5.6 EXECUTION STATES AND TRANSITIONS

An *execution state* is a tuple

$$(\gamma, sus, alst, clst, ctx)$$

where γ is a configuration, *sus* is a set of suspensions, *alst* is an and-list, *clst* is a choice-list, and *ctx* is a context list. The symbol ζ stands for execution states. The *execution model* is a structure $\langle E, \Rightarrow\rangle$, where E is the set of execution states and $\Rightarrow \subseteq E \times E$ is a transition relation on execution states.

The transition relation $\Rightarrow$ is defined by the derivation rules that follow. When a transition involves the application of one or several computation rules, this is explicitly stated, and also clearly reflected in the transition on the configuration part of the execution states involved. The context χ will be the same as in the corresponding computation rule, except for the choice splitting rule, which uses the contexts of Section 4.5 (Nondeterminism and Stability).

With each transition is associated a set of labels that are *new* in the transition, which will be used in Section 5.8.

5.6.1 Statement Tasks

Constraint Atom Success

$$\frac{(\chi[\ l{::}\mathbf{and}(R, i{::}A, S)_V^{\sigma}\ ], sus, \mathbf{s}(i).alst, clst, ctx)}{(\chi[\ l{::}\mathbf{and}(R, S)_V^{A\wedge\sigma}\ ], sus', \mathbf{w}(l, j_1)\ldots\mathbf{w}(l, j_m).alst, clst, ctx)} \tag{1}$$

The constraint atom rule is applied if env(χ) and $A\wedge\sigma$ are compatible.

The set sus' is $sus \cup \{v_1.l, \ldots, v_n.l\}$, where $\{v_1, \ldots, v_n\}$ is a sufficient set of variables for suspension of $\exists V(\sigma\wedge A)$ in an environment env(χ). The set $\{j_1, \ldots, j_m\}$ consists of all and-boxes in l suspended on variables in a sufficient set for waking when adding A to env(χ)$\wedge\sigma$. (Both n and m may be 0.)

Constraint Atom Failure

$$\frac{(\chi[\ j{::}\mathbf{and}(R, i{::}A, S)_V^{\sigma}\ ], sus, \mathbf{s}(i).alst, clst, ctx)}{(\chi[\ k{::}\mathbf{fail}\ ], sus, \mathbf{a}(k), clst, ctx)} \tag{2}$$

The constraint atom rule and the environment synchronisation rule are applied if env(χ) and $A\wedge\sigma$ are incompatible. The label k is new.

Program Atom Execution

$$\frac{(\chi[\ i{::}A\ ], sus, \mathbf{s}(i).alst, clst, ctx)}{(\chi[\ j{::}B\ ], sus, \mathbf{s}(j).alst, clst, ctx)} \tag{3}$$

The program atom rule is applied, where B is as in the rule. An **s** (statement) task will execute the body. The label j is new.

Composition Execution

$$\frac{(\chi[\ \mathbf{and}(R,\ i{::}(A,\ B),\ S)_V^{\sigma}\ ],\ sus,\ \mathbf{s}(i).alst,\ clst,\ ctx)}{(\chi[\ \mathbf{and}(R,\ j{::}A,\ k{::}B,\ S)_V^{\sigma}\ ],\ sus,\ \mathbf{s}(j).\mathbf{s}(k).alst,\ clst,\ ctx)} \tag{4}$$

The composition rule is applied, splitting a composition into its component statements. Two **s** (statement) tasks will execute the components. The labels *j* and *k* are new.

Hiding Execution

$$\frac{(\chi[\ \mathbf{and}(R,\ i{::}(U : A),\ S)_W^{\sigma}\ ],\ sus,\ \mathbf{s}(i).alst,\ clst,\ ctx)}{(\chi[\ \mathbf{and}(R,\ j{::}B,\ S)_{V\cup W}^{\sigma}\ ],\ sus,\ \mathbf{s}(j).alst,\ clst,\ ctx)} \tag{5}$$

The hiding rule is applied, where V and B are as in the rule. An **s** (statement) task will execute its component. The label *j* is new.

Choice Execution

$$\frac{(\gamma,\ sus,\ \mathbf{s}(i).alst,\ clst,\ ctx)}{(\gamma',\ sus,\ \varepsilon,\ \mathbf{c}(j_1)\ldots\mathbf{c}(j_n).\mathbf{o}(k),\ (alst,\ clst).ctx)} \tag{6}$$

The choice rule is applied, where

$$\gamma = \chi[\ i{::}(U_1 : A_1\ \%\ B_1\ ;\ \cdots\ ;\ U_n : A_n\ \%\ B_n)\ ]$$

$$\gamma' = \chi[\ k{::}\mathbf{choice}((\mathbf{and}(j_1{::}A_1')_{V_1}^{\mathbf{true}}\ \%\ B_1'),\ \ldots,\ (\mathbf{and}(j_n{::}A_n')_{V_n}^{\mathbf{true}}\ \%\ B_n'))\ ]$$

and A_i', B_i', and V_i are as in the rule

A number of **c** (clause) tasks will execute the clauses, and an **o** (or) task will deal with promotion, failure, and suspension. The labels j_1, …, j_n, and *k* are new. The and-boxes introduced are also given new labels.

Aggregate Execution

$$\frac{(\chi[\ i{::}\mathrm{aggregate}(u,\ A,\ v)\ ],\ sus,\ \mathbf{s}(i).alst,\ clst,\ ctx)}{(\chi[\ \mathbf{aggregate}(w,\ j{::}\mathbf{or}(\mathbf{and}(k{::}B)_{\{w\}}^{\mathbf{true}}),\ v)\ ],\ sus,\ \varepsilon,\ \mathbf{c}(k).\mathbf{o}(j),\ (alst,\ clst).ctx)} \tag{7}$$

The aggregate rule is applied, where *w* and B are as in the rule. A **c** (clause) task will execute the goal, and an **o** (or) task will deal with suspension and the unit rule. The labels *j* and *k* are new. The aggregate box and the and-box are also given new labels.

5.6.2 Clause Tasks

Clause Execution

$$\frac{(\chi[\ i{::}\mathbf{and}(j{::}A)_V^{\mathbf{true}}\ ],\ sus,\ \varepsilon,\ \mathbf{c}(j).clst,\ ctx)}{(\chi[\ i{::}\mathbf{and}(j{::}A)_V^{\mathbf{true}}\ ],\ sus,\ \mathbf{s}(j).\mathbf{a}(i),\ clst,\ ctx)} \tag{8}$$

Whether the first (or single) alternative, or one tried as a result of failure or suspension of preceding guards, this rule initiates execution in an and-box. The **c** (clause) task on the choice-list is replaced by two tasks on the and-list: (1) an **s** (statement) task for the statement in the and-box and (2) an **a** (and-box) task to deal with guards, aggregates, and the failure or suspension of an and-box.

5.6.3 And-Box Tasks

Promotion Execution

$$\frac{(\chi[\ l{::}\mathbf{and}(\mathrm{R},\ \mathbf{choice}(i{::}\sigma_{\mathrm{V}}\ \%\ \mathrm{B}),\ \mathrm{S})^{\theta}_{\mathrm{W}}\ ],\ sus,\ \mathbf{a}(i),\ clst,\ (alst',\ clst').ctx)}{(\chi[\ l{::}\mathbf{and}(\mathrm{R},\ j{::}\mathrm{B},\ \mathrm{S})^{\sigma\wedge\theta}_{\mathrm{V}\cup\mathrm{W}}\ ],\ sus',\ \mathbf{w}(l,j_1)\ldots\mathbf{w}(l,j_n).\mathbf{s}(j).alst',\ clst',\ ctx)} \tag{9}$$

The promotion rule is applied. If % is '$\rightarrow$' or '$|$' it is required that σ is quiet with respect to $\theta \wedge \mathrm{env}(\chi)$ and V.

The set *sus′* is *sus* where suspensions *v.i* are replaced by suspensions *v.l*. The set $\{j_1, \ldots, j_n\}$ consists of all and-boxes in *l* suspended on variables in a sufficient set for waking when adding σ to $\mathrm{env}(\chi)\wedge\theta$. The label *j* is new.

Condition Execution

$$\frac{(\chi[\ j{::}\mathbf{choice}(\mathrm{R},\ i{::}\sigma_{\mathrm{V}} \rightarrow \mathrm{B},\ \mathrm{S})\ ],\ sus,\ \mathbf{a}(i),\ clst,\ ctx)}{(\chi[\ j{::}\mathbf{choice}(\mathrm{R},\ i{::}\sigma_{\mathrm{V}} \rightarrow \mathrm{B})\ ],\ sus,\ \mathbf{a}(i),\ \mathbf{o}(j),\ ctx)} \tag{10}$$

The condition rule is applied if S is non-empty and σ is quiet with respect to $\mathrm{env}(\chi)$ and V.

Commit Execution

$$\frac{(\chi[\ j{::}\mathbf{choice}(\mathrm{R},\ i{::}\sigma_{\mathrm{V}}\ |\ \mathrm{B},\ \mathrm{S})\ ],\ sus,\ \mathbf{a}(i),\ clst,\ ctx)}{(\chi[\ j{::}\mathbf{choice}(i{::}\sigma_{\mathrm{V}}\ |\ \mathrm{B})\ ],\ sus,\ \mathbf{a}(i),\ \mathbf{o}(j),\ ctx)} \tag{11}$$

The condition rule is applied if at least one of R or S is non-empty and σ is quiet with respect to $\mathrm{env}(\chi)$ and V.

Collect Execution

$$\frac{(\gamma,\ sus,\ \mathbf{a}(i),\ clst,\ (alst',\ clst').ctx)}{(\gamma',\ sus,\ \varepsilon,\ clst,\ (\mathbf{s}(j).alst',\ clst').ctx)} \tag{12}$$

The collect rule is applied if σ is quiet with respect to $\mathrm{env}(\chi)$ and V, where

$$\gamma = \chi[\ k{::}\mathbf{and}(\mathrm{R}_1,\ \mathbf{aggregate}(u,\ \mathbf{or}(\mathrm{S}_1,\ i{::}\sigma_{\mathrm{V}},\ \mathrm{S}_2),\ v),\ \mathrm{R}_2)^{\theta}_{\mathrm{W}}\ ]$$

$$\gamma' = \chi[\ k{::}\mathbf{and}(\mathrm{R}_1,\ \mathbf{aggregate}(u,\ \mathbf{or}(\mathrm{S}_1,\ \mathrm{S}_2),\ v'),\ j{::}collect(u',\ v',\ v),\ \mathrm{R}_2)^{\sigma\wedge\theta}_{\mathrm{V}\cup\mathrm{W}}\ ]$$

Observe that since the choice splitting rule is augmented with variable renaming, there is no need for renaming in the collection step. The variable *u′* is the copy corresponding to *u* in V, and *v′* does not occur in γ.

An **s** (statement) task will execute the collect operation. It is entered in the saved context, to be executed upon completion or suspension of the aggregate. The constraints are promoted immediately. No waking can occur since they are quiet. Suspensions referring to the and-box *i* need not be promoted. The label *j* is new.

Guard Failure Execution

$$\frac{(\chi[\ \mathbf{choice}(\mathrm{R},\ i{::}\mathbf{fail}\ \%\ \mathrm{B},\ \mathrm{S})\ ],\ sus,\ \mathbf{a}(i),\ clst,\ ctx)}{(\chi[\ \mathbf{choice}(\mathrm{R},\ \mathrm{S})\ ],\ sus,\ \varepsilon,\ clst,\ ctx)} \tag{13}$$

The guard failure rule is applied.

Failure in Or Execution

$$\frac{(\chi[\ \mathbf{or}(\mathrm{R},\ i{::}\mathbf{fail},\ \mathrm{T})\ ],\ sus,\ \mathbf{a}(i),\ clst,\ ctx)}{(\chi[\ \mathbf{or}(\mathrm{R},\ \mathrm{T})\ ],\ sus,\ \varepsilon,\ clst,\ ctx)} \tag{14}$$

The or-flattening rule is applied with S as the empty sequence.

Stable And-box Detection

$$\frac{(\chi[\ i{::}\mathrm{G}\ ],\ sus,\ \mathbf{a}(i),\ clst,\ ctx)}{(\chi[\ i{::}\mathrm{G}\ ],\ sus,\ \varepsilon,\ \mathbf{cs}(i).clst,\ ctx)} \tag{15}$$

if no suspensions in *sus* refer to the and-box *i* or and-boxes in *i*, i.e., *i* is stable, and the preceding and-box task rules (9–14) are not applicable.

And-Box Suspension

$$\frac{(\chi[\ i{::}\mathrm{G}\ ],\ sus,\ \mathbf{a}(i),\ clst,\ ctx)}{(\chi[\ i{::}\mathrm{G}\ ],\ sus,\ \varepsilon,\ clst,\ ctx)} \tag{16}$$

if the preceding and-box task rules (9–15) are not applicable.

5.6.4 Or-Box Tasks

Determinacy Detection

$$\frac{(\chi[\ i{::}\mathbf{choice}(j{::}\sigma_{\mathrm{V}}\ \%\ \mathrm{B})\ ],\ sus,\ \varepsilon,\ \mathbf{o}(i),\ ctx)}{(\chi[\ i{::}\mathbf{choice}(j{::}\sigma_{\mathrm{V}}\ \%\ \mathrm{B})\ ],\ sus,\ \mathbf{a}(j),\ \mathbf{o}(i),\ ctx)} \tag{17}$$

if % is ? or if σ is quiet with respect to env(χ) and V.

Determinacy of an or-box (due to failure of siblings) is detected. The single guard is re-entered, to perform promotion. No waking is possible since the and-box is solved.

Goal Failure

$$\frac{(\chi[\ \mathbf{and}(\mathrm{R},\ i{::}\mathbf{choice}(),\ \mathrm{S})^{\sigma}_{\mathrm{V}}\ ],\ sus,\ \varepsilon,\ \mathbf{o}(i),\ (alst',\ clst').ctx)}{(\chi[\ j{::}\mathbf{fail}\ ],\ sus,\ \mathbf{a}(j),\ clst',\ ctx)} \tag{18}$$

The goal failure rule is applied. The label *j* is new.

Unit Execution

$$\frac{(\chi[\ \mathbf{aggregate}(u,\ i{::}\mathbf{fail},\ v)\ ],\ sus,\ \varepsilon,\ \mathbf{o}(i),\ (alst',\ clst').ctx)}{(\chi[\ j{::}unit(v)\ ],\ sus,\ \mathbf{s}(j).alst',\ clst',\ ctx)} \tag{19}$$

The unit rule is applied. The **s** (statement) task will execute the unit operation. The label *j* is new.

Choice-Box Suspension

$$\frac{(\chi[\ i{::}\mathbf{choice}(\mathrm{R})\ ],\ sus,\ \varepsilon,\ \mathbf{o}(i),\ (alst',\ clst').ctx)}{(\chi[\ i{::}\mathbf{choice}(\mathrm{R})\ ],\ sus,\ alst',\ clst',\ ctx)} \tag{20}$$

unless rules 17 or 18 are applicable.

Or-Box Suspension

$$\frac{(\chi[\ i{::}\mathbf{or}(\mathrm{R})\ ],\ sus,\ \varepsilon,\ \mathbf{o}(i),\ (alst',\ clst').ctx)}{(\chi[\ i{::}\mathbf{or}(\mathrm{R})\ ],\ sus,\ alst',\ clst',\ ctx)} \tag{21}$$

unless rule 19 is applicable.

5.6.5 Choice Splitting Tasks

As opposed to the other execution rules, in this section the contexts (χ and χ') are named as in Section 4.5 (Nondeterminism and Stability) instead of as in the corresponding computation rule (*choice splitting*).

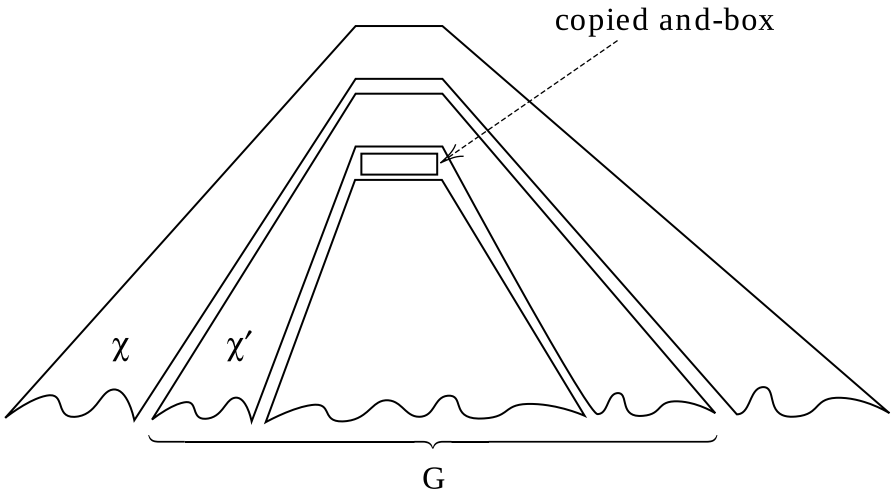

Figure 5.3. Contexts and goals involved in choice splitting

Choice Splitting Execution

$$\frac{(\chi[\ i{::}\mathrm{G}\ ],\ sus,\ \varepsilon,\ \mathbf{cs}(i).clst,\ ctx)}{(\gamma,\ sus',\ \varepsilon,\ clst',\ ctx')} \tag{22}$$

Since G is known to be stable, the choice splitting rule may be applied if G is of the form $\chi'[\ \mathbf{and}(S_1,\ \mathbf{choice}(T_1,\ T_2),\ S_2)^{\sigma}_{V}\ ]$, where $T_1 = (j{::}\theta_w\ ?\ A)$. (There is no need to commit to a particular choice of candidate.) If choice splitting is not applicable, let $\gamma = \chi[\ i{::}\mathrm{G}\ ]$, $sus' = sus$, $clst' = clst$, and $ctx' = ctx$, which is equivalent to and-box suspension.

Due to the handling of suspensions, the goals in the new branch need not only be duplicated, as suggested by the computation rule, but also *copied,* meaning that they will be given new labels and that local variables will be renamed. The copy we call G_1 and the copy of the candidate is labelled j'. All labels of goals in G_1 are new.

$$G_1 = \mathbf{and}(S_1', k_1{::}\mathbf{choice}(j'{::}\theta'_{W'} \mathbin{?} A'), S_2')^{\sigma'}_{V'}$$

The right branch retains the old labels.

$$G_2 = \mathbf{and}(S_1, k_2{::}\mathbf{choice}(T_2), S_2)^{\sigma}_{V}$$

Pertinent suspensions are duplicated.

$sus' = sus \cup sus_1 \cup sus_2$

$sus_1 = \{\, v.j \mid v.i \in sus,\ v$ external to G_2, j copy of $i \,\}$

$sus_2 = \{\, v.j \mid u.i \in sus,\ u$ local to G_2, j copy of i, v renaming of $u \,\}$

If $\chi[\chi'] = \chi''[\ \mathbf{choice}(R_1, \lambda \,\%\, B, R_2)\ ]$, then splitting is applied in the scope of a choice-box. The guard distribution rule and guard failure rules are applied, removing immediately the new or-box created by splitting. The body B is *copied*, renaming variables V to corresponding variables in V′.

$\gamma = \chi''[\ l{::}\mathbf{choice}(R_1, G_1 \,\%\, B', G_2 \,\%\, B, R_2)\ ]$

The choice-box in which splitting is performed is labelled *l*.

If $\chi[\chi'] = \chi''[\ \mathbf{or}(R_1, \lambda, R_2)\ ]$, then splitting is applied in the scope of an or-box. The or-flattening rule is applied, removing the new or-box. Observe that the interaction with aggregates is explained with rule 12 (collect execution).

$\gamma = \chi''[\ l{::}\mathbf{or}(R_1, G_1, G_2, R_2)\ ]$

The or-box in which splitting is performed is labelled *l*.

If $\chi' = \lambda$ then choice splitting was performed with respect to the stable and-box. We promote the copied solution and remember to retry choice splitting. If the remaining choice-box is promotable, we promote it instead.

$ctx' = ctx$

if $T_2 = (\theta_W \mathbin{?} A'')$ then $clst' = \mathbf{p}(k_1).\mathbf{p}(k_2).clst$
else $clst' = \mathbf{p}(k_1).\mathbf{cs}(i).clst$

Otherwise, choice splitting was performed within the stable and-box. Again, we should promote the copied solution, from the point of view of the parent choice- or or-box, but choice splitting should not be retried at that point, since the boxes are not stable. However, if the remaining choice-box is promotable, it has to be promoted.

Let $i_0, j_0, \ldots, i_n, j_n$ be the labels of and-boxes (*i*s) and choice- or or-boxes (*j*s) in the path from the stable and-box labelled $i = i_0$ to the choice- or or-box in which splitting is performed labelled $l = j_n$.

$ctx' = (\mathbf{a}(i_n), \mathbf{o}(j_{n-1})). \ldots .(\mathbf{a}(i_1), \mathbf{o}(j_0)).(\mathbf{a}(i_0), clst).ctx$

if $T_2 = (\theta_W \mathbin{?} A'')$ then $clst' = \mathbf{p}(k_1).\mathbf{p}(k_2).\mathbf{o}(l)$
else $clst' = \mathbf{p}(k_1).\mathbf{o}(l)$

Stability guarantees that no failure will occur and no suspensions need be waked by moving down.

5.6.6 Choice Promotion Tasks

Promote after Splitting

$$\frac{(\chi[\, j{::}\mathbf{and}(\mathrm{R},\, i{::}\mathbf{choice}(k{::}\mathrm{G}\ ?\ \mathrm{B}),\, \mathrm{S})^{\sigma}_{\mathrm{V}}\,],\ sus,\ \varepsilon,\ \mathbf{p}(i).clst,\ ctx)}{(\chi[\, j{::}\mathbf{and}(\mathrm{R},\, i{::}\mathbf{choice}(k{::}\mathrm{G}\ ?\ \mathrm{B}),\, \mathrm{S})^{\sigma}_{\mathrm{V}}\,],\ sus,\ \mathbf{a}(j),\ \mathbf{o}(i),\ ctx')} \tag{23}$$

where

$$ctx' = (\mathbf{a}(j),\ clst).ctx$$

which makes it ready for rule 9 (promotion execution). No waking is possible, since we are in a goal which was stable before choice splitting, and if we are in a aggregate, collect operations have not yet been executed.

5.6.7 Wake-up Tasks

Woken Up

$$\frac{(\gamma,\ sus,\ \mathbf{w}(i, j).alst,\ clst,\ ctx)}{(\gamma,\ sus,\ alst,\ clst,\ ctx)} \tag{24}$$

if $i = j$ or j is the label of an and-box dead in γ.

Wake Up

$$\frac{(\chi[\ i{::}\mathrm{G}\ ],\ sus,\ \mathbf{w}(i, j).alst,\ clst,\ ctx)}{(\chi[\ i{::}\mathrm{G}\ ],\ sus,\ \mathbf{in}(k).\mathbf{w}(k, j).\mathbf{a}(k),\ \mathbf{o}(l),\ (alst,\ clst).ctx)} \tag{25}$$

if $i \neq j$ and j is the label of an and-box live in γ, where

$$\mathrm{G} = \mathbf{and}(\mathrm{R}_1,\ l{::}\mathbf{choice}(\mathrm{S}_1,\ \mathrm{G}'\ \%\ \mathrm{B},\ \mathrm{S}_2),\ \mathrm{R}_2)^{\sigma}_{\mathrm{V}}$$

or $\quad \mathrm{G} = \mathbf{and}(\mathrm{R}_1,\ \mathbf{aggregate}(u,\ l{::}\mathbf{or}(\mathrm{S}_1,\ \mathrm{G}',\ \mathrm{S}_2),\ v),\ \mathrm{R}_2)^{\sigma}_{\mathrm{V}}$

and $\quad \mathrm{G}' = \chi'[j{::}\mathrm{G}''] = k{::}\mathbf{and}(\mathrm{S})^{\theta}_{\mathrm{W}}$

Installation Success

$$\frac{(\chi[\ i{::}\mathbf{and}(\mathrm{R})^{\sigma}_{\mathrm{V}}\ ],\ sus,\ \mathbf{in}(i).alst,\ clst,\ ctx)}{(\chi[\ i{::}\mathbf{and}(\mathrm{R})^{\sigma}_{\mathrm{V}}\ ],\ sus',\ \mathbf{w}(i, j_1)\ldots\mathbf{w}(i, j_m).alst,\ clst,\ ctx)} \tag{26}$$

if env(χ) and σ are compatible.

The set sus' is $sus \cup \{v_1.i,\ \ldots,\ v_n.i\}$, where $\{v_1,\ \ldots,\ v_n\}$ is a sufficient set of variables for suspension of $\exists \mathrm{V}\sigma$ in an environment env(χ). The set $\{j_1,\ \ldots,\ j_m\}$ consists of all and-boxes in i suspended on variables in a sufficient set for waking when adding σ to env(χ). (It is only necessary to wake at this point if the suspension-waking scheme requires multistage waking, as for the optimised treatment of rational trees.)

Installation Failure

$$\frac{(\chi[\ i::\textbf{and}(\mathrm{R})_{\mathrm{V}}^{\sigma}\ ],\ sus,\ \textbf{in}(i).alst,\ clst,\ ctx)}{(\chi[\ j::\textbf{fail}\ ],\ sus,\ \textbf{a}(j),\ clst,\ ctx)} \tag{27}$$

The environment synchronisation rule is applied if env(χ) and σ are incompatible. The label *j* is new.

5.7 EXECUTIONS

A *partial execution* is a finite or infinite sequence of execution states

$$\zeta_0 \Rightarrow \zeta_1 \Rightarrow \zeta_2 \Rightarrow \cdots$$

in which labels that are new in the transition to ζ_i do not occur as labels of goals in any preceding state ζ_j ($j < i$).

A execution state ζ which satisfies $\neg\exists\zeta'.\ \zeta \Rightarrow \zeta'$ is *terminal.*

An *initial* execution state is of the form

$$(\textbf{or}(\textbf{and}(j::\mathrm{A})_{\mathrm{V}}^{\textbf{true}}),\ sus,\ \varepsilon,\ \textbf{c}(j),\ \varepsilon)$$

where *sus* is an empty set of suspensions.

A *final* execution state is of the form

$$(\gamma,\ sus,\ \varepsilon,\ \varepsilon,\ \varepsilon)$$

where γ is a *final* configuration. A terminal execution state that is not final is *stuck.*

An *execution* is a partial execution beginning with an initial execution state and, if finite, ending with a terminal execution state.

By the *goal to which a computation rule can be applied,* we mean (1) for statement rules, the statement itself, (2) for the pruning and collect rules, the solved and-box, (3) for the promotion rule, the solved and-box or the choice-box, (4) for the environment failure rule, the and-box replaced by fail, (5) for the goal failure rule, the empty choice-box, (6) for the guard failure rule, the failed guard, (7) for the choice splitting rule, the stable and-box, (8) for the unit rule, the empty or-box corresponding to fail, and (9) for the or-flattening rule, in the case of flattening an empty or-box, the empty or-box.

Below we assume that the scheme for suspension and waking is complete.

LEMMA 1. *In all execution states that do not contain* **in**, *all subgoals of the configuration to which a computation rule can be applied are referred to by tasks.*

PROOF. This is shown by induction on execution steps.

We first observe, by inspecting all rules, that in all task lists, tasks refers to subgoals of the box referred to by the **a** or **o** task that end them. We also observe, by inspecting the rules that introduce multiple choice-tasks (6 and 22), that tasks in choice-lists refer to subgoals of the pertaining or- or choice-box in left-to-right order, and that tasks are processed in this order.

The initial state contains a statement to which a computation rule can be applied, which is referred to by a **c** task.

It can be argued for each execution rule, by tedious inspection, that the goals to which computation rules can be applied after a corresponding step are either given new tasks or are already referred to by tasks. The more interesting cases are described.

The constraint atom and installation execution rules can lead (1) to the incompatibility with its environment of the local store, (2) to the quietness or incompatibility of constraint stores in subgoals of the and-box, or (3) to the and-box being solved. Either (zero or more) **w** tasks are entered that refer to all and-boxes for which constraint stores may have become quiet or incompatible, or an **a** task is left that refers to the failed and-box. In the first case, if the and-box is solved, it is referred to by the **a** task in the list. In the second case, the tasks removed refer to dead goals.

The condition and commit execution rules remove all siblings or siblings to the right of the solved and-box. This and-box was reached by what was the first task on the pertaining choice-list. Therefore, the other tasks, except **o**, refer to subgoals to the right, and may safely be removed in both cases.

The collect execution rule moves tasks in an unusual way, but adds an **s** task for the new statement, and there is already an **o** task for the or-box, in case it becomes empty.

The guard failure execution and failure-in-or execution rules remove goals, and have already **o** tasks for the choice- or or-box, in case they become determinate or empty.

The choice splitting execution rule can suspend due to the lack of a candidate. Otherwise, it makes promotion applicable to the copy of the candidate. Promotion may have become applicable in the right branch. If the copied and-box was stable, the right branch may still be stable. These possibilities are handled by tasks introduced.

The remaining rules rather trivially preserve the property. ♦

LEMMA 2. *A terminal execution state in an execution is of the form*

$$(\gamma, sus, \varepsilon, \varepsilon, \varepsilon)$$

where γ *is a terminal configuration.*

PROOF.

(1) All tasks have rules that apply in any execution state that may arise. In particular, the **a** and **o** tasks have suspension rules that pop the context stack. Thus, the task and context stacks are empty in a terminal state.

(2) If there are no tasks, γ is a terminal configuration, by Lemma 1. ♦

LEMMA 3. *Only a finite number of transitions on an execution state are made between transitions on its configuration.*

PROOF. The tasks **s**, **c**, **a**, **o**, **cs**, and **p** either lead to transitions, or are skipped by suspension rules and the like, and there is only a finite number of tasks to skip. Unless the configuration is changed by a transition, each **in** and **w** leads to a finite number of **w** and **in**, since the tree of and-boxes visited is finite. ♦

PROPOSITION (correctness). *The sequence of configurations in an execution, in which subsequent equal configurations are deleted, forms a computation.*

PROOF. An initial execution state contains an initial configuration. Transitions between execution states that modify the configuration do so by applying computation rules, which are implicitly propagated by the subgoal rule, and where we for copying make implicit use of a rule that allows us to rename the local variables of and-boxes. Terminal execution states contain terminal configurations by Lemma 2. Execution is guaranteed to make progress and either produce a finite or an infinite computation by Lemma 3. ♦

5.8 DISCUSSION

The execution model presented is not fair, in that a task can remain unattended indefinitely. For example,

```
p, a = b
```

where

```
p :- p.
```

will loop, not fail.

A fair execution model is arguably more appealing to the intuition. All agents are active, and have their own “virtual processor”. It is easier to reason about some properties of programs. However, the intuition can also be misleading.

A common misunderstanding is that a producer and a consumer will execute in a, more or less, constant size execution state (if garbage collection simplification is performed). This is not so. The producer can produce at, say, twice the speed of the consumer. To guarantee constant size, a bounded buffer programming technique has to be used (see, e.g., [Shapiro 1987]).

Another misunderstanding is that fairness models the notion of background processes, for example, background re-pagination of a document in a word processor while it is being edited. Basic fairness, however, has nothing to say about the resources devoted to each task. Re-pagination might very well use 90% of the time, and slow down editing, or it might use 0.1%, and never catch up.

These misunderstandings are usually beginner's problems.

A problem of a quite different nature is that by stipulating fairness we also stipulate “breadth-first” search in programs employing don't know nondeterminism. This could be relaxed, by giving (unfair) priority to the leftmost alternative in don't know choice, conditional choice, and ordered bagof, since the