replaced with families of actions that do not say anything about the process leaving a slot. We have:

$$\begin{array}{llll} \textit{give} & P \xrightarrow{\triangleright\overline{\delta}(\mathscr{D}_\delta)} (P' \parallel \mathscr{D}_\delta(Q)) & \text{if } P \xrightarrow{\langle Q\rangle\triangleright\overline{\delta}} P' \\ \textit{exit} & P \xrightarrow{(\mathscr{C}_\gamma)\overline{\delta'}\triangleright(\mathscr{D}_\delta)} (\mathscr{C}_\gamma(P') \parallel \mathscr{D}_\delta(Q)) & \text{if } P \xrightarrow{\overline{\delta'}\triangleright\langle Q\rangle} P' \end{array}$$

Notice that in the *exit* action δ and γ are chosen freely: δ' denotes the path of the slot within P from where Q is removed; δ denotes the path of the slot where process Q is moved to and γ is the path to the slot in which P resides. The actions in the bisimulation becomes:

$$\psi ::= \beta \mid \triangleright\overline{\delta}(\mathscr{D}_\delta) \mid (\mathscr{C}_\gamma)\overline{\delta'} \triangleright (\mathscr{D}_\delta)$$

From [6] we have the following definition of $\sim$ and the result that it is a congruence:

Definition 1. A *simulation* is a binary relation $\mathcal{S}$ over $\mathcal{P}$ such that whenever $(P, Q) \in \mathcal{S}$:

$$\text{if } P \xrightarrow{a} P' \text{ then there exists } Q' \text{ such that } Q \xrightarrow{a} Q' \text{ and } (P', Q') \in \mathcal{S}$$

Where a is an element from the set of labels generated by ψ. $\mathcal{S}$ is a *bisimulation* if $\mathcal{S}$ and $\mathcal{S}^{-1} = \{(Q,P) | (P,Q) \in \mathcal{S}\}$ are simulations. We write $P \sim Q$ if there exists a bisimulation $\mathcal{S}$ such that $(P, Q) \in \mathcal{S}$.

2.2.4 Barbed Bisimulation

In terms of observable communication, barbs are defined as the set of (non-directed) actions in λ that a process P offers to the surrounding environment. This definition excludes observation of directed actions and move actions. As in [6] we define barbs as:

$$P \downarrow n \text{ if } P \equiv \alpha.P' \parallel Q \text{ where } \alpha \in \{n, \bar{n}\}$$

In this case n is the set of actions available from the process P to the surrounding environment.

Definition 2. A *barbed bisimulation* is a symmetric relation $\mathcal{S}$ on $\mathcal{P}$ such that whenever $P \; \mathcal{S} \; Q$ holds:

$$\begin{array}{l} P \downarrow n \text{ implies } Q \downarrow n \\ P \searrow P' \text{ then there exists } Q' \text{ such that } Q \searrow Q' \text{ and } P' \; \mathcal{S} \; Q' \end{array}$$

Barbed bisimulation congruence $\sim_b$ is the largest congruence that is a barbed bisimulation.

In [6] it is shown that $\sim \; = \; \sim_b$ and argued that it is easier to show bisimularity than to show barbed congruence of two MR processes.

3 A Logic for MR

Our logic $\mathcal{L}_{\equiv}$ is inspired by the logic for the Mobile Ambient calculus [1]. This becomes apparent in the definition of the connectives, which enables us to express properties of mobility as evolution of spatial configurations over time. To enable the logic to express the desired properties of spatial configurations, the design of the satisfaction relation is based on structural congruence and not transition rules as is the case of HML and the logic $\mathcal{L}_{\sim}$ we present in Section 5.1. As a result, $\mathcal{L}_{\equiv}$ is able to seperate terms on the basis of their internal structure, even though their behaviours are the same, as opposed to $\mathcal{L}_{\sim}$ and HML which can only seperate terms that have different behaviours.

3.1 The Syntax of $\mathcal{L}_{\equiv}$

The connectives can be devided into three categories: first order predicate logic, spatial and temporal constructs, where spatial express structure and temporal express evolution in time. The formulae of the logic are defined by the following syntax:

$$\begin{array}{lll} A, B ::= & \mathit{tt} \mid \neg A \mid A \vee B \mid \forall x.A & \textit{first order predicate logic} \\ & \mathbf{0} \mid \tilde{\eta}\lfloor A \rfloor \mid \tilde{\eta}\lfloor \bullet \rfloor \mid A_1 \parallel A_2 \mid \diamondsuit\!\!\!\!\diamond A \mid A@\tilde{\eta} \mid A \parallel\!\triangleright B & \textit{spatial} \\ & \diamond A \mid \lambda.A & \textit{temporal} \end{array}$$

The symbol η denotes a member and $\tilde{\eta}$ denotes a subset of the set $\Lambda \cup \mathcal{N}$ where Λ is the set of variables and $\mathcal{N}$ the set of names; we assume that $\Lambda \cap \mathcal{N} = \emptyset$. The *prefix* rule uses the symbol λ which ranges over the prefix actions of the calculus. Each element γ, δ and m from λ may consist of members from $\Lambda \cup \mathcal{N}$ and α from $\Lambda \cup \mathcal{A}$.

3.1.1 Precedence

Connectives only bind to the connectives adjacent to themselves. The precedence hierarchy is as follows, in order of appearance (top to bottom), such that unary operators bind the strongest and universal quantification binds the least:

$$\begin{array}{rcl} \lambda, \diamond, ✧, \square, ⌑ & : & \text{unary operators bind right s.t. } \lambda ✧ A \text{ should be read } \lambda(✧A) \\ \parallel\!\triangleright \text{ is stronger than } @ & : & A \parallel\!\triangleright B@n \text{ should be read } (A \parallel\!\triangleright B)@n \\ \vee \text{ is weaker than } \wedge & : & A \vee B \wedge C \text{ should be read } A \vee (B \wedge C) \\ \parallel & : & \text{seperates terms of expression} \\ \forall x & : & \text{quantifies entire expressions s.t. } \forall x.A \text{ should be read } (\forall x.A) \end{array}$$

To change the scope of a given connective, parentheses can be inserted in the formula, e.g. $A \vee (B \parallel C) \vee D$ instead of $A \vee B \parallel C \vee D$.

3.2 Satisfaction

The definition of satisfaction for closed formulae[1] of $\mathcal{L}_{\equiv}$ is given in Figure 10. The temporal connective *sometime* ($\Diamond$) is defined in terms of the transitive and reflective closure $\searrow_*$ of the reduction relation $\searrow$ in Section 2.2.1. Defining $\Diamond$ in terms of the reduction rules is preferred over the transition rules because, when using transition rules, a process containing an empty slot without the possibility of restriction may satisfy any property given the infinite possibilities of processes which may enter. Using the reduction rules allows expressions about how the process evolves on its own. Informally we have that a process satisfying $\Diamond A$ must at some point in the future reduce to a state where A is satisfied.

true	$P \models tt$	$\triangleq$	true for all processes P
negation	$P \models \neg A$	$\triangleq$	$\neg(P \models A)$
disjunction	$P \models A \vee B$	$\triangleq$	$P \models A \vee P \models B$
void	$P \models \mathbf{0}$	$\triangleq$	$P \equiv \mathbf{0}$
location	$P \models \tilde{m}\lfloor A \rfloor$	$\triangleq$	$\exists P'$ s.t. $P \equiv \tilde{n}\lfloor P' \rfloor_{n'}$ with $\tilde{m} \subseteq \tilde{n}$ and $P' \models A$ for some n'
empty location	$P \models \tilde{m}\lfloor \bullet \rfloor$	$\triangleq$	$P \equiv \tilde{n}\lfloor \bullet \rfloor_{n'}$ with $\tilde{m} \subseteq \tilde{n}$ for some n'
composition	$P \models A_1 \parallel A_2$	$\triangleq$	$\exists P_1, P_2$ s.t. $P \equiv P_1 \parallel P_2$ and $P_1 \models A_1 \wedge P_2 \models A_2$
sometime modality	$P \models \Diamond A$	$\triangleq$	$\exists P'$ s.t. $P \searrow_* P'$ and $P' \models A$
somewhere modality	$P \models$ ✧A	$\triangleq$	$\exists P'$ s.t. $P \downarrow^* P'$ and $P' \models A$
location adjunct	$P \models A@\tilde{m}$	$\triangleq$	$\exists P'$ s.t. $P' \equiv \tilde{n}\lfloor P \rfloor_{n'}$ with $\tilde{m} \subseteq \tilde{n}$ and $P' \models A$ for some n'
composition adjunct	$P \models A \parallel\!\triangleright B$	$\triangleq$	$\forall R$ s.t. $R \models A$ then $R \parallel P \models B$
prefix	$P \models \lambda.A$	$\triangleq$	$\exists P'$ s.t. $P \equiv \lambda.P'$ and $P' \models A$
universal	$P \models \forall x.A$	$\triangleq$	$P \models A\{^m/_x\}\ \forall m \in \mathcal{N}^+ \cup \overline{\mathcal{N}}$

Figure 10: The satisfaction relation is defined on closed processes and closed formulae

The *slot* connective is used to express structural properties of the process, but to allow for some flexisiblity in this structure the slot names in the logic only has to be a subset of the slot names in the process: the process $P = \{a, b, c\}\lfloor \mathbf{0} \rfloor_d$ satisfies the formula $A = \{a, c\}\lfloor tt \rfloor$ and all (non-empty) slots satisfies $\emptyset\lfloor tt \rfloor$. For similar reasons, slots in the logic have no delete name. The *somewhere* connective is defined (as in [2]) in terms of the transitive and reflexive closure $\downarrow^*$ of $\downarrow$ and means that somewhere in the slot hierarchy there is a process that satisfies A. The *location* connective specifies that the process P is a slot named $\tilde{n}$ containing a process P' which satisfies A. The *empty location* connective denotes that a process P contains an empty slot $\tilde{n}$. The *location adjunct* connective specifies that if the process P is placed in a slot $\tilde{n}$ it will satisfy A. The *composition adjunct* is a security connective inspired by [2] used to assert a property B of a process P in parallel with any process R satisfying A. *universal* adds the power of quantification over variables where the function $A\{^m/_x\}$ returns the formula A where all occurrences of variable x is initialized to m, where $m \in \mathcal{N}^+ \cup \overline{\mathcal{N}}$. If the initialization of x to m results in a syntactic unsound formula, the formula $\neg tt$ is returned in its place.

3.2.1 Derived Connectives

Figure 11, 12 and the following section defines and describes some useful connectives, which may be derived from the existing logic. All are derivatives found in standard logic. The *everytime* connective states that A holds for the process P in its current state and will continue to do so always.

3.2.2 Introducing The Somewhere Connective

As an alternative to introducing the ✧-connective through a definition (see Figure 10) our logic is infact strong enough to derive the connective through encoding of existing connectives. To achieve this derivation, we need some notational shorthand. We introduce $\gamma n\lfloor A \rfloor$ as defined in Figure 12 which express the existence of a slot in any context. We can then use the *existential* operator from Figure 11 to construct the formula $\exists\delta.\delta\lfloor A \rfloor \vee A$, which expresses the same as the current definition of the ✧-connective. Even though this definition is decidable, the complexity is exponential.

[1]In a closed formula there are no free variables, i.e. all variables in the formula is bound by a quantifier.

false	$P \models \mathit{ff}$	$\triangleq \neg\mathit{tt}$
conjunction	$P \models A \wedge B$	$\triangleq \neg(\neg A \vee \neg B)$
implication	$P \models A \implies B$	$\triangleq \neg A \vee B$
logical equivalence	$P \models A \iff B$	$\triangleq (A \implies B) \wedge (B \implies A)$
everytime	$P \models \Box A$	$\triangleq \neg\Diamond\neg A$
everywhere	$P \models \text{⌑} A$	$\triangleq \neg\text{✧}\neg A$
existential	$P \models \exists x.A$	$\triangleq \neg\forall x.\neg A$

Figure 11: Derived connectives

directed location	$P \models \gamma n\lfloor A \rfloor$	$\triangleq n_1\lfloor \ldots n_k\lfloor n\lfloor A\rfloor \parallel \mathit{tt}\rfloor \parallel \mathit{tt}\ldots\rfloor$ where $\gamma = n_1 \ldots n_k$
directed location adjunction	$P \models A@\gamma n$	$\triangleq A@n@n_k \ldots @n_1$ where $\gamma = n_1 \ldots n_k$

Figure 12: Notational shorthand

3.2.3 A Note on The Sometime and Somewhere Connectives

Although we refer to the sometime ($\Diamond$) connective as a temporal connective, it would be more accurate to refer to it as an alethic modality according to the definition given in [7]. Both temporal and alethic logics go beyond extensional logics and talk about other states than the current. But where alethic logics only express possibility and necessity in the future, temporal logics can express both future and past. For that reason it is more accurate to refer to $\Diamond$ as being an alethic modality with the special case that the reachability relation is defined over time.

Our logic and the $\Diamond$ connective fits in the general theory of alethic logic; the aim of the rest of this Section is to show this. As in all modal logics we need a *model* M in which we can evaluate logical expressions. By the definitions in [7] such a model consists of: 1) a *set of states* W; 2) a *reachability relation* R defined over $W \times W$; and 3) a *truth mapping* $\varphi : W \times L \mapsto \{\mathit{tt}, \mathit{ff}\}$ where L is the set of possible formulae in the logic. Evaluation of an expression $A \in L$ in the model M is based on the state $v \in W$ and we write $M \models_v A$ if $\varphi(v, A) = \mathit{tt}$. In classic logics we only talk about one state and hence we can omit W in the definition of φ. However, in alethic logics the state is important for the $\Diamond$ and $\Box$ connectives such that:

$$\begin{aligned} M \models_v \Diamond A &\triangleq \exists u \in W \text{ s.t. } M \models_u A \text{ where } (v,u) \in R \\ M \models_v \Box A &\triangleq \forall u \in W \text{ s.t. } M \models_u A \text{ where } (v,u) \in R \end{aligned}$$

Mapping this to our logic we have L to be the formulae in $\mathcal{L}_{\equiv}$; $W = \mathcal{P}$; R is the reflexive and transitive closure of $\searrow$ and φ is the satisfaction relation. Since M is fixed we alter the syntax and write $P \models A$ instead of $M \models_P A$.

The somewhere (✧) connective can be mapped in a similar way using the reflexive and transitive closure of $\downarrow$ for R.

3.3 An Example

The setting of the example is a person wanting to withdraw money from an ATM. The person is initially equipped with two pockets, one being empty and the other containing a credit card. The ATM initially contains an empty slot for inserting a credit card, an empty tray for delivering the money to the person and a slot containing the bank's money. When the ATM and the person are done interacting, the ATM no longer contains money and the person has a credit card in one pocket and money in the other pocket. The following processes model involved objects in a withdrawal.

$$
\begin{array}{lcl}
Cash & \triangleq & pay.\mathbf{0} \\
Card & \triangleq & validate.\mathbf{0} \\
ATM & \triangleq & cardslot\lfloor \bullet \rfloor \parallel moneytray\lfloor \bullet \rfloor \parallel moneysupply\lfloor Cash \rfloor \parallel \overline{pin} \\
 & & .cardslot\ \overline{validate}.moneysupply \triangleright moneytray.ready.\mathbf{0} \\
Pockets & \triangleq & pocket\lfloor left\lfloor Card \rfloor \parallel right\lfloor \bullet \rfloor \rfloor \\
Person & \triangleq & Pockets \parallel pocket\ left \triangleright cardslot.pin.\overline{ready} \\
 & & .moneytray \triangleright pocket\ right.cardslot \triangleright pocket\ left.\mathbf{0} \\
Withdraw & \triangleq & Person \parallel ATM
\end{array}
$$

Logical properties of the processes are defined as follows: $Success$ is the property of a person having both pockets full (money in one and credit card in the other). $Money\ present$ describes the property that somewhere in the system, money must be present i.e. withdrawal does not cause loss of money. $Person_{ready}$ states the property of a person is willing to insert his valid credit card into the ATM, enter a PIN-number and thereafter wait for the money. ATM_{ready} has two requirements; firstly the ATM has an initial state which must have an empty cardslot, an empty moneytray and be willing to accept a PIN-number. Secondly, an ATM must satisfy the property of having cash in its moneytray and be willing to output ready when put in parallel with any process satisfying $Person_{ready}$.

$$
\begin{array}{lcl}
Success & \triangleq & \Diamond(pocket\ left\lfloor tt \rfloor \wedge pocket\ right\lfloor tt \rfloor) \\
Money\ present & \triangleq & \Box\Diamond pay.tt \\
Person_{ready} & \triangleq & \exists validcard : pocket\ validcard \triangleright cardslot.pin.\overline{ready}.tt \\
ATM_{ready} & \triangleq & cardslot\lfloor \bullet \rfloor \parallel moneytray\lfloor \bullet \rfloor \parallel \overline{pin}.tt \\
 & & \wedge \left(Person_{ready} \parallel\!\triangleright \Diamond moneytray\lfloor pay.tt \rfloor \parallel ready\right)
\end{array}
$$

Now based on the satisfaction relation in Figure 10 we can assert whether the specified processes satisfy the logical properties or not.

$$
\begin{array}{lcl}
Withdraw & \models & Success \\
Withdraw & \models & Moneypressent \\
Person & \models & Person_{ready} \\
ATM & \models & ATM_{ready}
\end{array}
$$

Thus we have that the withdrawal process actually satisfies the properties of a successful transaction and that transactions do not cause loss of money. Likewise our instance of a person process satisfies the properties of $Person_{ready}$ which is the requirement of a person who wishes to make a withdrawal. Finally, the ATM process satisfies the ATM_{ready} property, guaranteeing that in parallel with a Person process it will result in a successful withdrawal.

4 Equivalence between $=_{\mathcal{L}_\equiv}$ and $\equiv$

In this section we show that the equivalence imposed by the logic $\mathcal{L}_\equiv$ coincides with the equivalence imposed by structural congruence. Since structural congruence is a very strong relation, this result may be unsuspected. However, because the satisfaction rules of the logic is based heavily upon structural congruence and because the logic can mimic a large part of the calculus constructions, this is not surprising.

The main result (Theorem 6) in this section is supported by two Lemmas. Lemma 4 shows the trivial result that structural congruence implies logical equivalence. Lemma 5 shows that if two processes are not structural congruent, then there exists a formula in the logic that is satisfied by one of them but not the other and hence they are also not logically equivalent.

The logic equivalence $=_{\mathcal{L}_\equiv}$ for $\mathcal{L}_\equiv$ is an equivalence relation over $\mathcal{P} \times \mathcal{P}$ and is defined as follows.

Definition 3. We write $P =_{\mathcal{L}_\equiv} Q$ if and only if P and Q satisfy exactly the same formulae, i.e. for all formulae $A \in \mathcal{L}_\equiv$:

$$
\begin{array}{l}
\text{if } P =_{\mathcal{L}_\equiv} Q \text{ and } P \models A \text{ then } Q \models A \\
\text{if } P =_{\mathcal{L}_\equiv} Q \text{ and } Q \models A \text{ then } P \models A
\end{array}
$$

Lemma 4. If $P \equiv Q$ then $P =_{\mathcal{L}_\equiv} Q$

Proof. Detailed proof is given in Section A.1.

This Lemma can be stated in another way, namely: if $P \equiv Q$ then P and Q satisfies exactly the same formulae according to Definition 3. By the symmetry of $\equiv$, i.e. $P \equiv Q \Leftrightarrow Q \equiv P$, it is sufficient to only consider the case where $P \models A$. We must prove the Lemma for all formulae in the logic and so the proof is by structural induction on the formula. For the inductive hypothesis, we assume that for processes R, S if $R \equiv S$ and $R \models B$ then $S \models B$ and in the inductive step we show that the equivalence is preserved by all structures in the logic. □

Lemma 5. If $P \not\equiv Q$ then $P \neq_{\mathcal{L}_\equiv} Q$.

Proof. Detailed proof is given in Section A.2.

To prove Lemma 5 we show, without loss of generality, that for two processes P and Q, where $P \not\equiv Q$, there exists a formula A that distinguishes P and Q, i.e. $P \models A$ and $Q \not\models A$. Since we must show this for any two processes, the proof is by structural induction on the processes. For the inductive hypothesis, we assume that for processes R, S if $R \not\equiv S$ then there exists a formula B distinguishing them and show that this is preserved in the inductive step. □

Theorem 6. $=_{\mathcal{L}_\equiv}$ and $\equiv$ coincide.

Proof. Follows directly by Lemma 4 and Lemma 5. □

5 Relation between $=_{\mathcal{L}_\equiv}$ and $\sim$

We now show the connection between $=_{\mathcal{L}_\equiv}$ and $\sim$. Theorem 6 shows that $=_{\mathcal{L}_\equiv} = \equiv$ and hence it is easy to see that we cannot have $=_{\mathcal{L}_\equiv} = \sim$.

Following the approach of [13] we look for a characterization of bisimularity in terms of $=_{\mathcal{L}_\equiv}$. However, we want a characterization of strong bisimularity from Section 2.2.3 contrary to what is done in [13] where the logic is compared to an *intensional bisimilarity* based on the reduction rules. We characterize strong bisimulation since [6] shows that this is equivalent to barbed congruence (Section 2.2.4) which is a preferred equivalence relation in ambient-based calculi.

In order to make the characterization more clear we first define a new HML-like logic $\mathcal{L}_\sim$ that by construction is closely connected to $\sim$. Because of this, it is easy to show the equivalence $=_{\mathcal{L}_\sim} = \sim$ (Theorem 10). Theorem 11 shows that $=_{\mathcal{L}_\equiv}$ implies $\sim$ and hence also $=_{\mathcal{L}_\sim}$. Proposition 14 states that $\mathcal{L}_\equiv$ is strong enough to simulate $\mathcal{L}_\sim$, i.e. that any formulae in $\mathcal{L}_\sim$ can be expressed in $\mathcal{L}_\equiv$.

5.1 The Logic $\mathcal{L}_\sim$

The syntax of $\mathcal{L}_\sim$ is defined as follows and the satisfaction relation is given in Figure 13 where a is an element from the set of labels generated by the ψ group defined in Section 2.2.3.

$$A, B ::= tt \mid A \vee B \mid A \wedge B \mid \langle a \rangle A$$

true	$P \models tt$	$\triangleq$	true for all processes P
disjunction	$P \models A \vee B$	$\triangleq$	$P \models A \vee P \models B$
conjunction	$P \models A \wedge B$	$\triangleq$	$P \models A \wedge P \models B$
possibility	$P \models \langle a \rangle A$	$\triangleq$	$\exists P'$ s.t. $P \xrightarrow{a} P'$ and $P' \models A$

Figure 13: The satisfaction relation of $\mathcal{L}_\sim$

A *possibility* expression e.g. $\langle a \rangle A$ is satisfied by a process P if it is possible for P, from its current state, to make an a-labelled transition and reach a new state P' satisfying A. Notice that *possibility* is defined using the transition rules instead of structural congruence.

We define $=_{\mathcal{L}_\sim}$ the same way as $=_{\mathcal{L}_\equiv}$, i.e. $P =_{\mathcal{L}_\sim} Q$ iff $\forall A \in \mathcal{L}_\sim : P \models A \iff Q \models A$.

Definition 7. A process P is *image finite* iff for each action a the collection $\{P' \mid P \xrightarrow{a} P'\}$ of processes reachable from P by transition a is finite. A LTS is image finite if each of its states are image finite.

Note that although a slot $n\lfloor\bullet\rfloor$ have infinitely many transitions, in that infinitely many different processes may enter it, it is still image finite as each entering af a process only leads to one state. This means that image finite is not the same as finite-state.

5.2 Equivalence Between $=_{\mathcal{L}_\sim}$ and $\sim$

Lemma 8. If $P =_{\mathcal{L}_\sim} Q$ then $P \sim Q$ for processes P and Q in an image finite labelled transition systems.

Proof. As in the proof of Theorem 4.1 in [11] we show that the relation $\mathcal{R} = \{(P, Q) \mid P =_{\mathcal{L}_\sim} Q\}$ imposed by $=_{\mathcal{L}_\sim}$ is a bisimulation, i.e. we show that if $P =_{\mathcal{L}_\sim} Q$ and $P \xrightarrow{a} P'$ then there exists Q' such that $Q \xrightarrow{a} Q'$ and $P' =_{\mathcal{L}_\sim} Q'$. Since $=_{\mathcal{L}_\sim}$ is symmetric it suffices to show that $\mathcal{R}$ is a bisimulation.

Assume for the purpose of reaching a contradiction that Q' does not exist. Since Q is image finite, there is only a finite set of possibilities for Q', say $\{Q'_1, Q'_2, .., Q'_k\}$ with $k \geq 0$ that can be reached by performing an a-labelled transition from Q. Then, by assumption, we have that $P' \neq_{\mathcal{L}_\sim} Q'_i$ for each Q'_i hence there exists a formula A_i such that $P' \models A_i$ and $Q'_i \not\models A_i$. As a result, the formula $B = \langle a\rangle A_1 \wedge \langle a\rangle A_2 \wedge .. \wedge \langle a\rangle A_k$ is satisfied by P but not by Q and it cannot be the case that $P =_{\mathcal{L}_\sim} Q$. □

Lemma 9. If $P \sim Q$ then $P =_{\mathcal{L}_\sim} Q$.

Proof. Detailed proof is given in Section A.3.

We have to show that for two processes $P \sim Q$ it is also the case that $P =_{\mathcal{L}_\sim} Q$. For $P =_{\mathcal{L}_\sim} Q$ to hold P and Q must satisfy exactly the same formulae, i.e. if $P \models A$ then $Q \models A$ for all formulae A which is sufficient since $=_{\mathcal{L}_\sim}$ is symmetric. In order to show this for all formulae, we do structural induction on the form of the formula. The inductive hypothesis for the proof is that for two processes R, S if $R \sim S$ then $R =_{\mathcal{L}_\sim} S$. □

Theorem 10. $=_{\mathcal{L}_\sim}$ and $\sim$ coincide for image finite labelled transition systems.

Proof. Follows from Lemma 8 and 9. □

5.3 Relation Between $=_{\mathcal{L}_\equiv}$ and $\sim/=_{\mathcal{L}_\sim}$

The logic $\mathcal{L}_\equiv$ can infact simulate all formulae in $\mathcal{L}_\sim$ which also means that the semantics of $\mathcal{L}_\sim$ (i.e. the satisfaction relation) can be expressed in terms of $\mathcal{L}_\equiv$. However, this may not be surprising since the transition rules are syntax-driven and Theorem 6 shows that $=_{\mathcal{L}_\equiv}$ is as strong as $\equiv$.

Theorem 11. If $P =_{\mathcal{L}_\equiv} Q$ then $P \sim Q$.

Proof. Detailed proof is given in Section A.4.

We know from Theorem 6 that $=_{\mathcal{L}_\equiv}$ and $\equiv$ coincide. We use this result and show that for two processes $P \equiv Q$, it is also the case that $P \sim Q$. For this to hold, P and Q must be able to do exactly the same transitions and still remain structual congruent, i.e. if $P \equiv Q$ and $P \xrightarrow{a} P'$ then there exists a Q' such that $Q \xrightarrow{a} Q'$ and $P' \equiv Q'$. To show this for all possible transitions we do induction on the depth of the inference of $P \xrightarrow{a} P'$. As the inductive hypothesis we have that for processes R, S if $R \equiv S$ and $R \xrightarrow{a} R'$ then $S \xrightarrow{a} S'$ such that $R' \equiv S'$. □

Corollary 12. If $P =_{\mathcal{L}_\equiv} Q$ then $P =_{\mathcal{L}_\sim} Q$.

Proof. Follows from Theorem 11 and Lemma 9. □

Corollary 13. $=_{\mathcal{L}_\equiv} \subsetneq \sim$ and $=_{\mathcal{L}_\equiv} \subsetneq =_{\mathcal{L}_\sim}$.

Proof. To prove $=_{\mathcal{L}_\equiv} \subsetneq \sim$ we have that Theorem 11 gives $=_{\mathcal{L}_\equiv} \subseteq \sim$. That it is a proper subset is shown by processes $P = a.a$ and $Q = a \parallel a$ for which $P \sim Q$ but $P \neq_{\mathcal{L}_\equiv} Q$.

For $=_{\mathcal{L}_\equiv} \subsetneq =_{\mathcal{L}_\sim}$ Corollary 12 gives $=_{\mathcal{L}_\equiv} \subseteq =_{\mathcal{L}_\sim}$. For the same P and Q Lemma 9 gives that $P =_{\mathcal{L}_\sim} Q$. As $P \neq_{\mathcal{L}_\equiv} Q$ we get $=_{\mathcal{L}_\equiv} \subsetneq =_{\mathcal{L}_\sim}$. □

We now focus on the encoding of formulae from $\mathcal{L}_\sim$ in terms of formulae from $\mathcal{L}_\equiv$. More precisely we want a mapping $\varphi : \mathcal{L}_\sim \to \mathcal{L}_\equiv$ such that for any formula $A \in \mathcal{L}_\sim$ we have $P \models_\sim A$ iff $P \models_\equiv \varphi(A)$. This would allow us to add the *possibility* connective to our logic and allow us to apply the model checker for $\mathcal{L}_\equiv$ to $\mathcal{L}_\sim$.

Proposition 14. There is an encoding $\varphi : \mathcal{L}_\sim \to \mathcal{L}_\equiv$ such that for all $A \in \mathcal{L}_\sim$ we have $P \models_\sim A$ if and only if $P \models_\equiv \varphi(A)$.

Proof. Detailed proof in Section A.5.

We must show that given any formula A from $\mathcal{L}_\sim$ there exists a formula A' from $\mathcal{L}_\equiv$ such that $P \models_\sim A$ iff $P \models_\equiv A'$. The proof idea is based on the fact that by knowing that a transition is possible tells something about the process structure, since the SOS transition rules are syntax-driven. Similar, by knowing the process structure it is possible to tell which transitions are possible. □

Notice that this result gives an alternative proof that $=_{\mathcal{L}_\equiv}$ implies $=_{\mathcal{L}_\sim}$ and is similar to the method used for the converse of Theorem 4.1 in [13].

6 Model Checking

We present a model checking algorithm for a decidable sub-logic which asserts the truth of input P, A where P is a process description in MR and A is a formula in $\mathcal{L}_\equiv$. Inspired by the initial design of model checking algorithm for the Mobile Ambients logic in [2] and the results from [5] our algorithm requires formula A to be $\|\triangleright$ (*composition adjunct*) free and process P to be finite-state. In [5] it is proven that the algorithm for the Mobile Ambients logic cannot be extended to include $\|\triangleright$ as this would lead to undecidability. While we have no proof that this result will hold for a model checking algorithm for $\mathcal{L}_\equiv$ this is likely since both calculi and logics are very similar.

A finite-state process can be guaranteed through syntactic restrictions such as disallowing the use of the replication operator ! (bang) in MR which is the approach taken here. However, in [4] it is shown that by introducing a type system it is possible to allow (some) replication in processes as long as they are typable and still achieve finite-state processes. We expect that a similar approach using a type system is applicable for MR but this is left for future work.

A proof-of-concept implementation of the algorithm has been developed in Ruby and is available at *http://www.cs.aau.dk/˜crt/2006/MR/modelchecker/*.

6.1 Prime Product Normal Form

As input, the algorithm requires the process to be converted into a structural equivalent process on *Prime Product Normal Form* (PPNF) (see Definition 15), in which a process is presented as a finite product of prime processes. Although this normal form is required by the algorithm presented here, this conversion and construction of a suitable AST[2] should be provided by a parser.

Definition 15. Process products are on the form: $\prod_{i\in 1..k} P_i \triangleq P_1 \parallel \ldots \parallel P_k \parallel \mathbf{0}$.
A prime process is a process on the form given by the basic calculus syntax of Figure 1 such that:

$$P ::= \gamma\alpha.P' \mid \delta \triangleright \overline{\delta'}.P' \mid \natural m.P' \mid \tilde{n}\lfloor\bullet\rfloor_m \mid \tilde{n}\lfloor P'\rfloor_m \mid \mathbf{0}$$

A process P is on PPNF if it does not contain nested composite processes. Thus a process on PPNF is a product $\prod_{i\in 1..k} P_i$ where P_i are prime processes and the subprocesses P' of P_i are either prime processes or process products.

As an example a process $P = P_1 \parallel P_2$ where $P_2 = P_3 \parallel P_4$ must be written $P = P_1 \parallel P_3 \parallel P_4$ where $P_{1,3,4}$ are prime processes. Thus P does not have nested composite processes.

[2]Abstract Syntax Tree

6.2 The Algorithm

The algorithm described in Figure 14 checks that a process P satisfies the closed formula A, this is done using recursion in a divide and conquer manner. The algorithm depends on the following:

To calculate the result of $\diamond$ (sometime) we define the function *reachable(P)* which returns the set $\mathcal{R}_P = \{P_1 \ldots P_k\}$ representing all reachable states, in one reduction, of the process P given the reduction rules from Section 2.2.1 such that if and only if $P \searrow Q$ then $Q \equiv P_i$ for some $P_i \in \mathcal{R}_P$.

Likewise for the ✧ (somewhere) computation, we define the function *sublocation(P)* which returns the set $\mathcal{S}_P = \{n_1 \ldots n_k\}$ representing all, possibly nested, slot processes in P.

Finally, for parallel composition we define the function $list(P)$ such that $list(P) = \{\mathbf{0}, P_1, \ldots, P_k\} \iff P \equiv \prod_{i \in 1..k} P_i$ and $list(P) = \{P\}$ for all prime processes.

Quantification is done over the domain $\mathcal{M}^+$ for which we define $\mathcal{M} = fn(P) \cup fn(A) = \{m_1, \ldots, m_k\} \wedge m_0 \notin \{m_1, \ldots, m_k\}$. Thus only relevant strings are considered. The proof in Section A.7 discusses this further.

The algorithm uses the truth value of the $\equiv$ relation which may in constant time assert the cases: $P \equiv \mathbf{0}$, $P \equiv \lambda.P'$, $P \equiv \tilde{n}\lfloor \bullet \rfloor_{n'}$ and $P \equiv \tilde{n}\lfloor P' \rfloor_{n'}$.

$$
\begin{array}{lcl}
check(P, tt) & \triangleq & tt \\
check(P, \mathbf{0}) & \triangleq & P \equiv \mathbf{0} \\
check(P, \neg A) & \triangleq & \neg check(P, A) \\
check(P, A \vee B) & \triangleq & check(P, A) \vee check(P, B) \\
check(P, \lambda.A) & \triangleq & P \equiv \lambda.P' \wedge check(P', A) \\
check(P, \tilde{m}\lfloor \bullet \rfloor) & \triangleq & P \equiv \tilde{n}\lfloor \bullet \rfloor_{n'} \text{ where } \tilde{m} \subseteq \tilde{n} \\
check(P, \tilde{m}\lfloor A \rfloor) & \triangleq & P \equiv \tilde{n}\lfloor P' \rfloor_{n'} \wedge check(P', A) \text{ where } \tilde{m} \subseteq \tilde{n} \\
check(P, A@\tilde{m}) & \triangleq & check(P', A) \text{ where } P' \equiv \tilde{n}\lfloor P \rfloor_{n'} \text{ for some } \tilde{n} \text{ such that } \tilde{m} \subseteq \tilde{n} \\
check(P, \forall x.A) & \triangleq & \displaystyle\bigwedge_{m_s \in \mathcal{M}^+} check(P, A\{^{m_s}/_x\}) \text{ where } |m_s| \leq \text{ the number of slots in } P \\
check(P, \diamond A) & \triangleq & check(P, A) \vee \displaystyle\bigvee_{P_i \in \mathcal{R}_P} check(P_i, \diamond A) \text{ where } \mathcal{R}_P = reachable(P) \\
check(P, \text{✧}A) & \triangleq & \displaystyle\bigvee_{P_i \in \mathcal{S}_P} check(P_i, A) \text{ where } \mathcal{S}_P = sublocation(P) \\
check(P, A \parallel B) & \triangleq & \text{let } list(P) = \{P_0, ..., P_k\} \displaystyle\bigvee_{\forall I, J} \Big(check(\prod_{i \in I} P_i, A) \wedge check(\prod_{j \in J} P_j, B) \Big) \\
 & & \text{where } I \cup J = \{0..k\} \wedge I \cap J = \emptyset
\end{array}
$$

Figure 14: The model checking algorithm

6.3 Decidability and Correctness

Termination of the algorithm can be guaranteed since all recursive calls meet the following criteria; either calls are on subprocesses of the original input guaranteeing a reduction in the input and inevitable termination. Alternatively the call explores the state space of the process P which is finite due to the fact that P is finite-state.

Although this algorithm is decidable it requires P-SPACE. The *sublocation()* function can be implemented as a simple search through the process structure, registering any subprocess structural equivalent to a slot process. The disjoined assertions used in the algorithm for asserting satisfaction of parallel formulae are also decidable, but asserting all possible subsets of a parallel process with a composite formulae requires $O(2^n)$ assertions, where n is the length of the input. Recursively exploring the state-space of a process using recursive calls to *reachable*(P) has the complexity $O(n!)$. A discussion of how to manage and reduce this complexity is not within the scope of this project, but a simple practical approach, also used in our implementation, is to ensure short circuiting of disjointed and conjoined assertions.

Lemma 16. *check*$(P, A) = tt$ if and only if $P \models A$

Proof. Detailed proof in Section A.7.

The proof for Lemma 16 is by structural induction in the formula A, we prove that the algorithm computes the correct result (*check*$(P, A) = tt$) exactly when $P \models A$. □

7 Related Works

Our approach was influenced by the work of Luca Cardelli and Andrew D. Gorden in [1] along with the works of Davide Sangiorgi in [13] regarding a logic for the Mobile Ambient calculus [2]. Cardelli and Gordon propose a logic which is able to express spatial structure as well as time in mobility models. The work of Sangiorgi shows that the logic allows us to observe the internal structure of the processes at a very fine-grained detail.

In [4] a type system for Mobile Ambients is developed that allows for a less restricted approach for having finite-state processes than by removal of the replication construct. [10] introduces a type system for the Mobile Resources calculus inspired by the type system for Mobile Ambients. It could be investigated whether this type system could be used for the purpose of extending the model checking algorithm to typable processes with (some) replication.

The calculus of Higher-order Mobile Embedded Resources (Homer) [9] is a pure (non-linear) higher-order calculus with mobile computing processes in nested locations. The calculus is defined such that it has very simple syntax and semantics, which conservatively extend the standard syntax and semantics of process passing calculi. The nested names as location addresses, as found in nested name spaces of distributed systems, were introduced in the predecessor of Homer, the Mobile Resources calculus [6].

8 Conclusions and Future Work

We have created a logic $\mathcal{L}_{\equiv}$ based on structual congruence and proved that the equivalence imposed by this coincides with the equivalence imposed by structual congruence (Theorem 6). To be able to show the coherence between bisimulation and $\mathcal{L}_{\equiv}$ we introduce a secondary logic $\mathcal{L}_{\sim}$ based on labelled transition rules. We have shown that the equivalence imposed by $\mathcal{L}_{\sim}$ coincides with bisimulation for image finite labelled transition systems (Theorem 10) and that $\mathcal{L}_{\equiv}$ can characterize $\mathcal{L}_{\sim}$. Finally, we present a model checking algorithm and a proof-of-concept implementation for a decidable fragment of $\mathcal{L}_{\equiv}$ without composition adjunct and replication.

Future work for $\mathcal{L}_{\equiv}$ might include extending it with name restriction along the lines of [3, 5]. That particular approach would require the reintroduction of name restriction in the calculus, alpha-conversion in the structural equivalence relation and the addition of a connective such as Ⓡ (and its adjunct ⃠) used to derive the restriction quantifier. Based on the results from [5] this approach would allow the model checking algorithm to be extended as well without any increase in complexity.

$\mathcal{L}_{\sim}$ and the characterization of it in $\mathcal{L}_{\equiv}$ could be extended to include all connectives found in Hennessy-Milner-logic [8]; simply adding the complete set of connectives would not effect $\mathcal{L}_{\sim}$ coherence with bisimularity but would require an extensive extension of the characterization. Also, it would be interesting to examine if the connectives from $\mathcal{L}_{\sim}$ could be added to $\mathcal{L}_{\equiv}$ as derived connectives, i.e. what happens when these connectives are mixed.

In [4] it is shown that by introducing a type system it is possible to allow (some) replication in processes as long as they are typable and still achieve finite-state-processes. We expect that a similar approach using a type-system could be applicable in MR, to allow the use of replication with the model checking algorithm.

Future work might also include the development of a logic for the calculus of Higher-order Mobile Embedded Resources (Homer) [9], the successor of the Mobile Resources calculus. We suspect that elements from $\mathcal{L}_{\equiv}$ could be used in the development of a logic for Homer.

References

[1] Luca Cardelli and Andrew D. Gordon. Anytime, anywhere: Modal logics for mobile ambients. In *POPL*, pages 365–377, 2000.

[2] Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Theor. Comput. Sci.*, 240(1):177–213, 2000.

[3] Luca Cardelli and Andrew D. Gordon. Logical properties of name restriction. pages 46–60, 2001.

[4] Witold Charatonik, Andrew D. Gordon, and Jean-Marc Talbot. Finite-control mobile ambients. In Daniel Le Métayer, editor, *ESOP*, volume 2305 of *Lecture Notes in Computer Science*, pages 295–313. Springer, 2002.

[5] Witold Charatonik and Jean-Marc Talbot. The decidability of model checking mobile ambients. In Laurent Fribourg, editor, *CSL*, volume 2142 of *Lecture Notes in Computer Science*, pages 339–354. Springer, 2001.

[6] Jens Chr. Godskesen, Thomas T. Hildebrandt, and Vladimiro Sassone. A calculus of mobile resources. In Lubos Brim, Petr Jancar, Mojmír Kretínský, and Antonín Kucera, editors, *CONCUR*, volume 2421 of *Lecture Notes in Computer Science*, pages 272–287. Springer, 2002.

[7] Vincent F. Hendricks and Stig Andur Pedersen. *Moderne elementær logik*.

[8] Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *J. ACM*, 32(1):137–161, 1985.

[9] Thomas Hildebrandt, Jens Chr. Godskesen, and Mikkel Bundgaard. Bisimulation congruences for Homer — a calculus of higher order mobile embedded resources. Technical Report ITU-TR-2004-52, IT University of Copenhagen, Rued Langgaards Vej 7, DK-2300 Copenhagen S, Denmark, November 2004.

[10] Morten Kühnrich and Hans Hüttel. Types for access control in a calculus of mobile resources. 2005.

[11] Kim G. Larsen Luca Aceto, Anna Ingólfsdóttir and Jiri Srba. *Reactive Systems: Modelling, Specification and Verification*.

[12] Robin Milner. *Communicating and Mobile Systems: the pi-calculus*.

[13] Davide Sangiorgi. Extensionality and intensionality of the ambient logics. pages 4–13, 2001.

A Proofs

A.1 Proof for Lemma 4

Lemma 4. If $P \equiv Q$ then $P =_{\mathcal{L}_\equiv} Q$.

Proof. If it is the case that $P \equiv Q$, by Definition 3, we must show that for all formulae $A \in \mathcal{L}_\equiv$ it is also the case that $P \models A \iff Q \models A$. We do this by structural induction in the formula A using as inductive hypothesis that for processes R, S if $R \equiv S$ then $R =_{\mathcal{L}_\equiv} S$. Assume that P satisfies the formula for the following cases:

tt : by definition of satisfaction we have $Q \models \mathit{tt}$ trivially.

$\mathbf{0}$: by definition of satisfaction $P \equiv \mathbf{0}$ so by definition of structural congruence we get that $P = \mathbf{0}$. Since $P \equiv Q$ the same goes for Q and hence $Q \models \mathbf{0}$.

$\neg A$: by definition of satisfaction we have that $P \models \neg A$ implies $\neg(P \models A)$. Since $Q \equiv P$ the inductive hypothesis gives that if $Q \models A$ then $P \models A$; since this is not the case we get $\neg(Q \models A)$ and finally $Q \models \neg A$.

$A \vee B$: by definition of satisfaction $P \models A$ or $P \models B$. By the inductive hypothesis we get that either $Q \models A$ or $Q \models B$ which gives that $Q \models A \vee B$.

$\tilde{m}\lfloor A \rfloor$: by definition of satisfaction $P \equiv \tilde{n}\lfloor P' \rfloor_{n'}$ and $P' \models A$ for $\tilde{m} \subseteq \tilde{n}$ and some n'. Since $P \equiv Q$ the definition of structural congruence (property E_7 in Section 2.1.2) gives that $Q \equiv \tilde{n}\lfloor Q' \rfloor_{n'}$ and that $P' \equiv Q'$. By the inductive hypothesis $Q' \models A$ and hence $Q \models \tilde{m}\lfloor A \rfloor$.

$\tilde{m}\lfloor \bullet \rfloor$: by definition of satisfaction $P \equiv \tilde{n}\lfloor \bullet \rfloor_{n'}$ for $\tilde{m} \subseteq \tilde{n}$ and some n'. Since $P \equiv Q$ the definition of structural congruence gives that $Q \equiv \tilde{n}\lfloor \bullet \rfloor_{n'}$ and so $Q \models \tilde{m}\lfloor \bullet \rfloor$.

$A_1 \parallel A_2$: by definition of satisfaction $P \equiv P_1 \parallel P_2$ where $P_1 \models A_1$ and $P_2 \models A_2$. If Q is on the form $Q_1 \parallel Q_2$ it follows by the difinition of structural congruence (E_9) that $P_1 \equiv Q_1$ and $P_2 \equiv Q_2$ and so the inductive hypothesis gives that $Q_1 \models A_1$ and $Q_2 \models A_2$ and hence $Q \models A_1 \parallel A_2$. If, however, Q is not on that form (allowed by structural congruence for $p \parallel \mathbf{0} \equiv p$) we can first transform Q into a suitable form using the rules of structural congruence and get the same result.

$\Diamond A$: by definition of satisfaction there exists P' such that $P \longrightarrow^* P'$ where $P' \models A$. Since $P \equiv Q$ we have by Lemma 11 that there exists Q' such that $Q \longrightarrow^* Q'$ and $P' \equiv Q'$. By the inductive hypothesis $Q' \models A$ and hence $Q \models \Diamond A$.

$A@\tilde{m}$: by definition of satisfaction there exists P' such that $P' \equiv \tilde{n}\lfloor P \rfloor_{n'}$ and $P' \models A$ for $\tilde{m} \subseteq \tilde{n}$ and some n'. Since $\equiv$ is a process congruence and hence preserves contexts, we have that $P' \equiv \tilde{n}\lfloor P \rfloor_{n'} \equiv \tilde{n}\lfloor Q \rfloor_{n'}$ and so there exists $Q' \equiv \tilde{n}\lfloor Q \rfloor_{n'}$ with $Q' \equiv P'$. By the inductive hypothesis we get that $Q' \models A$ and so $Q \models A@\tilde{m}$.

$A \parallel\!\triangleright B$: by definition of satisfaction we get that $R \parallel P \models B$ for any process $R \models A$. By definition of structural congruence (E_6) we have that $R \parallel P \equiv R \parallel Q$ for all R and thus by inductive hypothesis that $R \parallel P \models B$ implies $R \parallel Q \models B$. This shows that $Q \models A \parallel\!\triangleright B$.

$\lambda.A$: by definition of satisfaction there exists P' such that $P \equiv \lambda.P'$ and $P' \models A$. Since $P \equiv Q$ we have by the definition of structural congruence (E_8) that there exists Q' such that $Q \equiv \lambda.Q'$ and $P' \equiv Q'$. By inductive hypothesis $Q' \models A$ and so $Q \models \lambda.A$.

$\forall x.A$: by definition of satisfaction a process P satisfies $\forall x.A$ iff P satisfies A for all values of the variable x. Assume for the purpose of reaching a contradiction, that Q $\not\models \forall x.A$ which implies that $\exists m.A\{^m/_x\} = A_m$ where $Q \not\models A_m$ and $P \models A_m$. However, since $P \equiv Q$ the inductive hypothesis gives that $P =_{\mathcal{L}_\equiv} Q$ contradicting our assumption.

□

A.2 Proof for Lemma 5

First we need another Lemma allowing us to concentrate only on the special case where two processes are obviously not structural congruent. The Lemma shows that the general case holds by applying the rules of structural equivalence.

Lemma 17. $P \not\equiv Q$ if and only if there exists a context $\mathbb{C}$ such that $P \equiv \mathbb{C}[P']$, $Q \equiv \mathbb{C}[Q']$ and $P' \not\equiv Q'$ where $\mathbb{C}$ is an arbitrary, possibly empty, process context.

Proof. Follows directly from Definition 3 of structural congruence. □

Lemma 5. If $P \not\equiv Q$ then $P \neq_{\mathcal{L}_\equiv} Q$.

Proof. We assume that $P \not\equiv Q$ and so, by Definition 3, we show that it is not the case that for all formulae A' we have $P \models A'$ and $Q \models A'$; i.e. that there exists a formula A such that $P \models A$ and $Q \not\models A$.

We must show this for all processes and so the proof is by structural induction in the process. As the inductive hypothesis we use that for processes R, S if $R \not\equiv S$ then there exists a formula B such that $R \models B$ and $S \not\models B$.

$\mathbf{0}$: for processes $P \equiv \mathbf{0}$ and $Q \not\equiv \mathbf{0}$ and by the formula $A = \mathbf{0}$ we have that $P \models A$ and $Q \not\models A$.

$\tilde{n}\lfloor R \rfloor_{n'}$: for $P \equiv \tilde{n}\lfloor R \rfloor_{n'}$ we have that either Q is not a slot or $Q \equiv \tilde{m}\lfloor S \rfloor_{m'}$. In the first case the formula $A = \tilde{n}\lfloor tt \rfloor$ distinguishes P and Q; in the second case we have to examine $\tilde{m}$, m' and S:

- because of content, i.e. $R \not\equiv S$: by the inductive hypothesis we have that there exists a B such that $R \models B$ and $S \not\models B$ and hence $A = \tilde{n}\lfloor B \rfloor$.
- because of slot names: if $\tilde{n} \subset \tilde{m}$ then $A = \neg\tilde{m}\lfloor tt \rfloor$; if $\tilde{n} \supset \tilde{m}$ then $A = \tilde{n}\lfloor tt \rfloor$; or if $\tilde{n} \cap \tilde{m} = \emptyset$ then $A = \tilde{n}\lfloor tt \rfloor$.
- because of delete name, i.e. $n' \neq m'$: since slots have no delete name in the logic we use the $\|\triangleright$ and $\diamond$ operators; for $A = \natural n'.\mathbf{0} \|\triangleright \diamond \mathbf{0}$ we get that $P \models A$ and $Q \not\models A$ since $\tilde{n}\lfloor R \rfloor_{n'} \parallel \natural n'.\mathbf{0} \searrow \mathbf{0}$.

$\tilde{n}\lfloor \bullet \rfloor_{n'}$: similar to the case where $P = \tilde{n}\lfloor R \rfloor_{n'}$ but using $A = \tilde{n}\lfloor \bullet \rfloor$ in the case where $\tilde{n} = \tilde{m}$.

$\gamma\alpha.P'$: once again we have to consider two cases: either Q is not on the correct form or $Q \equiv \gamma\alpha.Q'$ but $P' \not\equiv Q'$. For the first case we choose $A = \gamma\alpha.tt$. For the second, the inductive hypothesis gives that there exists a B such that $P' \models B$ and $Q' \not\models B$; in this case we find satisfaction in $A = \gamma\alpha.B$.

$\delta \triangleright \overline{\delta'}.P'$: similar to the case where $P = \gamma\alpha.P'$ but using $\delta \triangleright \overline{\delta'}$ for A instead.

$\natural m.P'$: similar to the case for $P = \gamma\alpha.P'$ but using $\natural m$ for A instead.

$P_1 \parallel P_2$: we assume that $P \equiv P_1 \parallel P_2$ and consider $Q \not\equiv P$. By the rules of structural congruence this implies that either 1) there does *not* exists Q_1, Q_2 such that $Q \equiv Q_1 \parallel Q_2$ (i.e. Q does not have the correct syntax) or 2) there exists Q_1, Q_2 such that $Q \equiv Q_1 \parallel Q_2$ but $P_1 \not\equiv Q_1$ or $P_2 \not\equiv Q_2$.

- in case of 1) the formula $A = tt \parallel tt$ is enough.
- in case of 2) the inductive hypothesis gives that there exists A_1 and A_2 where $P_1 \models A_1 \wedge P_2 \models A_2$ but $Q_1 \not\models A_1 \vee Q_2 \not\models A_2$, in which case $P \models A$ but $Q \not\models A$ where $A = A_1 \parallel A_2$.

$!R$: we assume that $P \equiv\, !R$ and $Q \not\equiv P$. Since the logic does not have a connective to match ! directly we need to consider the following cases: 1) Q is a slot or a prefix or 2) there exists Q_1, Q_2 such that $Q \equiv Q_1 \parallel Q_2$ but $Q_1 \not\equiv R$ or $Q_2 \not\equiv R$ or 3) $Q \equiv R\,[\parallel R]^k$ for some constant k or 4) $Q \equiv\, !S$ but $S \not\equiv R$.

- case 1) and 2) are trivial
- in case 3) there exists a k such that a formula A can be constructed to require more parallel processes than the number contained in Q: $A = \neg\mathbf{0}\,[\parallel \neg\mathbf{0}]^k$.
- in case 4) by the inductive hypothesis there exists a formula B such that $R \models B$ and $S \not\models B$. Hence $P \models A$ and $Q \not\models A$ for $A = B \parallel tt$.

$\square$

A.3 Proof for Lemma 9

Lemma 9. If $P \sim Q$ then $P =_{\mathcal{L}_\sim} Q$.

Proof. We assume that $P \sim Q$ and show that then it holds that $P =_{\mathcal{L}_\sim} Q$. For $P =_{\mathcal{L}_\sim} Q$ to hold, P and Q must satisfy exactly the same formula, i.e. if $P \models A$ then $Q \models A$ for all formula A, which is sufficient since $=_{\mathcal{L}_\sim}$ is symmetric. In order to show this for all formulas we do structural induction on the form of the formula, having as inductive hypothesis that for two processes R, S if $R \sim S$ then $R =_{\mathcal{L}_\sim} S$.

tt : $Q \models tt$ trivially.

$A \vee B$: by definition of satisfaction this implies that either $P \models A$ or $P \models B$. If $P \models A$ the inductive hypothesis gives that $Q \models A$ and similar if $P \models B$ then $Q \models B$. So either $Q \models A$ or $Q \models B$ and hence $Q \models A \vee B$.

$A \wedge B$: by definition of satisfaction this implies that $P \models A$ and $P \models B$. By the inductive hypothesis we get that $Q \models A$ and $Q \models B$ and hence $Q \models A \wedge B$.

$\langle a \rangle A$: by definition of satisfaction $P \models \langle a \rangle A$ implies that there exists P' such that $P \xrightarrow{a} P'$ and $P' \models A$. By assumption we have that $P \sim Q$ and so there exists Q' such that $Q \xrightarrow{a} Q'$ and $P' \sim Q'$. By the inductive hypothesis we get that $Q' \models A$ and hence $Q \models \langle a \rangle A$.

$\square$

A.4 Proof for Lemma 11

Lemma 11. If $P =_{\mathcal{L}_\equiv} Q$ then $P \sim Q$.

The proof is inspired by a similar result (Proposition 5.2) from [12]. In the proof a is an element from the set of actions generated by the ψ.

Proof. We assume $P =_{\mathcal{L}_\equiv} Q$ (and so $P \equiv Q$ follows from Theorem 6) and show that it is also the case that $P \sim Q$. For this to hold, P and Q must be able to do exactly the same transitions and still remain structual congruent, i.e. if $P \equiv Q$ and $P \xrightarrow{a} P'$ then there exists a Q' such that $Q \xrightarrow{a} Q'$ and $P' \equiv Q'$.

To show this for all possible transitions we do induction on the depth of the inference of $P \xrightarrow{a} P'$. It is clearly enough to prove the result in the special case that the congruence $P \equiv Q$ is due to a single application of a structural congruence rule; the general case follows just by iterating the special case.

As the inductive hypothesis we have that for processes R, S if $R \equiv S$ and $R \xrightarrow{a} R'$ then $S \xrightarrow{a} S'$ such that $R' \equiv S'$. As the proofs are very simple and the result well-known, only partial proofs for the *sync*, *par* and *sym* transition rules are shown.

sync : we have $P = a.P_1 \parallel \overline{a}.P_2$, $P \xrightarrow{\tau} P'$ and $P' = P_1 \parallel P_2$. One possibility for Q is by use of rule E_2 (commutativity) so that $Q = \overline{a}.P_2 \parallel a.P_1$. In this case the *sym* rule gives that $Q \xrightarrow{\tau} Q'$ and we have that $P' \equiv Q'$.

par : we have $P = P_1 \parallel \mathbf{0}$, $P \xrightarrow{a} P'$ and $P' = P_1' \parallel \mathbf{0}$. We can use rule E_1 and have $Q = P_1$. By the inductive hypothesis we get that $P_1 \xrightarrow{a} P_1'$ and as a result $Q \xrightarrow{a} Q'$ and $Q' \equiv P'$.

sym : we have $P = P_1 \parallel P_2$, $P \xrightarrow{a} P'$ and $P' = P_1' \parallel P_2$. We can have $Q = P_2 \parallel P_1$ and so the *sym* rule gives that $Q \xrightarrow{a} Q'$ and $Q' \equiv P'$.

□

A.5 Encoding of $\mathcal{L}_\sim$ in $\mathcal{L}_\equiv$

We look for a mapping $\varphi : \mathcal{L}_\sim \rightarrow \mathcal{L}_\equiv$ such that given a formula $A \in \mathcal{L}_\sim$ we have $P \models_\sim A$ if and only if $P \models_\equiv \varphi(A)$. The idea is to utilize that the SOS transition rules are syntax-driven and hence that if a process P is able to do a transition it reveals information about the structure of P.

To prove that $\varphi(\cdot)$ is correct we would need to consider two directions. For the first direction we assume that $P \models_\sim A$ and show that this enables us to create a set of minimum structural requirements needed for P to perform the required transitions and that these requirements are expressed by the formula $\varphi(A)$. For the second direction we show that the structural requirements expressed in $\varphi(A)$ is enough to guarantee that for any process P such that $P \models_\equiv \varphi(A)$ we have that P can perform the required transition, i.e. that $P \models_\sim A$.

As noted, the algorithm to perform the mapping must first collect the structural requirements imposed by satisfying a formula $A \in \mathcal{L}_\sim$ and based on these requirements it must generate a formula $\varphi(A)$. In Section A.5.1 we discuss the requirements imposed by a single transition and how these requirements are expressed in $\mathcal{L}_\equiv$. How these requirements are combined to generate the complete formula $\varphi(A)$ are done in two sections: in Section A.5.2 we give an algorithm for simple formulae[3] and in Section A.5.3 we show example translations for complex formulae. We consider the algorithm for simple formulae and the provided translations enough to argue the existence of a full algorithm. However, since we do not provide a full algorithm we do not prove any of the directions mentioned above.

A.5.1 Requirements

The requirements revealed by being able to perform a single transition are summarized in Figure 15. The first column contains the name of the transition in $\mathcal{L}_\sim$ and the second contains the syntax. The third column gives the structural requirements imposed by observing the transition and is dictated by the transition rules from Section 2.2.2. The macros defined in Figure 16 are used to improve readability.

A.5.2 Mapping of Simple Formulae

A formula such as $\langle a\rangle\langle b\rangle\langle c\rangle tt$ is satisfied by several different non-structural congruent processes therefore the function *mapsimple* shown in Algorithm 1 returns a set of all possible cases (up to $\equiv$). If these cases are *or*-ed together we have an implementation of $\varphi(\cdot)$ for simple formulae.

The first parameter for *mapsimple* is a logical formula from $\mathcal{L}_\sim$ and the second parameter is a parallel composition of variables corresponding to each term in the formula. For *mapsimple*$(\langle a\rangle\langle b\rangle tt,\ A_1 \parallel A_2)$ we have two terms in the formula, hence also two parallel variables. The result is therefore $\{(a.b.tt \parallel tt),\ (a.tt \parallel b.tt)\}$ and $\varphi(\langle a\rangle\langle b\rangle tt) = (a.b.tt \parallel tt) \vee (a.tt \parallel b.tt)$.

[3] We say that a formula is *simple* when it only consists of *possibility* terms of actions not inside slots.

Label	$\mathcal{L}_\sim$	$\mathcal{L}_\equiv$
delete	$\langle \natural m \rangle$	$delete(m)$
action	$\langle \gamma\alpha \rangle$	$action(\gamma\alpha)$
enter	$\langle (P) \triangleright \delta \rangle$	$enter(\delta)$
move	$\langle \delta_1 \triangleright \overline{\delta_2} \rangle$	$move(\delta_1, \delta_2)$
exit	$\langle (\mathscr{C}_\gamma)\overline{\delta'} \triangleright (\mathscr{D}_\delta) \rangle$	$exit(\delta')$
coaction	$\langle \overline{\delta}\alpha \rangle$	$coaction(\delta\alpha)$
give	$\langle \triangleright\overline{\delta}(\mathscr{D})_\delta \rangle$	$give(\delta)$
comove	$\langle \overline{\delta_1} \triangleright \delta_2 \rangle$	$comove(\delta_1, \delta_2)$
take	$\langle \delta \triangleright (P) \rangle$	$take(\delta)$
tau	$\langle \tau \rangle$	$\diamondsuit A$, where $A =$
		$\exists\delta_1, \delta_2 \ . \ comove(\delta_1, \delta_2) \wedge move(\delta_1, \delta_2)$
		$\vee \ \exists\delta \ . \ give(\delta) \wedge enter(\delta)$
		$\vee \ \exists\delta \ . \ take(\delta) \wedge exit(\delta)$
		$\vee \ \exists m \ . \ codelete(m) \wedge delete(m)$
		$\vee \ \exists a \ . \ coaction(a) \wedge action(a) \vee \exists\delta \ . \ coaction(\delta a) \wedge \delta\lfloor action(a) \rfloor$

Figure 15: Requirements on the structure of a process imposed by the possibility of performing a transition

Macro name	Encoding in $\mathcal{L}_\equiv$
$delete(m)$	$\natural m.tt$
$codelete(\alpha)$	$\natural\alpha.\mathbf{0} \parallel\!\triangleright\!\Diamond\mathbf{0}$
$move(\delta_1, \delta_2)$	$\delta_1 \triangleright \overline{\delta_2}.tt$
$action(\gamma\alpha)$	$\gamma\alpha.tt$
$coaction(\delta\alpha)$	$\delta\lfloor \alpha.tt \rfloor$
$fullslot(\delta)$	$\delta\lfloor tt \rfloor$
$emptyslot(\delta)$	$\delta\lfloor \bullet \rfloor$
$exit(\delta)$	$fullslot(\delta)$
$enter(\delta)$	$emptyslot(\delta)$
$give(\delta)$	$\exists\delta' \ . \ exit(\delta') \wedge move(\delta', \delta)$
$comove(\delta_1, \delta_2)$	$exit(\delta_1) \wedge enter(\delta_2)$
$take(\delta)$	$\exists\delta' \ . \ move(\delta, \delta') \wedge enter(\delta')$

Figure 16: Macros for process structure requirements

The *mapsimple* function makes use of other functions, but since they are trivial no formal definition is needed:

$$head\big(\langle a\rangle\langle b\rangle\langle c\rangle tt\big) = \langle a\rangle$$
$$tail\big(\langle a\rangle\langle b\rangle\langle c\rangle tt\big) = \langle b\rangle\langle c\rangle tt$$

head returns the first term in the formula and *tail* returns all but the first term in the formula.

$$expand\big(\langle a\rangle,\ A_1,\ A_1 \parallel A_2 \parallel A_3\big) = a.A_1' \parallel A_2 \parallel A_3$$

expand replaces A_1 in formula $A_1 \parallel A_2 \parallel A_3$ with the formula that would satisfy $\langle a\rangle$.

A.5.3 Mapping of Complex Formulae

We give manual example mappings for complex formulae but omit examples for *delete*, *action* and *move* as they can only be observed if located at the root of the slot hierarchy, hence they are trivial.

In the following examples the first column shows the formula from $\mathcal{L}_\sim$ written from top to bottom. The second column states the requirement each connective in the formula imposes on the process. Sometimes a requirement may be satisfied by a previous term in the formula which has rearranged resources. E.g. in the case of moves; a $\overline{n} \triangleright m$ term may allow later requirements of the contents of slot m to be redirected to the original contents of n. We denote such shifts in requirements as $n\lfloor\bullet\rfloor \mapsto m\lfloor\bullet\rfloor$. The third column (to the right of the line) shows the resulting formula in $\mathcal{L}_\equiv$ with each line *or*-ed together.

coaction $\langle\overline{\delta}\alpha\rangle$: when these actions are observed it is because there exists a process inside the slot located at δ willing to perform action α. In the first example we have the formula $\langle\overline{n}a\rangle\langle\overline{m}b\rangle$ and we see that the

Algorithm 1 $mapsimple(A_\sim, formula)$

```
  $term \leftarrow head(A_\sim)$
  if $term = \mathit{tt}$ then
    replace all variables in $formula$ with $\mathit{tt}$ and return
  else
    for each variable in $formula$ do
      add $mapsimple\big(tail(A_\sim),\ expand(term,\ variable,\ formula)\big)$ to $list$
    end for
    remove all duplicates up to $\equiv$ from $list$
    return $list$
  end if
```

first and second connective have requirement $n\lfloor a \rfloor$ and $m\lfloor b \rfloor$ respectively. The resulting formula in $\mathcal{L}_\equiv$ is $n\lfloor a \parallel \mathit{tt} \rfloor \parallel m\lfloor b \parallel \mathit{tt} \rfloor \parallel \mathit{tt}$.

$A \in \mathcal{L}_\sim$		$\varphi(A) \in \mathcal{L}_\equiv$
$\langle \overline{n}a \rangle$	$n\lfloor a \rfloor$	$n\lfloor a \parallel \mathit{tt} \rfloor \parallel m\lfloor b \parallel \mathit{tt} \rfloor \parallel \mathit{tt}$
$\langle \overline{m}b \rangle$	$m\lfloor b \rfloor$	
tt		

The next example is similar except that now there are two requirements for slot n. As seen this yields three different process configurations.

$A \in \mathcal{L}_\sim$		$\varphi(A) \in \mathcal{L}_\equiv$
$\langle \overline{n}a \rangle$	$n\lfloor a \rfloor$	$n\lfloor a \parallel \mathit{tt} \rfloor \parallel n\lfloor b \parallel \mathit{tt} \rfloor \parallel \mathit{tt}$
$\langle \overline{n}b \rangle$	$n\lfloor b \rfloor$	$n\lfloor a.b \parallel \mathit{tt} \rfloor \parallel \mathit{tt}$
tt		$n\lfloor a \parallel b \parallel \mathit{tt} \rfloor \parallel \mathit{tt}$

enter $\langle (P) \triangleright \delta \rangle$: with the *enter* examples we see how requirements can be passed on: the b moved into slot m by $\langle (b) \triangleright m \rangle$ could be the b satisfying the requirement imposed on m. We denote this $m\lfloor b \rfloor \mapsto m\lfloor \bullet \rfloor$ since m has to be empty for the *enter* to take place.

$A \in \mathcal{L}_\sim$		$\varphi(A) \in \mathcal{L}_\equiv$
$\langle \overline{n}a \rangle$	$n\lfloor a \rfloor$	$n\lfloor a \parallel \mathit{tt} \rfloor \parallel m\lfloor b \parallel \mathit{tt} \rfloor \parallel m\lfloor \bullet \rfloor \parallel \mathit{tt}$
$\langle (b) \triangleright m \rangle$	$m\lfloor \bullet \rfloor$	$n\lfloor a \parallel \mathit{tt} \rfloor \parallel m\lfloor \bullet \rfloor \parallel \mathit{tt}$
$\langle \overline{m}b \rangle$	$m\lfloor b \rfloor \mapsto m\lfloor \bullet \rfloor$	
tt		

comove $\langle \overline{\delta_1} \triangleright \delta_2 \rangle$: here we have that the requirement on slot n m can be passed on to slot a m because $\langle \overline{a} \triangleright n \rangle$ could move a slot m from a into n satisfying the requirement imposed by $\langle \overline{b} \triangleright nm \rangle$.

$A \in \mathcal{L}_\sim$		$\varphi(A) \in \mathcal{L}_\equiv$
$\langle \overline{a} \triangleright n \rangle$	$a\lfloor \mathit{tt} \rfloor, n\lfloor \bullet \rfloor$	$a\lfloor \mathit{tt} \rfloor \parallel n\lfloor \bullet \rfloor \parallel b\lfloor \mathit{tt} \rfloor \parallel n\lfloor m\lfloor \bullet \rfloor \rfloor \parallel \mathit{tt}$
$\langle \overline{b} \triangleright nm \rangle$	$b\lfloor \mathit{tt} \rfloor, n\lfloor m\lfloor \bullet \rfloor \rfloor \mapsto a\lfloor m\lfloor \bullet \rfloor \rfloor$	$a\lfloor m\lfloor \bullet \rfloor \rfloor \parallel n\lfloor \bullet \rfloor \parallel b\lfloor \mathit{tt} \rfloor \parallel \mathit{tt}$
tt		

In the next example the content of slot n is moved into a slot m, so the requirement put on m by $\langle \overline{m}b \rangle$ can be passed on to n, yielding many different cases.

$A \in \mathcal{L}_\sim$		$\varphi(A) \in \mathcal{L}_\equiv$
$\langle \overline{n}a \rangle$	$n\lfloor a \rfloor$	$n\lfloor a \rfloor \parallel m\lfloor \bullet \rfloor \parallel m\lfloor b \rfloor \parallel \mathit{tt}$
$\langle \overline{n} \triangleright m \rangle$	$n\lfloor \mathit{tt} \rfloor, m\lfloor \bullet \rfloor$	$n\lfloor a.b \rfloor \parallel m\lfloor \bullet \rfloor \parallel \mathit{tt}$
$\langle \overline{m}b \rangle$	$m\lfloor b \rfloor \mapsto n\lfloor b \rfloor$	$n\lfloor a.b \rfloor \parallel m\lfloor \bullet \rfloor \parallel m\lfloor b \rfloor \parallel \mathit{tt}$
tt		$n\lfloor a \parallel b \rfloor \parallel m\lfloor \bullet \rfloor \parallel \mathit{tt}$
		$n\lfloor a \parallel b \rfloor \parallel m\lfloor \bullet \rfloor \parallel m\lfloor b \rfloor \parallel \mathit{tt}$
		$n\lfloor a \rfloor \parallel n\lfloor b \rfloor \parallel m\lfloor \bullet \rfloor \parallel \mathit{tt}$
		$n\lfloor a \rfloor \parallel n\lfloor b \rfloor \parallel m\lfloor \bullet \rfloor \parallel m\lfloor b \rfloor \parallel \mathit{tt}$

exit $\langle \overline{\delta} \triangleright \rangle$: here $n\lfloor \mathit{tt} \rfloor \mapsto n\lfloor \bullet \rfloor$ denotes that slot n could be empty after $\langle \overline{n} \triangleright \rangle$; if this is the case then requirement $n\lfloor b \rfloor$ has to be satisfied by another slot n.

$A \in \mathcal{L}_\sim$		$\varphi(A) \in \mathcal{L}_\equiv$
$\langle \overline{n}a \rangle$	$n\lfloor a \rfloor$	$n\lfloor a \rfloor \parallel n\lfloor b \rfloor \parallel \mathit{tt}$
$\langle \overline{n}\rhd \rangle$	$n\lfloor \mathit{tt} \rfloor \mapsto n\lfloor \bullet \rfloor$	$n\lfloor a.b \rfloor \parallel n\lfloor \mathit{tt} \rfloor \parallel \mathit{tt}$
$\langle \overline{n}b \rangle$	$n\lfloor b \rfloor$	$n\lfloor a \parallel b \rfloor \parallel n\lfloor \mathit{tt} \rfloor \parallel \mathit{tt}$
tt		

take $\langle \delta \rhd (P) \rangle$: a *take* transition is possible when there exists a *move* and an *enter* and so we "unfold" $\langle \delta \rhd (P) \rangle$ to $\langle \delta \rhd \delta' \rangle \langle (P) \rhd \delta' \rangle$:

$A \in \mathcal{L}_\sim$		$\varphi(A) \in \mathcal{L}_\equiv$	
$\langle ab \rhd \overline{\delta} \rangle$	$ab \rhd \overline{\delta}$	$\exists^* \delta . c\lfloor a \rfloor \parallel ab \rhd \overline{\delta} \parallel \delta\lfloor \bullet \rfloor \parallel \mathit{tt}$	if $\delta = c$
$\langle (a.b) \rhd \delta \rangle$	$\delta\lfloor \bullet \rfloor$	$ab \rhd \overline{c} \parallel c\lfloor \bullet \rfloor \parallel \mathit{tt}$	if $\delta \neq c$
$\langle \overline{c}a \rangle$	$c\lfloor a \rfloor \mapsto \delta\lfloor a \rfloor$	$c\lfloor a \rfloor \parallel ab \rhd \overline{c} \parallel c\lfloor \bullet \rfloor \parallel \mathit{tt}$	if $\delta \neq c$
tt			

give $\langle \rhd \overline{\delta} \rangle$: a *give* transition is possible when there exists an *exit* and a *move* and so we "unfold" $\langle \rhd \overline{\delta} \rangle$ to $\langle \overline{\delta'} \rhd \rangle \langle \delta' \rhd \delta \rangle$.

$A \in \mathcal{L}_\sim$		$\varphi(A) \in \mathcal{L}_\equiv$	
$\langle \delta' \rhd \rangle$	$\delta' \lfloor \mathit{tt} \rfloor$	$c\lfloor \mathit{tt} \rfloor \parallel c\lfloor d \rfloor \parallel c \rhd \overline{ab} \parallel \mathit{tt}$	if $\delta = c$
$\langle \delta' \rhd \overline{ab} \rangle$	$\delta' \rhd \overline{ab}$	$\delta\lfloor \mathit{tt} \rfloor \parallel c\lfloor d \rfloor \parallel \delta \rhd \overline{ab} \parallel \mathit{tt}$	if $\delta \neq c$
$\langle \overline{c}d \rangle$	$c\lfloor d \rfloor$		
tt			

$A \in \mathcal{L}_\sim$		$\varphi(A) \in \mathcal{L}_\equiv$	
$\langle \overline{c}d \rangle$	$c\lfloor d \rfloor$	$c\lfloor d \rfloor \parallel c \rhd \overline{ab} \parallel \mathit{tt}$	if $\delta = c$
$\langle \delta' \rhd \rangle$	$\delta' \lfloor \mathit{tt} \rfloor$	$c\lfloor d \rfloor \parallel \delta\lfloor \mathit{tt} \rfloor \parallel \delta \rhd \overline{ab} \parallel \mathit{tt}$	if $\delta \neq c$
$\langle \delta' \rhd \overline{ab} \rangle$	$\delta' \rhd \overline{ab}$		
tt			

or $A \vee B$: because of the nature of *or* not all requirements have to be satisfied. This is shown in the case of $\langle b \rangle \mathit{tt} \vee \langle c \rangle \mathit{tt}$ where only one of them are necessary; again we denote this $b \mapsto c$.

$A \in \mathcal{L}_\sim$		$\varphi(A) \in \mathcal{L}_\equiv$
$\langle a \rangle$	a	$a.(b \parallel c) \parallel \mathit{tt}$
$(\langle b \rangle \mathit{tt} \vee \langle c \rangle \mathit{tt})$	$b \mapsto c$	$a.b \parallel \mathit{tt}$
		$a.c \parallel \mathit{tt}$
		$a \parallel b \parallel \mathit{tt}$
		$a \parallel c \parallel \mathit{tt}$

and $A \wedge B$: in *and* all requirements must be satisfied and so $\langle b \rangle \mathit{tt} \wedge \langle c \rangle \mathit{tt}$ requires both a b and a c.

$A \in \mathcal{L}_\sim$		$\varphi(A) \in \mathcal{L}_\equiv$
$\langle a \rangle$	a	$a.(b \parallel c) \parallel \mathit{tt}$
$(\langle b \rangle \mathit{tt} \wedge \langle c \rangle \mathit{tt})$	b, c	$a \parallel b \parallel c \parallel \mathit{tt}$
		$a.b \parallel c \parallel \mathit{tt}$
		$a.c \parallel b \parallel \mathit{tt}$

A.6 Example of three-way synchronization

The following shows an example deriviation of a three-way synchronization.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{f\lfloor a \rfloor \xrightarrow{\overline{f}\rhd\langle a \rangle} f\lfloor \bullet \rfloor}\ \mathit{exit}}{e\lfloor f\lfloor a \rfloor \rfloor \xrightarrow{\overline{ef}\rhd\langle a \rangle} e\lfloor f\lfloor \bullet \rfloor \rfloor}\ \mathit{nesting} \qquad \dfrac{}{ef \rhd \overline{cd} \xrightarrow{ef\rhd\overline{cd}} \mathbf{0}}\ \mathit{prefix}}{e\lfloor f\lfloor a \rfloor \rfloor \parallel ef \rhd \overline{cd} \xrightarrow{\langle a \rangle\rhd\overline{cd}} e\lfloor f\lfloor \bullet \rfloor \rfloor}\ \mathit{give} \qquad \dfrac{\dfrac{}{d\lfloor \bullet \rfloor \xrightarrow{(a)\rhd d} d\lfloor a \rfloor}\ \mathit{enter}}{c\lfloor d\lfloor \bullet \rfloor \rfloor \xrightarrow{(a)\rhd cd} c\lfloor d\lfloor a \rfloor \rfloor}\ \mathit{nesting}}{e\lfloor f\lfloor a \rfloor \rfloor \parallel ef \rhd \overline{cd} \parallel c\lfloor d\lfloor \bullet \rfloor \rfloor \xrightarrow{\tau} e\lfloor f\lfloor \bullet \rfloor \rfloor \parallel c\lfloor d\lfloor a \rfloor \rfloor}\ \mathit{sync}$$

A.7 Proof for Modelchecking Algorithm

Lemma 16 $\mathit{check}(P, A) = \mathit{tt}$ if and only if $P \models A$

Proof. By structural induction in the formula A we prove that the algorithm computes the correct result such that if $P \models A$ exactly then $\mathit{check}(P, A) = \mathit{tt}$, which is our inductive hypothesis.

Base:

$A = \mathit{tt}$ then $P \models A$ and $\mathit{check}(P, A) = \mathit{tt}$.
$A = \mathbf{0}$ and $P \models A$ then $\mathit{check}(P, A) = \mathit{tt}$ Because $P \equiv \mathbf{0}$.
$A = \tilde{n}\lfloor\bullet\rfloor$ and $P \models A$ then $\mathit{check}(P, A) = \mathit{tt}$ Because $P \equiv \tilde{n}\lfloor\bullet\rfloor_{n'}$.

Inductive step: For each remaining connective in $\mathcal{L}_{\equiv}$ we assume the correct behaviour of $\mathit{check}(P, A)$ and show that $\mathit{check}(P, A')$ where A' is an extention of A by exactly one connective is computed correctly:

case: $A' = A_1 \parallel A_2$ and $P \models A'$ then by definition $\exists P_1, P_2$ such that $P \equiv P_1 \parallel P_2$ and $P_1 \models A_1 \wedge P_2 \models A_2$. Using that $R \parallel S \equiv \prod_{i \in 1..k} P_i$ iff $\exists I, J$ where $I \cup J = \{0..k\} \wedge I \cap J = \emptyset$ such that $R \equiv \prod_{i \in I} P_i \wedge S \equiv \prod_{j \in J} P_j$ $\mathit{check}(P, A_1 \parallel A_2)$ will construct and return the disjointed result of $R \models A_1 \wedge S \models A_2$ for all possible R and S. Thus $\mathit{check}(P, A_1 \parallel A_2) = \mathit{tt}$ if and only if $P \models A_1 \parallel A_2$.

case: $A' = \forall x.A$ and $P \models A'$ then by definition $P \models A\{^m/_x\}$ $\forall m \in \mathcal{N}^+ \cup \mathcal{A}$. $\mathit{check}(P, \forall x.A)$ implements this behavior asserting the conjunction of iterative replacement of the variable x with $m_s \in \mathcal{M}^+$, where $\mathcal{M} = (\{m_1, \ldots, m_k\} = \text{fn(P)} \cup \text{fn(A)}$ and $m_0 \notin \{m_1, \ldots, m_2\})$ for m_s where $|m_s| \leq$ number of slots in P. This is justified because replacement of x with any two names z, y where $z \neq y \wedge z, y \notin \text{fn(P)} \cup \text{fn(A)}$, will yield exactly the same result, since neither may coescale with pre-existing names in the expressions, we need only to test one (m_0). Furthermore strings longer than the number of slots in P will not coescale with anything, thus we need not test these, again the pressence of m_0 ensures we test atleast one string which does not exist. Thus $\mathit{check}(P, \forall x.A)$ will return tt exactly when $P \models \forall x.A$.

case: $A' = \Diamond A$ and $P \models A'$ then by definition there exists a P' s.t. $P \searrow_* P'$ and $P' \models A$. $\mathit{check}(P, \Diamond A)$ will assert the truth of P satisfying A at the current level in P's "tree" of possible reductions if this is not the case it will recursively call $\mathit{check}(P'', \Diamond A)$ for all possible P'', reachable in one reduction as defined by the reduction rules. Thus $\mathit{check}(P, \Diamond A)$ must reach a call where $P'' = P'$ which by definition satisfies A.

case: $A' = ✧A$ and $P \models A'$ then by definition there exists a P' s.t. $P \downarrow^* P'$ and $P' \models A$. $\mathit{check}(P, ✧A)$ will compute this by identifying the set $\mathcal{S}_p$ containing all sublocations of P which must include P'. $\mathit{check}(P, ✧A)$ will then return the disjointed answer of each recursive call to $\mathit{check}(R, A)$ $\forall R \in \mathcal{S}_p$. The call to $\mathit{check}(R, A)$ where $R = P'$ will eventually return *true* yielding $\mathit{check}(P, ✧A) = \mathit{tt}$.

the remaining cases are trivially shown and therefore omitted. □