

Techniques for Executing and Reasoning About Specification Diagrams

Prasanna Thati¹, Carolyn Talcott², and Gul Agha¹

¹ University of Illinois at Urbana Champaign

`thati@cs.uiuc.edu`

`agha@cs.uiuc.edu`

² SRI International

`clt@csl.sri.com`

Abstract. Specification Diagrams (SD) [19] are a graphical notation for specifying the message passing behavior of open distributed object systems. SDs facilitate specification of system behaviors at various levels of abstraction, ranging from high-level specifications to concrete diagrams with low-level implementation details. We investigate the theory of *may testing equivalence* [15] on SDs, which is a notion of process equivalence that is useful for relating diagrams at different levels of abstraction. We present a semantic characterization of the may equivalence on SDs which provides a powerful technique to establish semantic correspondence between abstract specifications and their refined implementations. We also describe our prototypical implementation of SDs and of a procedure that exploits the characterization of may testing to establish equivalences between finitary diagrams (without recursion). We have implemented these in the Maude tool [5] which supports specifications in rewriting logic.

Key Words: Graphical specification languages, π -calculus, may testing, trace equivalence, rewriting logic, Maude.

1 Introduction

Smith and Talcott introduced Specification Diagrams (SD) [19] as a graphical notation for specifying message passing behaviors of open distributed object systems. SDs not only have an intuitive appeal as other graphical specification languages such as UML [18] and MSC [17], but also have a formal underpinning which makes them amenable to rigorous analysis. SDs draw upon concepts from various formalisms for concurrency; they allow dynamic name generation and name passing as in the π -calculus [14], they have asynchronous communication and enforce the locality discipline on use of names as in concurrent object-based models such as the Actor model [1], they are equipped with imperative notions such as variables, environments, and assignments, and they also incorporate logical features such as assertions and constraints which are appropriate for specification languages.

The language of SDs is designed to be useful at various stages of the software cycle. In the initial stages, one can abstractly express the desired system behavior and its properties without having to switch to another logic, and then progressively refine the abstract specifications into concrete diagrams with implementation details. An important task to be accomplished in this process is to be able to formally prove that a refined diagram is indeed a correct implementation of an abstract specification. The framework of *may testing* [15] is useful for formalizing such a semantic correspondence between diagrams. It is known that may testing is useful for reasoning about safety properties of implementations; specifically, it formalizes the criteria for a refined diagram to be a safe implementation of an abstract specification.

Relating diagrams according to may testing is in general a difficult task. In this paper, we present a characterization of may testing on SDs that provides a powerful technique for relating diagrams.

We also present an executable implementation of SDs by modeling the language as a theory in rewriting logic [12]. The Maude tool [5] which supports specifications in rewriting logic can then be used to execute diagrams. Finally, we describe the implementation in Maude of a procedure that exploits the characterization of may testing to relate finitary diagrams that do not involve recursion.

SDs are more of a specification language rather than a programming language in that not every SD is executable. For instance SDs are equipped with the **constrain** construct that is analogous to Dijkstra’s **assume** predicate [7]. A constrain specifies a predicate that should hold during a computation; failure of the predicate indicates that such a computation never happens, i.e the entire computation “is cancelled in between the computation” as though it never happened. It is clear that the constrain construct is not implementable in general. SDs are also equipped with certain fairness notions that are not implementable [20] (see the end of Section 3). In this paper, we will consider only the executable fragment of SDs; in particular we discard the constrain construct and the fairness conditions. Although the language we consider is only a fragment of Smith and Talcott’s language, from now on we will refer to it as the language of SDs.

A central theme of our work is that we present SDs as an extension of asynchronous π -calculus [9] with certain imperative and logical constructs. We will exploit this connection both to obtain a characterization of may testing and an executable implementation of SDs. Specifically, we will adapt our characterization of may testing for asynchronous π -calculus with locality [24] to obtain a characterization of may testing on SDs. Similarly, we will extend our implementation of asynchronous π -calculus described in [22] to obtain an implementation of SDs. In summary, this paper has three main contributions. It presents SDs as an extension of asynchronous π -calculus and exploits this connection to obtain a characterization of may testing and an implementation of SDs. This demonstrates how our previous work on testing equivalences over variants of asynchronous π -calculus can be fruitfully applied to reason about high level specification languages.

Following is the layout of the rest of this paper. In Sections 2 and 3, we present SDs as an extension of asynchronous π -calculus. In Section 4, we instantiate the framework of may testing on SDs and present an alternate characterization of it. In Section 5, we describe our implementation of SDs in the Maude tool. We conclude the paper in Section 6 with comments on possible directions of future work.

2 Specification Diagram Syntax

We informally describe the executable fragment of SDs that we consider. Besides ignoring non-executable features, we also reformulate certain other constructs in the original language that involve variable and name bindings (without changing their semantics). This reformulation not only helps us clearly separate the functional and imperative aspects of the language, but also makes elegant use of the name binding constructs and scoping mechanisms of π -calculus. We do not elaborate on these reformulations as they are not essential, and refer the reader to [19] for details.

We assume a set of values Val , which is not specified completely, but is assumed to include booleans and an infinite set of names \mathcal{N} . We assume an infinite set of variables Var , which can take on values from Val . The sets Var and Val are assumed to be disjoint. We let u, v, w range over Val , a, b, c over \mathcal{N} , ρ, ξ over sets of names, and x, y, z over Var . An environment is a partial function from Var to Val that is defined for only finitely many variables. We let γ range over environments. We denote an environment as a subset of $Var \times Val$ in the usual way. We assume a set of operations on Val that is not specified completely, but is assumed to contain the equality operator $=$, and the boolean operators \neg, \vee and \wedge . We also assume a function $n(\cdot)$ on values such that $n(v)$ is the set of all names that are used in constructing the (possibly composite) value v ; we assume that this set is always finite. We lift the function $n(\cdot)$ from Val to environments in the expected way. We let e, f, g range over expressions and ϕ over boolean expressions. Expressions can contain free variables, and

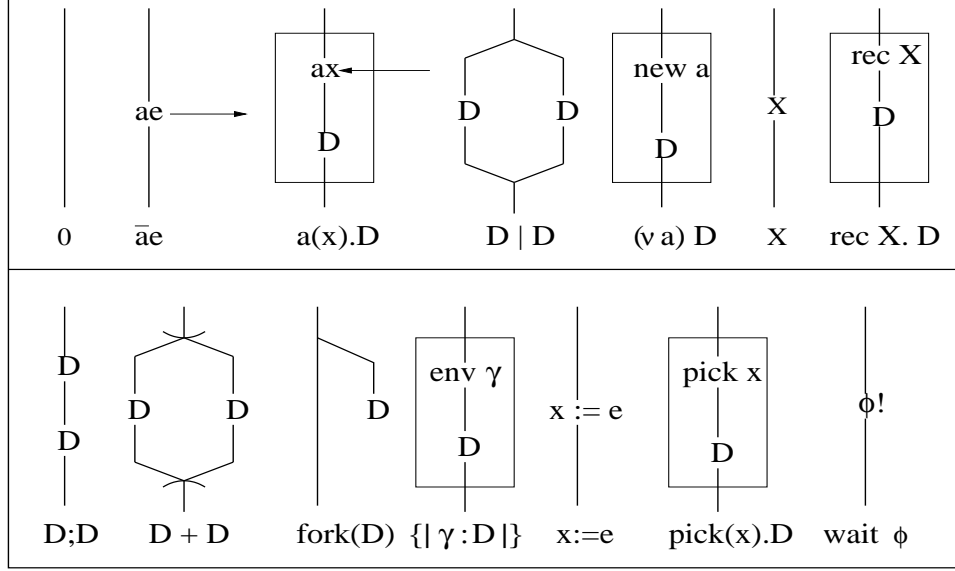


Fig. 1. A graphical representation of the specification diagram syntax

are evaluated in an environment that assigns values to these variables. We assume an evaluation function $\text{eval}(e, \gamma)$ that evaluates an expression e in an environment γ that assigns values to all free variables in e . From now on, we use the words diagrams and processes interchangeably.

SDs are defined by the following context-free grammar. We assume a set of process variables $PrVar$ that is disjoint from Var and Val , and let X, Y, Z, \dots range over it.

$$\begin{array}{ll}
D := 0 \mid \bar{a}e \mid a(x).D \mid D_1 | D_2 \mid (\nu a)D \mid \text{rec}X.D \mid X & \text{(asynch } \pi) \\
\mid D_1; D_2 \mid D_1 \oplus D_2 \mid \text{fork}(D) & \text{(control)} \\
\mid \{ | \gamma : D | \} \mid x := e & \text{(imperative)} \\
\mid \text{pick}(x).D \mid \text{wait}(\phi) & \text{(logical)}
\end{array}$$

Following is an informal description of each of these constructs.

- i. 0 (*nil*): Trivial behavior that does nothing.
- ii. $\bar{a}e$ (*output*): Send an asynchronous message to a with the result of evaluating e as the content. The name a is said to be the *subject* of the output.
- iii. $a(x).D$ (*input*): Receive an input u at a and continue as $D\{u/x\}$ (substitution). All occurrences of x in D are bound by the input argument. The name a is said to be the subject of the input.
- iv. $D_1 | D_2$ (*parallel composition*): Execute D_1 and D_2 parallelly (possibly involving interactions between the two).
- v. $(\nu a)D$ (*restriction*): Privatize the name a to D . All occurrences of a in D are bound by the restriction.
- vi. $\text{rec}X.D$ (*recursion*): Behave as $D\{\text{rec}X.D/X\}$. All occurrences of X in D are bound by the recursion operator.
- vii. $D_1; D_2$ (*sequential composition*): Execute D_1 , and then execute D_2 .

- viii. $\mathbf{D}_1 \oplus \mathbf{D}_2$ (*choice*): Execute exactly one of D_1 and D_2 .
- ix. $\mathbf{fork}(\mathbf{D})$ (*fork*): Make a copy of the current environment and execute D with this copy as its environment. D is to be executed concurrently with the (parent) diagram performing this fork. Specifically, note that the forked diagram and the parent diagram do not share their environments.
- x. $\{|\gamma : \mathbf{D}|\}$ (*scope*): Execute D in the environment γ .
- xi. $\mathbf{x} := \mathbf{e}$ (*assignment*): Assign to x the result of evaluating e .
- xii. $\mathbf{pick}(\mathbf{x}).\mathbf{D}$ (*pick*): Pick any value v such that $n(v)$ contains only the names that are currently in use, and execute $D\{v/x\}$. In particular, this construct does not generate fresh names. All occurrences of x in D are bound by the pick.
- xiii. $\mathbf{wait}(\phi)$ (*wait*): Wait until the environment is such that ϕ evaluates to *true*.

The reader is referred to [20] for a description of how many other constructs such as conditionals and finite and infinite loops can be encoded using the above constructs.

Figure 1 provides a graphical representation of these constructs. Vertical lines indicate progress of time, with the events above causally preceding the events below. Note that the first six constructs constitute the (monadic) asynchronous π -calculus [3, 9]. Constructs (vii)–(ix) provide additional control flow mechanisms, constructs (x) and (xi) provide imperative features, and the last two provide logical features. Further, note that there are five binding constructs in all; the *scope*, *pick*, and *input* constructs bind variables, *restriction* binds names, and *recursion* binds process variables. The pictorial depiction of these binding constructs contains a box that represents the scope of the bound variable. Since names can be communicated, the box for restriction can stretch during the course of the computation to include other parts of the diagram; this is the scope intrusion and extrusion mechanism described by Milner et al in [14].

SDs impose the discipline of *locality* in the use of names, where the recipient of a name communicated in a message is only allowed to use to the name for sending messages; in particular, the recipient does not have the capability to listen to messages targeted to the received name. This constraint syntactically localizes the *input* constructs with a given name as the subject, and hence the terminology. Locality is a common feature of object-based concurrent models such as the Actor model. It was first formally investigated in the setting of π -calculus by Merro and Sangiorgi [11]. The SD syntax automatically enforces the locality discipline. Specifically, an input subject is always a name (constant) and names can only be bound by a restriction. In particular, an input subject cannot be bound by the argument of another (enclosing) input, and hence the recipient cannot listen to messages targeted to the names it receives.

In addition to locality, SDs also enforce *uniqueness* of names. Specifically, let $rcp(D)$ be the set of all free names in D that occur as an input subject. This set contains all the free names at which D can currently receive a message. For a *top-level* diagram D , the uniqueness property states that no other process besides D can receive messages at a name in $rcp(D)$. Therefore, in particular, if D_1, D_2 are two top-level diagrams, then $rcp(D_1) \cap rcp(D_2) = \emptyset$. Note that locality guarantees that the uniqueness property is an invariant during execution; the set $rcp(D)$ may expand during the computation as private names of D are exported in outputs, and locality ensures the uniqueness property for these exported names. Further, note that if $D = D_1|D_2$ is a top-level diagram, then it can be that $rcp(D_1) \cap rcp(D_2) \neq \emptyset$ because D_1 and D_2 are not top-level diagrams.

We end this section with a few definitions and notational conventions. As usual, we do not distinguish between α -equivalent processes, i.e. processes that differ only in the use of bound names, bound variables, or bound process variables. The functions $fn(\cdot)$ and $bn(\cdot)$ which return the set of all free names and bound names that occur in a process (respectively), are defined as expected. Further, we define $n(D) = fn(D) \cup bn(D)$. A *value substitution* is a partial function from Var to Val that is defined only for finitely many variables. We write $\{\tilde{v}/\tilde{x}\}$ to denote the (value) substitution that maps x_i to v_i and is undefined for all other variables, where x_i and v_i are the i^{th} components of the tuples

α	τ	$(\xi)av$	$(\xi)\bar{a}v$	$(\xi)pick(v)$	$(\xi)fork(D, \gamma)$
$bn(\alpha)$	\emptyset	ξ	ξ	ξ	ξ
$fn(\alpha)$	\emptyset	$(\{a\} \cup n(v)) \setminus \xi$	$(\{a\} \cup n(v)) \setminus \xi$	$n(v) \setminus \xi$	$(n(D) \cup n(\gamma)) \setminus \xi$

Table 1. Free and bound names of actions.

\tilde{x} and \tilde{v} . We write $D\{\tilde{v}/\tilde{x}\}$ to denote the result of simultaneously substituting all occurrences of x_i in D with v_i . As usual, substitution is defined only modulo α -equivalence with the usual renaming of bound names to avoid captures. Similarly, a *process substitution* is a partial function from $PrVar$ to processes, that is defined only for finitely many process variables. The notations $\{\tilde{D}/\tilde{X}\}$ and $D'\{\tilde{D}/\tilde{X}\}$ have the expected meaning.

3 Operational Semantics

We define the SD semantics using a labeled transition system in the SOS style introduced by Plotkin [16]. The transition labels are of five kinds.

- i. τ : An internal action.
- ii. $(\xi)av$: An input of value v at name a . ξ is the set of names in $n(v)$ that are fresh with respect to the diagram D performing the input.
- iii. $(\xi)\bar{a}v$: An output of value v to name a . ξ is the set of names in $n(v)$ that are private to the diagram performing the output. These names will no longer be private to the diagram after the output.
- iv. $(\xi)pick(v)$: Execution of a *pick* construct that picks a value v . ξ is the set of all names in $n(v)$ that are private to the diagram performing this action.
- v. $(\xi)fork(D, \gamma)$: Execution of a *fork* construct that forks a diagram D with environment γ . ξ is the set of all names in $n(D) \cup n(\gamma)$ that are private to the diagram performing this action.

The functions $fn(\cdot)$ and $bn(\cdot)$ over actions are defined as in Table 1. We let α range over the set of all actions, and define $n(\alpha) = fn(\alpha) \cup bn(\alpha)$. For environments γ_1, γ_2 , we define $\gamma = \gamma_1; \gamma_2$ as $\gamma(x) = \gamma_1(x)$ if $\gamma_1(x)$ is defined, and $\gamma_2(x)$ otherwise. We write $\gamma[\tilde{x} \rightarrow \tilde{u}]$, where $\tilde{x} = x_1 \dots x_n$ and $\tilde{u} = u_1 \dots u_n$, as a shorthand for $\{(x_n, u_n)\}; \dots; \{(x_1, u_1)\}; \gamma$. We say that D is *trivial* if its syntax does not contain the *input*, *output*, *fork*, *assign*, *pick*, or *wait* constructs; such a process has the same behavior as 0. For $\xi = \{a_1, \dots, a_n\}$, we write $(\nu\xi)D$ as an abbreviation for $(\nu a_1) \dots (\nu a_n)D$; note that this notational convention is defined only modulo the ordering of the names a_1, \dots, a_n which in any case is irrelevant.

The transition system is defined at two levels - an *inner* level, and an *outer* level. The *inner* level transitions $\xrightarrow{\alpha}$ are between pairs consisting of a diagram and an environment in which the diagram is executed (see Table 2). The *outer* level transitions $\xrightarrow{\alpha}$ are defined between *closed* diagrams (see Table 3), i.e. diagrams in which every variable occurrence is bound by an *input*, *scope* or *pick* construct. The main reason for defining transitions at two levels, besides accounting for environments, is that the execution of *pick* and *fork* constructs is context sensitive. For instance, executing a *pick* can only return a value v such that every name in $n(v)$ is currently in use, and the set of names in use is determined by the entire top-level diagram that contains the *pick* construct. Similarly, in case of the *fork* construct, the forked diagram is to be instantiated in parallel with entire top-level diagram. Using two types of transitions facilitates the definition of a transition system in the SOS style despite the context sensitive nature of *pick* and *fork* constructs.

The transitions in Tables 2 and 3 are all defined modulo α -equivalence on diagrams, i.e. if D_1 and D_2 are α -equivalent then D_1 and D_2 have the same transitions, and so do the pairs $\langle D_1, \gamma \rangle$ and $\langle D_2, \gamma \rangle$. The rules *INP*, *OUT*, *REC*, *BINP*, *PAR*, *RES*, *OPEN* and *COM* are all analogous to

$INP \quad \langle a(x).D, \gamma \rangle \xrightarrow{\alpha v} \langle D\{v/x\}, \gamma \rangle$ $OUT \quad \langle \bar{a}e, \gamma \rangle \xrightarrow{\bar{\alpha} v} \langle 0, \gamma \rangle \quad \text{eval}(e, \gamma) = v$	$REC \quad \frac{\langle D\{\text{rec}X.D/X\}, \gamma \rangle \xrightarrow{\alpha} \langle D', \gamma' \rangle}{\langle \text{rec}X.D, \gamma \rangle \xrightarrow{\alpha} \langle D', \gamma' \rangle}$
$BINP \quad \frac{\langle D, \gamma \rangle \xrightarrow{(\xi)av} \langle D', \gamma' \rangle}{\langle D, \gamma \rangle \xrightarrow{(\xi \cup \{b\})av} \langle D', \gamma' \rangle} \quad b \in n(v) \setminus (fn(D) \cup n(\gamma))$	$PAR \quad \frac{\langle D_1, \gamma \rangle \xrightarrow{\alpha} \langle D'_1, \gamma'_1 \rangle}{\langle D_1 D_2, \gamma \rangle \xrightarrow{\alpha} \langle D'_1 D_2, \gamma'_1 \rangle} \quad bn(\alpha) \cap fn(D_2) = \emptyset$
$RES \quad \frac{\langle D, \gamma \rangle \xrightarrow{\alpha} \langle D', \gamma' \rangle}{\langle (\nu b)D, \gamma \rangle \xrightarrow{\alpha} \langle (\nu b)D', \gamma' \rangle} \quad b \notin n(\gamma) \cup n(\alpha)$	$COM \quad \frac{\langle D_1, \gamma \rangle \xrightarrow{(\xi)\bar{\alpha}v} \langle D'_1, \gamma'_1 \rangle \quad \langle D_2, \gamma \rangle \xrightarrow{(\xi)av} \langle D'_2, \gamma'_2 \rangle}{\langle D_1 D_2, \gamma \rangle \xrightarrow{\tau} \langle (\nu \xi)(D'_1 D'_2), \gamma' \rangle}$
$OPEN \quad \frac{\langle D, \gamma \rangle \xrightarrow{(\xi)\bar{\alpha}v} \langle D', \gamma' \rangle}{\langle (\nu b)D, \gamma \rangle \xrightarrow{(\xi \cup \{b\})\bar{\alpha}v} \langle D', \gamma' \rangle} \quad \begin{array}{l} b \neq a, b \in n(v), \\ b \notin \xi, b \notin n(\gamma) \end{array}$	
$SEQ1 \quad \frac{\langle D_1, \gamma \rangle \xrightarrow{\alpha} \langle D'_1, \gamma'_1 \rangle}{\langle D_1; D_2, \gamma \rangle \xrightarrow{\alpha} \langle D'_1; D_2, \gamma'_1 \rangle}$	$SEQ2 \quad \frac{\langle D_2, \gamma \rangle \xrightarrow{\alpha} \langle D'_2, \gamma'_2 \rangle}{\langle D_1; D_2, \gamma \rangle \xrightarrow{\alpha} \langle D_1; D'_2, \gamma'_2 \rangle} \quad D_1 \text{ is trivial}$
$SUM \quad \langle D_1 \oplus D_2, \gamma \rangle \xrightarrow{\tau} \langle D_1, \gamma \rangle$	$OPEN-FORK \quad \frac{\langle D, \gamma \rangle \xrightarrow{(\xi)fork(D_1, \gamma_1)} \langle D', \gamma' \rangle}{\langle (\nu b)D, \gamma \rangle \xrightarrow{(\xi \cup \{b\})fork(D_1, \gamma_1)} \langle D', \gamma' \rangle} \quad \begin{array}{l} b \in n(D_1) \cup n(\gamma_1) \\ b \notin \xi, b \notin n(\gamma) \end{array}$
$FORK \quad \langle \text{fork}(D), \gamma \rangle \xrightarrow{fork(D, \gamma)} \langle 0, \gamma \rangle$	
$SCOPE \quad \frac{\langle D, \gamma_1; \gamma_2 \rangle \xrightarrow{\alpha} \langle D', \gamma_1[\bar{x} \rightarrow \bar{v}]; \gamma'_2 \rangle}{\langle \{\gamma_1 : D\}, \gamma_2 \rangle \xrightarrow{\alpha} \langle \{\gamma_1[\bar{x} \rightarrow \bar{v}] : D'\}, \gamma'_2 \rangle}$	$ASSGN \quad \langle x := e, \gamma \rangle \xrightarrow{\tau} \langle 0, \gamma[x \rightarrow v] \rangle \quad \text{eval}(e, \gamma) = v$
$PICK \quad \langle \text{pick}(x).D, \gamma \rangle \xrightarrow{pick(v)} \langle D\{v/x\}, \gamma \rangle$	$RES-PICK \quad \frac{\langle D, \gamma \rangle \xrightarrow{(\xi)pick(v)} \langle D', \gamma' \rangle}{\langle (\nu b)D, \gamma \rangle \xrightarrow{(\xi \cup \{b\})pick(v)} \langle (\nu b)D', \gamma' \rangle} \quad b \in n(v)$
$WAIT \quad \langle \text{wait}(\phi), \gamma \rangle \xrightarrow{\tau} \langle 0, \gamma \rangle \quad \text{eval}(\phi, \gamma) = \text{true}$	

Table 2. Rules for inner level transitions.

the corresponding transition rules for asynchronous π -calculus [2]. The rules PAR , COM and SUM have symmetric versions that are not shown. We now elaborate on the rules concerned with $pick$ and $fork$ constructs; the others are self-explanatory.

The transition label of the $PICK$ rule includes the value v that is being picked. All the names in $n(v)$ are progressively accounted for by the $RES-PICK$ and $TOP-PICK$ rules; these names should either be private or already occur free in the top-level diagram. This ensures that every name in $n(v)$ is currently in use. The transition label of the $FORK$ rule contains both the diagram that is being forked, and the environment in which the forked diagram will be executed. The $TOP-FORK$ rule instantiates this diagram along with the environment, in parallel with the top-level diagram. This ensures that the newly forked diagram executes concurrently with the current diagram, and that the two diagrams do not share their environments.

Finally, a note on the $TOP-OUT$ rule. This rule accounts for asynchrony in message exchanges. A message emitted by the OUT rule can either be immediately exported by the TOP rule, or it can be buffered by the $TOP-OUT$ rule. Note that the arguments of a buffered message have already been evaluated by the OUT rule that created the message. As opposed to outputs, the only rule for inputs, namely INP , is synchronous. We avoid asynchronous input transitions that allow a process to be always input enabled, because otherwise even simple non-recursive process would have an infinite behavior. For instance, both 0 and $\text{rec}X.(a(x).(\bar{a}x|X))$ would have the same behavior. Instead we account for asynchrony in inputs explicitly in the theory of equivalence that we develop in Section 4.

Before we proceed any further, a few definitions and notational conventions are in order. Let \mathcal{L} denote the set of all input and output actions; these are the visible actions. Note that every top-level transition is labeled with a τ or an action in \mathcal{L} . We let s, r, t range over the set of traces \mathcal{L}^* . The

$$\begin{array}{ll}
TOP \frac{\langle D, \emptyset \rangle \xrightarrow{\alpha} \langle D', \emptyset \rangle}{D \xrightarrow{\alpha} D'} \alpha \neq \begin{array}{l} pick, \\ fork \end{array} & TOP-OUT \frac{\langle D_1, \emptyset \rangle \xrightarrow{(\xi)\bar{a}v} \langle D'_1, \emptyset \rangle}{D_1 \xrightarrow{\tau} (\nu\xi)(D'_1|\bar{a}v)} \\
TOP-FORK \frac{\langle D_1, \emptyset \rangle \xrightarrow{(\epsilon)fork(D_2, \gamma)} \langle D'_1, \emptyset \rangle}{D_1 \xrightarrow{\tau} (\nu\xi)(D'_1|\{\gamma : D_2\})} & TOP-PICK \frac{\langle D, \emptyset \rangle \xrightarrow{(\epsilon)pick(v)} \langle D', \emptyset \rangle}{D \xrightarrow{\tau} D'} \quad n(v) \subseteq \xi \cup fn(D)
\end{array}$$

Table 3. Rules for outer level transitions.

functions $fn(\cdot)$, $bn(\cdot)$ and $n(\cdot)$ are extended to \mathcal{L}^* the obvious way. We define a complementation function on \mathcal{L} as $(\xi)\bar{x}y = (\xi)\bar{x}y$, $(\xi)\bar{x}\bar{y} = (\xi)xy$, and extend this to \mathcal{L}^* the obvious way. The α -equivalence over traces is defined as expected, and α -equivalent traces are not distinguished. For example, the traces $(b)ab.\bar{b}a$ and $(c)ac.\bar{c}a$ are α -equivalent; we do not distinguish between the bound names b and c . Since we work modulo α -equivalence on traces, for convenience we assume the following *normality* condition on any trace s we consider – if $s = s_1.\alpha.s_2$ then $(n(s_1) \cup fn(\alpha)) \cap bn(\alpha.s_2) = \emptyset$.

We define the relation \Longrightarrow as the reflexive transitive closure of $\xrightarrow{\tau}$, and $\xRightarrow{\beta}$ as $\Longrightarrow \xrightarrow{\beta} \Longrightarrow$. For $s = l.s'$, we write $D \xrightarrow{l} \xrightarrow{s'} Q$ compactly as $D \xrightarrow{s} Q$, and similarly $D \xrightarrow{l} \xRightarrow{s'} Q$ as $D \xRightarrow{s} Q$. We write the assertion $D \xRightarrow{s} D'$ for some D' , as $D \xrightarrow{s}$, and similarly $D \xrightarrow{s}$ and $D \xrightarrow{\tau}$. We define $[D] = \{s \mid D \xRightarrow{s}\}$.

Not every trace produced by the transition system corresponds to a valid computation. For example, we have

$$(\nu a)(a(x).D|\bar{a}v|\bar{b}a) \xrightarrow{(a)\bar{b}a} a(x).D|\bar{a}v \xrightarrow{\bar{a}v}$$

But the message $\bar{a}v$ is not observable due to the *locality* property of SDs (see Section 2); the locality property prevents the recipient of the private name a from listening to messages targeted to a . Further, due to the uniqueness property the message $\bar{a}v$ in the top-level diagram $a(x).D|\bar{a}v$ is not observable, although we have the transition $a(x).D|\bar{a}v \xrightarrow{\bar{a}v}$. To account for this, we define for a set of names ρ , the notion of a ρ -well-formed trace such that only ρ -well-formed traces can be exhibited by a diagram D with $rcp(D) = \rho$.

Definition 1. We define $rcp(\rho, s)$ inductively as $rcp(\rho, \epsilon) = \rho$, $rcp(\rho, s.(\xi)av) = rcp(\rho, s)$, and $rcp(\rho, s.(\xi)\bar{a}v) = rcp(\rho, s) \cup \xi$. We say s is ρ -well-formed if $s = s_1.(\xi)\bar{a}v.s_2$ implies $a \notin rcp(\rho, s_1)$. We say s is well-formed if it is \emptyset -well-formed. \square

We adopt the following hygiene condition for convenience (in addition to the normality condition). Whenever we consider a ρ -well-formed trace s , we have $\rho \cap bn(s) = \emptyset$. The following lemma captures the intuition behind Definition 1.

Lemma 1. Let $rcp(D_2) \cap \rho = \emptyset$. Then the computation $D_1|D_2 \Longrightarrow$ can be unzipped into $D_1 \xRightarrow{s}$ and $D_2 \xRightarrow{\bar{s}}$ such that s is ρ -well-formed. \square

The original definition of SDs by Smith and Talcott [19] also accounts for certain fairness conditions. For instance, it is required that every message that is sent during the course of a computation is eventually received. Such fairness conditions are in general not implementable, making SDs more of a specification language rather than a programming language. For instance, it is in general impossible to decide if a diagram can eventually evolve to a state where it can receive a certain message. Since our intention is to focus on an executable fragment (or variant) of SDs, we drop these fairness conditions.

4 May Testing on Specification Diagrams

The may testing equivalence [15] is a notion of process equivalence which is useful to relate specifications at different levels of abstraction. It is an instance of the general notion of behavioral equivalence where, roughly, two processes are said to be equivalent if they are indistinguishable in all contexts of use. Specifically, the context consists of an observing process that runs in parallel and interacts with the process being tested. The observer can in addition signal a success by emitting a special event. The process being tested is said to pass the test proposed by the observer if there *exists* a run in which the observer signals a success; note that due to possible non-determinism the observer and the process can take one of many possible computation paths. Two processes are said to be may equivalent if they pass exactly the same set of tests.

We consider a generalized version of the usual may equivalence, where the equivalence is parameterized with a set of names that determines the set of observers that can be used to decide the equivalence. We originally introduced this generalized notion in the context of asynchronous π -calculus with locality [24].

Definition 2 (may testing). *Observers are diagrams that can emit a special message $\bar{p}\mu$. We let O range over the set of observers. We say $D \text{ may } O$ if $D|O \xrightarrow{\bar{p}\mu}$. We say $D_1 \sqsubseteq_\rho D_2$ if for every O with $\text{rcp}(O) \cap \rho = \emptyset$, we have $D_1 \text{ may } O$ implies $D_2 \text{ may } O$. We say $D_1 \simeq_\rho D_2$ if $D_1 \sqsubseteq_\rho D_2$ and $D_2 \sqsubseteq_\rho D_1$. \square*

Thus, only observers that do not listen at names in ρ are used to decide the preorder \sqsubseteq_ρ . Further, the larger the parameter ρ , the smaller the observer set that is used to decide \sqsubseteq_ρ . Hence if $\rho_1 \subseteq \rho_2$, we have $D_1 \sqsubseteq_{\rho_1} D_2$ implies $D_1 \sqsubseteq_{\rho_2} D_2$. However, $D_1 \sqsubseteq_{\rho_2} D_2$ need not imply $D_1 \sqsubseteq_{\rho_1} D_2$. For instance, $0 \simeq_{\{a\}} \bar{a}a$, but only $0 \sqsubseteq_\emptyset \bar{a}a$ and $\bar{a}a \not\sqsubseteq_\emptyset 0$. Similarly, $\bar{a}a \simeq_{\{a,b\}} \bar{b}b$, but $\bar{a}a \not\sqsubseteq_\emptyset \bar{b}b$ and $\bar{b}b \not\sqsubseteq_\emptyset \bar{a}a$. However, $D_1 \sqsubseteq_{\rho_2} D_2$ implies $D_1 \sqsubseteq_{\rho_1} D_2$ if $\text{fn}(D_1) \cup \text{fn}(D_2) \subseteq \rho_1$.

May testing is known to be useful for reasoning about safety properties of concurrent systems. Specifically, by viewing the observer's success as something bad happening, $D_1 \sqsubseteq_\rho D_2$ can be interpreted as D_1 is a safe implementation of the specification D_2 , because if the specification D_2 is guaranteed to not cause anything bad to happen in a given context (that does not listen to names in ρ), then the implementation D_1 would also not cause anything bad to happen in the same context.

The universal quantification over contexts in the definition of may testing makes it very hard to prove equalities. Specifically, to prove an equivalence, one has to consider all possible interactions between the given processes and all possible observers. The typical approach to circumvent this problem is to find an alternate characterization of the equivalence that involves only the process being tested [2, 4, 8]. For SDs, a variant of the trace semantics characterizes the parameterized may preorder. In fact, it turns out that the characterization is similar to the one for asynchronous π -calculus with locality that we presented in [24]; the only difference arising due to the fact that unlike SDs the calculus in [24] is not equipped with the mismatch operator on names. The characterization in [24] is in turn an adaptation of the characterization for asynchronous π -calculus [2] originally developed by Boreale, DeNicola and Pugliese. We skip the proofs of all the propositions in this section as they are relatively simple extensions of the proofs in [2] and [24]. The main difference arises due to the mismatch capability on names in SDs (this capability is absent in the formalisms in [2] and [24]), which can be handled using the techniques we presented in [23].

The trace based characterization of \sqsubseteq_ρ over SDs follows. Recall, that the transition system in Section 3 does not account for asynchrony in inputs. To account for this, we define a preorder \preceq on

<i>(drop)</i>	$s_1.(\xi)s_2 \preceq s_1.(\xi)av.s_2$	if $(\xi)s_2 \neq \perp$
<i>(delay)</i>	$s_1.(\xi)(\alpha.av.s_2) \preceq s_1.(\xi)av.\alpha.s_2$	if $(\xi)(\alpha.av.s_2) \neq \perp$
<i>(annihilate)</i>	$s_1.(\xi)s_2 \preceq s_1.(\xi)av.\bar{a}v.s_2$	if $(\xi)s_2 \neq \perp$

Table 4. A preorder relation on traces.

traces as the reflexive transitive closure of the laws shown in Table 4, where $(\xi)\cdot$ is defined as

$$(\xi)s = \begin{cases} s & \text{if } \xi = \emptyset \text{ or } \xi \cap fn(s) = \emptyset \\ (\xi \setminus \{b\})s_1.(\xi' \cup \{b\})av.s_2 & \text{if } b \in \xi, b \in n(v) \text{ and there are } s_1, s_2, a, \xi' \\ & \text{s.t. } s = s_1.(\xi')av.s_2 \text{ and } b \notin fn(s_1) \cup \{a\} \\ \perp & \text{otherwise} \end{cases}$$

The expression $(\xi)s$ returns \perp , if there is $b \in \xi$ such that b is used in s before it is received for the first time, i.e. the first free occurrence of b in s is *not* in the argument of an input. Otherwise, the expression binds the first such occurrence (in an input in s) of every $b \in \xi$, and returns the resulting trace.

The intuition behind the preorder \preceq is that if a process leads an observer to a success by exhibiting a trace s , then it can also lead the observer to a success by exhibiting any trace $r \preceq s$. Specifically, inputs are not observable since they are asynchronous, and hence they can be dropped, delayed, or annihilated with complementary output actions. This is formalized in the following lemma.

Lemma 2. *If $O \xrightarrow{\bar{s}.\bar{\mu}\mu}$, then $r \preceq s$ implies $O \xrightarrow{\bar{r}.\bar{\mu}\mu}$.* □

We now define a relation that characterizes the may preorder.

Definition 3. *We say $[D_2] \preceq_\rho [D_1]$ if for every ρ -well-formed trace s , $D_1 \xrightarrow{s}$ implies there is $r \preceq s$ such that $D_2 \xrightarrow{r}$.* □

Theorem 1. *$D_1 \sqsubseteq_\rho D_2$ if and only if $[D_2] \preceq_\rho [D_1]$.* □

So far, we have allowed a given pair of diagrams D_1 and D_2 to be compared with \sqsubseteq_ρ for arbitrary ρ . But if D_1 and D_2 are top level diagrams, due to the uniqueness property of names (see Section 2) it makes sense to compare D_1 and D_2 with \sqsubseteq_ρ only if $fn(D_1), fn(D_2) \subseteq \rho$. In this case, we can in fact strengthen Theorem 1 by dropping the *annihilation* law. Specifically, for ρ such that $fn(D_1), fn(D_2) \subseteq \rho$, we have $D_1 \sqsubseteq_\rho D_2$ if and only if for every ρ -well-formed trace s , $D_1 \xrightarrow{s}$ implies $D_2 \xrightarrow{r}$ for some $r \preceq s$ using only the laws *delay* and *drop*. The reason behind this is that since s is ρ -well-formed and $rcp(D) \subseteq \rho$, s cannot contain complimentary input and output actions.

5 Executable Specification in Rewriting Logic

We now specify the language of SDs as a theory in rewriting logic [12]. The Maude tool [5] which supports specifications in rewriting logic can then be used to execute SDs. We also present a procedure implemented in Maude, that exploits Theorem 1 to decide the may preorder relation between finitary diagrams (without recursion). To simplify matters we assume that the set of values Val only contains names. Extending this to arbitrary value sets will need sophisticated symbolic techniques [10], which is out of the scope of this paper.

Since we have represented specification diagrams as an extension of asynchronous π -calculus we can smoothly extend the specification of asynchronous π -calculus in rewriting logic that we

introduced in [22] to obtain an executable specification of SDs. The main idea behind specifying SDs in rewriting logic is to represent an (inner or outer) transition rule of form

$$\frac{P_1 \rightarrow Q_1 \ \dots \ P_n \rightarrow Q_n}{P_0 \rightarrow Q_0}$$

as a *conditional* rewrite rule of the form $P_0 \longrightarrow Q_0$ if $P_1 \longrightarrow Q_1 \wedge \dots \wedge P_n \longrightarrow Q_n$, where the condition includes rewrites. This was first introduced by Verdejo et al. [25] for implementing CCS [13]. Such conditional rules with rewrite conditions are executable in version 2.0 of the Maude language and system [5]; the rewrite conditions are solved by means of an implicit search process. The reader is referred to [5, 25] for more details about this.

In the Maude specification of the SD syntax that follows, the sorts **Chan**, **Var**, and **PrVar** are used to represent names, variables and process variables, and the sort **Env** is used to represent environments. Following are operator declarations for a few of the SD constructs.

```
sorts Chan Var PrVar Env Diag .
op _(_)_ : Chan Qid Diag -> Diag .      op {[_:_]} : Env Diag -> Diag .
op |_ : Diag Diag -> Diag .             ops new[_]_ rec[_]_ : Qid Diag -> Diag .
```

The sort **Qid** represents quoted identifiers. To manage name and variable bindings in specification diagrams, we use CINNI as a calculus for explicit substitutions [21] which has been implemented in Maude. CINNI gives a first-order representation of terms with bindings and capture-free substitutions, instead of going to the metalevel to handle identifiers and bindings. The main idea in such a representation is to keep the bound identifier inside the binders as it is, but to replace its use by the identifier followed by an index which is a count of the number of binders with the same identifier it jumps before it reaches the place of use. This combines the best of the approaches based on standard variables and de Bruijn indices [6]. Following this idea, we define terms of sorts **Chan**, **Var** and **PrVar** as indexed identifiers as follows.

```
op _{[_]} : Qid Nat -> Chan .           op _{[_]} : Qid Nat -> Var .
op _{[_]} : Qid Nat -> PrVar .
```

Note that the operator $_ \{ _ \} _$ is (ad hoc) overloaded. Following are the constructors for environments.

```
op emptyEnv : -> Env .   op (__) : Var Chan -> Env .   op _;_ : Env Env -> Env .
eq { | emptyEnv : D | } = D .
eq { | (X CX); E : D | } = { | E : { | (X CX) : D | } | } .
```

As a result of the equations above, from now on we can assume that the environment γ in $\{ | \gamma : D | \}$ is of form $(x a)$. For name substitutions we introduce the sort **ChanSubst** along with the following operations. The intuitive meaning of these operations is described in Table 5 (see [21] for more details).

```
op [_:=_] : Qid Chan -> ChanSubst .   op [shiftup_] : Qid -> ChanSubst .
op [shiftdown_] : Qid -> ChanSubst .   op [lift__] : Qid ChanSubst -> ChanSubst .
```

We introduce the sort **PrSubst** for process substitutions, along with similar operations as above. Using these, explicit substitutions for SDs can be defined equationally. Following are some interesting equations. Note how the substitution is lifted as it moves across a binder.

[a := x]	[shiftup a]	[shiftdown a]	[lift a S]
a{0} ↦ x	a{0} ↦ a{1}	a{0} ↦ a{0}	a{0} ↦ [shiftup a] (S a{0})
a{1} ↦ a{0}	a{1} ↦ a{2}	a{1} ↦ a{0}	a{1} ↦ [shiftup a] (S a{1})
...
a{n+1} ↦ a{n}	a{n} ↦ a{n+1}	a{n+1} ↦ a{n}	a{n} ↦ [shiftup a] (S a{n})
b{m} ↦ b{m}	b{m} ↦ b{m}	b{m} ↦ b{m}	b{m} ↦ [shiftup a] (S b{m})

Table 5. The CINNI operations.

```

Var NS : ChanSubst .  Var PS : PrSubst .
eq  NS (CX(Y) . D) = (NS CX)(Y) . ([lift Y NS] D) .
eq  NS (D1 | D2) = (NS D1) | (NS D2) .
eq  NS ({|(X CX) : D|} = {|(X (NS CX)) : [lift X NS] D|} .
eq  PS (new [X] D) = new [X] ([lift X PS] D) .
eq  PS (rec [X] D) = rec [X] ([lift X PS] D) .

```

We now describe the specification of the transition system. As mentioned earlier, the transition rules are represented as conditional rewrite rules with the premises as conditions of the rule. Since rewrites do not have labels unlike the labeled transitions, we make the label a part of the resulting term; thus these rewrites are of the form $P \Rightarrow \{\alpha\}Q$.

A problem to overcome in giving an executable specification of the transition system is that the transitions of a term can be *infinitely branching* because of the *INP* and *OPEN* rules. In case of the *INP* rule, there is one branch for every possible name that can be received in the input (recall that we have assumed names to be the only values). In case of the *OPEN* rule, there is one branch for every name that is chosen to denote the private channel that is being emitted (recall that the transition rules are defined only modulo α -equivalence).

To overcome this problem, we define transitions relative to an execution environment¹. The environment is represented abstractly as a set of free names \mathbf{CS} that it may use while interacting with the process, and both the inner and outer level transitions are modeled as rewrite rules over terms of the form $[\mathbf{CS}] P$. The set \mathbf{CS} expands during bound input and output interactions when private names are exchanged between the process and its environment. The infinite branching due to the *INP* rule is avoided by allowing only the names in \mathbf{CS} to be received in free inputs. Since \mathbf{CS} is assumed to contain all the free names in the environment, an input argument that is not in \mathbf{CS} would be a private name of the environment. Now, since the identifier chosen to denote the fresh name is irrelevant, all bound input transitions can be identified to a single input. With these simplifications, the number of input transitions of a term become finite. Similarly, in the *OPEN* rule, since the identifier chosen to denote the private name emitted is irrelevant, instances of the rule that differ only in the chosen name are not distinguished.

Following is the specification of a few of the inner level transitions (see Table 2).

```

sorts Action VisAction VisActionType EnvProc TraceProc .
subsort VisAction < Action .      subsort EnvProc < TraceProc .
ops i o : -> VisActionType .
op f : VisActionType Chan Chan -> VisAction .
op b : VisActionType Chan Qid -> VisAction .
op [_]<_,> : ChanSet Diag Env -> EnvProc .
op {_}_ : Action TraceProc -> TraceProc [frozen] .

```

```

rl [INP] : [CY CS] < CX(X) . D , E > =>

```

¹ We have overloaded the word environment. So far we have used it to denote variable bindings. We now also use it to refer to the external process with which the process under consideration is interacting. It should be clear from the context as to which of these we mean.

```

      {f(i,CX,CY)} [CY CS] < [X := CY] D , E > .
crl [BINP] : [CS] < D , E > =>
  {[shiftdown 'U] b(i,CX,'U)} ['U{0}] [shiftup 'U] CS] < D1 , E1 >
  if (not flag in CS) /\ CS1 := flag 'U{0} [shiftup 'U] CS /\
    [CS1] < [shiftup 'U] D , [shiftup 'U] E > =>
    {f(i,CX,'U{0})} [CS1] < D1 , E1 > .
crl [OPEN] : [CS] <new [X] D , E> =>
  {[shiftdown X] b(o,CY,X)} [X{0}] CS1] <D1 , E1>
  if CS1 := [shiftup X] CS /\ E2 := [shiftup X] E /\
    [CS1] < D , E2 > => {f(o,CY,X{0})} [CS1] < D1 , E1 > /\
    X{0} /= CY .
crl [SCOPE] : [CS] < {!(X CX) : D}| , E > => {A} [CS1] < {!(X CX1) : D1}| , E1 >
  if [CS] < D , (X CX) ; E > => {A} [CS1] < D1 , (X CX1) ; E1 > .

```

We have shown the constructors for only the sort `VisAction` that represents visible actions, i.e. input and output actions. Since names are assumed to be the only values, these actions are of form $(\hat{b})ab$ or $(\hat{b})\bar{a}b$, where the metavariable \hat{b} ranges over $\{\emptyset, \{b\}\}$. The operators `f` and `b` are used to construct free and bound actions respectively. Name substitutions on actions are defined equationally as expected. The implementation of the `INP`, `BINP` and `OPEN` rules is similar to that of the corresponding rules for asynchronous π -calculus [22]. We explain only the `BINP` rule in detail, and refer the reader to [22] for further details.

In the `BINP` rule, since the identifier chosen to denote the bound argument is irrelevant, we use the constant `'U` for all bound inputs, and thus `'U{0}` denotes the fresh name received. Note that in contrast to the `BINP` rule of Table 2, we do not check if `'U{0}` is in the free names of the process performing the input or its variable bindings, and instead we shift up the channel indices appropriately in `CS`, `D`, and `E` in the righthand side and condition of the rule. This is justified because the transition target is within the scope of the bound name in the input action. Note also that the channel `CX` in the action is shifted down because it is now out of the scope of the bound argument. The set `CS` is expanded by adding the received channel `'U{0}` to it. Finally, we use a special constant `flag` of sort `Chan`, to ensure termination. The constant `flag` is used to prevent the `BINP` rule from being fired again while evaluating the condition. Without this check, we will have a non-terminating execution in which the `BINP` rule is repeatedly fired.

Following is the implementation of one of the outer level transitions (see Table 3).

```

sorts EnvDiag TraceDiag .   subsort EnvDiag < TraceDiag .
ops tau bpick : -> Action .   op fpick : Chan -> Action .
op [_]_ : ChanSet Diag -> EnvDiag .
op {[_]} : Action TraceDiag -> TraceDiag [frozen] .
crl [TOP-PICK] : [CS] D => {tau} [CS1] D1
  if [CS] < D , emptyEnv > => {A} [CS1] < D1 , emptyEnv > /\
    (A == bpick \/\ (A == pick(CX) /\ CX in freenames(D))) .

```

The constant `bpick` is used to represent a pick action that picks a bound name, while `fpick` is used to denote an action that picks a free name. Note that the operator `{[_]}` above is declared `frozen`. This forbids rewriting of its arguments; otherwise rewrites can be applied to any subterm. We use the `frozen` attribute because otherwise for a recursive process `D` the term `[CS] D` may have a non-terminating rewrite sequence `[CS]D => {A1} [CS]D1 => {A1}{A2} [CS]D2 => ...`. But since `{[_]}` is declared `frozen` a term `[CS] D` can be rewritten only once. To compute all possible successors of a term, we explicitly generate the transitive closure of one step transitions as follows (the dummy operator `[_]` declared below is used to prevent infinite loops).

```

sort TEnvDiag .

```

```

op [_] : EnvDiag -> TEnvDiag [frozen] .
crl [reflx] : [ P ] => {A} Q if P => {A} Q .
crl [trans] : [ P ] => {A} R if P => {A} Q /\ [ Q ] => R .

```

Now, the set of all traces exhibited by $[CS] D$ can be computed by finding all the one step successors of $[[CS] D]$. The traces appear (with τ actions) as the prefix of these one-step successors. Of course, there can be infinitely many one-step successors if D is recursive, but using the meta-level facilities of Maude we can compute only as many of them as needed. To represent traces, we introduce the sort `Trace` as follows.

```

subsort VisAction < Trace .
op epsilon : -> Trace . op _.. : Trace Trace -> Trace [assoc id: epsilon] .
op [_] : Trace -> TTrace .
ceq [TR1 . b(IO,CX,Y) . TR2] = [TR1 . b(IO,CX,'U) . [Y := 'U{0}] [shiftup 'U] TR2]
if Y /= 'U .

```

The equation above defines α -equivalence on traces the expected way. The function `rwf` which checks if a trace is ρ -well-formed is defined along the lines of Definition 1 as follows.

```

op rwf : Trace Chanset -> Bool [frozen] .
eq rwf(epsilon , CS) = true .
eq rwf(f(i,CX,CY) . TR , CS) = rwf(TR , CS) .
eq rwf(b(i,CX,Y) . TR , CS) = rwf(TR , [shiftup Y] CS) .
eq rwf(f(o,CX,CY) . TR , CS) = (not CX in CS) and rwf(TR , CS) .
eq rwf(b(o,CX,Y) . TR , CS) = (not CX in CS) and rwf(TR , Y{0} [shiftup Y] CS) .

```

We encode the relation \preceq of Table 4 as rewrite rules on terms of sort `TTrace`. Specifically $r \preceq s$ if *cond* is encoded as $s \Rightarrow r$ if *cond*.

```

crl [Drop] : [ TR1 . b(i,CX,Y) . TR2 ] => [ TR1 . bind(Y , TR2) ]
if bind(Y , TR2) /= bot .
rl [Delay] : [ ( TR1 . f(i,CX,CY) . b(IO,CU,V) . TR2 ) ] =>
[ ( TR1 . b(IO,CU,V) . ([shiftup V] f(i, CX , CY)) . TR2 ) ] .
crl [Delay] : [ ( TR1 . b(i,CX,Y) . f(IO,CU,CV) . TR2 ) ] =>
[ ( TR1 . bind(Y , f(IO,CU,CV) . f(i,CX,Y{0}) . TR2 ) ) ]
if bind(Y , f(IO,CU,CV) . f(i,CX,Y{0}) . TR2) /= bot .

```

The operator `bind` implements the function $(\hat{y})\cdot$ on traces. Note that in the first `Delay` rule, the channel indices of the free input action are shifted up when it is delayed across a bound action, since it gets into the scope of the bound argument. Similarly, in the second `Delay` rule, when the bound input action is delayed across a free input/output action, the channel indices of the free action will be shifted down by the `bind` operation. The other two subcases of the `Delay` rule, namely, where a free input is to be delayed across a free input or output, and where a bound input is to be delayed across a bound input or output, are not shown as they are similar.

To decide $D_1 \sqsubseteq_\rho D_2$ for finitary diagrams D_1, D_2 without recursion, we exploit the alternate characterization of \sqsubseteq_ρ given by Theorem 1. But a problem with this approach is that finiteness of D only implies that the length of traces in $[D]$ is bounded, but the number of traces in $[D]$ can be infinite (even modulo α -equivalence) because the *INP* rule is infinitely branching. To avoid the problem of having to compare infinite sets, we observe that

$$[D_2] \lesssim_\rho [D_1] \quad \text{if and only if} \quad [D_2]_{fn(D_1, D_2)} \lesssim_\rho [D_1]_{fn(D_1, D_2)},$$

where for a set of traces S we define $S_\xi = \{s \in S \mid fn(s) \subseteq \xi\}$. Now, since the traces in $[D_1]$ and $[D_2]$ are finite in length, it follows that the sets of traces $[D_1]_{fn(D_1, D_2)}$ and $[D_2]_{fn(D_1, D_2)}$ are finite modulo

α -equivalence. In fact, the set of traces generated for $[[\text{fn}(D1, D2)] D1]$ by our implementation, contains exactly one representative from each α -equivalence class of $[D1]_{\text{fn}(D1, D2)}$.

Given processes D_1 and D_2 , we generate the set of all traces (modulo α -equivalence) of $[[\text{fn}(D1, D2)] D1]$ and $[[\text{fn}(D1, D2)] D2]$ using the metalevel facilities of Maude. Then for each ρ -well-formed trace T in $[D1]_{\text{fn}(D1, D2)}$, we compute the reflexive transitive closure of T with respect to the rewrite rules for the laws in Table 4. We then use the fact that $[D2]_{\text{fn}(D1, D2)} \lesssim_\rho [D1]_{\text{fn}(D1, D2)}$ if and only if for every ρ -well-formed trace T in $[D1]_{\text{fn}(D1, D2)}$ the closure of T and $[D2]_{\text{fn}(D1, D2)}$ have a common element. We skip the details of the implementation using metalevel facilities of Maude, as they are the same as that for asynchronous π -calculus [22].

6 Conclusion

We have presented the executable fragment of SDs as an extension of asynchronous π -calculus. We have exploited this relation to both obtain an implementation of SDs, and to develop a theory of may testing for SDs. An interesting direction for future work is to investigate the theory of may testing for the entire SD language. Features such as fairness conditions and the **constraint** predicate, which we have not considered here, change the characterization of may testing in a non-trivial way. Another problem of interest is to extend the implementation of may testing for the case where there are other infinite value domains besides names, such as integers and lists. This would involve the use of sophisticated symbolic techniques to handle these infinite value domains [10]. The resulting implementation of may testing over the full fledged SD language with a rich value set can be used for reasoning about practical examples; such case studies are also a topic of interest.

References

1. G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.
2. M. Boreale, R. De Nicola, and R. Pugliese. Trace and testing equivalence on asynchronous processes.
3. G. Boudol. Asynchrony and the π -Calculus. Technical Report 1702, Department of Computer Science, Inria Univeristy, May 1992.
4. I. Castellini and M. Hennessy. Testing theories for asynchronous languages. In *FSTTCS*, pages 90–101, 1998. LNCS 1530.
5. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Towards Maude 2.0. In Kokichi Futatsugi, editor, *Proceedings Third International Workshop on Rewriting Logic and its Applications, WRLA 2000, Kanazawa, Japan, September 18–20, 2000*, volume 36 of *Electronic Notes in Theoretical Computer Science*, pages 297–318. Elsevier, 2000. <http://www.elsevier.nl/locate/entcs/volume36.html>.
6. N. G. de Bruijn. Lambda calculus with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Proc. Koninkl. Nederl. Akademie van Wetenschappen*, 75:381–392, 1972.
7. E.W.Dijkstra and C.S.Scholten. *Predicate Calculus and Program Semantics*. Springer Verlag, 1990.
8. M. Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.
9. K. Honda and M. Tokoro. An Object Calculus for Asynchronous Communication. In *Fifth European Conference on Object-Oriented Programming*, July 1991. LNCS 512, 1991.
10. A. Ingólfssdóttir and H. Lin. A symbolic approach to value passing processes. In *Handbook of Process Algebra*, pages 427–478. Elsevier Publishing, 2001.
11. M. Merro and D. Sangiorgi. On Asynchrony in Name-Passing Calculi. In *Proceeding of ICALP '98*. Springer-Verlag, 1998. LNCS 1443.
12. José Meseguer. Rewriting as a unified model of concurrency. In J. C. M. Baeten and J. W. Klop, editors, *CONCUR'90, Theories of Concurrency: Unification and Extension, Amsterdam, The Netherlands, August 1990, Proceedings*, volume 458 of *Lecture Notes in Computer Science*, pages 384–400. Springer-Verlag, 1990.

13. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
14. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (Parts I and II). *Information and Computation*, 100:1–77, 1992.
15. R. De Nicola and M. Hennesy. Testing equivalence for processes. *Theoretical Computer Science*, 34:83–133, 1984.
16. G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Dept., Aarhus University, September 1981.
17. ITU-T Recommendation Z.120. Message sequence charts, 1996.
18. J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual*. Addison-Wisely, 1998.
19. S. Smith and C. Talcott. Modular reasoning for actor specification diagrams. In *Formal Methods in Object-Oriented Distributed Systems*. Kluwer Academic Publishers, 1999.
20. S. Smith and C. Talcott. Specification diagrams for actor systems. *Higer-Order and Symbolic Computation*, 2002. To appear.
21. M. O. Stehr. CINNI — A generic calculus of explicit substitutions and its application to λ , ζ - and π -calculi. In Kokichi Futatsugi, editor, *Proceedings Third International Workshop on Rewriting Logic and its Applications, WRLA 2000, Kanazawa, Japan, September 18–20, 2000*, volume 36 of *Electronic Notes in Theoretical Computer Science*, pages 71–92. Elsevier, 2000. <http://www.elsevier.nl/locate/entcs/volume36.html>.
22. P. Thati, K. Sen, and N. Martí-Oliet. An executable specification of asynchronous π -calculus and may testing in Maude 2.0. In *International Conference on Rewriting Logic and its Applications*, 2002. Electronic Notes in Theoretical Computer Science, vol. 71.
23. P. Thati, R. Ziaei, and G. Agha. A theory of may testing for actors. In *Formal Methods for Open Object-based Distributed Systems*, March 2002.
24. P. Thati, R. Ziaei, and G. Agha. A theory of may testing for asynchronous calculi with locality and no name matching. In *Proceedings of the 9th International Conference on Algebraic Methodology and Software Technology*, pages 222–238. Springer Verlag, September 2002. LNCS.
25. A. Verdejo and N. Martí-Oliet. Implementing CCS in Maude 2. In *International Conference on Rewriting Logic and its Applications*, 2002. Electronic Notes in Theoretical Computer Science, vol. 71.