Definition and Expansion of Composite Automata in IOA
MIT-LCS-TR-959

Joshua A. Tauber and Stephen J. Garland

July 31, 2004

Contents

List of Figures

List of Tables

1 Introduction

The IOA language provides notations for defining both primitive and composite I/O automata. This note describes, both formally and with examples, the constraints on these definitions, the composability requirements for the components of a composite automaton, and the transformation of a composite automaton into an equivalent primitive automaton.

Section 2 introduces four examples used throughout this note to illustrate new definitions and operations. Section 3 treats IOA programs for primitive I/O automata: it introduces notations for describing the syntactic structures that appear in these programs, and it lists syntactic and semantic conditions that these programs must satisfy to represent valid primitive I/O automata. Section 4 describes how to reformulate primitive IOA programs into an equivalent but more regular (desugared) form that is used in later definitions in this note. Section 5 treats IOA programs for composite I/O automata: it introduces notations for describing the syntactic structures that appear in these programs, describes resortings induced by them, and lists syntactic and semantic conditions that these programs must satisfy to represent valid composite I/O automata. Section 6 describes the translation of the name spaces of component automata into a unified name space for the composite automaton. Section 7 shows how to expand an IOA program for a composite automaton into an equivalent IOA program for a primitive automaton. The expansion is generated by combining syntactic structures of the desugared programs for the component automata after applying appropriate replacements of sorts and variables. Section 8 details the expansion of the composite automaton introduced in Section 2 using the desugared forms developed throughout Sections 4–6 and the techniques described in Section 7. Finally, Section 9 gives a precise definition of the resortings and substitutions used to replace sorts and variables.

Nancy Lynch and Mandana Vaziri contributed to the design of the composition mechanisms described in this note. Dilsun Kaynar suggested numerous and substantial clarifications in the note's presentation.

2 Illustrative examples

We use several examples of primitive and composite automata to illustrate both the notations provided by IOA and also the formal semantics of IOA. We refer to Examples 2.1–2.3 throughout Sections 3–8. Example 2.4 is relevant only to Sections 5–8.

Example 2.1 Figure 2.1 contains an IOA specification for a communication channel that can both drop duplicate messages and reorder messages. Type parameters for the specification, `Node` and `Msg`, represent the set of nodes that can be connected by channels and the set of messages that can be transmitted. Individual parameters, `i` and `j`, represent the nodes connected by a particular channel.

Two features of this example warrant particular attention later in this note. First, the example uses both type and variable automaton parameters. Second, it uses the keyword **const** to indicate that the parameters `i` and `j` in the action signature are terms referring to the parameters `i` and `j` of the automaton, rather than fresh variable declarations.

```
automaton Channel(Node, Msg:type, i, j:Node)
  signature
    input send(const i, const j, m:Msg)
    output receive(const i, const j, m:Msg)
  states contents:Set[Msg] := {}
  transitions
    input send(i, j, m)
      eff contents := insert(m, contents)
    output receive(i, j, m)
      pre m ∈ contents
      eff contents := delete(m, contents)
```

Figure 2.1: Sample automaton `Channel`

Example 2.2 Figure 2.2 contains the specification for a process that runs on a node indexed by a natural number and that communicates with its neighbors by sending and receiving messages that consist of natural numbers. The process records the smallest value it has received and passes on all values that exceed the recorded value; if the set of values waiting to be passed on grows too large, the process can also lose a nondeterministic set of those values. Interesting features of this example include the use of terms as parameters in transition definitions and a **local** variable representing an initial nondeterministic choice and temporary state local to the transition. (The keyword **local**, newly added to the IOA language, replaces and extends the keyword **choose** formerly used to introduce hidden parameters. See Section 3 for a fuller description of **local** parameters.)

Example 2.3 Figure 2.3 contains the specification for another process that watches for `overflow` actions and reports those that meet a simple criterion. Interesting features of this example include more complicated uses of type parameters and **where** clauses, both in the action signature and to distinguish two transition definitions for a single action.

Example 2.4 Finally, Figure 2.4 contains the specification of an automaton formed by composing instances of these three primitive automata. This specification relies on an auxiliary specification, shown in Figure 2.5, to define the term `between(1, nProcesses)`.

```
automaton P(n:Int)
  signature
    input receive(const n-1, const n, x:Int)
    output send(const n, const n+1, x:Int),
           overflow(const n, s:Set[Int])
  states
    val:Int := 0,
    toSend:Set[Int] := {}
  transitions
    input receive(n-1, n, x)
      eff if val = 0 then val := x
          elseif x < val then
            toSend := insert(val, toSend);
            val := x
          elseif val < x then
            toSend := insert(x, toSend)
          fi
    output send(n, n+1, x)
      pre x ∈ toSend
      eff toSend := delete(x, toSend)
    output overflow(n, s:Set[Int]; local t:Set[Int])
      pre s = toSend ∧ n < size(s) ∧ t ⊆ s
      eff toSend := t
```

Figure 2.2: Sample automaton `P`

```
automaton Watch(T:type, what:Set[T])
  signature
    input overflow(x:T, s:Set[T]) where x ∈ what
    output found(x:T) where x ∈ what
  states seen:Array[T,Bool] := constant(false)
  transitions
    input overflow(x, s ∪ {x})
      eff seen[x] := true
    input overflow(x, s) where ¬(x ∈ s)
      eff seen[x] := false
    output found(x)
      pre seen[x]
```

Figure 2.3: Sample automaton `Watch`

```
axioms Between(Int, ≤)

automaton Sys(nProcesses: Int)
  components C[n:Int]: Channel(Int, Int, n, n+1)
                   where 1 ≤ n ∧ n < nProcesses;
             P[n:Int] where 1 ≤ n ∧ n ≤ nProcesses;
             W: Watch(Int, between(1, nProcesses))
  hidden send(nProcesses, nProcesses+1, m)

invariant of Sys:
  ∀ m:Int ∀ n:Int (1 ≤ m ∧ m < n ∧ n ≤ nProcesses
                       ⇒ P[m].val < P[n].val ∨  P[n].val = 0)
```

Figure 2.4: Sample composite automaton `Sys`

```
Between(T, ≤:T,T→Bool): trait
  includes Set(T)
  introduces
    __≤__: T, T → Bool
    between: T, T → Set[T]
  asserts with x, y, z: T
    x ∈ between(y, z) ⇔ y ≤ x ∧ x ≤ z
```

Figure 2.5: Auxiliary definition of function `between`

3 Definitions for primitive automata

In order to describe syntactic manipulations of IOA programs, we introduce a nomenclature for their syntactic elements. We expose just those elements of an IOA program we use to describe the expansion of composite automata into primitive form. Section 3.1 introduces nomenclature for, and the meaning of, syntactic structures in primitive automata. Section 3.2 examines how states are represented and referenced in primitive IOA programs. Sections 3.3 and 3.4 describe semantic conditions that must hold for an IOA program to represent a valid primitive I/O automaton.

3.1 Syntax

Figure 3.1 illustrates the general form of an IOA definition for a primitive I/O automaton. The figure exposes just those elements of an IOA program we use to describe the expansion of composite automata into primitive form. It does not expose the individual statements that appear in an **eff** clause. (These are treated separately in Section 9.) Rather the figure simply refers to the "program" (i.e., the complete sequence of statements) in an **eff** clause.

automaton $A(params^A)$
  assumes $Assumptions$
  signature
    ...
    input $\pi(params_{in}^{A,\pi})$ **where** $P_{in}^{A,\pi}$
    output $\pi(params_{out}^{A,\pi})$ **where** $P_{out}^{A,\pi}$
    internal $\pi(params_{int}^{A,\pi})$ **where** $P_{int}^{A,\pi}$
    ...
  states $stateVars^A := initVals^A$ **initially** P_{init}^A
  transitions
    ...
    input $\pi(params_{in,t_j}^{A,\pi};$ **local** $localVars_{in,t_j}^{A,\pi})$ **case** t_j **where** $P_{in,t_j}^{A,\pi}$
      eff $Prog_{in,t_j}^{A,\pi}$ **ensuring** $ensuring_{in,t_j}^{A,\pi}$
    output $\pi(params_{out,t_j}^{A,\pi};$ **local** $localVars_{out,t_j}^{A,\pi})$ **case** t_j **where** $P_{out,t_j}^{A,\pi}$
      pre $Pre_{out,t_j}^{A,\pi}$
      eff $Prog_{out,t_j}^{A,\pi}$ **ensuring** $ensuring_{out,t_j}^{A,\pi}$
    internal $\pi(params_{int,t_j}^{A,\pi};$ **local** $localVars_{int,t_j}^{A,\pi})$ **case** t_j **where** $P_{int,t_j}^{A,\pi}$
      pre $Pre_{int,t_j}^{A,\pi}$
      eff $Prog_{int,t_j}^{A,\pi}$ **ensuring** $ensuring_{int,t_j}^{A,\pi}$
    ...

Figure 3.1: General form of a primitive automaton

Notations and writing conventions

In Figure 3.1, $params^A$ denotes the sequence of type and variable declarations that serve as the parameters of the automaton A. The *Assumptions* are LSL theories defining required properties for these parameters. Notations $params^{A,\pi}_{kind}$ and $params^{A,\pi}_{kind,t_j}$, where *kind* is one of *in*, *out*, or *int*, denote sequences of variables and/or terms that serve as parameters for the action π and its transition definitions. The notations $P^{A,\pi}_{kind}$, P^{A}_{init}, $P^{A,\pi}_{kind,t_j}$, $Pre^{A,\pi}_{kind,t_j}$, and $ensuring^{A,\pi}_{kind,t_j}$ denote predicates (i.e., boolean-valued expressions). The notation $initVals^A$ denotes the sequence of terms or **choose** expressions serving as initial values for the state variables. If the definition of A does not specify an initial value for some state variable, we treat the declaration of that state variable as equivalent to one of the form `x:T :=` **choose** `t:T` **where** `true`. The notation $Prog^{A,\pi}_{kind,t_j}$ denotes a program. The notation $localVars^{A,\pi}_{kind,t_j}$ denotes a sequence of variables. In general, a notation ending with an "s" denotes a sequence of zero or more elements.

Our conventions for decorating syntactic structures throughout this paper are as follows. Superscripts refer either to automaton names or to automaton-name/action-name pairs. Automaton names are capitalized (e.g., A, C_i, `P`). Action names are not capitalized and are either Greek letters (e.g., π, π_1) or written in `mono-spaced font` (e.g., `send`). Subscripts refer to more specific restrictions such as action kind (i.e., *in*, *out*, or *int*), transition label (e.g., t_1), or origin (e.g., *desug*). IOA keywords appear in a **small-bold roman font**. References to other text in sample IOA programs appear in a `mono-spaced font`. Syntactic structure labels and names in general IOA programs are *italicized*.

Syntactic elements of primitive IOA programs

Variables in IOA programs can be declared explicitly as automaton parameters ($vars^A$, which is a subsequence of $params^A$), as state variables ($stateVars^A$), or as local variables ($localVars^{A,\pi}_{kind,t_j}$); they can also be declared implicitly as post-state variables that correspond to state variables, post-local variables corresponding to local variables, or by their appearance in action parameters ($vars^{A,\pi}_{in}$, which appear in $params^{A,\pi}_{in}$) or in transition parameters ($vars^{A,\pi}_{in,t_j}$, which appear in $params^{A,\pi}_{in,t_j}$). Variables in IOA programs can appear in parameters, terms, predicates, and programs. For simplicity, Figure 3.1 does not indicate which variables may have free occurrences in which parameters, terms, predicates, or programs; Section 3.3 describes which can occur where. As an illustration, variables that occur freely in $P^{A,\pi}_{in}$ must be in one of the sequences $vars^A$ or $vars^{A,\pi}_{in}$.

Below, we define each labeled syntactic structure and then illustrate it using selections from Examples 2.1–2.3.

Parameters

- $params^A$ is the sequence of formal parameters for A, which can be either variables or **type** parameters. We decompose $params^A$ into two disjoint subsequences, one ($vars^A$) containing variable declarations and the other ($types^A$) containing **type** parameters (identifiers qualified by the keyword **type**). For example, $params^{\texttt{Watch}}$ is $\langle$`T:`**type**`, what:Set[T]`$\rangle$, which consists of a **type** parameter `T` followed by a variable `what:Set[T]`. Hence $types^{\texttt{Watch}}$ is $\langle$`T:`**type**$\rangle$ and $vars^{\texttt{Watch}}$ is $\langle$`what:Set[T]`$\rangle$.

- $params^{A,\pi}_{kind}$ is the sequence of parameters for the set of actions of type *kind* named by π

in A's signature. Action parameters can be either variables or **const** terms.[1] For example, $params_{in}^{\texttt{Channel},\texttt{send}}$ is $\langle$**const** `i`, **const** `j`, `m:Msg`$\rangle$.

- $params_{kind,t_j}^{A,\pi}$ is the sequence of terms serving as parameters for transition definition t_j for actions of type *kind* named by π. Whereas π can appear at most once as the name of an input, output, and internal action in A's signature, it can have more than one transition definition as an input, output, and internal action. For example, $params_{in,t_1}^{\texttt{Watch},\texttt{overflow}}$ is $\langle$`x, s` $\cup$ `{x}`$\rangle$ and $params_{in,t_2}^{\texttt{Watch},\texttt{overflow}}$ is $\langle$`x, s`$\rangle$.

Variables

- As noted above, $vars^A$ is the sequence of variables that are declared explicitly in $params^A$, that is, $vars^A$ is the sequence of identifiers in $params^A$ qualified by some sort other than **type**.[2] For example, $vars^{\texttt{Channel}}$ is $\langle$`i:Node, j:Node`$\rangle$.
- $vars_{kind}^{A,\pi}$ is the sequence of variable declarations (i.e., non-**const** parameters) in $params_{kind}^{A,\pi}$. For example, $vars_{in}^{\texttt{Channel},\texttt{send}}$ is $\langle$`m:Msg`$\rangle$.
- $stateVars^A$ is the sequence of state variables of A. For example, $stateVars^{\texttt{Channel}}$ is $\langle$`contents:Set[Msg]`$\rangle$.
- $postVars^A$ is the sequence of variables for post-states of A that can occur in any $ensuring_{kind,t_j}^{A,\pi}$. These variables are primed versions of variables in $stateVars^A$. For example, $postVars^{\texttt{P}}$ is $\langle$`val′:Int, toSend′:Set[Int]`$\rangle$.[3]
- $vars_{kind,t_j}^{A,\pi}$ is the sequence of variables that occur freely in $params_{kind,t_j}^{A,\pi}$, but are not in $vars^A$. For example, $vars_{out,t_1}^{\texttt{P},\texttt{send}}$ is $\langle$`x:Int`$\rangle$, because `n` is in $vars^{\texttt{P}}$.
- $localVars_{kind,t_j}^{A,\pi}$ is a sequence of additional **local** variables for transition definition t_j for actions of type *kind* named π; these variables are not listed as parameters of π in the signature of A. For example, $localVars_{out,t_1}^{\texttt{P},\texttt{overflow}}$ is $\langle$`t:Set[Int]`$\rangle$.
- $localPostVars_{kind,t_j}^{A,\pi}$ is the sequence of *post-local* variables that name the values of local variables after execution of $Prog_{kind,t_j}^{A,\pi}$. These variables are primed versions of variables in $localVars_{kind,t_j}^{A,\pi}$ that appear on the left side of an assignment statement in the transition definition and that can occur in $ensuring_{kind,t_j}^{A,\pi}$.

[1] We may want to consider an alternative treatment for action parameters, similar to that for $params_{kind,t_j}^{A,\pi}$, that would dispense with the keyword **const** and treat all action parameters as terms, rather than as a mixture of terms and variable declarations. The current treatment allows factored notations, such as $\pi(i, j{:}Int)$, which introduce a list of variables of a given sort; the alternative treatment would require unfactored notations, such as $\pi(i{:}Int, j{:}Int)$, in which a sort qualification applies only to the term it follows immediately.

[2] When we define a sequence by selecting some members of another sequence, we preserve order in projecting from the defining sequence to the defined sequence. For example, if $u{:}S$ precedes $v{:}T$ in $params^A$, then $u{:}S$ precedes $v{:}T$ in $vars^A$.

[3] Previously, only the primed versions of state variables that appeared on the left side of an assignment statement in the transition definition were allowed to appear in an **ensuring** clause. For example, we defined $postVars_{out,t_1}^{\texttt{P},\texttt{send}}$ to be $\langle$`toSend′:Set[Int]`$\rangle$, which did not include the variable `val′`, because `val` does not appear on the left side of an assignment in this transition definition. The more complicated definition was intended as a safeguard against specifiers writing `val′` in an **ensuring** clause when there was no way the value of `val′` could differ from that of `val`. However, the more complicated definition did not safeguard against all such errors, because specifiers could still write `A′.val` in an **ensuring** clause. Hence the simpler definition appears preferable.

Predicates

- $P^{A,\pi}_{kind}$ is the **where** clause for the set of actions of type *kind* named by π in A's signature. For example, $P^{\texttt{Watch},\texttt{found}}_{out}$ is $\texttt{x} \in \texttt{what}$. If $P^{A,\pi}_{kind}$ is not specified explicitly, it is taken to be *true*. If action π does not appear as a particular kind—input, output, or internal—in A's signature, then $P^{A,\pi}_{kind}$ is defined to be *false*.

- P^{A}_{init} is a predicate constraining the initial values for A's state variables. If it is not specified explicitly, it is taken to be *true*.

- $P^{A,\pi}_{kind,t_j}$ is the **where** clause for transition definition t_j for actions of type *kind* named by π. For example, $P^{\texttt{Watch},\texttt{overflow}}_{in,t_2}$ is $\neg(\texttt{x} \in \texttt{s})$. If $P^{A,\pi}_{kind,t_j}$ is not specified explicitly, it is taken to be *true*. If action π does not appear as a particular kind in A's signature, then $P^{A,\pi}_{kind,t_j}$ is defined to be *false*.

- $Pre^{A,\pi}_{kind,t_j}$ is the precondition for transition definition t_j for actions of type *kind* named π, where *kind* is *out* or *int*. For example, $Pre^{\texttt{P},\texttt{send}}_{out,t_1}$ is $\texttt{x} \in \texttt{toSend}$. If $Pre^{A,\pi}_{kind,t_j}$ is not specified explicitly, it is taken to be *true*. For every input transition, $Pre^{A,\pi}_{in,t_j}$ is defined to be *true* because transition definitions for input actions do not have preconditions.

- $ensuring^{A,\pi}_{kind,t_j}$ is the **ensuring** clause in the effects clause in transition definition t_j for actions of type *kind* named π. If $ensuring^{A,\pi}_{kind,t_j}$ is not specified explicitly, it is taken to be *true*. In the examples, all **ensuring** clauses are `true` by default.[4]

Programs and values

- $Prog^{A,\pi}_{kind,t_j}$ is the program in the effects clause in transition definition t_j for actions of type *kind* named π. For example, $Prog^{\texttt{P},\texttt{overflow}}_{out,t_1}$ is `toSend := t`.

- $initVals^{A}$ is the sequence of initial values for A's state variables, which can be specified as either terms or **choose** expressions. A state variable without an explicit initial value is equivalent to one with an unconstrained initial value, that is, to one specified by a **choose** expression constrained by the predicate *true*. For example, $initVals^{\texttt{P}}$ is $\langle \texttt{0, \{\}} \rangle$.

- t_j is an optional identifier used to distinguish transition definitions of the same *kind* for the same action π. If there is no **case** clause, t_j is taken to be an arbitrary, but unique label.[5]

[4] The keyword **ensuring** replaces the **so that** keyword, which has been removed from IOA. Formerly, **so that** was used to introduce three types of predicates in IOA: the initialization predicate for automaton state, the post-state predicate for transition definitions, and the loop variable predicate in **for** statements. This multiple use was confusing. Furthermore, the keyword **where** also introduces predicates, which led to additional confusion. In the new syntax, automaton state predicates are introduced by **initially**, post-state predicates are introduced by **ensuring**, and all other predicates (including **for** predicates) are introduced by **where**. The semantics of the clauses containing these predicates has not changed.

[5] The **case** clause was introduced for use by the IOA simulator; it is not described yet in the IOA manual.

3.2 Aggregate sorts for state and local variables

State variables

The value (or the lvalue) of any state variable (e.g., `toSend:Set[Int]`) may be referenced using that variable (e.g., `toSend`) as if it were a constant operator (e.g., `toSend:` $\rightarrow$ `Set[Int]`).[6] However, in contexts that involve more than a single automaton (e.g., simulation relations or composite automata), such variable references may be ambiguous. Hence IOA provides an equivalent, unambiguous notation for the values of state variables.

For each automaton A without type parameters, IOA automatically defines a sort $States[A]$, known as the *aggregate state sort* of A, as a tuple sort with a selection operator $\_\_.v{:}States[A] \rightarrow T$ for each state variable v of sort T. IOA also automatically defines variables A and A' of sort $States[A]$ to represent the *aggregate state* and *aggregate post-state* of A. The terms $A.v$ and $A'.v$ are equivalent to references to the state variable v and to its value v' in a post-state. For example, `States[P]` = **tuple of** `val:Int, toSend:Set[Int]`, and `P.val` is a term of sort `Int` equivalent to the state variable `val`.

If an automaton A has type parameters, the notation for its aggregate state sort is more complicated, because there can be different instantiations of A with different actual types, and a simple notation $States[A]$ for the aggregate state sort would be ambiguous. To avoid this ambiguity, IOA includes the type parameters of A (if any) in the notation $States[A, types^A]$ for the aggregate state sort of A, and the aggregate state and post-state variables A have this sort $States[A, types^A]$. For example, `States[Channel,Node,Msg]` = **tuple of** `contents:Set[Msg]`, and `Channel.contents` is a term of sort `Set[Msg]` equivalent to the state variable `contents`.

As we will see in Section 5.2, including type parameters in the name of the aggregate state sort enables us to generate distinct aggregate state sorts for each instantiation of A.

Local variables

In previous editions of the language, IOA introduced hidden action parameters with the keyword **choose** appearing subsequent to the **where** clause. Thus, hidden or **choose** parameters could not appear in the **where** clause. In the course of writing this document, we discovered a need for hidden parameters in the **where** clauses of desugared input actions (see Section 4). In addition, we believed that the ability to assign (temporary) values to hidden parameters would simplify the definitions of expanded transition definitions of composite automata.[7] We introduced local variables into IOA to serve both these purposes. Local variables replace and extend **choose** parameters. Thus, the keyword **local** replaces the keyword **choose** in transition definition parameter lists and local variables are those introduced following the keyword **local** in these parameter lists.

In the new notation, the scope of local variables extends to the whole transition definition, not just to the precondition and effects. In addition, local variables may be assigned values in the **eff** clause. Semantically, local variables are *not* part of the state of the I/O automaton represented by an IOA program. Rather, they define *intermediate states* that occur during the execution of an atomic transitions, but are not visible externally. Therefore, local variables may not appear in simulation relations or invariants.

Although local variables differ significantly from state variables in terms of semantics, their syntactic treatment is similar. As for state variables, IOA automatically defines an *aggregate local*

[6] An unambiguous variable identifier can be used alone. If two variables defined in the same scope have the same identifier, but different sorts, their identifier may need to be qualified by their sorts.

[7] In the end, our final definitions in Sections 7.6–7.9 do not to use this feature. However, the ability to assign to local variables was deemed useful and remains in the language.

sort, together with *aggregate local* and *post-local* variables, to provide a second, equivalent notation for references to local and post-local variables. For every transition definition t_j for an action π of type *kind* in automaton A, the aggregate local sort $Locals[A, types^A, kind, \pi, t_j]$ is a tuple sort with a selection operator $\_\_.v{:}States[A] \rightarrow T$ for each local variable v of sort T. Furthermore, aggregate local and post-local variables, A and A' of sort $localVars^{A,\pi}_{kind,t_j}$, are defined in the scope of that transition definition. If there is only one transition definition for an action π of type *kind*, we omit t_j in the notation for this sort. For example, the aggregate locals sort *Locals*[`P,out,overflow`] is **tuple of** `t:Set[Int]`, and `P.t` is a term of sort `Set[Int]` equivalent to the local variable `t` in the scope of `overflow`.

Note that the automaton name A is used as the identifier for *two* aggregate variables in every transition definition: $A{:}States[A, types^A]$ and $A{:}Locals[A, types^A, kind, \pi, t_j]$. As specified in Section 3.3, $stateVars^A$ and $localVars^{A,\pi}_{kind,t_j}$ must have no variables in common. Therefore, the aggregate sorts have no selection operators in common and there is no ambiguity.

The initial values of local variables are constrained by the **where** predicate of the declaring transition definition. In particular, a transition *kind* $\pi(\ldots)$ **case** t_j is defined only for values of its parameters that

1. satisfy the **where** clause of that *kind* of π in the signature of A, and

2. together with some choice of initial values for its local variables, satisfy the **where** clause of the transition definition.

A transition is *enabled* only for the values of its parameters and local variables for which it is defined and for which the precondition, if any, is satisfied.

Thus, the initial values of local variables are chosen nondeterministically from among the values that meet these constraints. Local variables serve as hidden parameters with the semantics formerly applied to **choose** parameters. We provide a formal treatment of the "values of its parameters" and "some choice of values" at the end of Section 4.

Example 3.1 The **type** declarations and variables automatically defined for the sample automata `Channel`, `P`, and `Watch` are shown in Figure 3.2.

```
type States[Channel,Node,Msg] = tuple of contents:Set[Msg]
type States[P] = tuple of val:Int, toSend:Set[Int]
type States[Watch,T] = tuple of seen:Array[T,Bool]
type Locals[P,out,overflow] = tuple of t:Set[Int]

Channel: States[Channel, Node, Msg]
P: States[P]
Watch: States[Watch,T]
P: Locals[P,out,overflow]
```

Figure 3.2: Automatically defined types and variables for sample automata

3.3 Static semantic checks

The following conditions must be true for an IOA program to represent a valid primitive I/O automaton. These conditions, which can be checked statically, are currently performed by `ioaCheck`, the IOA parser and static-semantic checker.

LOCATION OF TERM	VARIABLES THAT CAN OCCUR FREELY IN TERM
$params^A$	$vars^A$
$params^{A,\pi}_{kind}$	$vars^A$, $vars^{A,\pi}_{kind}$
$P^{A,\pi}_{kind}$	$vars^A$, $vars^{A,\pi}_{kind}$
$initVals^A$	$vars^A$
P^A_{init}	$vars^A$, $stateVars^A$
$params^{A,\pi}_{kind,t_j}$	$vars^A$, $vars^{A,\pi}_{kind,t_j}$
$P^{A,\pi}_{kind,t_j}$	$vars^A$, $vars^{A,\pi}_{kind,t_j}$, $localVars^{A,\pi}_{kind,t_j}$
$Pre^{A,\pi}_{kind,t_j}$	$vars^A$, $vars^{A,\pi}_{kind,t_j}$, $localVars^{A,\pi}_{kind,t_j}$, $stateVars^A$
$Prog^{A,\pi}_{kind,t_j}$	$vars^A$, $vars^{A,\pi}_{kind,t_j}$, $localVars^{A,\pi}_{kind,t_j}$, $stateVars^A$
$ensuring^{A,\pi}_{kind,t_j}$	$vars^A$, $vars^{A,\pi}_{kind,t_j}$, $localVars^{A,\pi}_{kind,t_j}$, $stateVars^A$, $postVars^A$, $localPostVars^{A,\pi}_{kind,t_j}$

Table 3.1: Variables that can occur freely in terms in the definition of a primitive automaton. Variables listed on the right may occur freely in the syntactic structure listed to their left.

✓ No sort appears more than once in $types^A$.

✓ Each action name (e.g., π) occurs at most three times in the signature of an automaton: at most once in a list of input actions, at most once in a list of output actions, and at most once in a list of internal actions.

✓ Each occurrence of an action name (e.g., π) in the signature of an automaton, or in one of its transition definitions, must be followed by the same number and sorts of parameters.

✓ The sequences $vars^A$ and $vars^{A,\pi}_{kind}$ of variables contain no duplicates; furthermore, no variable appears in both $vars^A$ and $vars^{A,\pi}_{kind}$ for any value of $kind$.[8]

✓ For each transition definition t_j for an action of type $kind$ named π, no variable appears more than once in the combination of the sequences $vars^A$, $stateVars^A$, $postVars^A$, $vars^{A,\pi}_{kind,t_j}$, $localVars^{A,\pi}_{kind,t_j}$, and $localPostVars^{A,\pi}_{kind,t_j}$.

✓ For each transition definition t_j for an action of type $kind$ named π, and for any identifier v and sort S, the sequences $stateVars^A$ and $localVars^{A,\pi}_{kind,t_j}$ do not contain both of the variables $v{:}S$ and $v'{:}S$.

✓ Any operator that occurs in a term used in the definition of an automaton must be introduced by a type definition or **axioms** clause in the IOA specification that contains the automaton

[8]This restriction is designed to avoid the confusion that would result if variables in $vars^{A,\pi}_{kind}$ are allowed to hide or override variables with the same identifiers and sorts in $vars^A$. A stronger restriction would prohibit an identifier from appearing in two different variables (of different sorts) in $vars^A$ and $vars^{A,\pi}_{kind}$; this restriction would avoid the need to pick a fresh variable when an instantiation of A causes two variables with the same identifier to clash by mapping their sorts to a common sort. However, IOA does not make this stronger restriction.

definition, by a theory specified in the **assumes** clause of the definition, or by a built-in datatype of IOA.

✓ Any variable that occurs freely in a term used in the definition of an automaton must satisfy the restrictions imposed by Table 3.1.

3.4 Semantic proof obligations

The following conditions must also be true for an IOA program to represent a valid I/O automaton. Except in special cases, these conditions cannot be checked automatically, because they may require nontrivial proofs (or even be undecidable); hence static semantic checkers must translate all but the simplest of them into proof obligations for an automated proof assistant. These proof obligations must be discharged using the axioms provided by IOA's built-in types, by the theories associated with the type definitions and the **axioms** in the IOA specification that contains the automaton definition, and by the theories associated with the **assumes** clause of that definition.

✓ The sets of input, output, and internal actions in an I/O automaton must be disjoint. Thus, for each sequence of values for the parameters of an action named π in the definition of an automaton A, at most one of $P^{A,\pi}_{in}$, $P^{A,\pi}_{out}$, and $P^{A,\pi}_{int}$ can be true.

  Special cases arise if two of the three signature **where** clauses for π are literally *false* or if two of three clauses are literally *true*. In the former case, the check automatically succeeds; in the latter, it automatically fails.

✓ There must be a transition defined for every action specified in the signature. Thus, for each sequence of values for the parameters of an action named π that make $P^{A,\pi}_{kind}$ true, there must be a transition definition t_j for π of type *kind* such that $P^{A,\pi}_{kind,t_j}$ is true for these values together with some values for the local variable of that transition definition.

✓ For each *kind* of each action π, at most one transition definition t_j can be defined for each sequence of parameters values. That is, for each sequence of values, $P^{A,\pi}_{kind,t_j}$ can be true for at most one value of j.

  Special cases arise if all but one of the transition definition **where** clauses for a kind of an action are literally *false* or any two are literally *true*. In the former case, the check automatically succeeds; in the latter, it automatically fails.

We define these proof obligations more formally at the end of Section 4.

4 Desugaring primitive automata

The syntax for IOA programs described in Section 3 allows some flexibility of expression. However, when defining semantic checks and algorithmic manipulations (e.g., composition) of IOA programs, it is helpful to restrict attention, without loss of generality, to IOA programs that conform to a more limited syntax.

In this section, we describe how to transform any primitive IOA program (as in Figure 3.1) into an equivalent program (Figure 4.7) written with a more limited syntax. We describe this transformation in four stages. First, in Section 4.1, we show how to *desugar* terms that appear as parameters by replacing them with variables constrained by **where** clauses; that is, we show how to reformulate action and transition definitions so as to eliminate the use of terms as parameters. Second, in Section 4.2, we show how to introduce *canonical parameters* into desugared actions and transition definitions. A canonicalized action is parameterized by the same sequence of variables in all appearances, both in the signature and in the transition definitions. Third, in Section 4.3, we *combine* all transition definitions of a single kind of an action into a single transition definition. Fourth, in Section 4.4, we convert each reference to a state variable x to the equivalent reference $A.x$ defined in Section 3.2. In Section 4.5, we summarize the effects of these desugarings, which are illustrated in Figure 4.7. Finally, in Section 4.6, we use the result of the first two transformations to formalize the semantic proof obligations introduced in Section 3.

4.1 Desugaring terms used as parameters

Signature

We desugar **const** parameters for an action in A's signature by introducing fresh variables and modifying the action's **where** clause. For each **const** parameter we introduce a fresh variable and add a conjunct to the **where** clause that equates the new variable with the term that served as the **const** parameter. For example, if t is a term of sort T, then we desugar the action

$$\textbf{input } \pi(vars_{in}^{A,\pi}, \textbf{const } t) \textbf{ where } P_{in}^{A,\pi}$$

as

$$\textbf{input } \pi(vars_{in}^{A,\pi}, v{:}T) \textbf{ where } v = t \ \wedge \ P_{in}^{A,\pi}$$

Here, $v{:}T$ is a fresh variable, that is, one that does not appear in $vars^A$, $vars_{in}^{A,\pi}$, $stateVars^A$, $postVars^A$, $localVars_{in,t_j}^{A,\pi}$, or $localPostVars_{in,t_j}^{A,\pi}$ for any j.[9]

Let $P_{kind,desug}^{A,\pi}$ be the **where** predicate that results after all **const** parameters in $params_{kind}^{A,\pi}$ have been desugared. Let $vars_{kind,desug}^{A,\pi}$ be the sequence of distinct variables that parameterize π after desugaring. Note that all variables that occur freely in $P_{kind,desug}^{A,\pi}$ are either in $vars_{kind,desug}^{A,\pi}$ or in $vars^A$. In general, $vars_{kind,desug}^{A,\pi}$ is a supersequence of $vars_{kind}^{A,\pi}$ (in that it contains a fresh variable for each **const** parameter in $params_{kind}^{A,\pi}$). In the above example, a **const** parameter appears in

[9] For the purposes of this transformation, it suffices to pick some $v{:}T$ that does not appear in either $vars^A$ or $vars_{in}^{A,\pi}$. However, by ensuring that $v{:}T$ is distinct from additional variables, we avoid having to replace it by yet another fresh variable when we introduce canonical transition parameters, as described in Section 4.2. Furthermore, to avoid any ambiguity that may arise when two variables share an identifier, and to avoid having to replace $v{:}T$ by yet another fresh variable in an instantiation of A that maps T and the sort of another variable with identifier v to a common sort, it is helpful to pick v to be an identifier that does not appear in $vars^A$, $vars_{in}^{A,\pi}$, $stateVars^A$, $postVars^A$, $localVars_{in,t_j}^{A,\pi}$, or $localPostVars_{in,t_j}^{A,\pi}$ for any j.

$$
\begin{array}{l}
\textbf{automaton } A(\mathit{types}^A, \mathit{vars}^A) \\
\quad \textbf{signature} \\
\qquad \ldots \\
\qquad \textbf{input } \pi(\mathit{vars}^{A,\pi}_{in,desug}) \textbf{ where } P^{A,\pi}_{in} \;\wedge\; \mathit{vars}^{A,\pi}_{in,desug} = \mathit{params}^{A,\pi}_{in} \\
\qquad \textbf{output } \pi(\mathit{vars}^{A,\pi}_{out,desug}) \textbf{ where } P^{A,\pi}_{out} \;\wedge\; \mathit{vars}^{A,\pi}_{out,desug} = \mathit{params}^{A,\pi}_{out} \\
\qquad \textbf{internal } \pi(\mathit{vars}^{A,\pi}_{int,desug}) \textbf{ where } P^{A,\pi}_{int} \;\wedge\; \mathit{vars}^{A,\pi}_{int,desug} = \mathit{params}^{A,\pi}_{int} \\
\qquad \ldots \\
\quad \textbf{states } \mathit{stateVars}^A := \mathit{initVals}^A \textbf{ initially } P^A_{init} \\
\quad \textbf{transitions} \\
\qquad \ldots \\
\qquad \textbf{input } \pi(\mathit{vars}^{A,\pi}_{in,t_j,desug};\ \textbf{local } \mathit{localVars}^{A,\pi}_{in,t_j}, \mathit{vars}^{A,\pi}_{in,t_j}) \textbf{ case } t_j \\
\qquad\qquad \textbf{where } P^{A,\pi}_{in,t_j} \;\wedge\; \mathit{vars}^{A,\pi}_{in,t_j,desug} = \mathit{params}^{A,\pi}_{in,t_j} \\
\qquad\quad \textbf{eff } \mathit{Prog}^{A,\pi}_{in,t_j} \textbf{ ensuring } \mathit{ensuring}^{A,\pi}_{in,t_j} \\
\qquad \textbf{output } \pi(\mathit{vars}^{A,\pi}_{out,t_j,desug};\ \textbf{local } \mathit{localVars}^{A,\pi}_{out,t_j}, \mathit{vars}^{A,\pi}_{out,t_j}) \textbf{ case } t_j \\
\qquad\qquad \textbf{where } P^{A,\pi}_{out,t_j} \;\wedge\; \mathit{vars}^{A,\pi}_{out,t_j,desug} = \mathit{params}^{A,\pi}_{out,t_j} \\
\qquad\quad \textbf{pre } \mathit{Pre}^{A,\pi}_{out,t_j} \\
\qquad\quad \textbf{eff } \mathit{Prog}^{A,\pi}_{out,t_j} \textbf{ ensuring } \mathit{ensuring}^{A,\pi}_{out,t_j} \\
\qquad \textbf{internal } \pi(\mathit{vars}^{A,\pi}_{int,t_j,desug};\ \textbf{local } \mathit{localVars}^{A,\pi}_{int,t_j}, \mathit{vars}^{A,\pi}_{int,t_j}) \textbf{ case } t_j \\
\qquad\qquad \textbf{where } P^{A,\pi}_{int,t_j} \;\wedge\; \mathit{vars}^{A,\pi}_{int,t_j,desug} = \mathit{params}^{A,\pi}_{int,t_j} \\
\qquad\quad \textbf{pre } \mathit{Pre}^{A,\pi}_{int,t_j} \\
\qquad\quad \textbf{eff } \mathit{Prog}^{A,\pi}_{int,t_j} \textbf{ ensuring } \mathit{ensuring}^{A,\pi}_{int,t_j} \\
\qquad \ldots
\end{array}
$$

Figure 4.1: Preliminary form of a desugared primitive automaton: all action parameters are variables

the last position of $params_{in}^{A,\pi}$. In general, **const** parameters may appear in any position. A fresh variable appears in $vars_{kind,desug}^{A,\pi}$ in the same position the **const** parameter it replaces appears in $params_{kind}^{A,\pi}$.

The preliminary form for desugaring an automaton signature shown in Figure 4.1 indicates that each variable in $vars_{kind,desug}^{A,\pi}$ is equated to the corresponding entry in $params_{kind}^{A,\pi}$. (In the figure, we use $params_{kind}^{A,\pi}$ to mean the sequence of terms without the **const** keyword.) An obvious simplification is to omit any identity conjuncts that arise when a variable in $vars_{kind}^{A,\pi}$ is equated to itself.

Transition definitions

We desugar the parameters for each transition definition for an action named π to eliminate parameters that are not just simple variable references.[10] As shown in Figure 4.1, we first replace the transition parameters $params_{kind,t_j}^{A,\pi}$ by references to distinct fresh variables $vars_{kind,t_j,desug}^{A,\pi}$, that is, to variables that do not appear in $vars^A$, $stateVars^A$, $postVars^A$, $vars_{kind,t_j}^{A,\pi}$, $localVars_{kind,t_j}^{A,\pi}$, or $localPostVars_{kind,t_j}^{A,\pi}$.[11] Second, we maintain the original semantics of the transition definition by adding conjuncts to the **where** clause to equate the new variables with the old parameters. Third, because transition definition parameters may introduce variables implicitly, but **where** clauses may not, we introduce the previously free variables (i.e., $vars_{kind,t_j}^{A,\pi}$) as additional local variables, letting $localVars_{kind,t_j,desug}^{A,\pi}$ be the concatenation of $localVars_{kind,t_j}^{A,\pi}$ and $vars_{kind,t_j}^{A,\pi}$. In effect, these steps move terms used as parameters into the **where** clause. For example, if t is a term and v is a fresh variable with the same sort as t, then we desugar the transition definition

$$\textbf{input}\ \pi(t)\ \textbf{where}\ P_{in,t_j}^{A,\pi}$$

as

$$\textbf{input}\ \pi(v; \textbf{local}\ vars_{in,t_j}^{A,\pi})\ \textbf{where}\ v = t\ \wedge\ P_{in,t_j}^{A,\pi}$$

Let $P_{kind,t_j,desug}^{A,\pi}$ be the **where** predicate that results after transition parameters have been desugared in this fashion. Then any variable that has a free occurrence in this predicate must be in $vars^A$, $vars_{kind,t_j,desug}^{A,\pi}$, or $localVars_{kind,t_j,desug}^{A,\pi}$.

After **const** and transition definition terms have been desugared, the valid occurrences of free variables in syntactic forms, shown in Table 3.1, is revised by those shown in Table 4.1. After desugaring, $params_{kind}^{A,\pi} = vars_{kind,desug}^{A,\pi}$ and $params_{kind,t_j}^{A,\pi} = vars_{kind,t_j,desug}^{A,\pi}$.

Example 4.1 The first step in desugaring the primitive automata defined in Figures 2.1–2.3 is shown in Figure 4.2. For the automaton `Channel`, `n1:Node` and `n2:Node` are fresh variables introduced to desugar the **const** parameters in the signature. Similarly, `n1:Node`, `n2:Node`, and `m1:Msg` are

[10] As mentioned in Footnote 1, we distinguish between action parameters in the signature that are terms (**const** parameters) and those that are variable declarations to provide strong typing for variable declarations. Since the sorts of $params_{kind}^{A,\pi}$ determine the sorts of $params_{kind,t_j}^{A,\pi}$, there is no need for such a distinction in transition parameters.

[11] It suffices to replace just those parameters that are not simply references to variables, because the fresh variables corresponding to such terms disappear when we substitute references to canonical variables for the parameters, as described in the next section. However, the replacement is easier to describe if we replace all parameters.

Furthermore, as for **const** parameters, to avoid any ambiguity that may arise in the **where** clause when two variables share an identifier, and to avoid having to replace $v{:}T$ by yet another fresh variable in an instantiation of A that maps T and the sort of another variable with identifier v to a common sort, it is helpful to pick v to be an identifier that is not in $vars^A$, $stateVars^A$, $postVars^A$, $vars_{kind,t_j}^{A,\pi}$, $localVars_{kind,t_j}^{A,\pi}$, or $localPostVars_{kind,t_j}^{A,\pi}$.

LOCATION OF TERM	VARIABLES THAT CAN OCCUR FREELY IN TERM
$P^{A,\pi}_{kind,desug}$	$vars^A$, $vars^{A,\pi}_{kind,desug}$
$P^{A,\pi}_{kind,t_j,desug}$	$vars^A$, $vars^{A,\pi}_{kind,t_j,desug}$, $localVars^{A,\pi}_{kind,t_j,desug}$

Table 4.1: Variables that can occur freely in terms in the definition of a desugared primitive automaton. Variables listed on the right may occur freely in the syntactic structure listed to their left.

fresh variables introduced to desugar transition parameters. Since both $vars^{\texttt{Channel},\texttt{send}}_{in,t_1}$ and $vars^{\texttt{Channel},\texttt{receive}}_{out,t_1}$ contain the single variable `m:Msg`, we introduce `m:Msg` as a local variable for each transition definition. Notice that the variables introduced for each action need be fresh only with respect to `i:Node`, `j:Node`, and `m:Msg`; furthermore, "freshness" need not extend across transitions or between actions and transitions.

The automata `P` and `Watch` are desugared in a similar fashion. Since there are no **const** parameters in the signature of `Watch`, that signature is unchanged. Since the parameters for the transition definitions for the `overflow` action in `Watch` contain two free variables, `x` and `s`, the desugared transition definitions declare these variables as local. Also, in the second of the desugared transition definitions, the desugared **where** clause incorporates the original **where** clause as a conjunct.

4.2 Introducing canonical names for parameters

Signature

IOA does not require that the sequences of variables $vars^{A,\pi}_{in}$, $vars^{A,\pi}_{out}$, and $vars^{A,\pi}_{int}$ be the same. For example, **const** parameters may cause these sequences to have different lengths. However, since IOA requires $params^{A,\pi}_{in}$, $params^{A,\pi}_{out}$, and $params^{A,\pi}_{int}$ to contain the same number and sorts of elements, the desugared versions of these sequences (i.e., $vars^{A,\pi}_{in,desug}$, $vars^{A,\pi}_{out,desug}$, and $vars^{A,\pi}_{int,desug}$) do have the same number and sorts of elements. We choose one of these desugared variable sequences to be the *canonical parameters* for the action π in A. We call the canonical sequence $vars^{A,\pi}$. We replace the other two sequences of parameters for π in the signature of A by $vars^{A,\pi}$, and we define substitutions $\sigma^{A,\pi}_{kind}$ to replace $vars^{A,\pi}_{kind,desug}$ with $vars^{A,\pi}$ in $P^{A,\pi}_{kind}$.[12]

Transition definitions

We canonicalize the parameters for each transition definition for an action named π so that the definition also uses $vars^{A,\pi}$ as its parameters. Specifically, we replace the references to variables that parameterize a desugared transition definition of π (i.e., $vars^{A,\pi}_{kind,t_j,desug}$) by references to the canonical variables (i.e., $vars^{A,\pi}$) throughout the transition definition. Therefore we define a substitution $\sigma^{A,\pi}_{kind,t_j}$ to perform this replacement and apply it to the whole transition definition. As described in Section 9, if the canonical variables clash with the desugared local variables (i.e., $localVars^{A,\pi}_{kind,t_j,desug}$), we must substitute fresh local variables for those that clash. The variables introduced by the substitution must be be distinct and fresh with respect to $vars^A$, $vars^{A,\pi}$, and the

[12]See Section 9 for a precise definition of a substitution, which maps a set of variables to a set of terms. Often we represent the domain and range of a substitution as sequences, with the ith variable in the domain being replaced by the ith variable or term in the range.

```
automaton Channel(Node, Msg:type, i, j:Node)
  signature
    input send(n1, n2:Node, m:Msg) where n1 = i ∧ n2 = j
    output receive(n1, n2:Node, m:Msg) where n1 = i ∧ n2 = j
  states contents:Set[Msg] := {}
  transitions
    input send(n1, n2, m1; local m:Msg) where n1 = i ∧ n2 = j ∧ m1 = m
      eff contents := insert(m, contents)
    output receive(n1, n2, m1; local m:Msg)
             where n1 = i ∧ n2 = j ∧ m1 = m
      pre m ∈ contents
      eff contents := delete(m, contents)

automaton P(n:Int)
  signature
    input receive(i1, i2, x:Int) where i1 = n-1 ∧ i2 = n
    output send(i1, i2, x:Int) where i1 = n ∧ i2 = n+1,
           overflow(i1:Int, s:Set[Int]) where i1 = n
  states
    val:Int := 0,
    toSend:Set[Int] := {}
  transitions
    input receive(i1, i2, i3; local x:Int)
      where i1 = n-1 ∧ i2 = n ∧ i3 = x
      eff ...    % effect clause unchanged from original definition of P
    output send(i1, i2, i3; local x:Int)
      where i1 = n ∧ i2 = n+1 ∧ i3 = x
      pre x ∈ toSend
      eff toSend := delete(x, toSend)
    output overflow(i1, s1; local t, s:Set[Int]) where i1 = n ∧ s1 = s
      pre s = toSend ∧ n < size(s) ∧ t ⊆ s
      eff toSend := t

automaton Watch(T:type, what:Set[T])
  signature
    input overflow(x:T, s:Set[T]) where x ∈ what
    output found(x:T) where x ∈ what
  states seen:Array[T,Bool] := constant(false)
  transitions
    input overflow(t1, s1; local x:T, s:Set[T])
             where t1 = x ∧ s1 = s ∪ {x}
      eff seen[x] := true
    input overflow(t1, s1; local x:T, s:Set[T])
             where ¬(x ∈ s) ∧ t1 = x ∧ s1 = s
      eff seen[x] := false
    output found(t1; local x:T) where t1 = x
      pre seen[x]
```

Figure 4.2: Preliminary desugarings of the sample automata `Channel`, `P`, and `Watch`

$$
\begin{array}{l}
\textbf{automaton } A(\mathit{types}^A, \mathit{vars}^A) \\
\quad \textbf{signature} \\
\qquad \ldots \\
\qquad \textbf{input } \pi(\mathit{vars}^{A,\pi}) \textbf{ where } \sigma^{A,\pi}_{in}(P^{A,\pi}_{in,desug}) \\
\qquad \textbf{output } \pi(\mathit{vars}^{A,\pi}) \textbf{ where } \sigma^{A,\pi}_{out}(P^{A,\pi}_{out,desug}) \\
\qquad \textbf{internal } \pi(\mathit{vars}^{A,\pi}) \textbf{ where } \sigma^{A,\pi}_{int}(P^{A,\pi}_{int,desug}) \\
\qquad \ldots \\
\quad \textbf{states } \mathit{stateVars}^A := \mathit{initVals}^A \textbf{ initially } P^A_{init} \\
\quad \textbf{transitions} \\
\qquad \sigma^{A,\pi}_{in,t_j} \left[\begin{array}{l} \textbf{input } \pi(\mathit{vars}^{A,\pi}_{in,t_j,desug};\ \textbf{local } \mathit{localVars}^{A,\pi}_{in,t_j,desug}) \textbf{ case } t_j \textbf{ where } P^{A,\pi}_{in,t_j,desug} \\ \quad \textbf{eff } \mathit{Prog}^{A,\pi}_{in,t_j} \textbf{ ensuring } \mathit{ensuring}^{A,\pi}_{in,t_j} \end{array} \right] \\
\qquad \sigma^{A,\pi}_{out,t_j} \left[\begin{array}{l} \textbf{output } \pi(\mathit{vars}^{A,\pi}_{out,t_j,desug};\ \textbf{local } \mathit{localVars}^{A,\pi}_{out,t_j,desug}) \textbf{ case } t_j \textbf{ where } P^{A,\pi}_{out,t_j,desug} \\ \quad \textbf{pre } \mathit{Pre}^{A,\pi}_{out,t_j} \\ \quad \textbf{eff } \mathit{Prog}^{A,\pi}_{out,t_j} \textbf{ ensuring } \mathit{ensuring}^{A,\pi}_{out,t_j} \end{array} \right] \\
\qquad \sigma^{A,\pi}_{int,t_j} \left[\begin{array}{l} \textbf{internal } \pi(\mathit{vars}^{A,\pi}_{int,t_j,desug};\ \textbf{local } \mathit{localVars}^{A,\pi}_{int,t_j,desug}) \textbf{ case } t_j \textbf{ where } P^{A,\pi}_{int,t_j,desug} \\ \quad \textbf{pre } \mathit{Pre}^{A,\pi}_{int,t_j} \\ \quad \textbf{eff } \mathit{Prog}^{A,\pi}_{int,t_j} \textbf{ ensuring } \mathit{ensuring}^{A,\pi}_{int,t_j} \end{array} \right] \\
\qquad \ldots
\end{array}
$$

Figure 4.3: Intermediate form of a desugared primitive automaton with canonical action parameters (cf. Figure 4.1)

desugared local variables. The substitutions for canonicalization are listed in Table 4.2. Variables listed in the center column are mapped by the substitution named in the left column to those listed in the right column.

Simplifying local variables

Finally, we simplify each desugared and canonicalized transition definition for actions named π by eliminating extraneous local variables. A local variable may be eliminated if it is never an lvalue in an assignment in the transition definition for π and if the **where** clause equates it with a canonical variable for π, that is, if it is used only as a constant in the transition definition and is already named by a canonical parameter.

This simplification is accomplished in four steps.

1. Define a substitution $\sigma^{A,\pi}_{kind,t_j,simp}$ that maps the redundant local variables to the corresponding canonical variables.

2. Apply $\sigma^{A,\pi}_{kind,t_j,simp}$ to each clause in the transition definition: the **where**, **pre**, **eff**, and **ensuring** clauses.

SUBSTITUTION	DOMAIN	RANGE
$\sigma^{A,\pi}_{kind}$	$vars^{A,\pi}_{kind,desug}$	$vars^{A,\pi}$
$\sigma^{A,\pi}_{kind,t_j}$	$vars^{A,\pi}_{kind,t_j,desug}$	$vars^{A,\pi}$
$\sigma^{A,\pi}_{kind,t_j,simp}$	Redundant variables in $\sigma^{A,\pi}_{kind,t_j}(localVars^{A,\pi}_{kind,t_j,desug})$	$vars^{A,\pi}$
σ^A	$x \in stateVars^A$	$A{:}States[A, types^A].x$
	$x' \in postVars^A$	$A'{:}States[A, types^A].x$
	$x \in localVars^{A,\pi}_{kind,t_j}$	$A{:}Locals[A, types^A, \pi].x$
	$x' \in localPostVars^{A,\pi}_{kind,t_j}$	$A'{:}Locals[A, types^A, \pi].x$

Table 4.2: Substitutions used in desugaring a primitive automaton. Substitutions listed on the left map variables in the domain to their right to variables in the range their far right.

3. Delete identity conjuncts from the **where** clause.

4. Delete the declarations of local variables that no longer appear in the transition.

Example 4.2 The second step in desugaring the primitive automata defined in Figures 2.1–2.3 is shown in Figure 4.4. The definitions in this figure are obtained from those in Figure 4.2 by selecting canonical parameters for each action.

Since each action occurs only once in the signature of the automaton `Channel`, selecting the canonical variables is trivial:

- $vars^{\texttt{Channel},\texttt{send}}$ defaults to $vars^{\texttt{Channel},\texttt{send}}_{in,desug} = \langle$`n1:Node, n2:Node, m:Msg`$\rangle$, and

- $vars^{\texttt{Channel},\texttt{receive}}$ defaults to $vars^{\texttt{Channel},\texttt{receive}}_{out,desug} = \langle$`n1:Node, n2:Node, m:Msg`$\rangle$.

These selections also make canonicalizing the signature trivial, because identity substitutions suffice.

We canonicalize the transition definitions by defining two substitutions.

- $\sigma^{\texttt{Channel},\texttt{send}}_{in,t_1}$ maps $vars^{\texttt{Channel},\texttt{send}}_{in,t_1,desug} = \langle$`n1:Node, n2:Node, m1:Msg`$\rangle$, to $vars^{\texttt{Channel},\texttt{send}}$ by replacing the parameter `m1:Msg` with the canonical parameter `m:Msg`. To avoid a conflict between the local variable `m:Msg` and the canonical parameter `m:Msg`, the substitution also replaces `m:Msg` by the fresh variable `m2:Msg`.

- In the same way, $\sigma^{\texttt{Channel},\texttt{receive}}_{out,t_1}$ maps $vars^{\texttt{Channel},\texttt{receive}}_{out,t_1,desug} = \langle$`n1:Node, n2:Node, m1:Msg`$\rangle$ to $vars^{\texttt{Channel},\texttt{receive}}$ by replacing the parameter `m1:Msg` with the canonical parameter `m:Msg` and the local variable `m:Msg` with the fresh variable `m2:Msg`.

Applying these substitution to the transition definitions produces

```
input send(n1, n2, m; local m2:Msg) where n1 = i ∧ n2 = j ∧ m = m2
  eff contents := insert(m2, contents)
output receive(n1, n2, m; local m2:Msg) where n1 = i ∧ n2 = j ∧ m = m2
  pre m2 ∈ contents
  eff contents := delete(m2, contents)
```

```
automaton Channel(Node, Msg:type, i, j:Node)
  signature
    input send(n1, n2:Node, m:Msg) where n1 = i ∧ n2 = j
    output receive(n1, n2:Node, m:Msg) where n1 = i ∧ n2 = j
  states contents:Set[Msg] := {}
  transitions
    input send(n1, n2, m) where n1 = i ∧ n2 = j
      eff contents := insert(m, contents)
    output receive(n1, n2, m) where n1 = i ∧ n2 = j
      pre m ∈ contents
      eff contents := delete(m, contents)

automaton P(n:Int)
  signature
    input receive(i1, i2, x:Int) where i1 = n-1 ∧ i2 = n
    output send(i1, i2, x:Int) where i1 = n ∧ i2 = n+1,
           overflow(i1:Int, s:Set[Int]) where i1 = n
  states
    val:Int := 0,
    toSend:Set[Int] := {}
  transitions
    input receive(i1, i2, x) where i1 = n-1 ∧ i2 = n
      eff if val = 0 then val := x
          elseif x < val then
            toSend := insert(val, toSend);
            val := x
          elseif val < x then
            toSend := insert(x, toSend)
          fi
    output send(i1, i2, x) where i1 = n ∧ i2 = n+1
      pre x ∈ toSend
      eff toSend := delete(x, toSend)
    output overflow(i1, s; local t:Set[Int]) where i1 = n
      pre s = toSend ∧ n < size(s) ∧ t ⊆ s
      eff toSend := t

automaton Watch(T:type, what:Set[T])
  signature
    input overflow(x:T, s:Set[T]) where x ∈ what
    output found(x:T) where x ∈ what
  states seen:Array[T,Bool] := constant(false)
  transitions
    input overflow(x, s; local s2:Set[T]) where s = s2 ∪ {x}
      eff seen[x] := true
    input overflow(x, s) where ¬(x ∈ s)
      eff seen[x] := false
    output found(x)
      pre seen[x]
```

Figure 4.4: Intermediate desugarings of the sample automata `Channel`, `P`, and `Watch`, obtained from the preliminary desugarings in Figure 4.2 by selecting canonical parameters for each action

However, the local variable `m2` is extraneous in both transition definitions, because it is equated with `m` in the **where** clause and no value is assigned to it. Hence `m2` equals `m` throughout the transition, and we can eliminate it entirely by applying a substitution (e.g., $\sigma^{\mathtt{channel,send}}_{in,t_1,simp}$, which maps `m2` to `m`) to the **where**, **eff** and **pre** (in the case of `receive`) clauses and simplifying the result, as shown in Figure 4.4.

As for `Channel`, each action occurs only once in the signature of the automaton `P`. Hence, it is trivial to select $vars^{\mathtt{P,receive}}$, $vars^{\mathtt{P,send}}$, and $vars^{\mathtt{P,overflow}}$ and to canonicalize the signature.

To map $vars^{\mathtt{P,receive}}_{in,t_1,desug}$ (i.e., ⟨`i1:Int`, `i2:Int`, `i3:Int`⟩) to $vars^{\mathtt{P,receive}}$, we define $\sigma^{\mathtt{P,receive}}_{in,t_1}$ to replace `i3:Int` by `x:Int`. To avoid conflicts between the local variable `x:Int` and the canonical parameter `x:Int`, the substitution also replaces `x:Int` by `i4:Int`. Applying this substitution to the transition definition produces:

```
input receive(i1, i2, x; local i4:Int) where i1 = n-1 ∧ i2 = n ∧ x = i4
  eff if val = 0 then val := i4
      elseif i4 < val then
        toSend := insert(val, toSend);
        val := i4
      elseif val < i4 then
        toSend := insert(i4, toSend)
      fi
```

Since the local variable `i4` equals `x` throughout the transition definition, we can eliminate it entirely by defining a substitution mapping `i4` to `x`, applying that substitution to the **where** and **eff** clauses, and simplifying the result, as shown in Figure 4.4.

Canonicalization of the `send` transition follows the same pattern as the `receive` transition. Application of the canonicalizing substitution $\sigma^{\mathtt{P,send}}_{out,t_1}$ yields:

```
output send(i1, i2, x; local i4:Int) where i1 = n ∧ i2 = n+1 ∧ x = i4
pre i4 ∈ toSend
eff toSend := delete(i4, toSend)
```

This definition simplifies to the one shown in Figure 4.2, which does not contain a local variable.

Similarly applying the canonicalizing substitution $\sigma^{\mathtt{P,overflow}}_{out,t_1}$ to the `overflow` transition yields:

```
output overflow(i1, s; local t, s2:Set[Int]) where i1 = n ∧ s = s2
  pre s2 = toSend ∧ n < size(s2) ∧ t ⊆ s2
  eff toSend := t
```

Once again, this definition simplifies to the one shown in Figure 4.2. Notice that the local variable `t` cannot be eliminated because it is not equated with a canonical parameter. Further notice that, in this case, canonicalization has eliminated all the local variables introduced in the desugaring step.

As for `Channel` and `P`, each action occurs only once in the signature of the automaton `Watch`. Hence it is trivial to select $vars^{\mathtt{Watch,overflow}}$ and $vars^{\mathtt{watch,found}}$.

Canonicalizing the two transition definitions for `overflow` proceeds by defining $\sigma^{\mathtt{watch,overflow}}_{in,t_1}$ and $\sigma^{\mathtt{watch,overflow}}_{in,t_2}$, which happen to be the same. They map `t1:T` to `x:T`, `s1:Set[T]` to `s:Set[T]`, `s:Set[T]` to `s2:Set[T]`, and `x:T` to `t2:T`. Applying these substitutions to the transition definitions yields:

```
input overflow(x, s; local t2:T, s2:Set[T])
        where x = t2 ∧ s = s2 ∪ {t2}
  eff seen[t2] := true
input overflow(x, s; local t2:T, s2:Set[T])
        where ¬(t2 ∈ s2) ∧ x = t2 ∧ s = s2
  eff seen[x] := false
```

The local variable `t2:T` can be eliminated from both transition definitions. The local variable `s2:Set[T]` can be eliminated from the second transition definition but *not* from the first. These simplifications result in the transition definitions shown in Figure 4.4.

Notice that after the simplification of the local variable, the semantic meaning of the parameter `s:Set[T]` in the desugared and canonicalized automaton shown in Figure 4.4 is different than the meaning of the parameter `s:Set[T]` in the original automaton shown in Figure 2.3. The parameter `s:Set[T]` in the original actually corresponds to the local variable `s2:Set[T]` in the canonicalized version.

Applying the canonicalizing substitution $\sigma^{\texttt{watch},\texttt{found}}_{in,t_1}$ to the `found` transition yields:

```
output found(x; local t2:T) where x = t2
  pre seen[t2]
```

After its local variables are simplified, the transition definition shown in Figure 4.4 is identical to the one originally defined in Figure 2.3.

4.3 Combining transition definitions

We will see in Sections 7.7–7.9 that combining multiple transition definitions for a given action into a single transition definition is useful for composing automata. It is necessary for combining input actions that execute atomically in the composition, and it avoids a code explosion multiplicative in the number of input and output actions. Because this transition combining step is easy to understand when applied to a single primitive automaton, we describe it here and assume all automata hereafter have only a single transition definition per kind per action, as shown in Figure 4.5. To combine the transition definitions for a given *kind* of an action π, we need to combine their sequences of parameters, their local variables, and their **where**, **pre**, **eff**, and **ensuring** clauses into one, semantically equivalent, transition definition.

Furthermore, as will be discussed further in Section 7, the kind of an action may be changed by composition. Input actions may be subsumed by output actions, and output actions may be hidden as internal actions. Thus, the expansion of a composite automaton may combine transition definitions across kinds. To facilitate such combinations, we collect together all the local variables for each action of an automaton A into a single sequence of variables $localVars^{A,\pi}$, which is the concatenation (with duplicates removed) of the all sequences $localVars^{A,\pi}_{kind,t_j}$. Again, this variable combining step is easy to understand when applied to a single primitive automaton, so we describe it here and assume all automata hereafter have only one sequence of local variables per action name.

In describing this combination, we assume that parameters of the automaton have already been desugared and canonicalized as described in Sections 4.1 and 4.2. In Figure 4.5 and the discussion below, we indicate the syntactic forms that result from that desugaring by use of the *desug* subscript. We rely on the key semantic condition (mentioned in Section 3.4 and discussed in Section 4.6) that exactly one transition definition be defined for each assignment of values to $vars^{A,\pi}$ that satisfies $P^{A,\pi}_{kind}$. That is, within an automaton, all like-named transition definitions must have **where** clauses that are satisfiable only for disjoint sets of parameter values.[13]

First, notice that since all the contributing transition definitions are already desugared and canonicalized, each is is parameterized by $vars^{A,\pi}$. Hence, combining the parameters is trivial.

At first glance, combining local variables looks trickier. Each transition definition has local scope with respect to local variables. So, there may be any amount of duplication of variables

[13]These semantic conditions also ensure that, in the absence of local variables, the resulting **where** clause can be eliminated because it will be equivalent to `true`.

automaton $A(types^A, vars^A)$

...

states $stateVars^A := initVals^A$ **initially** $\sigma^A(P^A_{init})$

transitions

input $\pi(vars^{A,\pi};$ **local** $localVars^{A,\pi})$ **where** $\bigvee_j P^{A,\pi}_{in,t_j,desug}$

eff

if $P^{A,\pi}_{in,t_j,desug}$ **then** $Prog^{A,\pi}_{in,t_j,desug}$

elseif ...

fi

ensuring $\bigwedge_j \left(P^{A,\pi}_{in,t_j,desug} \Rightarrow ensuring^{A,\pi}_{in,t_j,desug}\right)$

output $\pi(vars^{A,\pi};$ **local** $localVars^{A,\pi})$ **where** $\bigvee_j P^{A,\pi}_{out,t_j,desug}$

pre $\bigvee_j \left(P^{A,\pi}_{out,t_j,desug} \wedge Pre^{A,\pi}_{out,t_j,desug}\right)$

eff

if $P^{A,\pi}_{out,t_j,desug}$ **then** $Prog^{A,\pi}_{out,t_j,desug}$

elseif ...

fi

ensuring $\bigwedge_j \left(P^{A,\pi}_{out,t_j,desug} \Rightarrow ensuring^{A,\pi}_{out,t_j,desug}\right)$

internal $\pi(vars^{A,\pi};$ **local** $localVars^{A,\pi})$ **where** $\bigvee_j P^{A,\pi}_{int,t_j,desug}$

Analogous to output.

...

Figure 4.5: Intermediate form of a desugared primitive automaton, with canonical action parameters and with all transition definitions for each kind of an action combined into a single transition definition

```
automaton Watch(T:type, what:Set[T])
  signature
    input overflow(x:T, s:Set[T]) where x ∈ what
    output found(x:T) where x ∈ what
  states seen:Array[T,Bool] := constant(false)
  transitions
    input overflow(x, s; local s2:Set[T]) where s = s2 ∪ {x} ∨ ¬(x ∈ s)
      eff if s = s2 ∪ {x} then seen[x] := true
          elseif ¬(x ∈ s) then seen[x] := false
          fi
    output found(x)
      pre seen[x]
```

Figure 4.6: Improved intermediate desugaring of the sample automaton `Watch`, obtained from the intermediate desugaring in Figure 4.4 by combining the transition definitions for `overflow`

among the sequences $localVars^{A,\pi}_{kind,t_j,desug}$. One might think that a correctly combined transition definition might need distinct local variables to store the values of the duplicate local variable appropriate to each contributing transition definition. However, for each assignment of values to $vars^{A,\pi}$ only one contributing transition definition can be defined for any assignment of values to its local variables. Therefore, there is at most one "useful" initial value for each local variable. Similarly, at most one contributing **eff** clause can make assignments to its local variables. Hence, duplicate declarations of local variables have no effect on the combined transition definition. Accordingly, we define $localVars^{A,\pi}$ to be the sequence of variables obtained by removing any duplicates from the concatenation of all sequences $localVars^{A,\pi}_{kind,t_j,desug}$.

In combining the various clauses of the contributing transitions, we use the **where** clauses of the contributing transitions as guards to select the correct case to use. The four clauses of the combined transition are combined as follows:

- The combined **where** clause is the disjunction of the **where** clauses from all the contributing transition definitions.
- For output and internal transition definitions, the combined **pre** clause checks that one set of parameters fulfills both the **where** and **pre** clauses of some contributing transition definition.
- The combined **eff** clause is a single **if**...**then**...**elseif**...**fi** statement in which the contributing **eff** clause is guarded by the associated **where** clause.
- Similarly, the combined **ensuring** clause asserts the appropriate contributing **ensuring** clause when the associated **where** clause is true. Note that since $P^{A,\pi}_{kind,t_j,desug}$ is defined on the initial values of $localVars^{A,\pi}_{kind,t_j,desug}$, assignments made to local variables in the **eff** clause have no effect on which **ensuring** clause is asserted.

Example 4.3 Consider the desugared and canonicalized automaton `Watch` shown in Figure 4.4. The only action with multiple transition definitions is the `overflow` input action. Following the above recipe, they are combined into the one equivalent action shown in Figure 4.6.

4.4 Combining aggregate sorts and expanding variable references

Section 3.2 described aggregate sorts that are automatically defined for the state and local variables of an automaton A (i.e., $States[A, types^A]$ and $Locals[A, types^A, kind, \pi, t_j]$). Desugaring alters the

automaton $A(types^A, vars^A)$

$\ldots$

states $stateVars^A := initVals^A$ **initially** $\sigma^A(P^A_{init})$

transitions

$$\sigma^A \left[\sigma^{A,\pi}_{kind} \left[\begin{array}{l} \textbf{kind}\ \pi(vars^{A,\pi};\ \textbf{local}\ localVars^{A,\pi})\ \textbf{where}\ P^{A,\pi}_{kind,comb,t_1} \\ \quad \textbf{pre}\ Pre^{A,\pi}_{kind} \\ \quad \textbf{eff}\ Prog^{A,\pi}_{kind}\ \textbf{ensuring}\ ensuring^{A,\pi}_{kind} \end{array} \right] \right]$$

$\ldots$

Figure 4.7: Final form of a desugared primitive automaton, with canonical action parameters, with all transition definitions for each kind of an action combined into a single transition definition, and with all variable references expanded.

automaton A and, consequently, can alter these aggregate sorts. In particular, as discussed in Section 4.3, combining multiple transition definitions for a particular action π in automaton A involves combining the local variables that appear in each transition into a single sequence. We collect together all the local variables for each action π of an automaton A into a single sequence of variables $localVars^{A,\pi}$, which is the concatenation (with duplicates removed) of the all sequences $localVars^{A,\pi}_{kind,t_j}$.

As a result, the aggregate sort for local variables also changes. Notationally, the *kind* and **case** labels t_j are dropped from the aggregate local sort name $Locals[A, types^A, kind, \pi, t_j]$. We define a new sort $Locals[A, types^A, \pi]$ for the combined transition definition to be a tuple with selection operators that are named, typed, and have values in accordance with the local variables in $localVars^{A,\pi}$. That is, the set of identifiers for the selection operators on the sort $Locals[A, types^A, \pi]$ is the union of the sets of identifiers for the selection operators on the sorts $Locals[A, types^A, kind, \pi, t_j]$. We change the sorts of the aggregate local and post-local variables A and A' to this new sort. This has the effect of collapsing multiple aggregate local and post-local variables each defined in the scope of one transition into a single local and post-local variable defined in all transitions for a given action.[14]).

Formally, for each transition definition t_j for a given *kind* of an action π in A, we define a resorting[15] that maps the aggregate local sort $Locals[A, types^A, kind, \pi, t_j]$ to the new aggregate local sort $Locals[A, types^A, \pi]$, and we apply that resorting to the transition definition before performing the combining step. As a result, each variable $A{:}Locals[A, types^A, kind, \pi, t_j]$ is mapped to a variable $A{:}Locals[A, types^A, \pi]$. Thus, local variable references using the notation $A.v$ form remain well defined and the resorting does not change the text of the transition definition. After combining, the sorts $Locals[A, types^A, kind, \pi, t_j]$ may be ignored.

In addition to introducing notations for aggregate local sorts, Section 3.2 also introduced notations for aggregate state sorts. These notations provided an additional, and potentially less ambiguous, way of referencing the values of local and state variables. We now desugar simple references to local and state variables to use the notations for aggregate local and state variables.

[14]To avoid complications that arise when new fields are added to an aggregate local tuple during the combining of local variables across transitions, we should disallow use of the constructor `[__, ...]` for aggregate local sorts.

[15]See Section 9 for a formal definition of resortings, which map sorts to sorts.

Formally, we define a substitution[16] σ^A to map state and post-state variables to terms. If x is a state variable or a post-state variable (i.e., $x \in \mathit{stateVars}^A$ or $x \in \mathit{postVars}^A$), then $\sigma^A(x)) = A.x$, where A has sort $\mathit{States}[A, \mathit{types}^A]$ and the operator $\_\_.x$ has signature $\mathit{States}[A, \mathit{types}^A] \to T$, where T is the sort of x.

Similarly, for each transition definition π of type *kind*, we define a substitution $\sigma^{A,\pi}_{\mathit{kind}}$ to map local and post-local variables to terms. If x is a local or post-local variable (i.e., $x \in \mathit{localVars}^{A,\pi}$ or $x \in \mathit{localPostVars}^{A_i,\pi}_{\mathit{kind}}$), then $\sigma^{A,\pi}_{\mathit{kind}}(x) = A.x$, where A has sort $\mathit{Locals}[A, \mathit{types}^A, \mathit{kind}, \pi]$, and the operator $\_\_.x$ has signature $\mathit{Locals}[A, \mathit{types}^A, \mathit{kind}, \pi] \to T$, where T is the sort of x.

Figure 4.7 shows the final form of a desugared primitive automaton with canonical action parameters and local variables and with all transition definitions for each kind of an action combined into a single transition definition, and with all variable references expanded. In that figure, we indicate the syntactic forms that result from the combining step by use of the *comb* subscript. Figure 4.8 shows the result of applying these substitutions to the sample primitive automata.

4.5 Restrictions on the form of desugared automaton definitions

After the definition of a primitive automaton A has been desugared as described in Sections 4.1–4.4, it has the following properties.

- No **const** parameters appear in the signature of A.
- Each appearance of an action π in the signature of A is parameterized by the canonical action parameters $\mathit{vars}^{A,\pi}$ of π in A.
- Each transition definition of an action π is parameterized by the canonical action parameters $\mathit{vars}^{A,\pi}$ of π in A; i.e., every parameter is a simple reference to a variable in $\mathit{vars}^{A,\pi}$.
- Each action name has at most one transition definition of each kind.
- Each reference to a state variable x of A, other than in the list of state variables in the **states** statement, has been replaced by the term $A.x$.
- Each reference to a post-state variable x' of A has been replaced by the term $A'.x$.
- Each reference to a local variable x in a transition of A, other than in the **local** clause of that transition definition, has been replaced by the term $A.x$.
- Each reference to a post-local variable x' in a transition of A has been replaced by the term $A'.x$.

4.6 Semantic proof obligations, revisited

We are now ready to formalize the semantic proof obligations for primitive automata introduced in Section 3.4. Previously, we said that for each action named π and each sequence of parameters values:

1. At most one of $P^{A,\pi}_{\mathit{in}}$, $P^{A,\pi}_{\mathit{out}}$, and $P^{A,\pi}_{\mathit{int}}$ is true.
2. If $P^{A,\pi}_{\mathit{kind}}$ is true, at least one $P^{A,\pi}_{\mathit{kind},t_j}$ is true.

[16] See Section 9 for a formal definition of substitutions, which map variables to terms.

```
automaton Channel(Node, Msg:type, i, j:Node)
  signature
    input send(n1, n2:Node, m:Msg) where n1 = i ∧ n2 = j
    output receive(n1, n2:Node, m:Msg) where n1 = i ∧ n2 = j
  states contents:Set[Msg] := {}
  transitions
    input send(n1, n2, m) where n1 = i ∧ n2 = j
      eff Channel.contents := insert(m, Channel.contents)
    output receive(n1, n2, m) where n1 = i ∧ n2 = j
      pre m ∈ Channel.contents
      eff Channel.contents := delete(m, Channel.contents)

automaton P(n:Int)
  signature
    input receive(i1, i2, x:Int) where i1 = n-1 ∧ i2 = n
    output send(i1, i2, x:Int) where i1 = n ∧ i2 = n+1,
           overflow(i1:Int, s:Set[Int]) where i1 = n
  states
    val:Int := 0,
    toSend:Set[Int] := {}
  transitions
    input receive(i1, i2, x) where i1 = n-1 ∧ i2 = n
      eff if P.val = 0 then P.val := x
          elseif x < P.val then
            P.toSend := insert(P.val, P.toSend);
            P.val := x
          elseif P.val < x then
            P.toSend := insert(x, P.toSend)
          fi
    output send(i1, i2, x) where i1 = n ∧ i2 = n+1
      pre x ∈ P.toSend
      eff P.toSend := delete(x, P.toSend)
    output overflow(i1, s; local t:Set[Int]) where i1 = n
      pre s = P.toSend ∧ n < size(s) ∧ P.t ⊆ s
      eff P.toSend := P.t

automaton Watch(T:type, what:Set[T])
  signature
    input overflow(x:T, s:Set[T]) where x ∈ what
    output found(x:T) where x ∈ what
  states seen:Array[T,Bool] := constant(false)
  transitions
    input overflow(x, s; local s2:Set[T])
            where s = Watch.s2 ∪ {x} ∨ ¬(x ∈ s)
      eff
        if s = Watch.s2 ∪ {x} then Watch.seen[x] := true
        elseif ¬(x ∈ s) then Watch.seen[x] := false
        fi
    output found(x)
      pre Watch.seen[x]
```

Figure 4.8: Sample desugared automata `Channel`, `P`, and `Watch`, obtained from the intermediate desugarings in Figures 4.4 and 4.6 by desugaring references to state and local variables

3. If $P^{A,\pi}_{kind}$ is true, at most one $P^{A,\pi}_{kind,t_j}$ is true

We explicitly did not define the phrase "sequence of parameters values" because these predicates may be stated in terms of different variables. In other words, $vars^{A,\pi}_{in}$ may be different from $vars^{A,\pi}_{out}$ and $vars^{A,\pi}_{in,t_1}$. Similarly, $vars^{A,\pi}_{in,t_1}$ may be different from $vars^{A,\pi}_{in,t_2}$. However, after desugaring and canonicalizing (but *before* combining), we have predicates that are semantically equivalent to those in the original automaton, but defined over a common set of free variables. That is, all the free variables of all the predicates $\sigma^{A,\pi}_{kind}(P^{A,\pi}_{kind,desug})$ and $\sigma^{A,\pi}_{kind,t_j}(P^{A,\pi}_{kind,t_j,desug})$ are among $vars^{A}$ and $vars^{A,\pi}$.

The alert reader will realize that Tables 3.1 and 4.1 list $localVars^{A,\pi}_{kind,t_j}$ among the variables that may occur freely in $P^{A,\pi}_{kind,t_j}$ and $P^{A,\pi}_{kind,t_j,desug}$ and might therefore conclude that the aforementioned predicates are not "defined over a common set of free variables". However, as noted Section 3.2, a transition π is defined only for values of its parameters that, together with *some* choice of initial values for its local variables, satisfy the **where** clause of the transition definition. Thus, for the purposes of formalizing the semantic proof obligations for transition definitions, local variables should be existentially bound, not free in **where** clauses, that is, $P^{A,\pi}_{kind,t_j,desug}$ should be preceded by $\exists localVars^{A,\pi}_{kind,t_j}$.

The semantic proof obligations we introduced in Section 3.4 can be stated precisely as follows. We require that for each action name π, all values of $vars^{A}$, and all values of $vars^{A,\pi}$, the following statements must be provable from the axioms provided by IOA's built-in types, by the theories associated with the type definitions and the **axioms** in the IOA specification that contains the automaton definition, and by the theories associated with the **assumes** clause of that definition.

$$\checkmark \quad \neg\left(\sigma^{A,\pi}_{in}(P^{A,\pi}_{in,desug}) \wedge \sigma^{A,\pi}_{out}(P^{A,\pi}_{out,desug})\right), \tag{4.1}$$

$$\checkmark \quad \neg\left(\sigma^{A,\pi}_{in}(P^{A,\pi}_{in,desug}) \wedge \sigma^{A,\pi}_{int}(P^{A,\pi}_{int,desug})\right), \tag{4.2}$$

$$\checkmark \quad \neg\left(\sigma^{A,\pi}_{out}(P^{A,\pi}_{out,desug}) \wedge \sigma^{A,\pi}_{int}(P^{A,\pi}_{int,desug})\right), \tag{4.3}$$

$$\checkmark \quad \sigma^{A,\pi}_{kind}(P^{A,\pi}_{kind,desug}) \Rightarrow \bigvee_{j} \exists localVars^{A,\pi}_{kind,t_j} \sigma^{A,\pi}_{kind,t_j}(P^{A,\pi}_{kind,t_j,desug}), \text{ and} \tag{4.4}$$

$$\checkmark \quad \sigma^{A,\pi}_{kind}(P^{A,\pi}_{kind,desug}) \Rightarrow \tag{4.5}$$
$$\neg\left(\exists localVars^{A,\pi}_{kind,t_j} \sigma^{A,\pi}_{kind,t_j}(P^{A,\pi}_{kind,t_j,desug}) \wedge \exists localVars^{A,\pi}_{kind,t_k} \sigma^{A,\pi}_{kind,t_k}(P^{A,\pi}_{kind,t_k,desug})\right),$$

when $j \neq k$.

5 Definitions for composite automata

This section introduces notations and semantic checks for composite IOA automata. Section 5.1 describes the syntactic structures that may appear in an IOA description of a composite I/O automaton. Section 5.2 describes notations for the state variables of a composite automaton. When component automata have type parameters, the sorts of these state variables are obtained by mapping the formal type parameters of the component automata to the actual parameters used to instantiate those components in the composition. Finally, Sections 5.3 and 5.4 describe the conditions that descriptions of composite automata must satisfy to be semantically valid.

5.1 Syntax

As for primitive automata, we introduce a labeling of the syntactic elements of composite IOA programs in order to facilitate describing their syntactic manipulation. Figure 5.1 indicates a particular labeling of the expressions that can appear in the IOA definition of a composite I/O automaton. Again, we have selected the granularity of this labeling to expose just those elements of composite IOA programs that are needed in Section 7 to describe the expansion of composite automata into primitive form.

automaton $D(\mathit{types}^D, \mathit{vars}^D)$

  assumes $\mathit{Assumptions}$

  components

    $C_1[\mathit{vars}^{D,C_1}] : A_1(\mathit{actualTypes}^{D,C_1}, \mathit{actuals}^{D,C_1})$ **where** P^{D,C_1};

    $\ldots;$

    $C_n[\mathit{vars}^{D,C_n}] : A_n(\mathit{actualTypes}^{D,C_n}, \mathit{actuals}^{D,C_n})$ **where** P^{D,C_n}

  hidden

    $\pi_1(\mathit{params}^{D,\pi_1}_{\mathit{hide}_1})$ **where** $H^{D,\pi_1}_{\mathit{hide}_1}$;

    $\ldots;$

    $\pi_m(\mathit{params}^{D,\pi_m}_{\mathit{hide}_m})$ **where** $H^{D,\pi_m}_{\mathit{hide}_m}$

invariant of D : $\mathit{Inv}^D_1; \ldots \mathit{Inv}^D_z$

Figure 5.1: General form of a composite automaton

In Figure 5.1, parameterized components named $C_1, \ldots, C_n$ are based on instantiations of automata named $A_1, \ldots, A_n$. The formal parameters of component C_i are vars^{D,C_i}, and the actual parameters of automaton A_i consist of a sequence $\mathit{actualTypes}^{D,C_i}$ of sorts and a sequence $\mathit{actuals}^{D,C_i}$ of terms. IOA permits the specification of C_i to be abbreviated by deleting the colon and the following expression when C_i and A_i are named by the same identifier, $\mathit{actualTypes}^{D,C_i}$ is empty, and $\mathit{actuals}^{D,C_i} = \mathit{vars}^{D,C_i}$ (e.g., see component `P` in Example 2.4). In the specification of **hidden** actions, $\mathit{params}^{D,\pi_p}_{\mathit{hide}_p}$ is a sequence of terms, analogous to $\mathit{params}^{A,\pi_p}_{\mathit{out},t_1}$, and we define $\mathit{vars}^{D,\pi_p}_{\mathit{hide}_p}$ to be the set of variables that occur freely in $\mathit{params}^{D,\pi_p}_{\mathit{hide}_p}$ but are not in vars^D. Each **invariant** of D is stated as a predicate Inv^D_x.

SYNTACTIC STRUCTURE	FREE VARIABLES
$actuals^{D,C_i}$	$vars^D$, $vars^{D,C_i}$
P^{D,C_i}	$vars^D$, $vars^{D,C_i}$
$H_{hide_p}^{D,\pi_p}$	$vars^D$, $vars_{hide_p}^{D,\pi_p}$
$params_{hide_p}^{D,\pi_p}$	$vars^D$, $vars_{hide_p}^{D,\pi_p}$
Inv_x^D	$vars^D$, $stateVars^D$

Table 5.1: Variables that can occur freely in terms in the definition of a composite automaton. Variables listed on the right may occur freely in the syntactic structure listed to their left.

Example 2.4 conforms to this general form, as follows.

- The first component of `Sys` is named `C`. Its parameters, $vars^{\mathtt{Sys},\mathtt{C}}$, are $\langle$`n:Int`$\rangle$, and it is based on the automaton `Channel`, for which it supplies the actual parameters $actualTypes^{\mathtt{Sys},\mathtt{C}} =$ $\langle$`Int, Int`$\rangle$ and $actuals^{\mathtt{Sys},\mathtt{C}} = \langle$`n, n+1`$\rangle$.

- The second component of `Sys` is named `P`. It has the same parameters as `C`. By the conventions for abbreviating component descriptions, it is based on the automaton of the same name, for which it supplies the actual parameters $actuals^{\mathtt{Sys},\mathtt{P}} = \langle$`n`$\rangle$; in this case, $actualTypes^{\mathtt{Sys},\mathtt{P}}$ is empty (as required to use this abbreviated form).

- The third component of `Sys`, named `W`, has no parameters. It is based on the automaton `Watch`, for which it supplies the actual parameters $actualTypes^{\mathtt{Sys},\mathtt{W}} = \langle$`Int`$\rangle$ and $actuals^{\mathtt{Sys},\mathtt{W}} =$ $\langle$`between(1,nProcesses)`$\rangle$.

- The `send` actions that `Sys` inherits from `P[nProcesses]` are hidden as internal actions in `Sys`. The parameters $params_{hide_1}^{\mathtt{Sys,send}} = \langle\mathtt{nProcesses}, \mathtt{nProcesses+1}, \mathtt{m}\rangle$ in the single clause in the **hidden** statement involve a single free variable in $vars_{hide_1}^{\mathtt{Sys,send}} = \langle\mathtt{m:Int}\rangle$, and $H_{hide_1}^{\mathtt{Sys,send}}$ is *true*.

- The predicate
  $$\forall\ \mathtt{m:Int}\ \forall\ \mathtt{n:Int}\ (1 \leq \mathtt{m} \wedge \mathtt{m} < \mathtt{n} \wedge \mathtt{n} \leq \mathtt{nProcesses} \Rightarrow \mathtt{P[m].val} < \mathtt{P[n].val} \vee \mathtt{P[n].val} = 0)$$
  is invariant $Inv_1^{\mathtt{Sys}}$ of `Sys`.

5.2 State variables of composite automata

The definition of a composite automaton in IOA does not mention the automaton's state variables explicitly. Rather, its **components** statement implicitly introduces a single state variable for each component. We first describe the notations IOA provides for state variables associated with component automata that have no type parameters. Then we describe how these notations extend to state variables associated with component automata that have type parameters. Our goal is to provide a precise explanation of notations for state variables such as `P[m].val`, which appears in the invariant for the sample composite automaton `Sys`.

As for primitive automata (see Section 3.2), we automatically define a sort $States[D, types^D]$ representing the aggregate states of a composite automaton D, and we also define aggregate state and post-state variables D and D' of sort $States[D, types^D]$. Furthermore, we treat the sort $States[D, types^D]$ in the same fashion as for primitive automata, namely, as a tuple of state variables: we define the aggregate state of a composite automaton D to be a tuple containing a state variable for each component automaton, and we use the names of the components (i.e., $C_1, \ldots, C_n$) as the names of these state variables and of the corresponding selectors (i.e., $\_\_.C_1, \ldots, \_\_.C_n$) of $States[D, types^D]$.

State variables for components with no type parameters

Defining the sort of the state variable C_i is simplest when the component C_i does not have parameters and when the automaton A_i on which C_i is based does not have type parameters. For each such component C_i, the state variable C_i of D has sort $States[A_i]$, and the selector $\_\_.C_i$ has signature $States[D, types^D] \rightarrow States[A_i]$.

When the component C_i has parameters, but A_i still does not have type parameters, the situation is slightly more complicated, because the composite automaton D may contain multiple instances of A_i. For example, the composite automaton `Sys` contains `nProcesses` instances of the component automaton `P`, each with its own state variables `val` and `toSend`. These instances are parameterized by a single integer `n` and are distinguished by the component names `P[1]`, …, `P[nProcesses]`.

For each parameterized component C_i, the corresponding state variable C_i does not refer to the aggregate state of a single instance of A_i. Rather, it refers to a *map* from the values of the parameters $vars^{D,C_i}$ of C_i to the aggregate states of A_i. That is, the state variable C_i has sort $\texttt{Map[}types^{D,C_i}, States[A_i]\texttt{]}$, where $types^{D,C_i}$ is the sequence of sorts of the variables in $vars^{D,C_i}$. The selection operator $\_\_.C_i$ has signature $States[D, types^D] \rightarrow \texttt{Map[}types^{D,C_i}, States[A_i]\texttt{]}$.

For example, the state variable `P` of `Sys` has sort `Map[Int,States[P]]`. Hence, `P[n]` is a legitimate term with sort `States[P]`, and the term `P[n].val` has sort `Int`. Likewise, the selection operator `__.P` has signature `States[Sys]` $\rightarrow$ `Map[Int,States[P]]`, and `Sys.P[n].val` is an alternative notation for the state variable `val` that `Sys` inherits from component `P[n]`.

Resortings for automata with type parameters

Defining the sort of the state variable C_i is more complicated when A_i has type parameters. Since the semantics for IOA are defined using multisorted, first-order logic, we cannot quantify over sorts or use sorts as component indices. Instead, different instances of A_i, corresponding to different actual types, must be described in separate clauses in the **components** statement, where they are further distinguished by different component names. As a result, there can be only finitely many differently typed instantiations of A_i, even though altogether there may be infinitely many instances of A_i that are distinguished by the values of their non-type parameters. For example, a composite automaton might contain channel components that transmit finitely many different types of messages, but there may be infinitely many instances of such a component that transmits a given type of message.

When a component C_i is based on an automaton A_i parameterized by the sorts $types^{A_i}$, we define a resorting ρ_i (which we write as ρ^{C_i} in contexts, such as $\rho^{\texttt{W}}$, where it is more convenient to use the name of the component rather than its position in the list of all components) that maps

$types^{A_i}$ to $actualTypes^{D,C_i}$. For example, $\rho^{\mathtt{W}}$ maps $types^{\mathtt{Watch}} = \langle\mathtt{T}\rangle$ to $actualTypes^{\mathtt{Sys,W}} = \langle\mathtt{Int}\rangle$, and $\rho^{\mathtt{C}}$ maps $types^{\mathtt{Channel}} = \langle\mathtt{Node,\ Msg}\rangle$ to $actualTypes^{\mathtt{Sys,C}} = \langle\mathtt{Int,\ Int}\rangle$.

As described in Section 9, there is a natural way to extend the resorting ρ_i to map arbitrary sorts involving the formal type parameters in the defining automaton A_i to sorts involving the corresponding actual types that the component C_i supplies for A_i. For example, this extension maps the automatically defined sort $States[A_i, types^{A_i}]$ for the state of A_i to the sort $States[A_i, actualTypes^{D,C_i}]$ for the state of the instances of A_i corresponding to the component C_i.[17]

The resorting ρ_i also extends naturally to map operators with signatures involving the formal type parameters in the defining automaton A_i to operators with signatures involving the corresponding actual types that the component C_i supplies for A_i. Thus, for example, ρ^C maps

```
States[Channel,Node,Msg] = tuple of contents: Set[Msg]
```

to

```
States[Channel,Int,Int] = tuple of contents: Set[Int]
```

and it maps the signature of the selection operator `__.contents` from `States[Channel,Node,Msg]` $\rightarrow$ `Set[Msg]` to `States[Channel,Int,Int]` $\rightarrow$ `Set[Int]`.

State variables for components with type parameters

When A_i has type parameters, we employ a resorting of its aggregate state sort to define the sort of the state variable C_i of D. In the simple case when the component C_i does not have any parameters, the state variable C_i has sort $States[A_i, actualTypes^{D,C_i}]$, and the selection operator $\_\_.C_i$ has signature $States[D, types^D] \rightarrow States[A_i, actualTypes^{D,C_i}]$.

For example, the state variable `W` of `Sys` has sort `States[Watch,Int]`, the term `W.seen` has sort `Array[Int,Bool]`, the selection operator `__.W` has signature `States[Sys]` $\rightarrow$ `States[Watch,Int]`, and `Sys.Watch.seen` is an alternative notation for the state variable `seen` that `Sys` inherits from component `W`.

In the case when the component C_i has parameters (and the automaton A_i has type parameters), the state variable C_i has sort `Map[`$types^{D,C_i}, States[A_i, actualTypes^{D,C_i}]$`]`, where $types^{D,C_i}$ is the sequence of sorts of the variables in $vars^{D,C_i}$, and the selection operator $\_\_.C_i$ has signature $States[D, types^D] \rightarrow$ `Map[`$types^{D,C_i}, States[A_i, actualTypes^{D,C_i}]$`]`.

For example, the state variable `C` of `Sys` has sort `Map[Int,States[Channel,Int,Int]]`, the term `C[n]` has sort `States[Channel[Int,Int]`, the term `C[n].contents` has sort `Set[Int]`, the selection operator `__.C` has signature `States[Sys]` $\rightarrow$ `Map[Int,States[Channel,Int,Int]`, and `C[n].contents` is an alternative notation for the state variable `contents` that `Sys` inherits from component `C[n]`.

5.3 Static semantic checks

The following must be true for an IOA program to represent a valid composite I/O automaton and can be checked statically. These checks are currently performed by `ioaCheck`, the IOA parser and static-semantic checker.

- ✓ No sort appears more than once in $types^D$.
- ✓ Each component name (i.e., C_i) occurs at most once.
- ✓ The sequences $vars^D$ and $vars^{D,C_i}$ of variables contain no duplicates; furthermore, no variable appears in both $vars^D$ and $vars^{D,C_i}$ for any value of i.

[17] Although A_i, $types^A$, C_i, and $actualTypes^{D,C_i}$ appear as subsorts of a sort constructor `States[__,...]`, IOA assigns no semantics to these sorts. Syntactically, however, they are treated in the same fashion as other sorts; in particular, the resorting ρ_i replaces $types^{A_i}$ by $actualTypes^{D,C_i}$.

- ✓ Each component automaton is supplied with the appropriate number of actual types, that is, $actualTypes^{D,C_i}$ has the same length as $types^{A_i}$.
- ✓ For every operator f in a theory specified in the **assumes** clause of the automaton A_i, a corresponding operator $\rho_i(f)$ must be introduced by a type definition or **axioms** clause in the IOA specification that contains the definition of D, by a theory specified in the **assumes** clause of D, or by a built-in datatype of IOA.
- ✓ Each component automaton is supplied with the appropriate number and sorts of its other actual parameters, that is, $actuals^{D,C_i}$ has the same length as $vars^{A_i}$ and the same sorts as $\rho_i(vars^{A_i})$.
- ✓ Each component automaton is supplied with actual types that do not reduce the number of distinct state variables. That is, all selectors of $States[A_i, actualTypes^{D,C_i}]$ are distinct.
- ✓ All occurrences of an action name π in all component automata have the same number and sorts of parameters; that is, if π is an action name in both A_i and A_j, then $vars^{A_i,\pi}$ has the same length as $vars^{A_j,\pi}$, and $\rho_i(vars^{\hat{A}i,\pi})$ has the same sort as $\rho_j(vars^{\hat{A}j,\pi})$.
- ✓ Each action name in a **hidden** statement must be an action name in some component automaton.
- ✓ All occurrences of an action name π in a **hidden** statement have the same number and sorts of parameters as the occurrences of the action name π in the component automata; that is, if π is an action name in some A_i and $\pi = \pi_p$ for the **hidden** clause p, then $vars^{A_i,\pi}$ has the same length as $params^{D,\pi_p}_{hide_p}$, and $\rho_i(vars^{\hat{A}i,\pi})$ has the same sorts as $params^{D,\pi_p}_{hide_p}$.
- ✓ Any variable that occurs freely in a term used as a parameter or predicate, in the definition of a composite automaton must satisfy the restrictions imposed by Table 5.1.

5.4 Semantic proof obligations

The following must also be true for an IOA program to represent a valid I/O automaton. Except in special cases, these conditions cannot be checked automatically, because they may require nontrivial proofs (or even be undecidable); hence static semantic checkers must translate all but the simplest of them into proof obligations for an automated proof assistant. [18]

- ✓ Only output actions may be hidden.
- ✓ The components of a composite automaton must have disjoint sets of output actions.
- ✓ The set of internal actions for any component must be disjoint from the set of all actions of every other component.

We will express these these proof obligations in first-order logic in Section 7.4 using syntactic forms we define earlier in Section 7.

[18] An implementation of these checks might reduce the number of errors reported by first confirming that the composition contains no duplicate instances of any component automaton that contains internal or output actions. Any such duplication would necessarily cause violations of the latter two checks.

6 Expanding component automata

Before we can describe the contribution of a component C_i of a composite automaton D to the expansion of D into a primitive automaton *DExpanded*, we must take four preparatory steps. The result is a component that represents the instantiation of automaton A_i on which C_i is based using the actual parameters supplied by the component and whose variables have been translated into a unified name space used for *DExpanded*.

The first step is to desugar the definition of each component automaton A_i as described in Section 4. In the discussion below, we refer to this desugared version of A_i as $\hat{A}_i$ and assume that it satisfies the restrictions listed in Section 4.5. The second step, shown in Section 6.1, is to replace, throughout the entire definition of the automaton $\hat{A}_i$, the formal type parameters $\mathit{types}^{\hat{A}_i}$ of $\hat{A}_i$ by the actual types $\mathit{actualTypes}^{D,C_i}$ supplied by the component C_i. The third step is to replace the formal automaton (non-type) parameters vars^{A_i} by the actual parameters $\mathit{actuals}^{D,C_i}$ supplied by the component C_i. The fourth step is to translate the aggregate state variables, aggregate local variables, and action parameters from the name space of $\hat{A}_i$ into a unified name space for *DExpanded*. (It is not necessary to translate individual state and local variables, because references to them have been eliminated by the desugaring described in Section 4.4.) Sections 6.2 describes how we choose canonical action parameters for the unified name space. Section 6.3 describes the substitution we use to perform both this translation and the instantiation of actual automaton parameters for the previous step. Table 6.8 summarizes the notation, figures, and examples we use to present these stages.

Section 6.4 describes the result of applying these replacements and translations to individual component automata. It sets the stage Section 7, which describes how to combine the expanded components into a description of *DExpanded* by developing explicit representations for its signature and transition definitions.

6.1 Resorting component automata

We produce a definition of the instances of $\hat{A}_i$ whose sorts correspond to those of the component C_i by replacing the formal type parameters $\mathit{types}^{\hat{A}_i}$ of $\hat{A}_i$ with the actual types $\mathit{actualTypes}^{D,C_i}$ supplied by the component C_i. This replacement is accomplished by applying the resorting ρ_i, defined in Section 5.2 to the entire definition of the automaton $\hat{A}_i$. The precise definition of resortings and a full description of how resortings are extended to perform this replacement throughout the entire definition of the automaton $\hat{A}_i$ are given in Section 9. We denote the resulting definition by $\rho_i\hat{A}_i$.

Example 6.1 Tables 6.1–6.3 show how the resortings $\rho^{\mathtt{C}}$ and $\rho^{\mathtt{W}}$, induced by the **components** statement of the sample automaton `Sys` in Example 2.4, map the sorts, variables, and operators of the component automata.[19] The resorted components $\rho^{\mathtt{C}}$`Channel` and $\rho^{\mathtt{W}}$`Watch` of the composite automaton `Sys` are shown in Figure 6.1. Since the component automaton `P` of `Sys` does not have any type parameters, $\rho^{\mathtt{P}}$ is the identity, and the resorted component $\rho^{\mathtt{P}}$`P` is the same as shown in Figure 4.8.

[19]The table shows only the non-identity mappings of sorts, variables, and operators. Sorts, variables, and operators that appear in the sample automata, but are not shown in the table, are mapped to themselves.

RESORTING	DOMAIN	RANGE
$\rho^{\mathtt{C}}$	`Node`	`Int`
	`Msg`	`Int`
	`Set[Msg]`	`Set[Int]`
	`States[Channel,Node,Msg]`	`States[Channel,Int,Int]`
$\rho^{\mathtt{W}}$	`T`	`Int`
	`Set[T]`	`Set[Int]`
	`Array[T,Bool]`	`Array[Int,Bool]`
	`States[Watch,T]`	`States[Watch,Int]`
	`Locals[Watch,T,overflow]`	`Locals[Watch,Int,overflow]`

Table 6.1: Mappings of sorts by resortings in the composite automaton `Sys`. Resortings listed on the left map domain sorts to their right to the range sorts on their far right.

RESORTING	DOMAIN	RANGE
$\rho^{\mathtt{C}}$	`i:Node`	`i:Int`
	`j:Node`	`j:Int`
	`contents:Set[Msg]`	`contents:Set[Int]`
	`n1:Node`	`n1:Int`
	`n2:Node`	`n2:Int`
	`m:Msg`	`m:Int`
$\rho^{\mathtt{W}}$	`what:Set[T]`	`what:Set[Int]`
	`seen:Array[T,Bool]`	`seen:Array[Int,Bool]`
	`x:T`	`x:Int`
	`x:T`	`x:Int`
	`s:Set[T]`	`s:Set[Int]`
	`s2:Set[T]`	`s2:Set[Int]`

Table 6.2: Mappings of variables by resortings in the composite automaton `Sys`. Resortings listed on the left map domain variables to their right to the range variables on their far right.

RESORTING	OPERATOR	ORIGINAL AND NEW SIGNATURES
$\rho^{\mathtt{C}}$	=	Node,Node→Bool Int,Int→Bool
	=	Msg,Msg→Bool Int,Int→Bool
		→Set[Msg] →Set[Int]
	∈	Msg,Set[Msg]→Bool Int,Set[Int]→Bool
	insert	Msg,Set[Msg]→Set[Msg] Int,Set[Int]→Set[Int]
	delete	Msg,Set[Msg]→Set[Msg] Int,Set[Int]→Set[Int]
	__.contents	States[Channel,Node,Msg]→Set[Msg] States[Channel,Int,Int]→Set[Int]
$\rho^{\mathtt{W}}$	[__]	T→Bool Int→Bool
	{__}	T→Set[T] Int→Set[Int]
	=	Set[T],Set[T]→Bool Set[Int],Set[Int]→Bool
	∈	T,Set[T]→Bool Int,Set[Int]→Bool
	∪	Set[T],Set[T]→Set[T] Set[Int],Set[Int]→Set[Int]
	__.seen	States[Watch,T]→Array[T,Bool] States[Watch,Int]→Array[Int,Bool]
	__.s2	Locals[Watch,T,overflow]→Set[T] Locals[Watch,Int,overflow]→Set[Int]

Table 6.3: Mappings of operators by resortings in the composite automaton Sys. Resortings listed on the left map domain operators to their right to the range operators on their far right.

```
% Resorting of Channel for component C of Sys
automaton Channel(Node, Msg:type, i, j:Int)
  signature
    input send(n1, n2:Int, m:Int) where n1 = i ∧ n2 = j
    output receive(n1, n2:Int, m:Int) where n1 = i ∧ n2 = j
  states contents:Set[Int] := {}
  transitions
    input send(n1, n2, m) where n1 = i ∧ n2 = j
      eff Channel.contents := insert(m, Channel.contents)
    output receive(n1, n2, m) where n1 = i ∧ n2 = j
      pre m ∈ Channel.contents
      eff Channel.contents := delete(m, Channel.contents)

% Resorting of Watch for component W of Sys
automaton Watch(T:Type, what:Set[Int])
  signature
    input overflow(x:Int, s:Set[Int]) where x ∈ what
    output found(x:Int) where x ∈ what
  states seen:Array[Int,Bool] := constant(false)
  transitions
    input overflow(x, s; local s2:Set[Int])
            where s = Watch.s2 ∪ {x} ∨ ¬(x ∈ s)
      eff if s = Watch.s2 ∪ {x} then Watch.seen[x] := true
          elseif ¬(x ∈ s) then Watch.seen[x] := false
          fi
    output found(x)
      pre Watch.seen[x]
```

Figure 6.1: Sample component automata `Channel` and `Watch`, obtained by resorting the desugared automata shown in Figure 4.8

6.2 Introducing canonical names for parameters

For each action name π in some component C_i of D, we pick a sequence $vars^{D,\pi}$ of variables to be the *canonical action parameters* of π in D. Since the static checks ensure the number and sorts of variables in $\rho_i(vars^{\hat{A}i,\pi})$ are the same for all components C_i, we take $vars^{D,\pi}$ to be $\rho_i(vars^{\hat{A}i,\pi})$ for the smallest i such that π is the name of an action in C_i and this choice does not cause variables to clash. In particular, no variable in $vars^{D,\pi}$ should be a parameter of D (i.e., $vars^{D,\pi}$ and $vars^D$ should be disjoint) nor of any component C_i (i.e., $vars^{D,\pi}$ and $vars^{D,C_i}$ should be disjoint).[20]

If $vars^{D,\pi}$ cannot be defined in this fashion (without causing variables to clash), then we let i be the smallest integer such that π is the name of an action in C_i, and we take $vars^{D,\pi}$ to be $\rho_i(vars^{\hat{A}i,\pi})$ with any clashing variables replaced by fresh variables, that is, with variables not in $vars^D$ nor any $vars^{D,C_i}$.

6.3 Substitutions

For each component C_i of a composite automaton D, we define a substitution σ_i (which we write as σ^{C_i} in contexts, such as $\sigma^{\mathtt{W}}$, where it is more convenient to use the name of the component rather than its position in the list of all components) to map the non-type parameters $vars^{\rho_i\hat{A}_i} = \rho_i(vars^{\hat{A}_i})$ of the component automaton $\rho_i\hat{A}_i$ to the corresponding actual parameters $actuals^{D,C_i}$ and to map the aggregate state and post-state variables of $\rho_i\hat{A}_i$ to the appropriate state components in the composite automaton. For each action π of C_i, we also define a substitution $\sigma_{i,\pi}$ to be the same as σ_i, except that it also maps the canonical action parameters $vars^{\rho_i\hat{A}_i,\pi} = \rho_i(vars^{\hat{A}_i})$ of $\rho_i\hat{A}_i$ to the corresponding canonical action parameters $vars^{D,\pi}$ in D, and that it maps the aggregate local and post-local variables for transition definitions in $\rho_i\hat{A}_i$ to the appropriate local and post-local values in the composite automaton.

These substitutions[21] are summarized in Table 6.4 and defined by rules 1–9 below.

1. If x is a non-type parameter of $\hat{A}_i$ (i.e., $x \in vars^{\rho_i\hat{A}_i}$), then $\sigma_i\rho_i(x)$ is the corresponding element of $actuals^{D,C_i}$.

2. If C_i has no parameters and x is the variable A_i of sort $States[A_i, actualTypes^{D,C_i}]$ representing the aggregate states of $\rho_i\hat{A}_i$, then $\sigma_i(x)$ is the state variable for the component C_i of D, which has the same sort as A_i.

3. If C_i has parameters and x is the variable A_i of sort $States[A_i, actualTypes^{D,C_i}]$, then $\sigma_i(x)$ is the term $C_i[vars^{D,C_i}]$, where C_i is the state variable for the component C_i of D, which has sort $\mathtt{Map}[types^{D,C_i}, States[A_i, actualTypes^{D,C_i}]]$.

4. If C_i has no parameters and x is the variable A_i' of sort $States[A_i, actualTypes^{D,C_i}]$ representing the aggregate post-states of $\rho_i\hat{A}_i$, then $\sigma_i(x)$ is the post-state variable C_i' for the component C_i of D.

5. If C_i has parameters and x is the variable A_i' of sort $States[A_i, actualTypes^{D,C_i}]$, then $\sigma_i(x)$ is the term $C_i'[vars^{D,C_i}]$, where C_i' is the post-state variable for the component C_i of D, which has sort $\mathtt{Map}[types^{D,C_i}, States[A_i, actualTypes^{D,C_i}]]$.

[20] It is not necessary to avoid clashes with the state variables $\rho_i(stateVars^{A_i})$ or post-state variables $\rho_i(postVars^{A_i})$ of C_i, because desugaring has replaced references to such variables x by terms $C_i.x$.

[21] See Section 9 for a precise definition of substitutions, which ensures that they do not capture **local**, **for**, **choose**, or quantified variables.

SUBSTITUTION	DOMAIN	RANGE	RULE
σ_i	$vars^{\rho_i \hat{A}_i}$	$actuals^{D,C_i}$	rule 1
	$A_i{:}States[A_i, actualTypes^{D,C_i}]$	C_i	rule 2
	$A_i{:}States[A_i, actualTypes^{D,C_i}]$	$C_i[vars^{D,C_i}]$	rule 3
$\sigma_{i,\pi}$	$vars^{\rho_i \hat{A}_i}$	$actuals^{D,C_i}$	rule 1
	$A_i{:}States[A_i, actualTypes^{D,C_i}]$	C_i	rule 2
	$A_i{:}States[A_i, actualTypes^{D,C_i}]$	$C_i[vars^{D,C_i}]$	rule 3
	$A'_i{:}States[A_i, actualTypes^{D,C_i}]$	C'_i	rule 4
	$A'_i{:}States[A_i, actualTypes^{D,C_i}]$	$C'_i[vars^{D,C_i}]$	rule 5
	$vars^{\rho_i \hat{A}_i,\pi}$	$vars^{D,\pi}$	rule 7
	$A_i{:}Locals[A_i, actualTypes^{D,C_i}, \pi]$	C_i	rule 8
	$A'_i{:}Locals[A_i, actualTypes^{D,C_i}, \pi]$	C'_i	rule 8
	$A_i{:}Locals[A_i, actualTypes^{D,C_i}, \pi]$	$C_i[vars^{D,C_i}]$	rule 9
	$A'_i{:}Locals[A_i, actualTypes^{D,C_i}, \pi]$	$C'_i[vars^{D,C_i}]$	rule 9

Table 6.4: Substitutions used in canonicalizing component automata. Substitutions listed on the left map variables in the domains to their right to range variables according to the listed rules.

6. There is no rule 6! [1]

7. If x is a canonical action parameter (i.e., $x \in vars^{\hat{A}_i,\pi}$), then $\sigma_{i,\pi}\rho_i(x)$ is the corresponding element of $vars^{D,\pi}$.

8. If C_i has no parameters and x is the variable A_i of sort $Locals[A_i, actualTypes^{D,C_i,\pi}]$ (or the variable A'_i of the same sort) representing the aggregate local (or post-local) variables for a transition definition, then $\sigma_i(x)$ is the local variable C_i (or the post-local variable C'_i) for the transition definition in D, which has the same sort as A_i.

9. If C_i has parameters and x is the variable A_i of sort $Locals[A_i, actualTypes^{D,C_i,\pi}]$ (or the variable A'_i of the same sort), then $\sigma_i(x)$ is the term $C_i[vars^{D,C_i}]$ (or the term $C'_i[vars^{D,C_i}]$), where C_i and C'_i are the aggregate local and post-local variables in D, which have sort $\texttt{Map[}types^{D,C_i}, Locals[A_i, actualTypes^{D,C_i}, \pi]\texttt{]}$.

6.4 Canonical component automata

For each component C_i of D, we obtain a canonical automaton definition C_i for that component by applying ρ_i and then σ_i to the desugared definition $\hat{A}_i$ of A_i. Figure 6.2 shows the general form for such canonical component automata.

In the list of parameters for C_i, the type parameters $types^D$ of D replace the type parameters $types^{\hat{A}_i}$ of $\hat{A}_i$, and the variables $vars^D$ and $vars^{D,C_i}$ that parameterize D and its component C_i replace the individual parameters $vars^{A_i}$ of $\hat{A}_i$. The body of the automaton definition for C_i is obtained by applying the resorting ρ_i to the body of the automaton definition for $\hat{A}_i$, thereby eliminating all references to the type parameters in $\mathit{types}^{\hat{A}_i}$, to obtain a resorted definition for an automaton $\rho_i \hat{A}_i$ and then by applying the substitution σ_i to this resorted definition, thereby eliminating all references to the individual parameters in vars^{A_i}. We do not apply σ_i to $\mathit{stateVars}^{\rho_i A_i}$, because we wish to preserve the names of the state variables in $\mathit{stateVars}^{A_i}$. No ambiguity arises, because the desugaring described in Section 4.4 has replaced all references to state variables x in the definition of $\hat{A}_i$ with terms of the form $A_i.x$. For each action π, we also apply $\sigma_{i,\pi}$ to the **where** clause $P_{\mathit{kind}}^{\rho_i \hat{A}_i,\pi}$ for π in the signature of $\rho_i \hat{A}_i$ and to the transition definition for π in $\rho_i \hat{A}_i$.

Despite the absence of ambiguity, the automaton C_i may not pass the static semantic requirements in Section 3.3 that prohibit any clashes between state variables and automaton parameters. Furthermore, if C_i has non-type parameters, the aggregate state variable for the automaton is a map as specified in Section 5.2 rather than a tuple as specified for primitive automata in Section 3.2.

Table 6.8 shows the steps taken to expand canonical component automata. The "Original" column lists the names for syntactic elements of automata introduced in Section 3. The notation given in the "Desugared" column describes the result of desugaring such automata as described in Section 4. The elements listed in the the "Resorted" column result from the resorting of desugared component automata that Section 6.1 describes. Syntactic elements listed in the "Expanded" column are derived in Section 6.3 from resorted automata. Finally, names that appear in the "Component" column are just synonyms for the values in the previous column. We use these simpler synonyms in Section 7.

automaton $C_i(\mathit{types}^D, \mathit{vars}^D, \mathit{vars}^{D,C_i})$

  signature

    kind $\pi(\mathit{vars}^{D,\pi})$ **where** $\sigma_{i,\pi}(P_{\mathit{kind}}^{\rho_i \hat{A}_i,\pi})$

    ...

  states $\mathit{stateVars}^{\rho_i A_i} := \sigma_i(\mathit{initVals}^{\rho_i \hat{A}_i})$ **initially** $\sigma_i(P_{\mathit{init}}^{\rho_i \hat{A}_i})$

  transitions

$$\sigma_{i,\pi}\left[\begin{array}{l}\textbf{kind}\ \pi(\mathit{vars}^{\rho_i \hat{A}_i,\pi};\ \textbf{local}\ \mathit{localVars}^{\rho_i \hat{A}_i,\pi})\ \textbf{where}\ P_{\mathit{kind},t_1}^{\rho_i \hat{A}_i,\pi}\\ \quad \textbf{pre}\ \mathit{Pre}_{\mathit{kind}}^{\rho_i \hat{A}_i,\pi}\\ \quad \textbf{eff}\ \mathit{Prog}_{\mathit{kind}}^{\rho_i \hat{A}_i,\pi}\ \textbf{ensuring}\ \mathit{ensuring}_{\mathit{kind}}^{\rho_i \hat{A}_i,\pi}\end{array}\right]$$

    ...

Figure 6.2: General form of the expansion of the automaton for component C_i, obtained from the desugared definition $\hat{A}_i$ of the automaton on which C_i is based

Example 6.2 We derive the component automata `C`, `P`, and `W` of the composite automaton `Sys` by applying the substitutions shown in Tables 6.5–6.7 to the resorted automata $\rho^{\mathtt{C}}$`Channel` and $\rho^{\mathtt{W}}$`Watch` shown in Figure 6.1 and to the canonicalized automaton `P` shown in Figure 4.8. Since the per-action substitutions (e.g., $\sigma^{\mathtt{C,send}}$) are always extensions of the per-component substitutions (e.g., $\sigma^{\mathtt{C}}$), these tables show only the *additional* mappings that distinguish the per-action substitutions from the per-component substitutions. We also omit from these tables identity mappings. For example, we omit from Table 6.6 the identity mapping of `i1:Int` to itself due to rule 7 in $\sigma^{\mathtt{P,overflow}}$. The resulting component automata are shown in Figures 6.3–6.5.

```
automaton C(nProcesses:Int, n:Int)
  signature
    input send(n1, n2:Int, m:Int) where n1 = n ∧ n2 = n+1
    output receive(n1, n2:Int, m:Int) where n1 = n ∧ n2 = n+1
  states contents:Set[Int] := {}
  transitions
    input send(n1, n2, m) where n1 = n ∧ n2 = n+1
      eff C[n].contents := insert(m, C[n].contents)
    output receive(n1, n2, m) where n1 = n ∧ n2 = n+1
      pre m ∈ C[n].contents
      eff C[n].contents := delete(m, C[n].contents)
```

Figure 6.3: Sample instantiated component automaton `C`, obtained by applying the substitutions in Table 6.5 to the resorted automaton `Channel` in Figure 6.1

SUBSTITUTION	DOMAIN	RANGE	RULE
$\sigma^{\mathtt{C}}$	`Channel:States[Channel,Int,Int]`	`C[n]:Map[Int,States[Channel,Int,Int]]`	rule 3
	`i:Int`	`n:Int`	rule 1
	`j:Int`	`(n+1):Int`	rule 1
$\sigma^{\mathtt{C,send}}$	No additional substitutions		
$\sigma^{\mathtt{C,receive}}$	No additional substitutions		

Table 6.5: Substitutions used to derive sample component automaton `C`. Substitutions listed on the left map variables in the domain to their right to variables in the range their far right.

SUBSTITUTION	DOMAIN	RANGE	RULE
$\sigma^{\mathtt{P}}$	`P:States[P]`	`P[n]:Map[Int,States[P]]`	rule 3
$\sigma^{\mathtt{P,send}}$	`i1:Int`	`n1:int`	rule 7
	`i2:Int`	`n2:int`	rule 7
	`x:Int`	`m:int`	rule 7
$\sigma^{\mathtt{P,receive}}$	`i1:Int`	`n1:int`	rule 7
	`i2:Int`	`n2:int`	rule 7
	`x:Int`	`m:int`	rule 7
$\sigma^{\mathtt{P,overflow}}$	`P:Locals[P,overflow]`	`P[n]:Map[Int,Locals[P,overflow]]`	rule 9

Table 6.6: Substitutions used to derive sample component automaton `P`. Substitutions listed on the left map variables in the domain to their right to variables in the range their far right.

```
automaton P(nProcesses:Int, n:Int)
  signature
    input receive(n1, n2, m:Int) where n1 = n-1 ∧ n2 = n
    output send(n1, n2, m:Int) where n1 = n ∧ n2 = n+1,
           overflow(i1:Int, s:Set[Int]) where i1 = n
  states
    val:Int := 0,
    toSend:Set[Int] := {}
  transitions
    input receive(n1, n2, m) where n1 = n-1 ∧ n2 = n
      eff if P[n].val = 0 then P[n].val := m
          elseif m < P[n].val then
            P[n].toSend := insert(P[n].val, P[n].toSend);
            P[n].val := m
          elseif P[n].val < m then
            P[n].toSend := insert(m, P[n].toSend)
          fi
    output send(n1, n2, m) where n1 = n ∧ n2 = n+1
      pre m ∈ P[n].toSend
      eff P[n].toSend := delete(m, P[n].toSend)
    output overflow(i1, s; local t:Set[Int]) where i1 = n
      pre s = P[n].toSend ∧ n < size(s) ∧ P[n].t ⊆ s
      eff P[n].toSend := P[n].t
```

Figure 6.4: Sample instantiated component automaton `P`, obtained by applying the substitutions in Table 6.6 to the automaton `P` in Figure 4.8

```
automaton W(nProcesses:Int)
  signature
    input overflow(i1:Int, s:Set[Int]) where i1 ∈ between(1, nProcesses)
    output found(i1:Int) where i1 ∈ between(1, nProcesses)
  states seen:Array[Int,Bool] := constant(false)
  transitions
    input overflow(i1, s; local s2:Set[Int])
            where s = W.s2 ∪ {i1} ∨ ¬(i1 ∈ s)
      eff if s = W.s2 ∪ {i1} then W.seen[i1] := true
          elseif ¬(i1 ∈ s) then W.seen[i1] := false
          fi
    output found(i1)
      pre W.seen[i1]
```

Figure 6.5: Sample instantiated component automaton `W`, obtained by applying the substitutions in Table 6.7 to the resorted automaton `Watch` in Figure 6.1

SUBSTITUTION	DOMAIN	RANGE	RULE
$\sigma^{\mathtt{W}}$	Watch:States[Watch,Int]	W:States[Watch,Int]	rule 2
	what:Set[Int]	between(1, nProcesses)	rule 1
$\sigma^{\mathtt{W,overflow}}$	Watch:Locals[Watch,Int,overflow]	W:Locals[Watch,Int,overflow]	rule 8
	x:Int	i1:int	rule 7
$\sigma^{\mathtt{W,found}}$	x:Int	i1:Int	rule 7

Table 6.7: Substitutions used to derive sample component automaton W. Substitutions listed on the left map variables in the domain to their right to variables in the range their far right.

Syntactic Element	Original	Desugared	Resorted	Expanded	Component
Automaton	A_i	$\hat{A}_i$	$\rho_i \hat{A}_i$	$\sigma_i \rho_i \hat{A}_i$	C_i
General form	Figure 3.1	Figures 4.3, 4.5, 4.7		Figure 6.2	
Automaton parameters	$types^{A_i},\ vars^{A_i}$		$types^{A_i},\ \rho_i vars^{A_i}$	$types^{D},\ vars^{D},\ vars^{D,C_i}$	
Action parameters	$params^{A_i,\pi}_{kind}$	$vars^{\hat{A}_i,\pi}$	$\rho_i vars^{\hat{A}_i,\pi}$	$\sigma_{i,\pi}\rho_i vars^{\hat{A}_i,\pi}$	$vars^{D,\pi}$
Signature **where** predicates	$P^{A_i,\pi}_{kind}$	$P^{\hat{A}_i,\pi}_{kind}$	$\rho_i P^{\hat{A}_i,\pi}_{kind}$	$\sigma_{i,\pi}\rho_i P^{\hat{A}_i,\pi}_{kind}$	$P^{C_i,\pi}_{kind}$
State variables	$stateVars^{A_i}$		$\rho_i stateVars^{A_i}$		$stateVars^{C_i}$
Aggregate variable name	A_i			C_i	
Aggregate state sort	$States[A_i, types^{A_i}]$		$States[A_i, actualTypes^{D,C_i}]$		
Aggregate local sort	$Locals[A_i, types^{A_i}, kind, \pi, t_j]$	$Locals[A_i, types^{A_i}, \pi]$	$Locals[A_i, actualTypes^{D,C_i}, \pi]$		
Initial state variables values	$initVals^{A_i}$		$\rho_i initVals^{A_i}$	$\sigma_i \rho_i initVals^{A_i}$	$initVals^{C_i}$
initially predicate	$P^{A_i}_{init}$	$P^{\hat{A}_i}_{init}$	$\rho_i P^{\hat{A}_i}_{init}$	$\sigma_i \rho_i P^{\hat{A}_i}_{init}$	$P^{C_i}_{init}$
Transition parameters	$params^{A_i,\pi}_{kind,t_j}$	$vars^{\hat{A}_i,\pi}$	$\rho_i vars^{\hat{A}_i,\pi}$	$\sigma_{i,\pi}\rho_i vars^{\hat{A}_i,\pi}$	$vars^{D,\pi}$
local variables	$localVars^{A_i,\pi}_{kind,t_j}$	$localVars^{\hat{A}_i,\pi}_{kind}$	$\rho_i\ localVars^{\hat{A}_i,\pi}_{kind}$	$\sigma_{i,\pi}\rho_i\ localVars^{\hat{A}_i,\pi}_{kind}$	$localVars^{C_i,\pi}_{kind}$
Transition **where** predicates	$P^{A_i,\pi}_{kind,t_j}$	$P^{\hat{A}_i,\pi}_{kind,t_1}$	$\rho_i P^{\hat{A}_i,\pi}_{kind,t_1}$	$\sigma_{i,\pi}\rho_i P^{\hat{A}_i,\pi}_{kind,t_1}$	$P^{C_i,\pi}_{kind,t_1}$
Transition **pre** predicates	$Pre^{A_i,\pi}_{kind,t_j}$	$Pre^{\hat{A}_i,\pi}_{kind}$	$\rho_i Pre^{\hat{A}_i,\pi}_{kind}$	$\sigma_{i,\pi}\rho_i Pre^{\hat{A}_i,\pi}_{kind}$	$Pre^{C_i,\pi}_{kind}$
Transition **eff** programs	$Prog^{A_i,\pi}_{kind,t_j}$	$Prog^{\hat{A}_i,\pi}_{kind}$	$\rho_i Prog^{\hat{A}_i,\pi}_{kind}$	$\sigma_{i,\pi}\rho_i Prog^{\hat{A}_i,\pi}_{kind}$	$Prog^{C_i,\pi}_{kind}$
Transition **eff** predicates	$ensuring^{A_i,\pi}_{kind,t_j}$	$ensuring^{\hat{A}_i,\pi}_{kind}$	$\rho_i ensuring^{\hat{A}_i,\pi}_{kind}$	$\sigma_{i,\pi}\rho_i ensuring^{\hat{A}_i,\pi}_{kind}$	$ensuring^{C_i,\pi}_{kind}$
`Channel`	Figure 2.1	Figure 4.8	Figure 6.1	Figure 6.3	
`P`	Figure 2.2	Figure 4.8		Figure 6.4	
`Watch`	Figure 2.3	Figure 4.8	Figure 6.1	Figure 6.5	

Table 6.8: Stages in expanding components C_i of a composite automaton D.

7 Expanding composite automata

In this section, we present the main contribution of this document. We show how to expand a composite IOA program into an equivalent primitive IOA program. Section 7.1 reviews our assumptions about the form of the components of the composite automaton, and Section 7.2 describes a simplification of the structure of **hidden** statements, obtained by combining all clauses for a single action into a single clause.

In Section 7.3, we define the expansion of the signature of a composite automaton to primitive form. Section 7.4 gives first-order logic formulas for the semantic proof obligations we introduced in Section 5.4. These include compatibility requirements for component automata. In Section 7.5, we define the expansion of the **initially** predicate on states of a composite automaton. In Sections 7.6–7.9, we define the expansion of the transitions of a composite automaton.

7.1 Expansion assumptions

We expand a composite automaton D into primitive form by combining elements of its components $C_1, \ldots, C_n$. We assume each component automaton A_i has been desugared to satisfy the restrictions in Section 4.5, resorted to produce an automaton $\rho_i \hat{A}_i$ as described in Section 5.2 and 6.1, and transformed as described in Section 6.4 to produce an automaton $\sigma_i \rho_i \hat{A}_i = C_i$. In particular, for each component automaton C_i, we assume the following.

- No **const** parameters appear in the signature.
- Each appearance of an action π in the signature is parameterized by the canonical action parameters $\mathit{vars}^{D,\pi}$.
- Each transition definition of an action π is parameterized by the canonical action parameters $\mathit{vars}^{D,\pi}$.
- Each transition definition of an action π is further parameterized by the canonical sequence $\sigma_{i,\pi} \rho_i\ \mathit{localVars}^{\hat{A}_i,\pi}$ of local variables for that component.
- Each action has at most one transition definition of each kind.
- Every state, post-state, local variable, or post-local variable reference is of the unambiguous form $C_i.x$, $C_i'.x$, $C_i[\mathit{vars}^{D,C_i}].x$, or $C_i'[\mathit{vars}^{D,C_i}].x$.

7.2 Desugaring hidden statements of composite automata

The syntax for composite IOA programs as described in Section 5 provides programmers with flexibility of expression that can complicate expansion into primitive form. Hence, as with primitive automata, it is helpful to consider equivalent composite IOA programs that conform to a more limited "desugared" syntax. As discussed later in this section, **where** clauses of composite automaton **hidden** statements and of component transitions are combined during expansion. Thus, **hidden** statements must be desugared into a form analogous to that of a desugared transition. In particular, we desugar composite automata with **hidden** statements to have the following two properties.

- Each **hidden** clause for an action π is parameterized by the canonical action parameters $\mathit{vars}^{D,\pi}$.

- There is at most one **hidden** clause for each action π.

The static checks described in Section 5.3 ensure that the number and sorts of terms in $params_{hide_p}^{D,\pi_p}$ are the same as the number and sorts of variables in $vars^{D,\pi_p}$. If no variable in $vars^{D,\pi_p}$ occurs freely in $params_{hide_p}^{D,\pi_p}$ (i.e., if $vars^{D,\pi_p}$ and $vars_{hide_p}^{D,\pi_p}$ are disjoint), then we can desugar the clause

$$\pi_p(params_{hide_p}^{D,\pi_p}) \textbf{ where } H_{hide_p}^{D,\pi_p}$$

by replacing $params_{hide_p}^{D,\pi_p}$ by $vars^{D,\pi_p}$, reintroducing $vars_{hide_p}^{D,\pi_p}$ as existentially quantified variables in the **where** clause, and adding conjuncts to the **where** clause to equate $vars^{D,\pi_p}$ with the old parameters. This results in the desugaring

$$\pi_p(vars^{D,\pi_p}) \textbf{ where } \exists\ vars_{hide_p}^{D,\pi_p} \left(H_{hide_p}^{D,\pi_p}\ \wedge\ vars^{D,\pi_p} = params_{hide_p}^{D,\pi_p}\right).$$

Notice that introducing $vars_{hide_p}^{D,\pi_p}$ as existentially quantified variables is analogous to introducing $vars_{in,t_j}^{A,\pi}$ as local variables when desugaring transition parameters, as described in Section 4.1.

If $vars^{D,\pi_p}$ and $vars_{hide_p}^{D,\pi_p}$ are not disjoint, we define a substitution σ_p^{hide} that maps the intersection of these two sets to a set of fresh variables, and we desugar the **hidden** clause as

$$\pi_p(vars^{D,\pi_p}) \textbf{ where } \exists\ \sigma_p^{hide} vars_{hide_p}^{D,\pi_p} \left(\sigma_p^{hide} H_{hide_p}^{D,\pi_p}\ \wedge\ vars^{D,\pi_p} = \sigma_p^{hide} params_{hide_p}^{D,\pi_p}\right).$$

We simplify each existentially qualified **where** clause produced by the above transformations by dropping any existential quantifier, such as $\exists$ `i:Int` in the example, that introduces a variable equated to a term, as in `i = x` in the example, in the conjunction $vars^{D,\pi_p} = \sigma_p^{hide} params_{hide_p}^{D,\pi_p}$, and also by dropping the equating conjunct from that conjunction. We denote the resulting simplification of the **where** clause by $H_{hide_p,canon}^{D,\pi}$.

Following this clause-by-clause canonicalization, we combine all clauses in the **hidden** statement that apply to a single action π into one disjunction. This step is analogous to the combining step for transition definitions in Section 4.3. For example, if $\pi_p = \pi_q = \pi$, then the clauses

$$\pi_p(vars^{D,\pi_p}) \textbf{ where } H_{hide_p,canon}^{D,\pi_p}$$
$$\pi_q(vars^{D,\pi_q}) \textbf{ where } H_{hide_q,canon}^{D,\pi_q}$$

become the single clause

$$\pi(vars^{D,\pi}) \textbf{ where } H_{hide_p,canon}^{D,\pi_p}\ \vee\ H_{hide_q,canon}^{D,\pi_q}.$$

We denote this combined **where** clause by $H^{D,\pi}$.

7.3 Expanding the signature of composite automata

In the composite automaton D, actions that are internal to some component are internal actions of the composition, actions that are outputs of some component and are not hidden are output actions of the composition, and actions that are inputs to some components but outputs to none are input actions of the composition. The **where** clause predicates $P_{kind}^{D,\pi}$ express these facts in the signature of the expanded automaton *DExpanded*. We construct these predicates by defining subformulas, $P_{kind}^{D,C_i,\pi}$ and $Prov_{kind}^{D,\pi}$ which describe the actions components contribute to the composition. We

automaton $DExpanded(types^D, vars^D)$

signature

kind $\pi(vars^{D,\pi})$ **where** $P^{D,\pi}_{kind}$

...

Figure 7.1: General form of the signature in the expansion of a composite automaton

combine these formulas and the **where** predicate from any applicable **hidden** clause (i.e., $H^{D,\pi}$), to account for the subsumption of input actions by output actions and for hiding output actions. The final result consists of the three predicates $P^{D,\pi}_{in}$, $P^{D,\pi}_{out}$, and $P^{D,\pi}_{int}$.

All free variables that appear in these predicates are among the composite automaton parameters $vars^D$ and the canonical action parameters $vars^{D,\pi}$. Figure 7.1 shows the general form of the expanded signature. Below, we explain how to construct these predicates. (See Section 8.2 for an example application of the process to composite automaton `Sys` defined in Example 2.4.)

Subformulas for actions contributed by a component

In order for an action **kind** $\pi(vars^{D,\pi})$ to be defined in D, it must be defined in some component. An action is defined in a component C_i of D if, given action parameters $vars^{D,\pi}$ there are component parameters $vars^{D,C_i}$ that satisfy both the component **where** clause P^{D,C_i} and the action **where** clause $P^{C_i,\pi}_{kind}$ for π in the signature of C_i. Hence we define

$$P^{D,C_i,\pi}_{kind} \ ::= \ \exists vars^{D,C_i} \ (P^{D,C_i} \ \wedge \ P^{C_i,\pi}_{kind}),$$

which is satisfied by $vars^{D,\pi}$ if and only if $\pi(vars^{D,\pi})$ is an action of type *kind* in component C_i of D.

It is important to note that the type of the action $\pi(vars^{D,\pi})$ in D may be different from the type of π in some, or even all, the components contributing the action to the composition. Output actions in one instance of one component may subsume inputs in another, and output actions may be hidden as internal actions in the composition. We say that *kind* is the *provisional kind* of $\pi(vars^{D,\pi})$ in D when an action of that kind is contributed to the composition by some component. Hence we define the predicate $Prov^{D,\pi}_{kind}$ as follows:

$$Prov^{D,\pi}_{kind} \ ::= \ \bigvee_{1\le i\le n} P^{D,C_i,\pi}_{kind}.$$

Signature predicates

We account for subsumed inputs and hidden outputs in the signature of $DExpanded$ by appending special case formulas to the predicates $Prov^{D,\pi}_{kind}$ to form the signature predicates $P^{D,\pi}_{kind}$. The three cases we must consider are:

- An action $\pi(vars^{D,\pi})$ is an output action of D if and only if it is an output action in some component C_i of D and is not hidden in D.

- An action $\pi(vars^{D,\pi})$ is an input action of D if and only if it is an input action in some component C_i of D, but not an output action in any component of D.

- An action $\pi(vars^{D,\pi})$ is an internal action of D if and only if it is an internal action, or a hidden output action, in some component C_i of D.

Translating these requirements into first-order logic, we derive the following definitions for the signature predicates of $DExpanded$:

- $P_{out}^{D,\pi} ::= Prov_{out}^{D,\pi} \wedge \neg H^{D,\pi}$
- $P_{in}^{D,\pi} ::= Prov_{in}^{D,\pi} \wedge \neg Prov_{out}^{D,\pi}$
- $P_{int}^{D,\pi} ::= Prov_{int}^{D,\pi} \vee \left(Prov_{out}^{D,\pi} \wedge H^{D,\pi}\right)$.

7.4 Semantic proof obligations, revisited

We are now ready to formalize the following proof obligations on composite automata introduced in Section 5.4.

✓ Only output actions may be hidden.

✓ The components of a composite automaton have disjoint sets of output actions.

✓ The set of internal actions for any component is disjoint from the set of all actions of every other component.

Below we give corresponding formulas in first-order logic that must be verified for a composite IOA program to represent a valid I/O automaton. In order to express the latter two of these obligations in first-order logic, we break each of them into two parts. First, we consider different components from different clauses of the **components** statement (i.e., $C_i \neq C_j$). Second, we consider instances of the same parameterized component distinguished only by parameter values (i.e., $C_i[vars^{D,C_i}] \neq C_i[vars'^{D,C_i}]$). We use these formulas to help construct the expansion of transitions of composite automata in Sections 7.7–7.9.

Hidden actions

The first of these obligations is just the requirement that

✓ $H^{D,\pi} \Rightarrow Prov_{out}^{D,\pi}$.

Output actions

For output actions, we first require that different parameterized components have disjoint sets of output actions. Formally, we say that for all distinct components C_i and C_j of D, all values of the action parameters $vars^{D,\pi}$ for π, all values of the composite automaton parameters $vars^{D}$, and all values of the component parameters $vars^{D,C_i}$ and $vars^{D,C_j}$, we require that

$$\checkmark \quad \neg P_{out}^{D,C_i,\pi} \vee \neg P_{out}^{D,C_j,\pi} \tag{7.1}$$

Second, we require that different instances of the same parameterized component have disjoint sets of output actions. That is, for each component C_i of D, all values of the action parameters

$vars^{D,\pi}$ for π, all values of the individual parameters $vars^D$ of the composite automaton, and all pairs of values of the component parameters $vars^{D,C_i}$ and $vars'^{D,C_i}$, we require that

$$\checkmark \quad \left(P^{D,C_i} \ \wedge \ P'^{D,C_i} \ \wedge \ P^{C_i,\pi}_{out} \ \wedge \ P'^{C_i,\pi}_{out}\right) \ \Rightarrow \ vars^{D,C_i} = vars'^{D,C_i} \tag{7.2}$$

where $P'^{C_i,\pi}_{out}$ is $P^{C_i,\pi}_{out}$ evaluated on $vars'^{D,C_i}$.

In Example 2.4, these requirements are satisfied trivially, because the output actions in the different components of `Sys` have different labels. However, the composition

```
automaton BadSys1
  components P1[n:Int] where 0 < n ∧ n < 10;
             P2: P(5)
```

would violate the first requirement, because components `P1[5]` and `P2` share an output action, and the composition

```
automaton BadSys2
  components W[what:Set[Int]]: Watch(Int, what)
           where what = between(1,1) ∨ what = between(1,2)
```

would violate the second requirement because components `W[[1]]` and `W[[1,2]]` both have `found(1)` as an output action.

Internal actions

Similarly, we break the last of these semantic proof obligations, which concerns internal actions, into two parts. We first require that internal actions are defined in one component only for parameter values where no action is defined in any other component. Formally, we say that for all distinct components C_i and C_j of D, all values of the action parameters $vars^{D,\pi}$, and all values of the composite automaton non-type parameters $vars^D$, we require that

$$\checkmark \quad P^{D,C_i,\pi}_{int} \Rightarrow \neg P^{D,C_j,\pi}_{all} \tag{7.3}$$

where $P^{D,C_j,\pi}_{all}$ is the disjunction of $P^{D,C_j,\pi}_{in}$, $P^{D,C_j,\pi}_{out}$, and $P^{D,C_j,\pi}_{int}$.

Second, we require that internal actions of one instance of a parameterized component are defined only for parameter values where no action is defined in any other instance of that component. That is, for each component C_i of D, all values of the action parameters $vars^{D,\pi}$, all values of the composite automaton non-type parameters $vars^D$, and all pairs of values of component non-type parameters $vars^{D,C_i}$ and $vars'^{D,C_i}$, we require that

$$\checkmark \quad \left(P^{D,C_i} \ \wedge \ P'^{D,C_i} \ \wedge \ P^{C_i,\pi}_{int} \ \wedge \ P'^{C_i,\pi}_{all}\right) \ \Rightarrow \ vars^{D,C_i} = vars'^{D,C_i}, \tag{7.4}$$

where $P'^{C_i,\pi}_{all}$ is the disjunction of $P'^{C_i,\pi}_{in}$, $P'^{C_i,\pi}_{out}$, and $P'^{C_i,\pi}_{int}$ and where the primed form of each predicate is the evaluation of the predicate on $vars'^{D,C_i}$.

Note, although allowed by obligation 7.4, the cases where $P^{C_i,\pi}_{int} \ \wedge \ P^{C_i,\pi}_{in}$ or $P^{C_i,\pi}_{int} \ \wedge \ P^{C_i,\pi}_{out}$ hold are already disallowed by semantic proof obligations 4.2 and 4.3, respectively.

Claim 1 (Signature compatibility) Semantic proof obligations 7.1–7.4 taken together with the signature **where** predicates $P^{D,\pi}_{kind}$ imply that *DExpanded* fulfills the semantic proof obligations for primitive automata 4.1–4.3.

In Sections 7.7–7.9, we argue that remaining obligations for primitive automata (4.4 and 4.5) are discharged by the transition **where** clauses of *DExpanded*.

7.5 Expanding **initially** predicates of composite automata

In Section 5.2, we described the state variables of a composite automaton D. Corresponding to each component C_i is a state variable C_i with sort $States[A_i, actualTypes^{D,C_i}]$ if C_i has no parameters and with sort $\texttt{Map[}types^{D,C_i}, States[A_i, actualTypes^{D,C_i}]\texttt{]}$ otherwise. Here, we describe the construction of an **initially** predicate that constrains the initial values of these state variables. This predicate is a conjunction of clauses, one per unparameterized component and two per parameterized component.

If a component C_i is not parameterized (i.e., the state variable C_i is a tuple, not a map), then a single clause asserts that, for all values of the component parameters for which the component is defined (i.e., when P^{D,C_i} is true), each element of the tuple has an appropriate initial value. Furthermore, the clause asserts that, when P^{D,C_i} is true, the tuple as a whole satisfies the **initially** predicate $P^{C_i}_{init}$ of the component. In order to account for initial values specified as nondeterministic choices, we proceed as follows. Let

- X_i be the set of indices k of state variable declarations of the form

  $$x_k{:}T_k \;:=\; \textbf{choose}\ v_k{:}T_k\ \textbf{where}\ P^{C_i}_{init,k}$$

  in the definition of the component C_i,

- $cVars^{C_i}$ be a set of distinct fresh variables $v'_k{:}T_k$, one for each k in X_i,

- $*initVals^{C_i}$ be $initVals^{C_i}$ with each of the above **choose** expressions replaced by the corresponding $v'_k{:}T_k$ for each k in X_i, and

- $*P^{C_i}_{init,k}$ be $P^{C_i}_{init,k}$ with v'_k substituted for v_k when $k \in X_i$ and the predicate $true$ otherwise.

Then we formulate the clause (shown in Figure 7.2) corresponding to C_i in the **initially** predicate of $DExpanded$ by factoring out, and existentially qualifying, the variables (i.e., $cVars^{C_i}$) used to choose nondeterministic values for the state variables of the component automaton C_i.

When a component C_i is parameterized (i.e., the state variable C_i is a map, not a tuple), then there are two clauses for the component. The first is analogous to the single clause for the simple case in which the state variable is a tuple, but now it asserts that each element of each tuple in the map has an appropriate initial value and that, when P^{D,C_i} is true, the map as a whole satisfies the **initially** predicate of the component. The second clause asserts that the map is defined exactly for the values of the component parameters for which the component itself is defined (i.e., when P^{D,C_i} is true). This second clause is also asserted automatically as an invariant of the automaton. That is, no transition either extends or reduces the domain over which the map is defined. Figure 7.2 summarizes these two cases and the invariant.

7.6 Combining local variables of composite automata

Just as it helped to collect the local variables from all transition definitions for an action π when desugaring a primitive automaton (see Section 4.3), it helps to collect the local variables from the transitions definitions from different components for an action π when expanding the definition of a composite automaton. Hence, we parameterize every transition definition by n per-component aggregate local variables that are named for the components $C_1, \ldots, C_n$ just as the n per-component aggregate state variables are named for those components (see Section 5.2).

states

$\ldots,$

$C_i\colon \mathit{States}[A_i, \mathit{actualTypes}^{D,C_i}],$ % if vars^{D,C_i} is empty

$\ldots,$

$C_j\colon \texttt{Map}[\mathit{vars}^{D,C_j}, \mathit{States}[A_j, \mathit{actualTypes}^{D,C_j}]],$ % if vars^{D,C_j} is not empty

$\ldots$

initially

$\ldots \wedge$

$P^{D,C_i} \Rightarrow \exists c\mathit{Vars}^{C_i} \left(P_{\mathit{init}}^{C_i} \wedge C_i.\mathit{stateVars}^{C_i} = *\mathit{initVals}^{C_i} \wedge \bigwedge_{k\in X_i} *P_{\mathit{init},k}^{C_i}\right) \wedge$

$\ldots \wedge$

$$\forall \mathit{vars}^{D,C_j} \Big(P^{D,C_j} \Rightarrow \exists c\mathit{Vars}^{C_j}(P_{\mathit{init}}^{C_j} \wedge C_j[\mathit{vars}^{D,C_j}].\mathit{stateVars}^{C_j} = *\mathit{initVals}^{C_j} \wedge \textstyle\bigwedge_{k\in X_j} *P_{\mathit{init},k}^{C_j})\Big) \wedge$$

$\forall \mathit{vars}^{D,C_j} \left(P^{D,C_j} \Leftrightarrow \texttt{defined}(C_j[\mathit{vars}^{D,C_j}])\right) \wedge$

$\ldots$

invariant of $\mathit{DExpanded}$:

$\ldots;$

$\forall \mathit{vars}^{D,C_j} \left(P^{D,C_j} \Leftrightarrow \texttt{defined}(C_j[\mathit{vars}^{D,C_j}])\right);$

$\ldots$

Figure 7.2: General form of the states in the expansion of a composite automaton

The sort of each per-component local variable depends on the name of the action and the parameterization of the component. If the component C_i has no parameters, then the aggregate local variable C_i has sort $\mathit{Locals}[A_i, \mathit{actualTypes}^{D,C_i}, \pi]$. On the other hand, if the component C_i has parameters, then the aggregate local variable C_i has sort $\texttt{Map[}\mathit{types}^{D,C_i}, \mathit{Locals}[A_i, \mathit{actualTypes}^{D,C_i}, \pi]\texttt{]}$, where types^{D,C_i} is the sequence of types of the variables in vars^{D,C_i}.

We define $\mathit{localVars}^{D,\pi}$ to be the sequence of the per-component local variables $C_1, \ldots, C_n$. If a transition π has no local variables in component C_i or if π is not a transition in component C_i, we omit C_i from $\mathit{localVars}^{D,\pi}$. We also define the sort $\mathit{Locals}[D, \mathit{types}^D, \pi]$ to be a tuple sort with selection operators that are named, typed, and have values in accordance with the variables in $\mathit{localVars}^{D,\pi}$.

7.7 Expanding input transitions

Composition combines the transitions for identical input actions in different component automata into a single atomic transition. An input transition is defined for an action π exactly for those values of $\mathit{vars}^{D,\pi}$ that satisfy the signature **where** predicate $P_{in}^{D,\pi}$. Figure 7.3 shows the general form for the definition of a combined input transition based on this observation. Below, we discuss the definitions of the **where**, **eff**, and **ensuring** clauses which appear in that figure.

Each of these clauses also appears as part of the expanded transitions for output and internal transitions, so we name them $P_{in,t_1}^{D,\pi}$, $\mathit{Prog}_{in}^{D,\pi}$, and $\mathit{ensuring}_{in}^{D,\pi}$, respectively, and include them in the figures for the output and internal transitions only by reference. In those transitions, $P_{in,t_1}^{D,\pi}$ refers only to the predicate explicitly appearing in Figure 7.3. That is, without the implicitly conjoined signature predicate $P_{in}^{D,\pi}$.

transitions

  ...

  input $\pi(\mathit{vars}^{D,\pi};$ **local** $\mathit{localVars}^{D,\pi})$ **where** $\bigwedge_{1\le i\le n} P_{in,t_1}^{D,C_i,\pi}$

    eff

      ...

      % When vars^{D,C_i} is empty

      if $P^{D,C_i} \wedge P_{in}^{C_i,\pi}$ **then** $\mathit{Prog}_{in}^{C_i,\pi}$ **fi**;

      ...

      % When vars^{D,C_j} is not empty

      for vars^{D,C_j} **where** $P^{D,C_j} \wedge P_{in}^{C_j,\pi}$ **do**

        $\mathit{Prog}_{in}^{C_j,\pi}$

      od;

      ...

    ensuring $\bigwedge_{1\le i\le n} \mathit{ensuring}_{in}^{D,C_i,\pi}$

Figure 7.3: General form of an input transition in the expansion of a composite automaton

where clause

Since there is only one input transition for the action π in *DExpanded*, the expanded transition **where** clause trivially satisfies semantic proof obligation 4.5 and its only functional role is to define the initial values of the local variables $localVars^{D,\pi}$ that correspond to a given sequence of action parameters $vars^{D,\pi}$. While the signature **where** predicate $P_{in}^{D,\pi}$ need only establish that there exists *some* instance of *some* component that contributes an input action $\pi(vars^{D,\pi})$, the transition **where** predicate must define local variable initial values for *each* contributing instance of *all* contributing components.

We define the input transition **where** clause $P_{in,t_1}^{D,\pi}$ by constructing subformulas $P_{in,t_1}^{D,C_i,\pi}$. Each such subformula constrains the initial value of one local variable C_i of contributing component C_i. The **where** clause shown in Figure 7.3 is then just the conjunction of these predicates $P_{in,t_1}^{D,C_i,\pi}$ for all components.

The subformula $P_{in,t_1}^{D,C_i,\pi}$ is the implication that for each instance of the component that contributes to the transition, the local variable C_i satisfies the proper initial constraints. The initial value of local variable C_i in $localVars^{D,\pi}$ is properly constrained when it satisfies the **where** clause $P_{in,t_1}^{C_i,\pi}$ for the input transition definition of π in component C_i (for the given values of the component parameters $vars^{D,C_i}$ and action parameters $vars^{D,\pi}$). Thus, the consequent of the subformula implication is $P_{in,t_1}^{C_i,\pi}$.

When the component is parameterized, the local variable C_i is a map and each entry $C_i[vars^{D,C_i}]$ in that map corresponds to the aggregate local variable for one instance of the component. In this case, the initial values for entries corresponding to all contributing instances must initialized. An instance of component C_i contributes to the transition $\pi(vars^{D,\pi})$ when component parameters $vars^{D,C_i}$ satisfy both the component **where** clause P^{D,C_i} and the signature **where** clause $P_{in}^{C_i,\pi}$ in that component (for the given values of the action parameters $vars^{D,\pi}$). Thus, the antecedent of the implication is the conjunction of these two predicates. To cover all instances, the implication is universally quantified over all values of the component parameters $vars^{D,C_i}$. Hence, we define

$$P_{in,t_1}^{D,C_i,\pi} ::= \forall\, vars^{D,C_i} \left(\left(P^{D,C_i} \wedge P_{in}^{C_i,\pi}\right) \Rightarrow P_{in,t_1}^{C_i,\pi}\right).$$

Since component C_i satisfies the semantic proof obligation 4.4, there must exists a value for local variable C_i that satisfies the above consequent whenever the antecedent holds. Thus, the implication is always true when read with the existential quantifier over the local variables $localVars^{D,\pi}$ that is implicit in the transition header. Thus, *DExpanded* also (trivially) satisfies semantic proof obligation 4.4 for input transitions, since whenever the input action $\pi(vars^{D,\pi})$ is defined in the signature of *DExpanded*, the input transition $\pi(vars^{D,\pi})$ is also defined.

Notice that for each distinct value of $vars^{D,C_i}$ the predicate $P_{in,t_1}^{C_i,\pi}$ mentions a distinct local variable C_i or $C_i[vars^{D,C_i}]$ in $localVars^{D,\pi}$. So, the truth values of instantiations of the the implication are independent even though there is only one existential instantiation of the local variables $localVars^{D,\pi}$.

However, the fact that the implication is always true does not mean that it is equivalent to omit the expanded transition **where** clause. It is a consequence of the expanded signature **where** clause $P_{in}^{D,\pi}$ that some value of $vars^{D,C_i}$ satisfies the above implication antecedent. In that case the **where** clause asserts that the initial value of the relevant local variable must satisfy the contributing component transition **where** predicate $P_{in,t_1}^{C_i,\pi}$.

When the component is not parameterized, $P_{in,t_1}^{D,C_i,\pi}$ reduces to $P_{in,t_1}^{C_i,\pi}$. To see this, first, note that the universal quantifier simplifies away for lack of variables to quantify. Second, note that P^{D,C_i} and $P_{in}^{C_i,\pi}$ are true whenever $P_{in}^{D,\pi}$ is true. So the implication reduces to just the consequent.

Since the only functional role of the **where** clause is to define the initial values of the local variables $localVars^{D,\pi}$, when there are no local variables or when no local variable appears in any $P_{in}^{C_i,\pi}$, the **where** clause can be omitted altogether.

eff clause

The **eff** clause performs the effects of all input transitions of each contributing instance of all contributing components. It contains a conditional statement for each unparameterized component C_i of D and a loop statement for each parameterized component C_i of D.

The predicate in the conditional statement for an unparameterized component C_i (when implicitly conjoined with the **where** clause for the entire transition and **where** clause for the action in the automaton signature) is true if C_i contributes an input transition for π to the composite automaton D. In that case, the body of the conditional statement executes the program in the **eff** clause in the transition definition for π in C_i.

The situation is slightly more complicated when the component C_i is parameterized, because the transition must execute the effects of all instances of the component that contribute to the action. Thus, the **eff** clause loops over all the different values of the component parameters $vars^{D,C_i}$ that satisfy the component **where** clause P^{D,C_i} and the signature **where** clause $P_{in}^{C_i,\pi}$ in that component to execute the program in the **eff** clause in the transition for π in that instance of component C_i. Notice that each instance of a contributing component C_i (corresponding to one iteration of the loop for C_i) manipulates a distinct tuple of local variables $C_i[vars^{D,C_i}]$.[22]

If only one unparameterized component C_i contributes to the input transition definition, the conditional statement for that component may be replaced by the **eff** clause in the transition definition for π in C_i itself because the guard is implied by $P_{in}^{D,\pi}$.

ensuring clause

The **ensuring** predicate must be true if and only if the **ensuring** predicate from each contributing instance of all contributing components is true. That is, given the parameters $vars^{D,\pi}$, for each sequence of values of component parameters $vars^{D,C_i}$ of each component C_i that satisfies both the component **where** clause P^{D,C_i} and the signature **where** clause $P_{in}^{C_i,\pi}$ in that component, the value of the local variable C_i in $localVars^{D,\pi}$ must also satisfy the **ensuring** clause $ensuring_{in}^{C_i,\pi}$ for the input transition definition of π in C_i. Thus, we define the predicate $ensuring_{in}^{D,C_i,\pi}$ analogously to the the predicate $P_{in,t_1}^{D,C_i,\pi}$:

$$ensuring_{in}^{D,C_i,\pi} ::= \forall\, vars^{D,C_i} \left(\left(P^{D,C_i} \wedge P_{in}^{C_i,\pi} \right) \Rightarrow ensuring_{in,t_1}^{C_i,\pi} \right).$$

[22] Currently, IOA syntax permits only a single single loop variable in **for** statements. However, if V is a sequence of variables $v_1, v_2, v_3, \ldots$, then it is simple to rewrite multi-variable loops such as the ones used in Figure 7.3

for V **where** p **do** g **od**

as nested single-variable loops using the inductive step

for v_1 **where** $\exists V' p$ **do**
    for V' **where** p **do** g **od**
od

where is the variable sequence $V' = v_2, v_3 \ldots$, p is a predicate and g is a program.

7.8 Expanding output transitions

We build up to the general form of expanded output transitions by first considering three specialized cases. The simplest case we consider is an output transition that appears in exactly one unparameterized component and in no component as an input transition. Second, we consider the expansion of an output transition when that sole contributing component is parameterized. Third, we extend our definitions to apply output transitions contributed by multiple components. Finally, the fully general expansion of output transitions covers the case where output actions and input actions share a name.

Output-only transition contributed by a single unparameterized component

We begin by considering the simplest case of an output transition $\pi(\mathit{vars}^{D,\pi})$ that appears in exactly one unparameterized component C_i and in no component as an input transition. That is, there is no component C_j, whose signature contains an input action $\pi(\mathit{vars}^{D,\pi})$. In this case, the expanded output transition does not need to be performed atomically with any input transition.

As there is only one transition contributing to the expansion, there is only one transition for the action $\pi(\mathit{vars}^{D,\pi})$ in *DExpanded*. Thus, the expanded transition **where** clause trivially satisfies semantic proof obligation 4.5 and its only functional role is to define the initial values of the local variable C_i that corresponds to a given sequence of parameters $\mathit{vars}^{D,\pi}$. In this case, simply reusing the component transition **where** clause $P^{C_i,\pi}_{out,t_1}$ as the expanded transition **where** clause gives the correct definition. In fact, the only difference between the expanded transition and the component transition in this simplest case is the way locals variables are declared in transition header. The aggregate local variable of the component transition becomes the sole local variable of the expanded transition. The resulting form is show in Figure 7.4.

transitions

...

output $\pi($**local** $\mathit{vars}^{D,C_i},\ C_i{:}\mathit{Locals}[C_i, \mathit{actualTypes}^{D,C_i}, \pi])$

where $P^{C_i,\pi}_{out,t_1}$

pre $\mathit{Pre}^{C_i,\pi}_{out}$

eff $\mathit{Prog}^{C_i,\pi}_{out}$

ensuring $\mathit{ensuring}^{C_i,\pi}_{out}$

Figure 7.4: Expanded transition for an output action with no matching input actions, derived uniquely from a component C_i with no parameters.

Output-only transition contributed by a single parameterized component

When the component C_i has parameters the expansion is slightly more complicated. As in the previous case, no like-named input transitions exist in any component and, therefore, the expanded output transition does not need to be performed atomically with any input transition. Also like the previous case, there is only one transition definition for $\pi(\mathit{vars}^{D,\pi})$ in the expanded automaton, so the transition **where** clause trivially satisfies semantic proof obligation 4.5 and its only functional role is to define the initial values of the local variables. Unlike the previous case, the state and

transitions

...

output $\pi(vars^{D,\pi};$ **local** $vars^{D,C_i},\ C_i$:Map$[types^{D,C_i}, Locals[C_i, actualTypes^{D,C_i}, \pi]])$

where $P^{D,C_i} \ \wedge\ P^{C_i,\pi}_{out} \ \wedge\ P^{C_i,\pi}_{out,t_1}$

pre $Pre^{C_i,\pi}_{out}$

eff $Prog^{C_i,\pi}_{out}$

ensuring $ensuring^{C_i,\pi}_{out}$

Figure 7.5: Expanded transition for an output action with no matching input actions, derived uniquely from a parameterized component C_i.

local variables C_i are maps rather than simple tuples and the contributing component parameters $vars^{D,C_i}$ are introduced as local variables.

The initial values of $vars^{D,C_i}$ need to be the correct indices for the relevant entry in the state and local variable maps. That is, $C_i[vars^{D,C_i}]$ should evaluate to the tuple derived from the aggregate variable of the contributing instance of the component. Note, the semantic proof obligation 7.2 requires that at most one instance of a component may contribute an output action $\pi(vars^{D,\pi})$. In fact, proof obligation 7.2 provides the formula for selecting the correct indices. The component parameters of the sole contributing instance uniquely satisfy both the component **where** clause P^{D,C_i} and the signature **where** clause $P^{C_i,\pi}_{out}$. Thus, these two predicates appear as conjuncts in the **where** clause.

Since at most one instance of component C_i contributes to the expanded transition, at most one entry in each of state and local variable maps C_i, corresponding to the aggregate variable of the contributing instance of the component, has any relevance to the transition. The other entries are completely ignored.[23] The initial values for that entry $C_i[vars^{D,C_i}]$ are those that satisfy the component transition **where** clause $P^{C_i,\pi}_{out,t_1}$. Thus, this predicate forms the last conjunct in the expanded **where** clause.

The fact that at most one instance of component C_i contributes to the expanded transition also means the expanded definition for the transition of an output action π need not use a **for** statement, as does the expanded definition for the transition of an input action. Instead, the expanded definition simply reuses the **eff** clause of the sole contributing component transition. Similarly, the **pre** and **ensuring** clauses of the expanded transition are the same as those of the sole contributing component transition, as shown in Figure 7.5.

Output-only transitions contributed by multiple components

When an output action name appears in several components, it would be valid for the expanded composite automaton to include a separate output transition derived from each contributing component transition using the above definitions. Unfortunately, as we see below, this approach yields a code-size explosion multiplicative in the number of like-named input and output transitions. To avoid this code explosion, we define the expanded composite automaton to combine all like-named

[23] In this special case, the references to local variable maps (rather than simple tuples) introduced by substitution $\sigma_{i,\pi}$ rule 9 in Section 6.3 are actually an unnecessary complication. However, they are required in the more general cases discussed below.

transitions

...

output $\pi(vars^{D,\pi}$; **local** $vars^{D,C_1}, \ldots, vars^{D,C_n},\ localVars^{D,\pi})$

where $\bigvee_{1 \le i \le n} \left(P^{D,C_i} \ \wedge \ P^{C_i,\pi}_{out} \ \wedge \ P^{C_i,\pi}_{out,t_1} \right)$

pre $\bigvee_{1 \le i \le n} \left(P^{D,C_i} \ \wedge \ P^{C_i,\pi}_{out} \ \wedge \ Pre^{C_i,\pi}_{out} \right)$

eff

if ...

elseif $P^{D,C_i} \ \wedge \ P^{C_i,\pi}_{out}$ **then** $Prog^{C_i,\pi}_{out}$

elseif ...

fi

ensuring $\bigwedge_{1 \le i \le n} \left(P^{D,C_i} \ \wedge \ P^{C_i,\pi}_{out,t_1} \ \ \Rightarrow \ ensuring^{A,\pi}_{out} \right)$

Figure 7.6: Expanded transition for an output action with no matching input actions, contributed by several components

output transitions into a single output transition, as shown in Figure 7.6. An additional advantage of combining all like-named output transitions is that, once again, the expanded transition **where** clause trivially satisfies semantic proof obligation 4.5 and its only functional role is to define the initial values of the local variables.

In the expansion, we declare as local variables the parameters of each (contributing) component and the local variable C_i from each (contributing) component. As in the previous case, the semantic proof obligations for output actions given in Section 7.4 provide the key to defining the **where** clause. Obligation 7.1 requires that for any value of parameters $vars^{D,\pi}$, at most one disjunct of

$$\bigvee_{1 \le i \le n} P^{D,C_i,\pi}_{out} = \bigvee_{1 \le i \le n} \exists vars^{D,C_i} \ (P^{D,C_i} \ \wedge \ P^{C_i,\pi}_{out})$$

can be true. That is, at most one component may contribute an output transition $\pi(vars^{D,\pi})$. Since, all the component parameters $vars^{D,C_i}$ appear as local variables in the expanded transition header, these variables are implicitly existentially quantified in the **where** clause. Therefore, in the expanded transition, the above obligation can be expressed simply as

$$\bigvee_{1 \le i \le n} (P^{D,C_i} \ \wedge \ P^{C_i,\pi}_{out}).$$

Similarly, obligation 7.2 requires that at most one set of values for the component parameters $vars^{D,C_i}$ of that contributing component C_i satisfies the conjunction

$$P^{D,C_i} \ \wedge \ P^{C_i,\pi}_{out}.$$

That is, at most one instance of that component may contribute an output transition $\pi(vars^{D,\pi})$. Notice that this conjunction appears exactly in the previous obligation. In fact, we use the conjunction of the component **where** clause P^{D,C_i} of the contributing component and the signature **where** clause $P^{C_i,\pi}_{out}$ as a "guarding conjunction" for selecting the contributing instance of the contributing component throughout the expanded output transition.

In the **where** clause the guarding conjunction is paired with the corresponding component transition **where** clause and that triple conjunct is disjoined over all the components. Doing so has the effect that the initial values of the relevant local variable C_i (or its relevant map entry $C_i[vars^{D,C_i}]$) satisfies the component transition **where** clause whenever C_i is the contributing component.

Notice, it is a consequence of the expanded signature **where** clause $P^{D,\pi}_{out}$ that some value of $vars^{D,C_i}$ satisfies the guarding conjunction. Furthermore, since component C_i satisfies the semantic proof obligation 4.4, there must exists a value for local variable C_i that satisfies the consequent whenever the guarding conjunction is true. Therefore, whenever the output action $\pi(vars^{D,\pi})$ is defined in the signature of *DExpanded*, the output transition $\pi(vars^{D,\pi})$ is also defined. Thus, *DExpanded* also satisfies semantic proof obligation 4.4 for output transitions.

In the precondition, the guarding conjunction is paired with the corresponding component precondition and that triple conjunct is disjoined over all the components. Thus, the expanded transition is enabled when there is a component for which all three of the transition precondition, the transition **where** clause, and the component **where** clause are true for the given parameters and initial local variable values. Checking the conjunction of all three predicates avoids enabling the transition when the **where** clause is satisfied by the transition from one component while the **pre** clause is satisfied by the transition of another component.

In the **eff** clause, the guarding conjunction selects the conditional branch containing the effects of the single contributing output transition that is defined for the given parameters. Similarly, the **ensuring** clause of the contributing output transition must be satisfied.

Output transitions subsuming input transitions (general case)

When both input and output transitions are defined and (the output transition is) enabled, the output transition subsumes the input transitions. That is, the input actions execute atomically with the output action. Just as we cannot statically decide that two input actions will never be simultaneously executed, we cannot, in general, statically decide that an input transition can never be subsumed by a like-named output transition. Therefore, each expanded output transition must include the effects of *all* like-named input transitions (appropriately guarded). (It is this fact that would cause the code-size explosion mentioned in the previous section were we to include a separate output transition derived from each contributing component transition.) Figure 7.7 shows the general form for expanding output transitions of composite automata.

In the cases where the output transition subsumes one or more input transition, the local variables from the instance(s) of the component(s) contributing the input transition(s) must be initialized by the expanded transition **where** clause. On the other hand, the **where** clause must still always be satisfiable when an output action is defined. As we argue in Section 7.7, the expanded input transition **where** predicate $P^{D,\pi}_{in,t_1}$ does exactly these two things. First, it requires the local variables derived from contributing input transitions to satisfy the **where** clauses of those transitions. Second, $P^{D,\pi}_{in,t_1}$ is satisfiable by some choice of values for $localVars^{D,\pi}$. Thus, we simply conjoin $P^{D,\pi}_{in,t_1}$ to the **where** clause developed in the previous case.

The **eff** clause selects the effects of the single contributing output transition that is defined for the given parameters and then performs all the effects of the subsumed input transitions by executing $Prog^{D,\pi}_{in}$. Each effect in $Prog^{D,\pi}_{in}$ is already guarded so as to occur only when the source transition contributes. Therefore, we simply append $Prog^{D,\pi}_{in}$ to the **eff** clause from the previous case. Similarly, the **ensuring** clause $ensuring^{D,\pi}_{in}$ can also be simply conjoined with the the **ensuring**

transitions

...

output $\pi(vars^{D,\pi};$ **local** $vars^{D,C_1}, \ldots, vars^{D,C_n},\ localVars^{D,\pi})$

where $\bigvee_{1 \le i \le n} \left(P^{D,C_i} \wedge P^{C_i,\pi}_{out} \wedge P^{C_i,\pi}_{out,t_1} \right) \wedge P^{D,\pi}_{in,t_1}$

pre $\bigvee_{1 \le i \le n} \left(P^{D,C_i} \wedge P^{C_i,\pi}_{out} \wedge Pre^{C_i,\pi}_{out} \right)$

eff

if ...

elseif $P^{D,C_i} \wedge P^{C_i,\pi}_{out}$ **then** $Prog^{C_i,\pi}_{out}$

elseif ...

fi;

$Prog^{D,\pi}_{in}$

ensuring $\bigwedge_{1 \le i \le n} \left(P^{D,C_i} \wedge P^{C_i,\pi}_{out} \Rightarrow ensuring^{A,\pi}_{out} \right) \wedge ensuring^{D,\pi}_{in}$

Figure 7.7: General form of an output transition in the expansion of a composite automaton

clause from the previous case.

Note that, $Prog^{D,\pi}_{in}$ may, in fact, amount to a no-op in all executions. However, in general, this cannot be statically decided. Also note that the order of execution of the subsumed input transitions with respect to each other or to the enabled output transition does not matter. The semantic checks require that each conditional branch or **for** body executed in either the subsumed input transition or the remainder of the clause must be derived from distinct automata. These effects can alter only the value of state, local, or **choose** variables derived from the automaton contributing that effect. Furthermore, the effects can depend only on those same set of state, local, and **choose** variables or on the parameters of the transition. No effect can change a parameter value.

We define $Prog^{D,\pi}_{out}$ to be the program in the **eff** clause that combines the effects of output transitions and subsumed input transitions. Similarly, we define $ensuring^{D,\pi}_{out}$ to be the predicate that appears in the **ensuring** clause.

7.9 Expanding internal transitions

The basic form of expanded internal transitions is analogous to that of output actions. The most significant difference is that the internal transition expansion must account for output actions that are (potentially) hidden. So before we consider the general expansion for internal transitions, we build on the discussion of the expansion of output transitions above to consider the the simpler case of expanding transitions for internal actions when there are no **hidden** clauses for those actions. We then discuss how to generalize this construction to account for hidden output transitions.

Internal-only transitions

The expanded form of the transition for an internal action when there is no **hidden** clause for that action follows a pattern similar to that of output transitions when there are no like-named input

transitions

$\ldots$

internal $\pi(vars^{D,\pi}$; **local** $vars^{D,C_1}, \ldots, vars^{D,C_n},\ localVars^{D,\pi})$

where $\bigvee_{1 \le i \le n} \left(P^{D,C_i} \ \wedge\ P_{int}^{C_i,\pi} \ \wedge\ P_{int,t_1}^{C_i,\pi} \right)$

pre

$\bigvee_{1 \le i \le n} \left(P^{D,C_i} \ \wedge\ P_{int}^{C_i,\pi} \ \wedge\ Pre_{int}^{C_i,\pi} \right)$

eff

if $\ldots$

elseif $P^{D,C_i} \ \wedge\ P_{int}^{C_i,\pi} \ \wedge$ **then** $Prog_{int}^{C_i,\pi}$

elseif $\ldots$

fi

ensuring $\bigwedge_{1 \le i \le n} \left(P^{D,C_i} \ \wedge\ P_{int,t_1}^{C_i,\pi} \ \Rightarrow\ ensuring_{int}^{A,\pi} \right)$

Figure 7.8: Expanded transition for an internal action with no matching **hidden** clause

transitions. In that expansion, shown in Figure 7.8, we introduce local variables for the parameters of each contributing automaton as well as all the local variables from all the contributing transitions. Following reasoning analogous to the output case, we use the conjunction of the component **where** clause P^{D,C_i} of the contributing component and the signature **where** clause $P_{int}^{C_i,\pi}$ as the guarding conjunction for selecting the contributing instance of the contributing component throughout the expanded internal transition.

In the **where** clause, the guarding conjunction is paired with the component **where** clause for the contributing transition $P_{int,t_1}^{C_i,\pi}$ to initialize the local variable values. In the precondition, the guarding conjunction is paired with the **pre** predicate of the contributing transition. In the **eff** clause, the guarding conjunction selects the conditional branch containing the effects of the single contributing transition that is defined for the given parameters. In, the **ensuring** clause, the contributing transition **ensuring** clause must be satisfied when the guarding conjunction holds.

Internal transitions with hiding (general case)

The most important difference between the expansion for internal transitions and that for output transitions is that the internal transition expansions must account for output actions that are (potentially) hidden. We cannot, in general, statically decide whether the **hidden** predicate $H^{D,\pi}$ covers the output signature predicate $P_{out}^{D,\pi}$. Nor can we, in general, statically decide whether $H^{D,\pi}$ covers the **where** clause for any contributing transition $P_{out,t_1}^{C_i,\pi}$. Thus, each transition for each action $\pi(vars^{D,\pi})$ mentioned by a **hidden** clause must be incorporated into the expanded composite automaton *twice*, once in an output transition and once in an internal transition.

One way to do this, would be to include two internal transitions for each transition $\pi(vars^{D,\pi})$. The first transition would be derived as in the previous section, ignoring any hidden output actions. The second transition would be a second copy of the expanded output transition $\pi(vars^{D,\pi})$. This transition would be identical to the general case output transition expansion except it would be labeled internal.

An alternative expansion is shown in Figure 7.9. This expansion follows the pattern of including just one transition of each kind. An advantage of having just one transition is that the expanded transition **where** clause trivially satisfies semantic proof obligation 4.5 and its only functional role is to define the initial values of the local variables.

Proof obligations 7.3 and 7.4 imply that, over all components, at most one of the conjunctions $P^{D,C_i} \wedge P^{C_i,\pi}_{int}$ and $P^{D,C_i} \wedge P^{C_i,\pi}_{out}$ can be true. So these conjunctions are used as the guarding conjunctions for the expanded transition. The former guards elements derived from internal component transitions. The latter guards elements derived from output component transitions.

In the **where** clause, each guarding conjunction is paired with the component **where** clause for the contributing transition $P^{C_i,\pi}_{kind,t_1}$ of matching kind to initialize the local variable values. Since a hidden output transition might also subsume a like-named input action, the **where** predicate also asserts $P^{D,\pi}_{in}$.[24] In the precondition, the guarding conjunction selects the appropriate component transition precondition $Pre^{C_i,\pi}_{int}$ or $Pre^{C_i,\pi}_{out}$ to satisfy. These latter disjuncts are abbreviated by referencing the expanded output **pre** predicate $Pre^{D,\pi}_{out}$. The **eff** clause selects the effects of the single contributing internal or output transition that is defined for the given parameters and then performs all the effects of the subsumed input transitions. The conditional selecting the effects of an internal action is shown in the figure. Effects derived from hidden output and hidden subsumed inputs are executed in the appended program $Prog^{D,\pi}_{out}$. Similarly, the **ensuring** clause from the previous case can be simply conjoined with expanded output transition **ensuring** clause $ensuring^{D,\pi}_{out}$

Notice, it is a consequence of the expanded signature **where** clause $P^{D,\pi}_{int}$ that some value of $vars^{D,C_i}$ satisfies one of the guarding conjunctions. Furthermore, since component C_i satisfies the semantic proof obligation 4.4, there must exists a value for local variable C_i that satisfies the consequent whenever a guarding conjunction is true. Therefore, whenever the internal action $\pi(vars^{D,\pi})$ is defined in the signature of $DExpanded$, the internal transition $\pi(vars^{D,\pi})$ is also defined. Thus, $DExpanded$ also satisfies semantic proof obligation 4.4 for internal transitions.

[24] We cannot simply conjoin $P^{D,\pi}_{out}$ to the transition **where** clause because $P^{D,\pi}_{in}$ would not distribute correctly.

transitions

$\ldots$

internal $\pi(\mathit{vars}^{D,\pi};$ **local** $\mathit{vars}^{D,C_1}, \ldots, \mathit{vars}^{D,C_n},\ \mathit{localVars}^{D,\pi})$

where $\bigvee_{1 \le i \le n} \left(\left(P^{D,C_i} \ \wedge \ P^{C_i,\pi}_{int} \ \wedge \ P^{C_i,\pi}_{int,t_1} \right) \ \vee \ \left(P^{D,C_i} \ \wedge \ P^{C_i,\pi}_{out} \ \wedge \ P^{C_i,\pi}_{out,t_1} \right) \right) \ \wedge \ P^{D,\pi}_{in,t_1}$

pre

$\bigvee_{1 \le i \le n} \left(P^{D,C_i} \ \wedge \ P^{C_i,\pi}_{int} \ \wedge \ \mathit{Pre}^{C_i,\pi}_{int} \right) \ \vee \ \mathit{Pre}^{D,\pi}_{out}$

eff

if $\ldots$

elseif $P^{D,C_i} \ \wedge \ P^{C_i,\pi}_{int} \ \wedge$ **then** $\mathit{Prog}^{C_i,\pi}_{int}$

elseif $\ldots$

fi;

$\mathit{Prog}^{D,\pi}_{out}$

ensuring $\bigwedge_{1 \le i \le n} \left(P^{D,C_i} \ \wedge \ P^{C_i,\pi}_{int,t_1} \ \Rightarrow \ \mathit{ensuring}^{A,\pi}_{int} \right) \ \wedge \ \mathit{ensuring}^{D,\pi}_{out}$

Figure 7.9: General form of an internal transition in the expansion of a composite automaton

8 Expansion of an example composite automaton

In this section, we detail the expansion the composite automaton introduced in Example 2.4. In this expansion, we apply the techniques described in Section 7 to the composite automaton `Sys` shown in Figure 2.4 and to the canonical versions of its component automata shown in Figures 6.3–6.5. In Section 8.2, we derive the signature of `SysExpanded` in three stages. In Section 8.3, we describe the state of the expanded automaton, including its initial values, and an invariant about the scope of definition for its state variables.

Where convenient, we recapitulate definitions developed in previous sections in summary tables to save the reader (and the authors!) from having to flip back to look up definitions.

8.1 Desugared hidden statement of `Sys`

Following the procedure described in Section 7.2, we eliminate terms other than variable references from the parameters of the **hidden** statement of automaton `Sys` by replacing $params^{\texttt{Sys,send}}_{hide_1} = \langle\texttt{nProcesses, nProcesses+1, x:Int}\rangle$ with $vars^{\texttt{Sys,send}} = \langle\texttt{n1:Int,n2:Int,m:Int}\rangle$, defining σ^{hide}_1 to map `m:Int` to a fresh variable `i:Int`, and rewriting the **where** clause in the **hidden** statement to produce

```
hidden send(n1, n2, m)
   where ∃ i:Int (i = m ∧ n1 = nProcesses ∧ n2 = nProcesses+1)
```

which simplifies to

```
hidden send(n1, n2, m) where n1 = nProcesses ∧ n2 = nProcesses+1
```

Thus, we define $H^{\texttt{Sys,send}}$ to be `n1 = nProcesses ∧ n2 = nProcesses+1`.

8.2 Signature of `SysExpanded`

To expand the signature of composite automaton `Sys` as described in Section 7.3, we first calculate the per-kind, per-action, per-component predicates $P^{\texttt{Sys},C_i,\pi}_{kind}$. Then we combine these by component to form the provisional kind predicates $Prov^{\texttt{Sys},\pi}_{kind}$. Finally, we combine these predicates with the **hidden** statement predicate to derive the signature predicates $P^{\texttt{Sys},\pi}_{in}$, $P^{\texttt{Sys},\pi}_{out}$, and $P^{\texttt{Sys},\pi}_{int}$.

In computing these predicates it is helpful to remember the component predicates and canonical variables of the sample composite automaton `Sys`. Table 8.1 collects the former from Example 2.4. Table 8.2 recalls the latter as they were defined in Example 6.2. The local variables shown are derived from Example 6.2 as described in Section 7.6.

PREDICATE	VALUE
$P^{\texttt{Sys,C}}$	`j = i+1 ∧ 1 ≤ i ∧ i < nProcesses`
$P^{\texttt{Sys,P}}$	`1 ≤ n ∧ n ≤ nProcesses`
$P^{\texttt{Sys,W}}$	true

Table 8.1: Component predicates of the sample composite automaton `Sys`

Actions per component

First, we define predicates for each kind of each action for each component. `Sys` has three components and four action names, each of up to three kinds. Thus, there are thirty-six possible per-kind,

CANONICAL SEQUENCE	VARIABLES
$vars^{\texttt{Sys}}$	`nProcesses:Int`
$vars^{\texttt{C}}$	`n:Int`
$vars^{\texttt{P}}$	`n:Int`
$vars^{\texttt{Sys},\texttt{send}}$	`n1:Int, n2:Int, m:Int`
$vars^{\texttt{Sys},\texttt{receive}}$	`n1:Int, n2:Int, m:Int`
$vars^{\texttt{Sys},\texttt{overflow}}$	`i1:Int, s:Set[Int]`
$vars^{\texttt{Sys},\texttt{found}}$	`i1:Int`
$localVars^{\texttt{Sys},\texttt{overflow}}$	`P:Map[Int, Locals[P, overflow]],` `W:Locals[Watch, Int, overflow]`

Table 8.2: Canonical variables used to expand the sample composite automaton `Sys`

per-action, per-component predicates $P_{kind}^{\texttt{Sys},C_i,\pi}$. Table 8.3 shows the seven of these predicates that are not trivially false. All the existential quantifiers have been eliminated from the predicates shown in the table.

We can simplify such a predicate by dropping existential quantifiers and conjuncts that are superfluous. A quantifier is superfluous if the predicate equates the quantified variable directly with a term not involving a quantified variable. The conjunct that equates the quantified variable to a defining term is also superfluous. The simplification proceeds in four steps:

1. Define a substitution that maps any superfluous existential variables to the corresponding term.
2. Apply the substitution to the predicate.
3. Delete identity conjuncts from the **where** clause.
4. Delete the existential quantifiers for variables that no longer appear in the predicate.

For example, by the definition given in Section 7.3,

$$\begin{aligned} P_{in}^{\texttt{Sys},\texttt{C},\texttt{send}} &::= \exists vars^{\texttt{Sys},\texttt{C}}\ (P^{\texttt{Sys},\texttt{C}} \wedge P_{in}^{\texttt{C},\texttt{send}}) \\ &= \exists\ \texttt{n:Int}\ (\texttt{1} \leq \texttt{n} \wedge \texttt{n} < \texttt{nProcesses} \wedge \texttt{n1} = \texttt{n} \wedge \texttt{n2} = \texttt{n+1}) \end{aligned}$$

We simplify this predicate by defining and applying a substitution that maps `n:Int` to `n1:Int`, delete the resulting identity conjunct, the quantified variable, and the quantifier, resulting in the predicate shown in Table 8.3.

Provisional action kinds

Since no two components of `Sys` share the same kind of any action, it is simple to define the provisional kind predicates $Prov_{kind}^{\texttt{Sys},\pi}$. Seven of the twelve possible predicates are not trivially false. Each of these has exactly one nontrivial disjunct—the corresponding predicate $P_{kind}^{\texttt{Sys},C_i,\pi}$, as shown in Table 8.4

Predicate	value
$P_{in}^{\texttt{Sys,C,send}}$	`(1` $\leq$ `n1` $\wedge$ `n1` $<$ `nProcesses)` $\wedge$ `(n2` $=$ `n1+1)`
$P_{out}^{\texttt{Sys,P,send}}$	`(1` $\leq$ `n1` $\wedge$ `n1` $\leq$ `nProcesses)` $\wedge$ `(n2` $=$ `n1+1)`
$P_{out}^{\texttt{Sys,C,receive}}$	`(1` $\leq$ `n1` $\wedge$ `n1` $<$ `nProcesses)` $\wedge$ `(n2` $=$ `n1+1)`
$P_{in}^{\texttt{Sys,P,receive}}$	`(1` $\leq$ `n2` $\wedge$ `n2` $\leq$ `nProcesses)` $\wedge$ `(n1` $=$ `n2-1)`
$P_{out}^{\texttt{Sys,P,overflow}}$	`1` $\leq$ `i1` $\wedge$ `i1` $\leq$ `nProcesses`
$P_{in}^{\texttt{Sys,W,overflow}}$	`i1` $\in$ `between(1, nProcesses)`
$P_{out}^{\texttt{Sys,W,found}}$	`i1` $\in$ `between(1, nProcesses)`

Table 8.3: Simplified predicates defining contributions to the signature of `Sys`

Predicate	value
$Prov_{in}^{\texttt{Sys,send}}$	$P_{in}^{\texttt{Sys,C,send}}$
$Prov_{out}^{\texttt{Sys,send}}$	$P_{out}^{\texttt{Sys,P,send}}$
$Prov_{out}^{\texttt{Sys,receive}}$	$P_{out}^{\texttt{Sys,C,receive}}$
$Prov_{in}^{\texttt{Sys,receive}}$	$P_{in}^{\texttt{Sys,P,receive}}$
$Prov_{out}^{\texttt{Sys,overflow}}$	$P_{out}^{\texttt{Sys,P,overflow}}$
$Prov_{in}^{\texttt{Sys,overflow}}$	$P_{in}^{\texttt{Sys,W,overflow}}$
$Prov_{out}^{\texttt{Sys,found}}$	$P_{out}^{\texttt{Sys,W,found}}$

Table 8.4: Provisional **where** predicates for the signature of `Sys`

Signature predicates

We now compute the nontrivial signature predicates $P_{in}^{\mathtt{Sys},\pi}$, $P_{out}^{\mathtt{Sys},\pi}$, and $P_{int}^{\mathtt{Sys},\pi}$ for the four action labels `send`, `receive`, `overflow`, and `found` of automaton `SysExpanded`.

Output actions We compute the signature predicate for output action `send`, by applying the formula

$$P_{out}^{\mathtt{Sys,send}} = Prov_{out}^{\mathtt{Sys,P,send}} \wedge \neg H^{\mathtt{Sys,send}}.$$

Using the desugared form of the **hidden** predicate shown in Example 7.2, we find that $P_{out}^{\mathtt{Sys,send}}$ is

```
1 ≤ n1 ∧ n1 ≤ nProcesses ∧ n2 = n1+1
  ∧ ¬(n1 = nProcesses ∧ n2 = nProcesses+1)
```

Computing the predicates for output actions `receive`, `found`, and `overflow` is simple because there is no **hidden** clause applying to them (i.e., $H^{\mathtt{Sys},\pi}$ is false) and the action predicate is, in fact, just the provisional kind predicate, as shown in Figure 8.1.

Input actions We compute the signature predicate for input action `send` by applying the formula

$$P_{in}^{\mathtt{Sys,send}} = Prov_{in}^{\mathtt{Sys,send}} \wedge \neg Prov_{out}^{\mathtt{Sys,send}}.$$

Thus, $P_{in}^{\mathtt{Sys,send}}$ evaluates to

```
1 ≤ n1 ∧ n1 < nProcesses ∧ n2 = n1+1 ∧
    ¬((1 ≤ n1 ∧ n1 ≤ nProcesses) ∧ (n2 = n1+1))
```

The signature predicates for input actions `receive`, and `overflow` are computed similarly and appear in Figure 8.1.

Internal actions In Example 2.4, the component automata have no internal actions. Therefore, the only internal action in `Sys` is the hidden action `send`. Thus, the predicate $P_{int}^{\mathtt{Sys,send}}$ is equivalent to

$$Prov_{out}^{\mathtt{Sys,send}} \wedge H^{\mathtt{Sys,send}},$$

which evaluates to

```
1 ≤ n1 ∧ n1 ≤ nProcesses ∧ n2 = n1+1 ∧ n1=nProcesses ∧ n2=nProcesses+1
```

The complete expanded signature of automaton `Sys` is given in Figure 8.1.

8.3 States and initially predicates of `SysExpanded`

The complete expanded state of automaton `Sys` is given in Figure 8.1. Since each component of the desugared composite automaton has non-type parameters, all three state variables are maps. Three of the **initially** subclauses (and the subsequent invariant) assert the well-formedness requirement that each map is defined only for values of the component parameters on which the component itself is defined. The other three **initially** subclauses assert that the `contents` of each `channel` is initially empty, the `watch` process is looking for values between 1 and `nProcesses` and that each process `P` initially has value 0 and nothing to send. The **type** declaration appearing at the beginning of the figure is the automatically generated sort for the state of the composite automaton.

```
type States[Sys] = tuple of C:Map[Int, States[Channel,Int,Int]],
                            P:Map[Int, States[P]],
                            W:States[Watch,Int]

automaton SysExpanded(nProcesses:Int)
  signature
    output send(n1, n2, m:Int)
             where 1 ≤ n1 ∧ n1 ≤ nProcesses ∧ n2 = n1+1
                      ∧ ¬(n1 = nProcesses ∧ n2 = nProcesses+1),
           receive(n1, n2, m:Int)
             where 1 ≤ n1 ∧ n1 < nProcesses ∧ n2 = n1+1,
           overflow(i1:Int, s:Set[Int]) where 1 ≤ i1 ∧ i1 ≤ nProcesses,
           found(i1:Int) where i1 ∈ between(1, nProcesses)
    input send(n1, n2, m:Int)
             where 1 ≤ n1 ∧ n1 < nProcesses ∧ n2 = n1+1
                     ∧ ¬(1 ≤ n1 ∧ n1 < nProcesses ∧ n2 = n1+1),
          receive(n1, n2, m:Int)
             where 1 ≤ n2 ∧ n2 ≤ nProcesses ∧  n1 = n2-1
                   ∧  ¬(1 ≤ n1 ∧ n1 < nProcesses ∧ n2 = n1+1),
          overflow(i1:Int, s:Set[Int])
             where i1 ∈ between(1, nProcesses)
                     ∧ ¬(1 ≤ i1 ∧ i1 ≤ nProcesses)
    internal send(n1, n2, m:Int)
             where 1 ≤ n1 ∧ n1 ≤ nProcesses ∧ n2 = n1+1
                      ∧ n1 = nProcesses ∧ n2 = nProcesses+1
  states C:Map[Int, States[Channel, Int, Int]],
         P:Map[Int, States[P]],
         W:States[Watch, Int]
    initially
        ∀ n:Int ((1 ≤ n ∧ n < nProcesses) ⇒ C[n].contents = {})
      ∧ ∀ n:Int ((1 ≤ n ∧ n < nProcesses) ⇔ defined(C, n))
      ∧ ∀ n:Int ((1 ≤ n ∧ n ≤ nProcesses) ⇒ P[n].val = 0 ∧ P[n].toSend = {})
      ∧ ∀ n:Int ((1 ≤ n ∧ n ≤ nProcesses) ⇔ defined(P, n))
      ∧ W.seen = constant(false)
  ...
invariant of SysExpanded:
  ∀ n:Int (1 ≤ n ∧ n < nProcesses ⇔ defined(C[n]));
  ∀ n:Int (1 ≤ n ∧ n ≤ nProcesses ⇔ defined(P[n]))
```

Figure 8.1: Expanded signature and states of the sample composite automaton Sys

8.4 Input Transition Definitions of `SysExpanded`

We compute the input transitions of `SysExpanded` by following the pattern of Figure 7.3 for each of the input actions in its signature (`receive`, `send`, and `overflow`) and simplifying. Figure 8.2 shows the three resulting forms.

In that figure, each input transition is formed from only a single contributing component. Thus, the conjunctions in the **where** over the contributing components in Figure 7.3 each resolves to a single term. Furthermore, we omit the **where** clauses for the `receive` and `send` transitions because the transition definitions have no local variables. In each of the three transitions, we omit the **ensuring** predicate altogether because the sole contributing predicate for each transition ($ensuring_{in}^{\mathtt{P,receive}}$, $ensuring_{in}^{\mathtt{C,send}}$, and $ensuring_{in}^{\mathtt{W,overflow}}$) is trivially true. The **eff** clause of each transition resolves to a single **for** loop or conditional. In the `overflow` transition, the conditional is replaced by its body because there is only a single contributing transition.

Figure 8.3 shows the final text of the expanded input transitions. In that figure, we omit the local variable `P:Map[Int, Locals[P, overflow]]` from the `overflow` transition because it does not appear in the transition precondition or effects. The **where** clause predicate $P_{in,t_1}^{\mathtt{Sys,W,overflow}}$ reduces to the implication shown in Table 8.5 because $vars^{\mathtt{Sys,W}}$ is empty and $P^{\mathtt{Sys,W}}$ is trivially true.

The **for** loops in the `receive` and `send` transitions have been eliminated by the following simplification. Filling in the specified variables from Tables 8.2, predicates from Tables 8.1 and 8.5 and statements from Example 6.2 in the `receive` transition **for** loop yields the loop

```
for n:Int where (1 ≤ n ∧ n ≤ nProcesses ∧ n1 = n-1 ∧ n2 = n) do
  if P[n].val = 0 then P[n].val := m
  elseif m < P[n2].val then
    P[n].toSend := insert(P[n].val, P[n].toSend);
    P[n].val := m
  elseif P[n].val < m then
    P[n].toSend := insert(m, P[n].toSend)
  fi
od.
```

Since the last conjunct of the loop **where** clause limits the loop variable to a single value, the transition parameter `n2`, we can eliminate the loop altogether. Thus, in Figure 8.3, we replace the loop with its body after applying to the body a substitution that maps the loop variable `n` to its value `n2`. Similarly, the **for** loop in the `send` transition is eliminated using a substitution that maps its loop variable `n` to the transition parameter `n1`.

8.5 Output Transition Definitions of `SysExpanded`

We compute the output transitions of `SysExpanded` by following the pattern of Figure 7.7 for each of the output actions in its signature (`receive`, `send`, `overflow`, and `found`) and simplifying. Figure 8.4 shows the four resulting forms.

Notice that only one component contributes an output transition to each expanded output transition. Therefore, only syntactic elements from the sole contributing component and the corresponding expanded input action appear in each transition. Each local variable list contains of the component variables for that contributing component. Since, $localVars^{\mathtt{Sys,receive}}$, $localVars^{\mathtt{Sys,send}}$, and $localVars^{\mathtt{Sys,found}}$ are empty, they are omitted from their respective transitions. Since component `W` is unparameterized, the `found` transition has no local variables at all.

The **where** clause of each transition resolves to a single term rather than being a disjunction over the contributing components. Furthermore, we omit the **where** clauses for the `receive`, `send`,

PREDICATE	VALUE
$P_{in}^{\texttt{P,receive}}$	`n1 = n-1` $\wedge$ `n2 = n`
$P_{in}^{\texttt{C,send}}$	`n1 = n` $\wedge$ `n2 = n+1`
$P_{in}^{\texttt{W,overflow}}$	`i1` $\in$ `between(1, nProcesses)`
$P_{in,t_1}^{\texttt{W,overflow}}$	`s = W.s2` $\cup$ `{i1}` $\vee$ $\neg$`(i1` $\in$ `s)`
$P_{in,t_1}^{\texttt{Sys,W,overflow}}$	`i1` $\in$ `between(1, nProcesses)` $\Rightarrow$ `(s = W.s2` $\cup$ `{i1}` $\vee$ $\neg$`(i1` $\in$ `s))`

Table 8.5: Nontrivial predicates used in expanding input transition definitions of the sample composite automaton `Sys` derived from Figures 6.3, 6.4 and 6.5

input `receive`($vars^{\texttt{Sys,receive}}$)
  eff for $vars^{\texttt{Sys,P}}$ **where** $P^{\texttt{Sys,P}} \wedge P_{in}^{\texttt{P,receive}}$ **do** $Prog_{in}^{\texttt{P,receive}}$**od**

input `send`($vars^{\texttt{Sys,send}}$)
  eff for $vars^{\texttt{Sys,C}}$ **where** $P^{\texttt{Sys,C}} \wedge P_{in}^{\texttt{C,send}}$ **do** $Prog_{in}^{\texttt{C,send}}$ **od**

input `overflow`($vars^{\texttt{Sys,overflow}}$; **local** $localVars^{\texttt{Sys,overflow}}$) **where** $P_{in,t_1}^{\texttt{Sys,W,overflow}}$
  eff $Prog_{in}^{\texttt{W,overflow}}$

Figure 8.2: Form of input transitions of `SysExpanded`

```
input receive(n1, n2, m)
  eff if P[n2].val = 0 then P[n2].val := m
      elseif m < P[n2].val then
        P[n2].toSend := insert(P[n2].val, P[n2].toSend);
        P[n2].val := m
      elseif P[n2].val < m then
        P[n2].toSend := insert(m, P[n2].toSend)
      fi

input send(n1, n2, m)
  eff C[n1].contents := insert(m, C[n1].contents)

input overflow(i,s; locals W:Locals[W,int,overflow])
  where i1 ∈ between(1, nProcesses) ⇒ (s = W.s2 ∪ {i1} ∨ ¬(i1 ∈ s))
  eff if s = W.s2 ∪ {i1} then W.seen[i1] := true
      elseif ¬(i1 ∈ s) then W.seen[i1] := false
      fi
```

Figure 8.3: Input transition definitions of `SysExpanded`

PREDICATE	VALUE
$P^{\texttt{C,receive}}_{out}$	`n1 = n` $\wedge$ `n2 = n+1`
$P^{\texttt{C,receive}}_{out,t_1}$	`n1 = n` $\wedge$ `n2 = n+1`
$Pre^{\texttt{C,receive}}_{out}$	`m` $\in$ `C[n].contents`
$P^{\texttt{P,send}}_{out}$	`n1 = n` $\wedge$ `n2 = n+1`
$P^{\texttt{P,send}}_{out,t_1}$	`n1 = n` $\wedge$ `n2 = n+1`
$Pre^{\texttt{P,send}}_{out}$	`m` $\in$ `P[n].toSend`
$P^{\texttt{P,overflow}}_{out}$	`i1 = n`
$P^{\texttt{P,overflow}}_{out,t_1}$	`i1 = n`
$Pre^{\texttt{P,overflow}}_{out}$	`s = P[n].toSend` $\wedge$ `n < size(s)` $\wedge$ `P[n].t` $\subseteq$ `s`
$P^{\texttt{W,found}}_{out}$	`i1` $\in$ `between(1, nProcesses)`
$Pre^{\texttt{W,found}}_{out}$	`W.seen[i1]`

Table 8.6: Nontrivial predicates used in expanding output transition definitions of the sample composite automaton `Sys` derived from Figures 6.3, 6.4 and 6.5

and `found` transitions because the transition definitions have no local variables. Similarly, the ensuring clause is only a single conjunction. In each of the four transitions, we omit the **ensuring** predicate altogether because the consequent for each transition ($ensuring^{\texttt{C,receive}}_{out}$, $ensuring^{\texttt{P,send}}_{out}$, $ensuring^{\texttt{P,overflow}}_{out}$, and $ensuring^{\texttt{W,found}}_{out}$) is trivially true. Furthermore, the conditional and guarding conjunction can be omitted from the **eff** clause because only one output contributes. So each effect is just the effect of the contributing output transition followed by the effect of the corresponding expanded input transition. Since the output transition definition for the `found` action in component `W` has no effect and there is no `found` input action, the expanded found transition has no effect either.

Filling in the specified variables from Tables 8.2, predicates from Tables 8.1 and 8.6 and statements from Example 6.2 and Figure 8.3 yields the complete the complete text of the expanded output transitions shown in Figure 8.5. We simplify the transition definitions using two techniques. First, we eliminating unneeded local variables. Second, we use the fact that the signature **where** predicate for an action (e.g., $P^{\texttt{Sys,receive}}_{out}$) is implicitly conjoined to the corresponding transition **where** predicate (e.g., $P^{\texttt{Sys,receive}}_{out,t_1}$) and precondition ($Pre^{\texttt{Sys,receive}}_{out}$) to eliminate redundant assertions in the transition **where** predicate and precondition. The resulting final form of output transitions is shown in Figure 8.6.

To eliminate unneeded local variables, we follow the four step process to eliminate unneeded local variables described in Section 4.2. For example, we note that the **where** clause of the `receive` transiting equates `n` with parameter `n1`. Furthermore, there is no assignment to `n` in the effects of that transition. Thus, the local variable `n` is extraneous. So, we define a substitution that maps the local variable `n` to the parameter `n1` and apply it to the **where**, **pre**, and **eff** clauses. We then delete the resulting identity conjunct from the **where** clause and the declaration of the local variable `n`. Similarly simplifications eliminate the local variable `n` from the `send` and `overflow` transition

output $\texttt{receive}(\mathit{vars}^{\texttt{Sys,receive}};$ **local** $\mathit{vars}^{\texttt{Sys,C}})$
  where $P^{\texttt{Sys,C}} \;\wedge\; P_{\mathit{out}}^{\texttt{C,receive}} \;\wedge\; P_{\mathit{out},t_1}^{\texttt{C,receive}} \;\wedge\; P_{\mathit{in},t_1}^{\texttt{Sys,receive}}$
  pre $P^{\texttt{Sys,C}} \;\wedge\; P_{\mathit{out}}^{\texttt{C,receive}} \;\wedge\; \mathit{Pre}_{\mathit{out}}^{\texttt{C,receive}}$
  eff $\mathit{Prog}_{\mathit{out}}^{\texttt{C,receive}};$
    $\mathit{Prog}_{\mathit{in}}^{\texttt{Sys,receive}}$

output $\texttt{send}(\mathit{vars}^{\texttt{Sys,send}};$ **local** $\mathit{vars}^{\texttt{Sys,P}})$
  where $P^{\texttt{Sys,P}} \;\wedge\; P_{\mathit{out}}^{\texttt{P,send}} \;\wedge\; P_{\mathit{out},t_1}^{\texttt{P,send}} \;\wedge\; P_{\mathit{in},t_1}^{\texttt{Sys,send}}$
  pre $P^{\texttt{Sys,P}} \;\wedge\; P_{\mathit{out}}^{\texttt{P,send}} \;\wedge\; \mathit{Pre}_{\mathit{out}}^{\texttt{P,send}}$
  eff $\mathit{Prog}_{\mathit{out}}^{\texttt{P,send}};$
    $\mathit{Prog}_{\mathit{in}}^{\texttt{Sys,send}}$

output $\texttt{overflow}(\mathit{vars}^{\texttt{Sys,overflow}};$ **local** $\mathit{vars}^{\texttt{Sys,C}}, \mathit{vars}^{\texttt{Sys,P}}, \mathit{localVars}^{\texttt{Sys,overflow}})$
  where $P^{\texttt{Sys,P}} \;\wedge\; P_{\mathit{out}}^{\texttt{P,overflow}} \;\wedge\; P_{\mathit{out},t_1}^{\texttt{P,overflow}} \;\wedge\; P_{\mathit{in},t_1}^{\texttt{Sys,overflow}}$
  pre $P^{\texttt{Sys,P}} \;\wedge\; P_{\mathit{out}}^{\texttt{P,overflow}} \;\wedge\; \mathit{Pre}_{\mathit{out}}^{\texttt{P,overflow}}$
  eff $\mathit{Prog}_{\mathit{out}}^{\texttt{P,overflow}};$
    $\mathit{Prog}_{\mathit{in}}^{\texttt{Sys,overflow}}$

output $\texttt{found}(\mathit{vars}^{\texttt{Sys,found}})$
  pre $P^{\texttt{Sys,W}} \;\wedge\; P_{\mathit{out}}^{\texttt{W,found}} \;\wedge\; \mathit{Pre}_{\mathit{out}}^{\texttt{W,found}}$

Figure 8.4: Form of output transitions of `SysExpanded`

definitions. Since the resulting `receive` and `send` transitions no longer have any local variables, we omit their **where** clauses altogether.

After this simplification, the precondition for the `receive` transition is

pre $1 \leq \texttt{n1} \wedge \texttt{n1} < \texttt{nProcesses} \wedge \texttt{n2} = \texttt{n1+1} \wedge \texttt{m} \in \texttt{C[n1].contents}$

However, the first three conjuncts are also asserted by the the signature **where** clause for the `receive` output action $P_{\mathit{out}}^{\texttt{Sys,receive}}$ and, therefore, are redundant. Similarly simplifications to the **where** and **pre** clauses of the other transitions result in the final text of the expanded output transitions shown in Figure 8.6.

8.6 Internal Transition Definitions of `SysExpanded`

Since no component has any internal transitions, the only internal transitions in `SysExpanded` is the hidden output `send` transitions. In the case where no component contributes an internal transition, the form in Figure 7.9 reduces exactly that in Figure 7.7. That is, the internal `send` transition definition is identical to the output transition definition except for its label. The two actions are distinguished exactly by the assertion or negation of $H^{\texttt{Sys,send}}$ in the signature of `SysExpanded`.

```
output receive(n1, n2, m; local n:Int)
       where 1 ≤ n1 ∧ n1 < nProcesses ∧ n1 = n ∧ n2 = n1+1
  pre 1 ≤ n ∧ n1 < nProcesses ∧ n1 = n ∧ n2 = n+1
      ∧ m ∈ C[n].contents
  eff
    C[n].contents := delete(m, C[n].contents)
    if P[n2].val = 0 then P[n2].val := m
    elseif m < P[n2].val then
      P[n2].toSend := insert(P[n2].val, P[n2].toSend);
      P[n2].val := m
    elseif P[n2].val < m then
      P[n2].toSend := insert(m, P[n2].toSend)
    fi

output send(n1, n2, m; local n:Int)
         where 1 ≤ n ∧ n ≤ nProcesses ∧ n1 = n ∧ n2 = n+1
               ∧ n1 = n ∧ n2 = n+1
   pre 1 ≤ n ∧ n ≤ nProcesses ∧ n1 = n ∧ n2 = n+1 ∧ m ∈ P[n].toSend
   eff
     P[n].toSend := delete(m, P[n].toSend)
     C[n1].contents := insert(m, C[n1].contents)

output overflow(i1, s; local n:Int,
                             P:Map[Int, Locals[P, overflow]],
                             W:Locals[Watch, Int, overflow])
   where 1 ≤ n ∧ n ≤ nProcesses ∧ i1 = n ∧
         i1 ∈ between(1, nProcesses) ⇒ (s = W.s2 ∪ {i1} ∨ ¬(i1 ∈ s))
   pre 1 ≤ n ∧ n ≤ nProcesses ∧ i1 = n ∧
       s = P[n].toSend ∧ n < size(s) ∧ P[n].t ⊆ s
   eff P[n].toSend := P[n].t
       if s = W.s2 ∪ {i1} then W.seen[i1] := true
       elseif ¬(i1 ∈ s) then W.seen[i1] := false
       fi

 output found(i1)
   pre i1 ∈ between(1, nProcesses) ∧ W.seen[i1]
```

Figure 8.5: Output transition definitions of `SysExpanded`

```
output receive(n1, n2, m)
  pre m ∈ C[n1].contents
  eff
    C[n1].contents := delete(m, C[n1].contents)
    if P[n2].val = 0 then P[n2].val := m
    elseif m < P[n2].val then
      P[n2].toSend := insert(P[n2].val, P[n2].toSend);
      P[n2].val := m
    elseif P[n2].val < m then
      P[n2].toSend := insert(m, P[n2].toSend)
    fi

output send(n1, n2, m)
   pre m ∈ P[n1].toSend
   eff
     P[n1].toSend := delete(m, P[n1].toSend)
     C[n1].contents := insert(m, C[n1].contents)

output overflow(i1, s; local P:Map[Int, Locals[P, overflow],
                             W:Locals[Watch, Int, overflow])
   where s = W.s2 ∪ {i1} ∨ ¬(i1 ∈ s)
   pre s = P[i1].toSend ∧ i1 < size(s) ∧ P[i1].t ⊆ s
   eff P[i1].toSend := P[i1].t
       if s = W.s2 ∪ {i1} then W.seen[i1] := true
       elseif ¬(i1 ∈ s) then W.seen[i1] := false
       fi

 output found(i1)
   pre W.seen[i1]
```

Figure 8.6: Simplified output transition definitions of `SysExpanded`

```
internal send(n1, n2, m)
   pre m ∈ P[n1].toSend
   eff
     P[n1].toSend := delete(m, P[n1].toSend)
     C[n1].contents := insert(m, C[n1].contents)
```

Figure 8.7: Internal transition definitions of `SysExpanded`

The final transition of `SysExpanded` is shown in Figure 8.7.

9 Renamings, Resortings, and Substitutions

In this section, we give formal definitions for resortings and variable substitutions in IOA.

9.1 Sort renamings

A *sort renaming* or *resorting* is a map from simple sorts to sorts.[25] Any resorting ρ extends naturally to a map $\dot{\rho}$ defined for all simple sorts by letting $\dot{\rho}$ be the identity on elements not in the domain of ρ. In turn, $\dot{\rho}$ extends further to a map $\ddot{\rho}$ from sorts to sorts by the following recursive definition:

$$\ddot{\rho}(u) ::= \begin{cases} \dot{\rho}(T) & \text{if } u \text{ is a simple sort } T\text{, and} \\ T[\ddot{\rho}(T_1), \ldots, \ddot{\rho}(T_n)] & \text{if } u \text{ is a compound sort } T[T_1, \ldots, T_n]. \end{cases}$$

Let $\rho_{S \to T}$ denote a resorting that maps the sort S to sort T and is otherwise the same as ρ (even if S is already in the domain of ρ).

9.2 Variable renamings

A *variable renaming* ρ_q is an extension of a resorting ρ that maps variables in a sequence q to distinct variables. If v is a variable $i{:}T$ in q, then $\rho_q(v)$ is defined to be $j{:}\ddot{\rho}(T)$ where j is an identifier (i itself, if possible) such that $j{:}\ddot{\rho}(T) \neq \rho_q(v')$ for all variables v' that precede v in q. We say that ρ_q is a *variable renaming with respect to precedence sequence* q.

If ρ_r is a variable renaming where $r = q \| p$ then we say ρ_r is an *extension of* ρ_q *with respect to precedence sequence* p and we write that $\rho_r = \rho_q \vdash p$.

9.3 Operator renamings

An *operator renaming* ω is a map from operators to operators that preserves signatures. Any operator renaming ω extends naturally to a map $\dot{\omega}$ defined for all operators by letting $\dot{\omega}$ map each operator not in the domain of ω to itself.

We extend any operator renaming ω further to a map $\ddot{\omega}$ on some syntactic elements of an IOA automaton (terms to terms, statements to statements, etc.) We now define $\ddot{\omega}$ for each type of IOA syntax to which it may apply.

Terms and sequences of terms

If u is a term, then $\ddot{\omega}(u)$ is

- v, if u is a variable v,
- $\dot{\omega}(f)(\ddot{\omega}(u_1), \ldots, \ddot{\omega}(u_n))$, if u is a term $f(u_1, \ldots, u_n)$ for some operator f and terms $u_1, \ldots, u_n$,
- $\forall v\, \ddot{\omega}(u')$, if u is a term $\forall v\ (u')$ for some variable v and term u', and
- $\exists v\, \ddot{\omega}(u')$, if u is a term $\exists v\ (u')$ for some variable v and term u'.

If q is a sequence of terms $\{u_1, u_2, \ldots, u_n\}$, then $\ddot{\omega}(q)$ is $\{\ddot{\omega}(u_1), \ddot{\omega}(u_2), \ldots, \ddot{\omega}(u_n)\}$.

[25] In IOA, sorts are divided into *simple* or *primitive* sorts, such as `Int` and `T`, and *compound* or *constructed* sorts, such as `Set[T]` and `WeightedGraph[Node,Nat]`.

Values

If l is a value, then $\ddot{\omega}(l)$ is

- $\ddot{\omega}(t)$, if l is a term t
- **choose** v **where** $\ddot{\omega}(p)$, if l is a choice **choose** v **where** p for some variable v and predicate p.

Statements and programs

If s is a statement, then $\ddot{\omega}(s)$ is

- $\ddot{\omega}(lhs) := \ddot{\omega}(rhs)$, if s is an assignment $lhs := rhs$ for some lvalue lhs and some value rhs,
- **if** $\ddot{\omega}(p_1)$ **then** $\ddot{\omega}(s_1)$ **elseif** $\ddot{\omega}(p_2)$ **then** ... **else** $\ddot{\omega}(s_n)$ **fi**, if s is a conditional statement **if** p_1 **then** s_1 **elseif** p_2 ... **else** s_n **fi** for some predicates $p_1, \ldots, p_{n-1}$ and statements $s_1, \ldots, s_n$, and
- **for** v **where** $\ddot{\omega}(p)$ **do** $\ddot{\omega}(g)$ **od**, if s is a loop statement **for** v **where** p **do** g **od** for some variable v, predicate p, and program g.

If g is a program $s_1; s_2; \ldots$, then $\ddot{\omega}(g)$ is $\ddot{\omega}(s_1); \ddot{\omega}(s_2); \ldots$.

Shorthand tuple sort declarations

If ω is an operator renaming and d_1 and d_2 are two shorthand **tuple** sort declarations:

$$d_1 \; ::= \; T \textbf{ tuple of } i_1{:}T_1, i_2{:}T_2, \ldots, \text{and}$$
$$d_2 \; ::= \; T \textbf{ tuple of } j_1{:}T_1, j_2{:}T_2, \ldots,$$

where i_1, $i_2, \ldots$, and j_1, $j_2, \ldots$, are identifiers and T, T_1, $T_2, \ldots$, are sorts then we write $\omega_{d_1 \to d_2}$ or $\omega_{T,\{i_1, i2, \ldots\} \to \{j_1, j_2, \ldots\}}$ for the operator renaming that maps

1. tuple selection operators $\_\_.i_k{:}T \to T_k$ to $\_\_.j_k{:}T \to T_k$, and
2. tuple set operators $\texttt{set\_}i_k{:}T, T_k \to T$ to $\texttt{set\_}j_k{:}T, T_k \to T$.

9.4 Renamings for automata

In Section 5 we defined resortings that map $types^A$ to $actualTypes^{D,A}$ for some desugared automaton A with formal type parameters $types^A$ instantiated with actual type parameters $actualTypes^{D,A}$.

Let ρ be such a resorting and ϱ be the variable renaming $\rho_{\{\}}$. We extend ϱ to a map $\dot{\varrho}$ on some syntactic elements of an IOA automaton (terms to terms, statements to statements, etc.) by defining $\dot{\varrho}$ for each type of IOA syntax to which it may apply.

Automata

If A is desugared primitive automaton with syntax as given in Section 4 and shown in Figure 4.5, then $\dot{\varrho}(A)$ is[26]

[26]Strictly speaking, the definition of the automaton $\dot{\varrho}(A)$ is not a legal definition of a primitive IOA automaton. Its type parameters, shown as $types^A$, should really consist of the non-built-in types that appear in sorts in $\dot{\varrho}(types^A)$. Furthermore, the declared state variables may not match the aggregate state variable selectors that appear in terms in signature **where** clauses, in the **initially** clause, or in transition definitions.

automaton $A(\dot{\varrho}^A(vars^A); types^A)$

signature

$\ldots$

kind $\pi(\dot{\varrho}^{A,\pi}(vars^{A,\pi}))$ **where** $\dot{\varrho}^{A,\pi}(P^{A,\pi}_{kind})$

$\ldots$

states $\rho(stateVars^A) := \dot{\varrho}^A(initVals^A)$ **initially** $\dot{\varrho}^A(P^A_{init})$

transitions

$\ldots$

$$\dot{\varrho}^{A,\pi}_{kind,t_1}\left[\begin{array}{l}\textbf{kind}\ \pi(vars^{A,\pi};\ \textbf{local}\ localVars^{A,\pi}_{kind})\ \textbf{where}\ P^{A,\pi}_{kind,t_1}\\ \quad\textbf{pre}\ Pre^{A,\pi}_{kind,t_1}\\ \quad\textbf{eff}\ Prog^{A,\pi}_{kind,t_1}\ \textbf{ensuring}\ ensuring^{A,\pi}_{kind,t_1}\end{array}\right]$$

$\ldots.$

where

1. $\dot{\varrho}^A$ is a variable renaming $\dot{\varrho} \vdash (\{A, A'{:}States[A, types^A]\} \| \ vars^A \| \ stateVars^A \| \ postVars^A)$.[27]
2. $\dot{\varrho}^{A,\pi}$ is a variable renaming $\dot{\varrho}^A \vdash vars^{A,\pi}$.
3. $\dot{\varrho}^{A,\pi}_{kind,t_1}$ is a variable renaming
$\dot{\varrho}^{A,\pi} \vdash (\{A, A'{:}Locals[A, types^A, kind, \pi]\} \| \ localVars^{A,\pi}_{kind} \| \ localPostVars^{A,\pi}_{kind})$.[28]

Transition definitions

Let t be a transition definition in automaton A as given above. That is, t is

kind $\pi(vars^{A,\pi};$ **local** $localVars^{A,\pi}_{kind})$ **case** c **where** p_1

pre p_2

eff g **ensuring** p_3

where $vars^{A,\pi}$ is a sequence of variables, $localVars^{A,\pi}_{kind} = \{i_1{:}T_1, i_2{:}T_2, \ldots,\}$ is a sequence of variables, p_1, p_2, and p_3 are predicates, and g is a program. Let S be the aggregate local sort $Locals[A, types^A, kind, \pi]$ of t, and $\dot{\varrho}$ be the variable renaming $\dot{\varrho}^{A,\pi}_{kind,t_1}$ given above. That is, $\dot{\varrho}$ is an extension of ρ with respect to the precedence sequence $\{A, A'{:}States[A, types^A]\} \| \ vars^A \| \ stateVars^A \| \ postVars^A \| \ vars^{A,\pi} \| \ \{A, A'{:}Locals[A, types^A, kind, \pi]\} \| \ localVars^{A,\pi}_{kind} \| \ localPostVars^{A,\pi}_{kind}$.

[27] Even though variables in $stateVars^A$ and $postVars^A$ do not appear in any terms in a desugared automaton definition, we include those variables in the precedence sequence to ensure that selectors for local variables do not clash with selectors for state variables in transition definitions (see below).

[28] Like state variables, variables in $localVars^{A,\pi}_{kind}$ and $localPostVars^{A,\pi}_{kind}$ do not appear in any terms in a desugared automaton definition. We include those variables in the precedence sequence only to ensure that selectors for local variables do not clash with each other. (see below).

We define $\dot{\varrho}(t)$ to be

$$\begin{aligned}&\textbf{kind}\ \pi(\dot{\varrho}(vars^{A,\pi});\ \dot{\varrho}(localVars^{A,\pi}_{kind}))\ \textbf{case}\ c\ \textbf{where}\ \ddot{\omega}_{\rho(S),\{i_1,i2,\ldots\}\to\{j_1,j_2,\ldots\}}(\dot{\varrho}(p_1))\\ &\quad\textbf{pre}\ \ddot{\omega}_{\rho(S),\{i_1,i2,\ldots\}\to\{j_1,j_2,\ldots\}}(\dot{\varrho}(p_2))\\ &\quad\textbf{eff}\ \ddot{\omega}_{\rho(S),\{i_1,i2,\ldots\}\to\{j_1,j_2,\ldots\}}(\dot{\varrho}(g))\ \textbf{ensuring}\ \ddot{\omega}_{\rho(S),\{i_1,i2,\ldots\}\to\{j_1,j_2,\ldots\}}(\dot{\varrho}(p_3)).\end{aligned}$$

where $\dot{\varrho}(localVars^{A,\pi}_{kind})$ is a variable sequence $\{j_1{:}\rho(T_1), j_2{:}\rho(T_2), \ldots, \}$. Note that if $localVars^{A,\pi}_{kind} = \dot{\varrho}(localVars^{A,\pi}_{kind})$, then $\omega_{\rho(S),\{i_1,i2,\ldots\}\to\{j_1,j_2,\ldots\}}$ is the identity operator renaming.

Statements and programs

If s is a statement and ϱ is some variable renaming, then $\dot{\varrho}(s)$ is

- $\dot{\varrho}(lhs) := \dot{\varrho}(rhs)$, if s is an assignment $lhs := rhs$ for lvalue lhs and value rhs,

- **if** $\dot{\varrho}(p_1)$ **then** $\dot{\varrho}(s_1)$ **elseif** $\dot{\varrho}(p_2)$ **then**...**else** $\dot{\varrho}(s_n)$ **fi**, if s is a conditional statement **if** p_1 **then** s_1 **elseif** $p_2 \ldots$ **else** s_n **fi** for some predicates $p_1, \ldots, p_{n-1}$and statements $s_1, \ldots, s_n$, and

- **for** $\dot{\varrho}'(v)$ **where** $\dot{\varrho}'(p)$**do** $\dot{\varrho}'(g)$ **od**, if s is a loop **for** v **where** p **do** g **od** for some variable v, predicate p, and program g, where $\dot{\varrho}' = \dot{\varrho} \vdash \{v\}$.

If g is a program $s_1; s_2; \ldots$, then $\dot{\varrho}(g)$ is $\dot{\varrho}(s_1); \dot{\varrho}(s_2); \ldots$.

Values

If l is a value and ϱ is some variable renaming, then $\dot{\varrho}(l)$ is

- $\dot{\varrho}(t)$, if l is a term t, and

- **choose** $\dot{\varrho}'(v)$ **where** $\dot{\varrho}'(p)$, if l is a choice **choose** v **where** p for some variable v and predicate p, where $\dot{\varrho}' = \dot{\varrho} \vdash \{v\}$.

Terms and sequences of terms

If u is a term and ϱ is some variable renaming, then $\dot{\varrho}(u)$ is

- $\varrho(v)$, if u is a variable v,

- $f(\dot{\varrho}(u_1), \ldots, \dot{\varrho}(u_n))$, if u is a term $f(u_1, \ldots, u_n)$ for some operator f and terms $u_1, \ldots, u_n$,

- $\forall\dot{\varrho}'(v)\ \dot{\varrho}'(u')$, if u is a term $\forall v\ (u')$ for some variable v and term u', where $\varrho' = \varrho \vdash \{v\}$, and

- $\exists\dot{\varrho}'(v)\ \dot{\varrho}'(u')$, if u is a term $\exists v\ (u')$ for some variable v and term u', where $\varrho' = \varrho \vdash \{v\}$.

If q is a sequence of terms $\{u_1, u_2, \ldots, u_n\}$, then $\dot{\varrho}(q)$ is $\{\dot{\varrho}(u_1), \dot{\varrho}(u_2), \ldots, \dot{\varrho}(u_n)\}$.

9.5 Substitutions

A *substitution* is a map from variables to terms such that the image of any variable has the same sort as the variable. Any substitution σ extends naturally to a map $\dot{\sigma}$ defined for all variables by letting $\dot{\sigma}$ map each variable not in the domain of σ to a term that is a simple reference to the variable itself.

Let $\sigma_{v \to t}$ denote a substitution that maps the variable v to the term t and is otherwise the same as σ (even if v is already in the domain of σ). We extend any substitution σ further to a map $\ddot{\sigma}$ on some syntactic elements of an IOA automaton (terms to terms, statements to statements, etc.). We now define $\ddot{\sigma}$ for each type of IOA syntax to which it may apply.

Terms and sequences of terms

If u is a term, then $\ddot{\sigma}(u)$ is

- $\dot{\sigma}(v)$, if u is a variable v,
- $f(\ddot{\sigma}(u_1), \ldots, \ddot{\sigma}(u_n))$, if u is a term $f(u_1, \ldots, u_n)$ for some operator f and terms $u_1, \ldots, u_n$,
- $\forall w\ \ddot{\sigma}_{v \to w}(u')$, if u is a term $\forall v\ (u')$ for some variable v and term u', where w is a variable (v itself, if possible) with the same sort as v, where $w \notin \mathcal{FV}(\ddot{\sigma}(v'))$ for all variables $v' \in \mathcal{FV}(u)$, and
- $\exists w\ \ddot{\sigma}_{v \to w}(u')$, if u is a term $\exists v\ (u')$ for some variable v and term u', where w is as above.

If q is a sequence of terms $\{u_1, u_2, \ldots, u_n\}$, then $\ddot{\sigma}(q)$ is $\{\ddot{\sigma}(u_1), \ddot{\sigma}(u_2), \ldots, \ddot{\sigma}(u_n)\}$.

Values

If l is a value, then $\ddot{\sigma}(l)$ is

- $\ddot{\sigma}(t)$, if l is a term t
- **choose** w **where** $\ddot{\sigma}_{v \to w}(p)$, if l is a choice **choose** v **where** p for some variable v and predicate p, where w is a variable (v itself, if possible) with the same sort as v, and where $w \notin \mathcal{FV}(\ddot{\sigma}(v'))$ for all variables $v' \in \mathcal{FV}(l)$.

Statements and programs

If s is a statement, then $\ddot{\sigma}(s)$ is

- $\ddot{\sigma}(lhs) := \ddot{\sigma}(rhs)$, if s is an assignment $lhs := rhs$ for some lvalue lhs and some value rhs,
- **if** $\ddot{\sigma}(p_1)$ **then** $\ddot{\sigma}(s_1)$ **elseif** $\ddot{\sigma}(p_2)$ **then** … **else** $\ddot{\sigma}(s_n)$ **fi**, if s is a conditional statement **if** p_1 **then** s_1 **elseif** p_2 … **else** s_n **fi** for some predicates $p_1, \ldots, p_{n-1}$ and statements $s_1, \ldots, s_n$,
- **for** w **where** $\ddot{\sigma}_{v \to w}(p)$ **do** $\ddot{\sigma}_{v \to w}(g)$ **od**, if s is a loop statement **for** v **where** p **do** g **od** for some variable v, predicate p, and program g, where w is a variable (v itself, if possible) with the same sort as v, where $w \notin \mathcal{FV}(\ddot{\sigma}(v'))$ for all variables $v' \in \mathcal{FV}(s)$.

If g is a program $s_1; s_2; \ldots$, then $\ddot{\sigma}(g)$ is $\ddot{\sigma}(s_1); \ddot{\sigma}(s_2); \ldots$.

Transition definitions

If, in automaton A parameterized by type parameters $types^A$, t is a transition definition

$$\begin{array}{l} \textbf{kind}\ \pi(params^\pi;\ \textbf{local}\ v_1, v_2, \dots)\ \textbf{case}\ c\ \textbf{where}\ p_1 \\ \quad \textbf{pre}\ p_2 \\ \quad \textbf{eff}\ g\ \textbf{ensuring}\ p_3 \end{array}$$

where $params^\pi$ is a sequence of terms, $v_1, v_2, \dots$ is a sequences of variables $i_1{:}T_1, i_2{:}T_2, \dots$, p_1, p_2, and p_3 are predicates, g is a program, and S is the aggregate local sort of t, then $\ddot{\sigma}(t)$ is

$$\begin{array}{l} \textbf{kind}\ \pi(\ddot{\sigma}_{\{v_1,v_2,\dots\}\to\{w_1,w_2,\dots\}}(params^\pi);\ \textbf{local}\ w_1, w_2, \dots) \\ \qquad \textbf{case}\ c\ \textbf{where}\ \ddot{\omega}_{S,\{i_1,i2,\dots\}\to\{j_1,j_2,\dots\}}(\ddot{\sigma}_{\{v_1,v_2,\dots\}\to\{w_1,w_2,\dots\}}(p_1))) \\ \quad \textbf{pre}\ \ddot{\omega}_{S,\{i_1,i2,\dots\}\to\{j_1,j_2,\dots\}}(\ddot{\sigma}_{\{v_1,v_2,\dots\}\to\{w_1,w_2,\dots\}}(p_2))) \\ \quad \textbf{eff}\ \ddot{\omega}_{S,\{i_1,i2,\dots\}\to\{j_1,j_2,\dots\}}(\ddot{\sigma}_{\{v_1,v_2,\dots\}\to\{w_1,w_2,\dots\}}(g))) \\ \qquad \textbf{ensuring}\ \ddot{\omega}_{S,\{i_1,i2,\dots\}\to\{j_1,j_2,\dots\}}(\ddot{\sigma}_{\{v_1,v_2,\dots\}\to\{w_1,w_2,\dots\}}(p_3))) \end{array}$$

where

1. w_k is a variable $j_k{:}T_k$ (v_k itself, if possible), and
2. $w_k \notin \mathcal{FV}(\ddot{\sigma}(v'))$, for all variables $v' \in \{A, A'{:}States[A, types^A]\} \cup\ stateVars^A \cup\ postVars^A \cup vars^A \cup\ \mathcal{FV}(params^\pi) \cup\ \{A, A'{:}Locals[A, types^A, kind, \pi, c]\} \cup\ \{v_l, v'_l \mid l < k\}$.

Note that if $i_k = j_k$ for all k, then $\omega_{S,\{i_1,i2,\dots\}\to\{j_1,j_2,\dots\}}$ is the identity operator renaming.

Hidden clauses

If c is a clause in a **hidden** statement

$$\pi(params^\pi)\ \textbf{where}\ p$$

where $params^\pi$ is a sequence of terms and p is a predicate, then $\ddot{\sigma}(c)$ is

$$\pi(\ddot{\sigma}_{\{v_1,v_2,\dots\}\to\{w_1,w_2,\dots\}}(params^\pi)\ \dots\textbf{where}\ \ddot{\sigma}_{\{v_1,v_2,\dots\}\to\{w_1,w_2,\dots\}}(p)$$

where

1. v_k is a variable $i_k{:}T_k \in \mathcal{FV}(params^\pi)$
2. w_k is a variable (v_k itself, if possible) with sort T_k
3. $w_k \notin \mathcal{FV}(\ddot{\sigma}(v'))$ for all variables $v' \in \mathcal{FV}(params^\pi) \cup\ \mathcal{FV}(p)\ \cup \{v_l \mid l \neq k\}$.

9.6 Notation

Except in definitions such as these, we do not employ separate notations for the extensions $\dot{\rho}$, $\ddot{\rho}$, ρ_ω, ρ_q, and $\dot{\varrho}$ of a resorting ρ. In particular, when applying a resorting ρ to an IOA automaton A, we write ρ for $\dot{\varrho}$. Similarly, we do not distinguish $\dot{\sigma}$ and $\ddot{\sigma}$ from a substitution σ and we write σ for $\ddot{\sigma}$.

References

[1] G. Chapman, J. Cleese, E. Idle, T. Jones, T. Gilliam, and M. Palin. Drinking philosophers. University of Woolloomooloo, Novemember 1970.

[2] Stephen J. Garland, Nancy A. Lynch, Josua A. Tauber, and Mandana Vaziri. *IOA User Guild and Reference Manual*. Massachusetts Institute of Technology, August 2003. `http://theory.lcs.mit.edu/tds/ioa/manual.ps`.