Appendix of DimSum: A Decentralized Approach to Multi-language Semantics and Verification

MICHAEL SAMMLER, MPI-SWS, Germany
SIMON SPIES, MPI-SWS, Germany
YOUNGJU SONG, MPI-SWS, Germany
EMANUELE D’OSUALDO, MPI-SWS, Germany
ROBBERT KREBBERS, Radboud University Nijmegen, The Netherlands
DEEPAK GARG, MPI-SWS, Germany
DEREK DREYER, MPI-SWS, Germany

A COMBINATORS

Filter. The filter combinator \( M \setminus \sigma M' \equiv (S_{\text{filter}} \times S_M \times S_{M'}, \rightarrow_{\text{filter}}, (\sigma, \sigma^0_M, \sigma^0_{M'})) \) takes in a module \( M \in \text{Module}(E_1) \) and a filter \( M' \in \text{Module}(\text{FilterEvents}(E_1, E_2)) \) and then produces a module with events drawn from \( E_2 \). The states of the filter combinator are given by \( S_{\text{filter}} \equiv \{P, F\} \cup \{P(e) \mid e \in E_1\} \cup \{F(e) \mid e \in E_1\} \) and the transitions are depicted in Fig. 1. The events of the filter module are drawn from the following set:

\[
\text{FilterEvents}(E_1, E_2) \equiv \{\text{FromInner}(e_1) \mid e_1 \in E_1\} \cup \{\text{ToInner}(e_1) \mid e_1 : \text{option}(E_1)\} \cup \\
\{\text{ToEnv}(e_2) \mid e_2 : E_2\} \cup \{\text{FromEnv}(e_2) \mid e_2 : E_2\}
\]

The event FromInner(e_1) means that \( M' \) is willing to accept \( e_1 \) from \( M \). The event ToInner(e_1) means that \( M' \) wants to return control to the module \( M \), optionally sending it the event \( e_1 \). Sending \( e_1 \) to \( M \) means that \( M \) will all visible transitions of the inner module \( M \) except ones emitting event \( e_1 \) are blocked. The event ToEnv(e_2) means that \( M' \) wants to emit \( e_2 \) to the environment, and FromEnv(e_2) means that \( M' \) is willing to accept \( e_2 \) from the environment. Note that, while there is a difference between the intuition for ToEnv(e_2) and FromEnv(e_2), both events are treated the same by \( M \setminus M' \) as DimSum does not distinguish between incoming and outgoing events.

Linking. The linking operator \( M_1 \otimes_X M_2 \) is defined on modules \( M_1, M_1 \in \text{Module}(E_{E_3}) \) where \( E_{E_3} \) is (an event type that is isomorphic to) \( E \times \{?, !\} \). The parameter \( X = (S, \sim \sim, s^0) \) determines how the events are linked. It consists of a set of linking-interal states \( S \), an initial state \( s^0 \in S \), and a relation \( \rightarrow_{\sim \sim} \subseteq (D \times S \times E) \times ((D \times S \times E) \cup \{\} \} \) describing how events should be translated. Formally, linking can be defined as \( M_1 \otimes_X M_2 \equiv M_1 \times M_2 \downarrow_{\text{link}_X} \). The module link_X is defined as \( \text{link}_X \equiv (S_{\text{link}} \times S_X, \rightarrow_{\text{link}}, (\text{Wait}, s^0_X)) \) where \( S_{\text{link}} \equiv \{\text{Wait}, \text{Ub}\} \cup \{\text{ToEnv}(e, \sigma), \text{FromEnv}(e, \sigma) \mid e \in E_{E_3}, \sigma \in S_{\text{link}}\} \cup \{\text{ToInner}(e) \mid e \in \text{option}(E_{E_3})\} \) and \( \rightarrow_{\text{link}} \) is defined in Fig. 2.

\[ \text{Authors’ addresses: Michael Sammler, MPI-SWS, Saarland Informatics Campus, Germany, msammler@mpi-sws.org; Simon Spies, MPI-SWS, Saarland Informatics Campus, Germany, spies@mpi-sws.org; Youngju Song, MPI-SWS, Saarland Informatics Campus, Germany, youngju@mpi-sws.org; Emanuele D’Osualdo, MPI-SWS, Saarland Informatics Campus, Germany, dosualdo@mpi-sws.org; Robbert Krebbers, Radboud University Nijmegen, The Netherlands, mail@robbertkrebbers.nl; Deepak Garg, MPI-SWS, Saarland Informatics Campus, Germany, dg@mpi-sws.org; Derek Dreyer, MPI-SWS, Saarland Informatics Campus, Germany, dreyer@mpi-sws.org.} \]
Where the filter module is given by the following, some

\[ \begin{align*}
\text{filter-step-prog} & \quad \sigma = F \land \sigma = F(e) \\
\text{\hspace{1cm}} & \quad \sigma_2 \xrightarrow{e} \Sigma \\
\text{filter-step-prog-recv} & \quad \sigma_1 \xrightarrow{\sigma} \Sigma \\
\text{\hspace{1cm}} & \quad (P(e), \sigma_1, \sigma_2) \xrightarrow{r_{\text{filter}}} \{ (P(e), \sigma'_1, \sigma_2) \mid \sigma'_1 \in \Sigma \} \\
\text{filter-step-prog-from-inner} & \quad \sigma_2 \xrightarrow{\text{FromInner}(e)} \Sigma \\
\text{\hspace{1cm}} & \quad (F(e), \sigma_1, \sigma_2) \xrightarrow{r_{\text{filter}}} \{ (F(e), \sigma'_1, \sigma_2) \mid \sigma'_1 \in \Sigma \} \\
\text{filter-step-prog-to-inner} & \quad \sigma_2 \xrightarrow{\text{ToInner}(e)} \Sigma \\
\text{\hspace{1cm}} & \quad (F, \sigma_1, \sigma_2) \xrightarrow{r_{\text{filter}}} \{ (F, \sigma_1, \sigma'_2) \mid \sigma'_2 \in \Sigma \} \\
\text{filter-step-prog-env} & \quad \sigma_2 \xrightarrow{\text{FromEnv}(e)} \Sigma \\
\text{\hspace{1cm}} & \quad (F, \sigma_1, \sigma_2) \xrightarrow{r_{\text{filter}}} \{ (F, \sigma_1, \sigma'_2) \mid \sigma'_2 \in \Sigma \} \\
\end{align*} \]

is a definition of \( \xrightarrow{e} \).

\[ \begin{align*}
\text{(Kripke) wrappers.} & \quad \text{The combinator}[M]_X\text{ translates a module with events } E_1 \text{ to a module with events } E_2. \text{ This combinator is parametrized by } X = (S, R, \leftarrow, \rightarrow, s^0, F^0) \text{ where } S \text{ is a set of states and } s^0 \text{ is an initial state (} R \text{ is explained below). These states were omitted in the main paper for simplicity. They do not give additional expressive power but make writing the wrapper \([\_]_{r:=a} \) more pleasant. The relations \( \leftarrow \) and \( \rightarrow \) describe how the wrapper transforms the incoming and outgoing events. Concretely, \( \leftarrow \) describes how to translate an event } e_1 \in E_2 \text{ to an event } e_1 \in E_1 \text{ and } \rightarrow \text{ describes the translation from } e'_1 \in E_1 \text{ to } e'_2 \in E_2. \text{ As mentioned in the paper, these relations are separation logic relations. Which separation logic the relations are defined in is determined by the parameter } X \text{ of the wrapper. In the paper, it contains an arbitrary separation logic } L \text{ as one of its components. However, for our instantiations of the wrapper, we are only interested in instances of the separation logic Iris [Jung et al. 2015]. Thus, instead of an arbitrary separation logic } L, \text{ we parameterize the wrapper by a resource algebra } R \text{ and use the separation logic } L = UPred(R) \text{ where } UPred(R) = \text{Iris’s logic of uniform predicates [Jung et al. 2018]. The separation logic relations } \leftarrow \text{ and } \rightarrow \text{ are of type } E_1 \times S \times E_2 \rightarrow UPred(R). \text{ The proposition } F^0 : UPred(R) \text{ denotes the initial set of resources owned by the wrapper. We define } [M]_X \triangleq M \uplus \left[ \text{wrap}(s^0, F^0) \right]_X \text{ where the filter module is given by the following Spec program:}^2
\begin{align*}
\text{wrap}(s_2, F_2) & \triangleq \text{coind} \\
\exists e_2; \text{vis(FromEnv}(e_2)); \forall e_1, s_1, F_1; \text{assume(sat}(F_1 \ast F_2 \ast (e_1, s_1) \leftarrow (e_2, s_2))); \text{vis(ToInner}(e_1)); \\
\exists e'_1; \text{vis(FromInner}(e'_1)); \exists e'_2, s'_2, F'_2; \text{assert(sat}(F_1 \ast F_2 \ast (e'_1, s_1) \rightarrow (e'_2, s'_2))); \text{vis(ToEnv}(e'_2)); \\
\text{wrap}(s'_2, F'_2)
\end{align*} \]
The instruction \textit{syscall} does a syscall and then executes \textit{c}. The instruction \textit{upd}(x, r, v); c updates

\footnote{The Coq development defines an equivalent module directly using a step relation, but we give the definition here using \textit{Spec} for readability.}
\[ \text{Instr} \ni c \triangleq \text{syscall}; c | \text{upd}(x, r, v); c | \text{ldr}(x_1, x_2, v, v'); c | \text{str}(x_1, x_2, v, v'); c | \text{jump} \]

Fig. 3. Micro-Instructions of \textit{Asm}

\[
\begin{align*}
\text{ASM-LINK-JUMP} & \\
(d' = L \land r(pc) \in d_1) & \lor (d' = R \land r(pc) \in d_2) & \lor (d' = E \land r(pc) \notin d_1 \cup d_2) & d \neq d' \\
(d, \text{None}, \text{Jump}(r, m)) & \leadsto_{d_1, d_2} (d', \text{None}, \text{Jump}(r, m))
\end{align*}
\]

\[
\begin{align*}
\text{ASM-LINK-SYSCALL} & \\
(d, \text{None}, \text{Syscall}(v_1, v_2, m)) & \leadsto_{d_1, d_2} (E, \text{Some}(d), \text{Syscall}(v_1, v_2, m))
\end{align*}
\]

\[
\begin{align*}
\text{ASM-LINK-SYSCALL-RETURN} & \\
(d', \text{None}, \text{SyscallRet}(v, m)) & \leadsto_{d_1, d_2} (d', \text{None}, \text{SyscallRet}(v, m))
\end{align*}
\]

Fig. 4. Definition of semantic linking relation \(\leadsto\) for \textit{Asm}.

the register \(x\) according to the map \(r \mapsto v\) applied to the current register values \(r\) and then executes \(c\). The instruction \(\text{ldr}(x_1, x_2, v, v'); c\) takes the value stored in \(x_2\), applies the transformation \(v \mapsto v'\) to it to obtain an address, stores the result in \(x_1\), and then executes \(c\). The instruction \(\text{ldr}(x_1, x_2, v, v'); c\) takes the value stored in \(x_2\), applies the transformation \(v \mapsto v'\) to it to obtain an address, stores in the memory at that address the value in \(x_1\), and then executes \(c\). The instruction \(\text{jump}\) reads the \(pc\) register and then jumps to the address stored there.

The reason for the micro instruction representation is that we can represent a large instruction set by chaining few primitives. For example, the instructions used in \textit{print} and \textit{locle} are derived as follows:

\[
\begin{align*}
\text{ret} & \triangleq \text{upd}(pc, r, (x30)); \text{jump} \\
\text{syscall} & \triangleq \text{syscall}; \text{next} \\
\text{mov} x, v & \triangleq \text{upd}(x, r, v); \text{next} \\
\text{sle} x_1, x_2, x_3 & \triangleq \text{upd}(x_1, r, \text{if } r(x_2) \leq r(x_3) \text{ then } 1 \text{ else } 0); \text{next}
\end{align*}
\]

where we abbreviate \(\text{next} \triangleq \text{upd}(pc, r, (pc + 1)); \text{jump}\).

C \hspace{1em} \text{SEMANTIC LINKING FOR} \hspace{1em} \text{Asm}

The full definition of the semantic linking relation \(\leadsto\) for \textit{Asm} can be found in \textit{Fig. 4}. Compared to the excerpt shown in the paper, it contains two additional cases, \textit{ASM-LINK-SYSCALL} and \textit{ASM-LINK-SYSCALL-RETURN}. The rule \textit{ASM-LINK-SYSCALL} makes sure syscalls are passed on to the environment (and never come from the environment). When a syscall is triggered, we store the current turn \(d\) in the private state of the linking operator. This way, we can make sure that when we return from a syscall (\textit{ASM-LINK-SYSCALL-RETURN}), the execution continues with the module that triggered the syscall.

D \hspace{1em} \text{Rec}

The language \textit{Rec} is a simple, high-level language with arithmetic operations, let bindings, memory operations, conditionals, and (potentially recursive) function calls (depicted in \textit{Fig. 5}). The libraries \(R\) of \textit{Rec} are lists of function declarations. Each function declaration contains the name of the function \(f\), the argument names \(\overline{x}\), local variables \(\overline{y}\) which are allocated in the memory, and a
function body $e$. The set of function names $|R|$ of a library $R$ is defined as the names of the functions in the list $R$.

**Module semantics.** The semantics of a Rec library $R$ is the module $\llbracket R \rrbracket_R$. The states of the module are of the from $\sigma = (E, m, R)$ where $E$ is the current runtime expression (explained below). We write $(\rightarrow_r)$ for the transition system (shown in Fig. 6) and the initial state is $(\text{Wait}(\text{false}), \emptyset, R)$.

To define the transition relation $\rightarrow_r$, we extend the static expressions $e$ to runtime expressions $E$, which have operations for allocating and deallocating stack frames as well as two distinguished expressions $\text{Ret}(b, E)$ and $\text{Wait}(b)$. These expressions are used to control when the module emits call and return events: Initially, the module is waiting and willing to accept any incoming call to the functions of the library (see rec-start). Once it starts, the function call is wrapped in the $\text{Ret}(b, \cdot)$ expression to ensure an event is emitted after the function finishes executing (see rec-ret-return). A call to functions of the library (see rec-call-internal), will trigger the allocation of the local variables and, subsequently, the execution of the function body. A call to an external function (see rec-call-external) will emit a Call!($f, \vec{v}, m$) and proceed to the waiting state. The flag for the waiting becomes true, because the module is now willing to accept a return to the function call that was just issued (see rec-ret-incoming). The language Rec is an evaluation-context based language, meaning reductions can happen inside of an arbitrary evaluation context (see rec-eval-ctx). The definition of the evaluation contexts $K$ can be found in the Coq development [Sammler et al. 2023].

**Linking.** Syntactically, linking of two Rec libraries (i.e., $R_1 \cup_r R_2$) denotes merging the function definitions in $R_1$ and $R_2$. In case of overlapping function names, the function declaration of the left library is chosen. (This choice is arbitrary.) If we semantically link two Rec modules (i.e., $M_1^{d_1, a_1} \oplus_r^{d_2, a_2} M_2$), then we have to synchronize based on the function call and return events. To define the linking $M_1^{d_1, a_1} \oplus_r^{d_2, a_2} M_2$, we use the combinator $M_1 \oplus_r X M_2$. In the case of Rec, we pick the relation $R$ depicted in Fig. 7. The most interesting difference to Asm is that linking in Rec has to build up and then wind down a call-stack, which is maintained as the internal state of $(\rightsquigarrow)$.

### E $\llbracket \cdot \rrbracket_{r=a}$ WRAPPER

Before we can give the definition of the wrapper $\llbracket \cdot \rrbracket_{r=a}$, we first need to describe its full form: $\llbracket M \rrbracket_{r=a}^{d, a, m}$. In particular, the wrapper is parametrized by a mapping $a_-$ from Rec function names to Asm addresses, by the instruction address of the Asm code $d$, by the function names of the Rec code $d$, and by a (fragment of) the initial memory $m$, which can be used for global variables.

To define the wrapper, we pick a suitable flavor of separation logic. Instead of directly presenting the technical details of the resource algebra that we choose for $R_{r=a}$, we instead describe the connectives of the resulting separation logic:

- $p \leftrightarrow v$ states that the Rec block id $p$ is mapped to Asm address $v$. We lift this relation to locations by $l \leftrightarrow v_2 \triangleq \exists v_1. l.\text{blockid} \leftrightarrow v_1 \cdot v_2 = v_1 + l.\text{offset}$ and to values (i.e., $v \leftrightarrow v$) by

```
Fig. 5. Grammar of Rec.
```

```
\[
\text{REC-BINOP} \quad (v_1 \oplus v_2, m, R) \xrightarrow{r} \{ (v, m, R) \mid \text{eval}_{\oplus}(v_1, v_2) \}
\]

\[
\text{REC-LOAD} \quad (!v_1, m, R) \xrightarrow{r} \{ (v_2, m, R) \mid \exists \ell. v_1 = \ell \land m(\ell) = v_2 \}
\]

\[
\text{REC-STORE} \quad (v_1 \leftarrow v_2, m, R) \xrightarrow{r} \{ (v_2, m[R = v_1], R) \mid \exists \ell. v_1 = \ell \land \text{heap_alive}(m, \ell) \}
\]

\[
\text{REC-IF} \quad (\text{if } v \text{ then } e_1 \text{ else } e_2, m, R) \xrightarrow{r} \{ (e, m, R) \mid \exists b. v = b \land \text{if } e = e_1 \text{ else } e = e_2 \}
\]

\[
\text{REC-LET} \quad (\text{let } x := v \text{ ine}, m, R) \xrightarrow{r} \{ (e[v/x], m, R) \}
\]

\[
\text{REC-VAR} \quad (x, m, R) \xrightarrow{r} \emptyset
\]

\[
\text{REC-ALLOC} \quad \text{heap_alloc_list}(\overline{n}, \overline{\ell}, m_1, m_2)
\]

\[
\text{alloc}(\overline{y}, n) e, m_1, R) \xrightarrow{r} \{ (\text{free_frame}(\overline{\ell}, n)(e[\overline{\ell}/\overline{y}])(m_2, R)) \mid \forall m \in \overline{n}. m > 0 \}
\]

\[
\text{REC-FREE} \quad \text{free_frame}(\overline{\ell}, n) v, m_1, R) \xrightarrow{r} \{ (v, m_2, R) \mid \text{heap_free_list}((\overline{\ell}, n), m_1, m_2) \}
\]

\[
\text{REC-START} \quad f \in R
\]

\[
\text{Wait}(b), m, R) \xrightarrow{\text{Call}^f(\overline{\ell}, n')} \{ \text{Ret}(b, f(\overline{v})), m', R) \}
\]

\[
\text{REC-CALL-INTERNAL} \quad \text{fn } f(\overline{x}) \triangleq \text{local } y[n]; e) \in R
\]

\[
(f(\overline{v}), m, R) \xrightarrow{r} \{ (\text{alloc}(\overline{y}, n)(e[\overline{\ell}/\overline{x}])(m, R)) \mid |\overline{x}| = |\overline{v}| \}
\]

\[
\text{REC-CALL-EXTERNAL} \quad f \notin R
\]

\[
(f(\overline{v}), m, R) \xrightarrow{\text{Call}(f, \overline{v}, m)} \{ \text{Wait}(true, m, R) \}
\]

\[
\text{REC-RET-RETURN} \quad (\text{Ret}(b, v), m, R) \xrightarrow{\text{Return}(v, m')} \{ \text{Wait}(b, m, R) \}
\]

\[
\text{REC-EVAL-CTX} \quad (E, m, R) \xrightarrow{\alpha, \Sigma} \{ (K[E'], m', R') \mid (E', m', R') \in \Sigma \}
\]

Fig. 6. Operational semantics of Rec.

relating Rec integers with the same integer in Asm and Boolean values with 0 and 1. The definition of \( v \leftrightarrow v \) corresponds to \( v \sim_w v \) in the main paper.
Appendix of DimSum: A Decentralized Approach to Multi-language Semantics and Verification

This separation logic is used to define the relations (→) and (←) (depicted in Fig. 8) that are used in the definition of [·]_{r⇒a}. Note that these definitions build on the definition of the Kripke wrapper in Appendix A as they maintain the state s for tracking the call stack in addition to the

- v₁ ↦ₐ v₂ asserts ownership of the address v₁ in Asm memory m and asserts that it contains the value v₂. The v₁ ↦ₐ v₂ connective is useful for asserting private ownership of Asm memory in assembly libraries (e.g., it is used internally by the coroutine library to manage its global state).
- p ↦ₐ V where V is a map from offsets to values asserts that the block with id p contains exactly V. The p ↦ₐ V connective is useful for asserting ownership of locations in the Rec memory, e.g., for locations that are not mapped to the Asm memory.
- inv(v, m, m) asserts that m and m are in an invariant such that all the aforementioned assertions (i.e., p ↦ₐ v, v₁ ↦ₐ v₂, and p ↦ₐ V) have the meaning described above and v points to a valid stack.

Fig. 7. Definition of semantic linking relation →_{d₁,d₂} for Rec.

Fig. 8. Definition of (→) and (←) for [·]_{r⇒a}. 
separation logic predicates. We define:

\[
\begin{align*}
\text{CORO-LINK-YIELD} & \quad (d = L \land d' = R) \lor (d = R \land d' = L) \\
\text{CORO-LINK-YIELD-UB} & \quad d = L \lor d = R \quad \lnot v \neq 1 \\
\text{CORO-LINK-L-YIELD-INIT} & \quad (L, (L, \text{Some}(f)), \text{Call}(\text{yield}, [v], m)) \leadsto_{\text{coro}}^{d_1, d_2} (R, (R, \text{None}), \text{Call}(f, [v], m)) \\
\text{CORO-LINK-L-YIELD-INIT-UB} & \quad \lnot v \neq 1 \\
\text{CORO-LINK-INIT} & \quad f \in |M_1| \\
\text{CORO-LINK-INIT-UB} & \quad f \notin |M_1| \\
\text{CORO-LINK-L-RETURN} & \quad (L, (L, f^0), \text{Return}(v, m)) \leadsto_{\text{coro}} (E, (E, f^0), \text{Return}(v, m)) \\
\text{CORO-LINK-R-RETURN} & \quad (R, R, \text{Return}(v, m)) \leadsto_{\text{coro}} \\
\text{CORO-LINK-CALL} & \quad f \neq \text{yield} \\
\text{CORO-LINK-CALL-UB} & \quad (d = L \land f \notin |M_2|) \lor (d = R \land f \notin |M_1|) \\
\text{CORO-LINK-E-RETURN} & \quad (s = L \land d = L) \lor (s = R \land d = R) \quad e = \text{Return}(\_\_\_\_) \\
\text{CORO-LINK-E-CALL-UB} & \quad (s = L \land d = L) \lor (s = R \land d = R) \quad e = \text{Call}(\_\_\_\_) \\
\end{align*}
\]

\[
\text{Fig. 9. Definition of linking relation } \leadsto_{\text{coro}}^{d_1, d_2}.
\]

\[
[M]^{a, d, d, m}_{r = a} \triangleq [M]_X \quad \text{where} \quad X \triangleq (\text{List}(\text{Registers}), R_{r = a}, \_\_\_, \rightarrow, [], \_\_\_, v_1 \rightarrow_{a} \_\_\_ v_2)
\]
F COROUTINE LINKING

Formally, $M_1 \oplus_\text{coro} M_2$ is defined using the generic linking operator $M_1 \oplus_X M_2$. Concretely, we define $M_1 \oplus_{\text{coro}}^d, d_f, M_2 \equiv M_1 \oplus_{X_{\text{coro}}} M_2$ where

$$X_{\text{coro}} \equiv ((D \times \text{option}(\text{FnName})), \rightsquigarrow_{\text{coro}}^{d_1, d_2}, (E, \text{Some}(f)))$$

Note that this linking operator is parametrized by a function name $f$ of the initial function on the right side of the linking (stream in the example). The effect of linking is described by $\rightsquigarrow_{\text{coro}}$ shown in Fig. 9. There are many transitions, but most of them are straightforward. The rule CORO-LINK-YIELD encodes the core idea of $\oplus_{\text{coro}}$: If either the left side or the right side performs a call to $\text{yield}$, control switches to the other side, and the event is transformed to a $\text{Return?(v, m)}$ event. There is one special case to consider: When $M_1$ calls $\text{yield}$ the first time, there is no $\text{yield}$ in $M_2$ from which to return. Instead this first call to $\text{yield}$ becomes the invocation of a designated start function $f$ in $M_2$ (stream in the example), as stated by CORO-LINK-L-YIELD-INIT. CORO-LINK-INITS handles the initial call from the environment to $M_1$. If the environment tries to call a function not in $M_1$, the behavior is undefined (CORO-LINK-INIT-UB). CORO-LINK-L-RETURN handles the return from $M_1$ to the environment. $M_2$ should never return and thus CORO-LINK-R-RETURN states that doing so would lead to undefined behavior. Finally, CORO-LINK-CALL and CORO-LINK-E-RETURN allow both $M_1$ and $M_2$ to call external functions (like print). However, $M_1$ and $M_2$ cannot directly call a function in the other module (without going through $\text{yield}$) (CORO-LINK-CALL-UB) and the environment may not call them back recursively (CORO-LINK-CALL-UB).

REFERENCES


Appendix of DimSum: A Decentralized Approach to Multi-language Semantics and Verification 9