5-8-1996

Automating transformational design for distributed programs

Champak Das
Florida International University

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

AUTOMATING TRANSFORMATIONAL DESIGN FOR DISTRIBUTED PROGRAMS

A thesis submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

IN

COMPUTER SCIENCE

by

Champak Das

1996

To: Dean Arthur W. Herriott
College of Arts and Sciences

This thesis, written by Champak Das, and entitled AUTOMATING TRANSFORMATIONAL DESIGN FOR DISTRIBUTED PROGRAMS, having been approved in respect to style and intellectual content, is referred to you for your judgement.

We have read this thesis and recommend that it be approved.

Michael Evangelist

Rakesh Sinha

Rida Bazzi

Paul Attie, Major Professor

Date of Defense : May 8th, 1996

The thesis of Champak Das is ap

Dean Arthur W. Herriott
College of Arts and Sciences

Dr. Richard L. Campbell
Dean of Graduate Studies

Florida International University, 1996

Acknowledgment

I would like to thank my advisor, Dr. Paul Attie, for guiding me into the area of distributed computing. His patience and deep understanding of the field saw me through the difficult stretches. I would like to also thank Dr. Michael Evangelist, Dr. Rida Bazzi and Dr. Rakesh Sinha for their helpful comments and suggestions.

I wish to express my deepest thanks to my parents for their support and encouragement which enabled me to complete this work.

ABSTRACT OF THE THESIS

AUTOMATING TRANSFORMATIONAL DESIGN FOR DISTRIBUTED PROGRAMS

by

Champak Das

Florida International University, 1996

Professor Paul Attie, Major Professor

We address the problem of designing concurrent, reactive, nonterminating programs. Our approach to developing concurrent programs involves the use of correctness-preserving transformations to realize each step of program development. The transformations we have designed automatically guarantee the preservation of the deadlock freedom property, and hence deadlock freedom does not have to be manually verified after each development step. Since our transformations are syntactic, they are easily mechanizable as well. This makes syntactic transformations particularly appealing for the development of large, complex, and correct distributed systems, where a manual approach would be prohibitively expensive. In this work we present a set of syntactic transformations along with an example of their application to the development of a simplified mobile telephone system.

Contents

List of Figures

Chapter 1

Introduction

We address the problem of designing concurrent, reactive, nonterminating programs. Nontermination implies that correctness properties cannot be stated as simple functional relationships between inputs and outputs. Instead, it becomes necessary to express correctness properties in terms of the time varying behavior of a system.

Correctness properties of concurrent programs are classified as follows :

- Safety properties: These imply that "nothing bad happens" e.g., simultaneous write access to the same file by two processes does not occur.
- Liveness properties: These imply that "progress occurs in the system" e.g., every request for a resource is eventually granted.

The problem of specifying and ensuring correctness properties is difficult in general because they are global properties and therefore depend on the entire program. Thus the entire program has to be considered in analysis. There have been some efforts at modularization [Pn85], but success has been limited.

Most of the current approaches to ensuring correctness properties are either impractical or excessively restricted in scope. Formal program verification techniques are widely accepted as a means of guaranteeing the correctness of distributed

programs [Ho69, Fr92, La80, V87]. The applicability of these techniques is limited by a number of different factors such as the excessive manual labour required, the possibility of errors in the proofs and the lack of appropriately trained personnel [DM90, DB89, FJ89, GCR94, KM90, CM81]. Even the use of theorem provers has not made the process significantly more feasible.

Successive refinement approaches with manual verification of each refinement step involve large amounts of manual formal labor for each refinement step [BKS83, BKS85, CM88, RM87]. Refinement is largely a matter of symbol manipulation, and, for humans, that activity is as error-prone as manual theorem proving or even program writing. The use of theorem provers to carry out formal manipulation is one way of reducing the possibility of error, but the labor involved is still high.

The model checking approaches suffer from exponential time complexity and the state explosion problem [BCMD90, CLM89, CBGM91]. They are efficient only for systems with large amounts of symmetry and regularity. Besides the problems mentioned above, all the current approaches we referred to fail to address the problem of designing systems.

In this work, an alternative approach to the problem of developing concurrent systems is proposed, namely the use of *syntactic transformations* to realize each step of program development. In order to reduce the formal labor of verifying each step, we aim to design *correctness-peserving transformations*, i.e., transformations which automatically guarantee the preservation of desirable properties (e.g., deadlock freedom). If a correctness-peserving transformation is not available for a particular step, then we have to rely on a *manual transformation*, which necessitates the manual verification that the desired properties still hold after applying the transformation. Since

our transformations are syntactic, they are easily mechanizable as well. This makes correctness-peserving transformations particularly appealing for the development of large, complex distributed systems, where a manual approach would be prohibitively expensive. As mentioned before, preservation of global properties cannot be achieved by considering only local structures and therefore transformations generally must encompass the entire system.

We take a two-fold approach to transformations. The first approach is successive refinement which lets us start with an abstract specification and incrementally refine it to a stage where implementation becomes relatively straightforward. In itself this is not a new idea. However manual verification combined with refinement has a number of drawbacks as explained above. This motivates our proposal for correctness-preserving syntactic transformations. Such transformations are mechanizable and, therefore, do not involve significant amounts of manual labor. Using this approach, the process of development may be viewed as the human-assisted high-level compilation of a specification into code. Human interaction is still essential for choosing the appropriate transformations to apply at each stage. But verifying that a transformation preserves desired properties is unnecessary, in our approach, because this is guaranteed by the fact that the transformations are correctness-preserving. In chapter two we present two transformations for successive refinement.

A second approach is program composition where we create small programs and then attempt to merge them in a manner which is consistent with our design goals. The merging process also preserves important properties. The small programs we start with could be checked manually without spending excessive intellectual effort. In chapter three we present eight transformations for program composition.

All of the transformations described in this thesis preserve *deadlock freedom*, an important safety property of concurrent programs. In principle, more general safety and liveness properties can be dealt with. We envision this as a possible extension of the work to be presented in the thesis.

It may argued that it would be very difficult, if not impossible, to create a complete set of transformations which is powerful enough to deal with every conceivable problem. Any transformation toolbox would be restricted in scope. It is true that devising a large set of transformations would require a very significant amount of effort over a long period. However this would be a one time effort (as in writing a good compiler). The current work is a step in that direction.

Chapter 2

Model of Concurrent Computation

2.1 Notation and Syntax

A *program* is the composition of a fixed set of sequential processes executing concurrently. We use the nondeterministic interleaving model of concurrency. That is, we view concurrency as the nondeterministic interleaving of events. An *event* is the atomic (i.e., indivisible) execution of an action, described in Definition 1. We use $\rightarrow$, $[\!]$, $\|$ to denote sequence, choice, and parallel composition, respectively. The semantics of these operators is similar to that given in CSP [Ho85]. To model state transitions, we employ the concept of a labeled transition system, as used in [Mil89]. $\stackrel{a}{\rightarrow}$ will denote the transition relation induced by action, a. The formal meaning of $\rightarrow$, $[\!]$, $\|$ and $\stackrel{a}{\rightarrow}$ is given below.

Definition 1 (*action*)

An action, a drawn from some set, $\mathcal{A}$, of action names, is a character string (i.e., an identifier) .

We use lower-case letters towards the beginning of the alphabet to denote actions.

Definition 2 (*action expression*)

An action expression E is a finite expression given by the following BNF grammar:

<*action_expression*> ::=

<*action_expression*> $[\!]$ <*action_expression*> |

<*action_expression*> $\rightarrow$ <*action_expression*> |

(<*action_expression*>) |

<*action*> | ε | 0

E, F, G, H will range over the set of action expressions. We make the convention that $\rightarrow$ has higher binding power than $[\!]$. Parenthesis have the highest precedence and are used to alter precedence where necessary. Thus $E \rightarrow F \,[\!]\, G$ denotes $(E \rightarrow F) \,[\!]\, G$, which is different from $E \rightarrow (F \,[\!]\, G)$. We assume that a normal form exists for an action expression where all actions are parenthesized as appropriate. The precedence rule may make some of these parenthesis redundant. For our discussions we will sometimes consider action expressions without the redundant parenthesis. Intuitively, $E \rightarrow F$ means execute E and then execute F, while $E \,[\!]\, F$ means execute either E or F.

0, ("Stop"), is the identity element of $[\!]$, and ε ("Skip"), is the identity element of $\rightarrow$. We postulate the following axioms :

1. $0 \,[\!]\, A = A$, $A \,[\!]\, 0 = A$, $A \rightarrow \varepsilon = A$, $\varepsilon \rightarrow A = A$

2. $A \,[\!]\, B = B \,[\!]\, A$

3. $A \,[\!]\, (B \,[\!]\, C) = (A \,[\!]\, B) \,[\!]\, C$,
   $A \rightarrow (B \rightarrow C) = (A \rightarrow B) \rightarrow C$

Equating action expressions (E, F):

- If E and F are syntactically identical, then $E = F$.

- If E and F are not syntactically identical, we first fully parenthesize both expressions using the precedence rules. Let the new expressions derived from E and F be E' and F' respectively. If E' and F' are syntactically identical modulo the above axioms, then $E = F$.

We define αE, the *alphabet* of action expression E, as follows.

Definition 3 (α)

$\alpha(a) \stackrel{\text{df}}{=} \{a\}$

$\alpha(E \parallel F) \stackrel{\text{df}}{=} \alpha(E) \cup \alpha(F)$

$\alpha(E \rightarrow F) \stackrel{\text{df}}{=} \alpha(E) \cup \alpha(F)$

Definition 4 (*sequential process*)

A sequential process, P_i, consists of a process body and a process alphabet. The process body, βP_i, is an expression of the form $F_i \rightarrow *E_i$ where F_i, E_i are action expressions. The process alphabet, αP_i, is defined to be a set of action names. The process alphabet must contain $\alpha F_i \cup \alpha E_i$.

Note that this definition extends the definition of "alphabet" to processes. We have also introduced "*", which denotes infinite iteration. We extend $=$ to process bodies in a straightforward manner. If $\beta P_i = F_i \rightarrow *E_i$, and $\beta P_j = F_j \rightarrow *E_j$, then $\beta P_i = \beta P_j$ iff $F_i = F_j$ and $E_i = E_j$. Finally, $P_i = P_j$ iff $\alpha P_i = \alpha P_j$ and $\beta P_i = \beta P_j$. (Note that when we write $\alpha P_i = \alpha P_j$, the $=$ symbol denotes standard set-theoretic equality, because alphabets are set.)

Definition 5 (*program*)

A program, P, is the parallel composition of one or more sequential processes; i.e.,

$P = (\| \; i \in \varphi : P_i)$, *where* P_i *are processes and* φ *is some suitable index set. Also,* $\alpha P = (\cup i \in \varphi : \alpha P_i)$.

For sake of simplicity, we assume that all actions in a program are uniquely named. We extend $=$ to programs in the expected manner: $(\| \; i \in \varphi : P_i) = (\| \; i \in \psi : Q_i)$ iff $\varphi = \psi$ and, for all $i \in \varphi, P_i = Q_i$.

Definition 6 (PA_P)

$$PA_P(a) \stackrel{\text{df}}{=} \{i \in \varphi \mid a \in \alpha P_i\}$$

$PA_P(a)$ is the set of processes within program P that jointly and synchronously participate in the execution of action a. If $|PA_P(a)| > 1$ then a is a *multiparty interaction* of program P. If $|PA_P(a)| = 1$, then a is a *local action* of some process P_i (namely the P_i such that $a \in \alpha P_i$) in program P.

2.2 Operational Semantics

We define the binary transition relation $\stackrel{a}{\rightarrow}$ on the set of sequential processes as follows. In each case, the alphabet of the process making the transition is unchanged, and so we only show the process bodies. In order to avoid the well-known phenomenon that the behavior of $E \,[\!]\, F$ and $\varepsilon \rightarrow E \,[\!]\, \varepsilon \rightarrow F$ is different even though they are "equal", we stipulate that the transition relation cannot be applied to 0 and ε, i.e., $\stackrel{\varepsilon}{\rightarrow}$ and $\stackrel{0}{\rightarrow}$ are not defined. This does not cause any difficulties, since ε and 0 can always be eliminated from an expression using the above axioms, after which the transition relation can be applied.

Definition 7 (*Transition Relation* $\stackrel{a}{\rightarrow}$)

$$\textbf{Act.} \quad \frac{}{(a \rightarrow E) \stackrel{a}{\rightarrow} E}$$

$$\textbf{Ch.}\quad \frac{E \xrightarrow{a} E'}{(E \parallel F) \xrightarrow{a} E'} \qquad \frac{F \xrightarrow{a} F'}{(E \parallel F) \xrightarrow{a} F'}$$

$$\textbf{Seq}\quad \frac{E \xrightarrow{a} E'}{(E \to F) \xrightarrow{a} (E' \to F)}$$

$$\textbf{Iter}\quad \frac{((E) \to *E) \xrightarrow{a} E'}{*E \xrightarrow{a} E'}$$

We extend $\xrightarrow{a}$ *to processes by stipulating that* $P_i \xrightarrow{a} P_i'$ *iff* $\beta P_i \xrightarrow{a} \beta P_i'$ *and* $\alpha P_i = \alpha P_i'$. *in other words, the alphabets are the same and the bodies are related by* $\xrightarrow{a}$. *Finally, we extend* $\xrightarrow{a}$ *to programs as follows:*

Let $P = (\| \; i \in \varphi : P_i)$, $P' = (\| \; i \in \varphi : P_i')$. *Then* $P \xrightarrow{a} P'$ *iff:*

1. *for all* $i \in PA_P(a) : P_i \xrightarrow{a} P_i'$
2. *for all* $i \in \varphi - PA_P(a) : P_i = P_i'$

We write $P \xrightarrow{a}$ to mean that there exists a P' such that $P \xrightarrow{a} P'$. In this case, we say that a is *enabled* in P. We also write $P \overset{a}{\not\to}$ to mean that there does not exist a P' such that $P \xrightarrow{a} P'$, and we say that a is *disabled* in P in this case.

Suppose $P_i \xrightarrow{a}$. Then the general form for the body of P_i is $F \to *E$, where F has one of the forms $c, c \to G, c \parallel H, c \to G \parallel H$. All of these forms are subsumed by the form $c \to G \parallel H$ however, since $c = c \to \varepsilon \parallel 0$, $c \to G = c \to G \parallel 0$, $c \parallel G = c \to \varepsilon \parallel G$. Thus the introduction of 0 and ε allows us to avoid a large amount of tedious case-analysis.

We now present some preliminary definitions and results.

Definition 8 (*Independent*)

Two actions b, c *are* independent *in program* P *iff* $PA_P(b) \bigcap PA_P(c) = \emptyset$

Definition 9 (*Derivative, Path*)

If $P \xrightarrow{a_1} \cdots \xrightarrow{a_n} P'$ *for some sequence* $a_1, \ldots, a_n$ *of actions, then ([Mil89]) we say that* P' *is a* derivative *of* P. *The sequence* $a_1, \ldots, a_n$ *is called a* path. *If path* $\pi = a_1, \ldots, a_n$, *then we abbreviate* $P \xrightarrow{a_1} \cdots \xrightarrow{a_n} P'$ *by* $P \xrightarrow{\pi} P'$.

Consider a program consisting of a single process $P_1 = *[a \rightarrow b \parallel a \rightarrow c]$. Clearly, $P_1 \xrightarrow{a} b \rightarrow P_1$, and $P_1 \xrightarrow{a} c \rightarrow P_1$. This example can easily be extended to arbitrary paths. Thus we see that $P \xrightarrow{\pi} P'$ and $P \xrightarrow{\pi} P''$ for some P, π does not allow us to conclude $P' = P''$. Thus, if P and π are given, then the assertion $P \xrightarrow{\pi} P'$ can be regarded as an abbreviation for "there exists a P' such that $P \xrightarrow{\pi} P'$".

Definition 10 (*Equivalent*)

Two paths π, ρ *are* equivalent *iff one can be obtained from the other by a finite number of exchanges of adjacent independent actions.*

Proposition 1 *Let* $P \xrightarrow{\pi} Q$. *If* π *and* ρ *are equivalent, then* $P \xrightarrow{\rho} Q$.

Proof: The proof is by induction on the number m of exchanges of independent adjacent actions required to obtain ρ from π.

Base Case, $m = 1$.

Now ρ is obtained from π by one exchange. Hence we can write $\pi = \pi' ab \pi''$, $\rho = \pi' ba \pi''$, where a, b are the exchanged independent actions. Thus we have

$$P \xrightarrow{\pi'} P' \xrightarrow{ab} P'' \xrightarrow{\pi''} Q \qquad (*)$$

for some P', P''. Since a and b are independent actions, by definition 8, $PA_P(a) \bigcap PA_P(b) = \emptyset$. Different sets of processes execute the actions a and b. Clearly, irrespective of the order in which a and b are executed, the resulting program is the

same. Therefore $P' \xrightarrow{ba} P''$, i.e., P'' can be derived from P' by executing b and then a. Using this result and (*) we have $P \xrightarrow{\pi'} P' \xrightarrow{ba} P'' \xrightarrow{\pi''} Q$. Hence $P \xrightarrow{\rho} Q$. Thus the base case is established.

Induction Step, $m = n + 1$, $n \geq 1$, where the inductive hypothesis is assumed for n exchanges.

Since ρ is obtained from π by $n + 1$ exchanges, there must exist a η such that η is obtained from π by n exchanges, and ρ is obtained from η by one exchange. By the inductive hypothesis, we have:

$$P \xrightarrow{\eta} Q$$

Since ρ is obtained from η by one exchange, we use same argument as employed in the base case (i.e., for a single exchange) to conclude:

$$P \xrightarrow{\rho} Q$$

This establishes the induction step. □

We shall use the notation described here in the rest of the work. Enhancements will be pointed out when necessary.

Chapter 3

Sequence Introduction Transformations

3.1 Introduction

The idea of successively refining an abstract specification until it contains enough details to suggest an implementation has been investigated by numerous researchers [BKS83, BKS85, CM88, RM87]. The emphasis to date has been on techniques that, unfortunately, lead to a large amount of manual formal labor for each refinement step. With such techniques, both the cost and the possibility of errors arising in formal manipulation are high. Using a theorem prover can reduce the number of manipulation errors, but, given current technology, the amount of labor is still daunting.

Our research explores an alternative solution to the refinement problem, namely the use of syntactic transformations to realize each refinement step. We reduce formal labor by employing automatic transformations that guarantee the preservation of desirable properties — e.g., deadlock-freedom. Automatic transformations are particularly appealing for the development of large, complex distributed systems, where a manual approach to refinement would be prohibitively expensive. Distributed computations are, by nature, reactive and concurrent, so their correctness cannot be

specified as a simple functional relationship between inputs and outputs. Instead, specifications must describe the time-varying behavior of the system. Further difficulty is caused by the fact that such important characteristics of distribution as deadlock-freedom are global properties that cannot be achieved through considering local structures only. Transformations generally must encompass the entire system.

In this chapter we present two automatic transformations that decompose actions into a sequence of actions. These transformations are guaranteed to preserve freedom from deadlock. We give formal proofs of this characteristic, as well as an example of refinement using the transformations.

3.2 The Right-sequence Introduction Transformation

Our first transformation allows us to introduce a new action, d, in sequence with, and immediately after, an already-present action, c. Intuitively, we use such a transformation to refine c. In the original high-level program, c might model a complex set of activities. In the transformed (lower-level) program, this set of activities is split between c and d.

Definition 11 (*Right-sequence Introduction Transformation*)

We define the right-sequence introduction transformation $[c/c \rightarrow d]$ *in a bottom-up manner as follows. Let* a *be an arbitrary action, and* E, F *be arbitrary action expressions. Then, we have*

$$\varepsilon[c/c \rightarrow d] = \varepsilon$$

$$0[c/c \rightarrow d] = 0$$

$$a[c/c \rightarrow d] = a \text{ if } a \neq c$$

$c[c/c \rightarrow d] = c \rightarrow d$

$(E \parallel F)[c/c \rightarrow d] = ((E[c/c \rightarrow d]) \parallel (F[c/c \rightarrow d]))$

$(E \rightarrow F)[c/c \rightarrow d] = ((E[c/c \rightarrow d]) \rightarrow (F[c/c \rightarrow d]))$

In the sequel, we will use the abbreviation E_t *for* $E[c/c \rightarrow d]$ *for an arbitrary action expression* E.

For an arbitrary process P_i *such that* $c \in \alpha P_i$, *and* $\beta P_i = F \rightarrow *E$ *for some action expressions* F, E, *define* $P_i[c/c \rightarrow d] = Q_i$, *where* $\alpha Q_i = \alpha P_i \cup \{d\}$, $\beta Q_i = F_t \rightarrow *E_t$.

Let $P = (\parallel i \in \varphi : P_i)$ *be an arbitrary program. Let* ψ *be an arbitrary non-empty subset of* $PA_P(c)$. *We define* $P[c/c \rightarrow d] = (\parallel i \in \psi : P_i[c/c \rightarrow d]) \parallel (\parallel i \in \varphi - \psi : P_i)$.

If $Q = P[c/c \rightarrow d]$, then we say that Q results from P by means of a *right-sequence introduction transformation.* The right-sequence introduction transformation $[c/c \rightarrow d]$ takes a program P containing an action c, and introduces a new action d after c and in sequence with c. Note that ψ is an implicit parameter of the functional mapping from P to Q expressed by $Q = P[c/c \rightarrow d]$. In the sequel, whenever we write $Q = P[c/c \rightarrow d]$, we shall implicitly assume that the conditions $\alpha P \cup \{d\} = \alpha Q$, $d \notin \alpha P$, $c \in \alpha P$, $c \in \alpha Q$ are all true.

3.2.1 Discussion and Preliminary Technical Definitions

The basic idea is that the action c in program P is decomposed into the sequence of c followed by d in program Q. The transformation can be iterated any number of times, so that c is decomposed into a sequence $c, d_1, \ldots, d_n$.

Having defined the right-sequence introduction transformation, we need to relate the behavior of the transformed program $Q = P[c/c \rightarrow d]$ to that of the original

program P. We do this by means of a modification of the notion of *strong bisimulation* (see [Mil89], ch. 4).

Definition 12 (*cd-bisimulation*)

*Let $\mathcal{S}$ be a binary relation over programs. Then $\mathcal{S}$ is a cd-*bisimulation *iff $P\,\mathcal{S}\,Q$ implies:*

1. $\alpha P \cup \{d\} = \alpha Q$, $d \notin \alpha P$, $c \in \alpha P$, $c \in \alpha Q$
2. *for all $a \in \alpha P - \{c\}$, if $P \xrightarrow{a} P'$, then $Q \xrightarrow{a} Q'$ for some Q' such that $P'\,\mathcal{S}\,Q'$*
3. *if $P \xrightarrow{c} P'$, then $Q \xrightarrow{cd} Q'$ for some Q' such that $P'\,\mathcal{S}\,Q'$*
4. *for all $a \in \alpha P - \{c\}$, if $Q \xrightarrow{a} Q'$, then $P \xrightarrow{a} P'$ for some P' such that $P'\,\mathcal{S}\,Q'$*
5. *if $Q \xrightarrow{cd} Q'$, then $P \xrightarrow{c} P'$ for some P' such that $P'\,\mathcal{S}\,Q'$*

Definition 13 *Following the treatment of strong bisimulation given in chapter 4 of [Mil89], we define*

$$\sim \stackrel{\text{df}}{=} \bigcup\{\mathcal{S} \mid \mathcal{S} \text{ is a cd-bisimulation}\}.$$

From this, we can establish

Proposition 2 *$P \sim Q$ iff*

1. $\alpha P \cup \{d\} = \alpha Q$, $d \notin \alpha P$, $c \in \alpha P$, $c \in \alpha Q$
2. *for all $a \in \alpha P - \{c\}$, if $P \xrightarrow{a} P'$, then $Q \xrightarrow{a} Q'$ for some Q' such that $P' \sim Q'$*
3. *if $P \xrightarrow{c} P'$, then $Q \xrightarrow{cd} Q'$ for some Q' such that $P' \sim Q'$*
4. *for all $a \in \alpha P - \{c\}$, if $Q \xrightarrow{a} Q'$, then $P \xrightarrow{a} P'$ for some P' such that $P' \sim Q'$*

5. *if* $Q \xrightarrow{cd} Q'$, *then* $P \xrightarrow{c} P'$ *for some* P' *such that* $P' \sim Q'$

Proof: The proof proceeds in the same way as the proof of proposition 4 in chapter 4 of [Mil89] (bearing in mind that the notions of bisimulation differ technically). We paraphrase the proof here for completeness.

First, we define the relation $\sim'$ as follows:

$P \sim' Q$ iff

1. $\alpha P \cup \{d\} = \alpha Q$, $d \notin \alpha P$, $c \in \alpha P$, $c \in \alpha Q$
2. for all $a \in \alpha P - \{c\}$, if $P \xrightarrow{a} P'$, then $Q \xrightarrow{a} Q'$ for some Q' such that $P' \sim Q'$
3. if $P \xrightarrow{c} P'$, then $Q \xrightarrow{cd} Q'$ for some Q' such that $P' \sim Q'$
4. for all $a \in \alpha P - \{c\}$, if $Q \xrightarrow{a} Q'$, then $P \xrightarrow{a} P'$ for some P' such that $P' \sim Q'$
5. if $Q \xrightarrow{cd} Q'$, then $P \xrightarrow{c} P'$ for some P' such that $P' \sim Q'$

From definition 13, we see that $\sim$ is a *cd*-bisimulation. Thus from definition 12 and the definition of $\sim'$, we deduce:

$$P \sim Q \text{ implies } P \sim' Q \tag{P1}$$

We now show

$$\sim' \text{ is a } cd\text{-bisimulation} \tag{P2}$$

Proof of P2.

Let $P \sim' Q$. If we establish all the clauses of definition 12 (for $\mathcal{S} = \sim'$), then, by that definition, we can conclude that $\sim'$ is a *cd*-bisimulation. Clause 1 follows from (and is identical to) clause 1 of the definition of $\sim'$. For clause 2, let $a \in \alpha P - \{c\}$ and $P \xrightarrow{a} P'$. By the definition of $\sim'$, there exists Q' such that $Q \xrightarrow{a} Q'$ and $P' \sim Q'$. From $P' \sim Q'$ and P1, we have $P' \sim' Q'$. Since $Q \xrightarrow{a} Q'$ and $P' \sim' Q'$, clause 2 of

definition 12 (for $\mathcal{S} = \sim'$) is satisfied. Clauses 3–5 are verified in an identical manner. (end proof of P2).

Since $\sim'$ is a *cd*-bisimulation, and $\sim$ is the largest *cd*-bisimulation, we have $P \sim' Q$ implies $P \sim Q$. From this and P1, we get $P \sim' Q$ iff $P \sim Q$. Replacing $P \sim' Q$ by $P \sim Q$ in the definition of $\sim'$ then gives us proposition 2. □

3.2.2 Proof of Correctness

Proposition 3 *Let* $\mathcal{S} = \{(P, Q) \mid Q = P[c/c \rightarrow d]\}$, *then* $\mathcal{S}$ *is a cd-bisimulation.*

Proof: We establish each clause of definition 12 in turn. Let $P = (\| \ i \in \varphi : P_i), Q = (\| \ i \in \varphi : Q_i)$. Let G be an action expression. We shall use G_t for G with all occurrences of c substituted with $c \rightarrow d$.

Clause 1: $\alpha P \cup \{d\} = \alpha Q$, $d \notin \alpha P$, $c \in \alpha P$, $c \in \alpha Q$.

The assumption we make on the alphabets of P, Q whenever we write $Q = P[c/c \rightarrow d]$ (see page 14), is identical to this clause. Hence clause 1 holds by assumption.

Clause 2: for all $a \in \alpha P - \{c\}$, if $P \xrightarrow{a} P'$, then $Q \xrightarrow{a} Q'$ for some Q' such that $P' \mathcal{S} Q'$

Let $P \xrightarrow{a} P'$ for some $P' = (\| \ i \in \varphi : P'_i)$ and $a \in \alpha P - \{c\}$. We show that there exists a $Q' = (\| \ i \in \varphi : Q'_i)$ such that $Q \xrightarrow{a} Q'$ and $P' \mathcal{S} Q'$. By $P \xrightarrow{a} P'$ and the transition relation definition (7), $P_i \xrightarrow{a} P'_i$ for all $i \in PA_P(a)$, and $P_i = P'_i$ for all $i \in \varphi - PA_P(a)$. We have two cases.

Case 1: $i \in PA_Q(a)$.

Consider P_i, Q_i, P'_i for an arbitrary $i \in PA_Q(a)$. Since $P_i \xrightarrow{a}$, $\beta P_i = (a \rightarrow F \ \| \ G) \rightarrow *E$ as discussed above (page 9), for some action expressions F, G, E.

By the transition relation definition (7), rules **Act, Ch, Seq** applied in sequence, we

have

$$((a \rightarrow F \parallel G) \rightarrow *E) \xrightarrow{a} (F \rightarrow *E)$$

In general, P_i can have more than one a-derivative, since G itself could have the form $a \rightarrow F' \parallel G'$. However, there is no loss of generality in assuming that the execution of the a in $a \rightarrow F$ leads to P_i'. Hence we have $\beta P_i' = (F \rightarrow *E)$.

By $Q = P[c/c \rightarrow d]$ and the sequence introduction transformation definition (11), we have two subcases, $i \in PA_Q(a) \bigcap PA_Q(d)$ and $i \in PA_Q(a) - PA_Q(d)$.

Subcase 1.1: $i \in PA_Q(a) \bigcap PA_Q(d)$.

From $Q = P[c/c \rightarrow d]$, the sequence introduction transformation definition (11), and the subcase condition, we have $Q_i = P_i[c/c \rightarrow d]$. Since $\beta P_i = (a \rightarrow F \parallel G) \rightarrow *E$, we get $\beta Q_i = (a \rightarrow F_t \parallel G_t) \rightarrow *E_t$, by the sequence introduction transformation (11). By the transition relation definition (7), rules **Act, Ch, Seq**, we have

$$((a \rightarrow F_t \parallel G_t) \rightarrow *E_t) \xrightarrow{a} (F_t \rightarrow *E_t)$$

Since $\beta Q_i = (a \rightarrow F_t \parallel G_t) \rightarrow *E_t$, there exists a Q_i' such that $Q_i \xrightarrow{a} Q_i'$, and $\beta Q_i' = (F_t \rightarrow *E_t)$. Since $\beta P_i' = F \rightarrow *E$, we have $Q_i' = P_i'[c/c \rightarrow d]$ by the sequence introduction transformation (11).

Subcase 1.2: $i \in PA_Q(a) - PA_Q(d)$.

From $Q = P[c/c \rightarrow d]$, the sequence introduction transformation definition (11), and the subcase condition, we have $Q_i = P_i$. Letting Q_i' be P_i', we get $Q_i \xrightarrow{a} Q_i'$ and $Q_i = P_i$, since $P_i \xrightarrow{a} P_i'$.

Case 2: $i \in \varphi - PA_Q(a)$.

Consider P_i, Q_i, P_i' for an arbitrary $i \in \varphi - PA_Q(a)$. As before, by $Q = P[c/c \rightarrow d]$ and the sequence introduction transformation definition (11), we have two subcases, $i \in (\varphi - PA_Q(a)) - PA_Q(d)$ and $i \in (\varphi - PA_Q(a)) \bigcap PA_Q(d)$

Subcase 2.1: $i \in (\varphi - PA_Q(a)) - PA_Q(d)$.

From $Q = P[c/c \rightarrow d]$, the sequence introduction transformation definition (11), and the subcase condition, we have $Q_i = P_i$. Therefore $P_i' = Q_i'$ since neither of P_i, Q_i participate in the action a (and so $P_i = P_i', Q_i = Q_i'$).

Subcase 2.2: $i \in (\varphi - PA_Q(a)) \bigcap PA_Q(d)$.

From $Q = P[c/c \rightarrow d]$, the sequence introduction transformation definition (11), and the subcase condition, we have $Q_i = P_i[c/c \rightarrow d]$. Again neither of P_i, Q_i participate in a, and so $P_i' = P_i$ and $Q_i' = Q_i$. Hence, we have $Q_i' = P_i'[c/c \rightarrow d]$.

If we now consider $Q' = (\| \ i \in \varphi : Q_i')$, where the Q_i' are as given by the preceding case analysis, we see that $Q \xrightarrow{a} Q'$ by the transition relation definition (7), and also that $Q' = P'[c/c \rightarrow d]$, by the sequence introduction transformation definition (11), since the above two cases cover the entire indexing set φ. Since $\mathcal{S} = \{(P, Q) \mid Q = P[c/c \rightarrow d]\}$, we have $P' \mathcal{S} Q'$ as required.

Clause 3: if $P \xrightarrow{c} P'$, then $Q \xrightarrow{cd} Q'$ for some Q' such that $P' \mathcal{S} Q'$

Let $P \xrightarrow{c} P'$ for some $P' = (\| \ i \in \varphi : P_i')$. We show that there exists a $Q' = (\| \ i \in \varphi : Q_i')$ such that $Q \xrightarrow{cd} Q'$ and $P' \mathcal{S} Q'$. By $P \xrightarrow{c} P'$ and the transition relation definition (7), $P_i \xrightarrow{c} P_i'$ for all $i \in PA_P(c)$, and $P_i = P_i'$ for all $i \in \varphi - PA_P(c)$. We have three cases.

Case 1: $i \in PA_Q(d)$.

Consider P_i, Q_i, P_i' for an arbitrary $i \in PA_Q(d)$. By $Q = P[c/c \rightarrow d]$ and the sequence introduction transformation definition (11), we have $Q_i = P_i[c/c \rightarrow d]$. Since $P_i \xrightarrow{c}$, we have $\beta P_i = (c \rightarrow F \parallel G) \rightarrow *E$ as discussed above (page 9), for some action expressions F, G, E.

By the transition relation definition (7), rules **Act, Ch, Seq** applied in sequence, we have

$$((c \rightarrow F \parallel G) \rightarrow *E) \xrightarrow{c} (F \rightarrow *E)$$

In general, P_i can have more than one c-derivative, since G itself could have the form $c \rightarrow F' \parallel G'$. However, there is no loss of generality in assuming that the execution of the c in $c \rightarrow F$ leads to P_i'. Hence we have $\beta P_i' = F \rightarrow *E$.

Now $Q_i = P_i[c/c \rightarrow d]$. Since $\beta P_i = (c \rightarrow F \parallel G) \rightarrow *E$, we get $\beta Q_i = (c \rightarrow d \rightarrow F_t \parallel G_t) \rightarrow *E_t$, by the sequence introduction transformation (11). By the transition relation definition (7), rules **Act, Ch, Seq** applied in sequence, we have

$$((c \rightarrow d \rightarrow F_t \parallel G_t) \rightarrow *E_t) \xrightarrow{c} (d \rightarrow F_t \rightarrow *E_t)$$

By rules **Act, Seq**, we have

$$(d \rightarrow F_t \rightarrow *E) \xrightarrow{d} (F_t \rightarrow *E_t)$$

Hence, we conclude

$$((c \rightarrow d \rightarrow F_t \parallel G_t) \rightarrow *E_t) \xrightarrow{cd} (F_t \rightarrow *E_t)$$

Since $\beta Q_i = (c \rightarrow d \rightarrow F_t \parallel G_t) \rightarrow *E_t$, there exists a Q_i' such that $Q_i \xrightarrow{cd} Q_i'$, and $\beta Q_i' = F_t \rightarrow *E_t$. Since $\beta P_i' = F \rightarrow *E$, we have $Q_i' = P_i'[c/c \rightarrow d]$ by the sequence introduction transformation (11).

Case 2: $i \in PA_Q(c) - PA_Q(d)$.

Consider P_i, Q_i, P_i' for an arbitrary $i \in PA_Q(c) - PA_Q(d)$. By $Q = P[c/c \rightarrow d]$ and the sequence introduction transformation definition (11), we have $Q_i = P_i$. Letting Q_i' be P_i', we get $Q_i \xrightarrow{c} Q_i'$ and $Q_i' = P_i'$.

Case 3: $i \in \varphi - PA_Q(c)$.

Consider P_i, Q_i, P_i' for an arbitrary $i \in \varphi - PA_Q(c)$. By $Q = P[c/c \rightarrow d]$ and the

sequence introduction transformation definition (11), we have $Q_i = P_i$. Since $P_i = P_i'$ in this case, we let $Q_i' = Q_i$, and hence $Q_i' = P_i'$.

If we now consider $Q' = (\| \ i \in \varphi : Q_i')$, where the Q_i' are as given by the preceding case analysis, we see that $Q \xrightarrow{cd} Q'$ by the transition relation definition (7), and also that $Q' = P'[c/c \rightarrow d]$, by the sequence introduction transformation definition (11), since the above three cases cover the entire indexing set φ. Since $\mathcal{S} = \{(P, Q) \mid Q = P[c/c \rightarrow d]\}$, we have $P' \mathcal{S} Q'$ as required.

Clause 4: for all $a \in \alpha P - \{c\}$, if $Q \xrightarrow{a} Q'$, then $P \xrightarrow{a} P'$ for some P' such that $P' \mathcal{S} Q'$

Let $Q \xrightarrow{a} Q'$ for some $Q' = (\| \ i \in \varphi : Q_i')$ and $a \in \alpha P - \{c\}$. We show that there exists a $P' = (\| \ i \in \varphi : P_i')$ such that $P \xrightarrow{a} P'$ and $P' \mathcal{S} Q'$. By $Q \xrightarrow{a} Q'$ and the transition relation definition (7), $Q_i \xrightarrow{a} Q_i'$ for all $i \in PA_Q(a)$, and $Q_i = Q_i'$ for all $i \in \varphi - PA_Q(a)$. We have two cases.

Case 1: $i \in PA_Q(a)$.

Consider P_i, Q_i, Q_i' for an arbitrary $i \in PA_Q(a)$. From $Q = P[c/c \rightarrow d]$ and the sequence introduction transformation definition (11) we have two subcases, $i \in PA_Q(a) \bigcap PA_Q(d)$ and $i \in PA_Q(a) - PA_Q(d)$.

Subcase 1.1: $i \in PA_Q(a) \bigcap PA_Q(d)$.

From $Q = P[c/c \rightarrow d]$, the sequence introduction transformation definition (11), and the subcase condition, we have $Q_i = P_i[c/c \rightarrow d]$. Since $Q_i \xrightarrow{a}$, we have $\beta Q_i = (a \rightarrow F' \parallel G') \rightarrow *E'$ for some action expressions F', G', E', as discussed above (page 9). From $Q_i = P_i[c/c \rightarrow d]$ and the sequence introduction transformation definition (11), we conclude that $\beta P_i = (a \rightarrow F \parallel G) \rightarrow *E$ where action expressions E, F, G are such that $F' = F_t, G' = G_t, E' = E_t$.

By the transition relation definition (7), rules **Act, Ch, Seq** applied in sequence, we have

$$((a \rightarrow F_t \parallel G_t) \rightarrow *E_t) \xrightarrow{a} (F_t \rightarrow *E_t)$$

In general, Q_i can have more than one a-derivative, since G_t itself could have the form $a \rightarrow F'' \parallel G''$. However, there is no loss of generality in assuming that the execution of the a in $a \rightarrow F_t$ leads to Q_i'. Hence we have $\beta Q_i' = (F_t \rightarrow *E_t)$.

Now $\beta P_i = (a \rightarrow F \parallel G) \rightarrow *E$. By the transition relation definition (7), rules **Act, Ch, Seq**, we have

$$((a \rightarrow F \parallel G) \rightarrow *E) \xrightarrow{a} (F \rightarrow *E)$$

Since $\beta P_i = (a \rightarrow F \parallel G) \rightarrow *E$, there exists a P_i' such that $P_i \xrightarrow{a} P_i'$ and $\beta P_i' = (F \rightarrow *E)$. Since $\beta Q_i' = F_t \rightarrow *E_t$, we have $Q_i' = P_i'[c/c \rightarrow d]$ by the sequence introduction transformation (11).

Subcase 1.2: $i \in PA_Q(a) - PA_Q(d)$.

From $Q = P[c/c \rightarrow d]$, the sequence introduction transformation definition (11), and the subcase condition, we have $Q_i = P_i$. Letting P_i' be Q_i', we get $P_i \xrightarrow{a} P_i'$ and $P_i' = Q_i'$.

Case 2: $i \in \varphi - PA_Q(a)$.

Consider P_i, Q_i, P_i' for an arbitrary $i \in \varphi - PA_Q(a)$. By $Q = P[c/c \rightarrow d]$ and the sequence introduction transformation definition (11), we have two subcases, $i \in (\varphi - PA_Q(a)) \bigcap PA_Q(d)$ and $i \in (\varphi - PA_Q(a)) - PA_Q(d)$.

Subcase 2.1: $i \in (\varphi - PA_Q(a)) \bigcap PA_Q(d)$.

From $Q = P[c/c \rightarrow d]$, the sequence introduction transformation definition (11), and the subcase condition, we have $Q_i = P_i[c/c \rightarrow d]$. Since $i \in \varphi - PA_Q(a)$ these Q_i do not participate in action a. Therefore we have $Q_i' = Q_i$. We also have $P_i' = P_i$

since these P_i do not participate in action a either. Since $Q_i = P_i[c/c \rightarrow d]$ we have $Q'_i = P'_i[c/c \rightarrow d]$

Subcase 2.2: $i \in (\varphi - PA_Q(a)) - PA_Q(d)$.

From $Q = P[c/c \rightarrow d]$, the sequence introduction transformation definition (11), and the subcase condition, we have $Q_i = P_i$. Since $i \in \varphi - PA_Q(a)$ these Q_i do not participate in action a. Letting P'_i be Q'_i, we have $Q_i = P_i = Q'_i = P'_i$.

If we now consider $P' = (\| \ i \in \varphi : P'_i)$, where the P'_i are as given by the preceding case analysis, we see that $P \xrightarrow{a} P'$ by the transition relation definition (7), and also that $Q' = P'[c/c \rightarrow d]$, by the sequence introduction transformation definition (11), since the above two cases cover the entire indexing set φ. Since $\mathcal{S} = \{(P, Q) \mid Q = P[c/c \rightarrow d]\}$, we have $P' \mathcal{S} Q'$ as required.

Clause 5: if $Q \xrightarrow{cd} Q'$, then $P \xrightarrow{c} P'$ for some P' such that $P' \mathcal{S} Q'$

Let $Q \xrightarrow{cd} Q'$ for some $Q' = (\| \ i \in \varphi : Q'_i)$. We show that there exists a $P' = (\| \ i \in \varphi : P'_i)$ such that $P \xrightarrow{c} P'$ and $P' \mathcal{S} Q'$. By $Q \xrightarrow{cd} Q'$ and the transition relation definition (7), $Q_i \xrightarrow{cd} Q'_i$ for all $i \in PA_Q(d)$, $Q_i \xrightarrow{c} Q'_i$ for all $i \in PA_Q(c) - PA_Q(d)$ and $Q_i = Q'_i$ for all $i \in \varphi - PA_Q(c)$. We have three cases.

Case 1: $i \in PA_Q(d)$

Consider Q_i, P_i, Q'_i for an arbitrary $i \in PA_Q(d)$. From $Q = P[c/c \rightarrow d]$, the sequence introduction transformation definition (11), and the subcase condition, we have $Q_i = P_i[c/c \rightarrow d]$. Hence, d cannot occur as an operand of $\|$ in Q_i. From this and $Q_i \xrightarrow{cd} Q'_i$, we have $\beta Q_i = (c \rightarrow d \rightarrow F' \| G') \rightarrow *E'$ for some action expressions F', G', E', as discussed above (page 9). From $Q_i = P_i[c/c \rightarrow d]$ and the sequence introduction transformation definition (11), we conclude that $\beta P_i = (c \rightarrow F \| G) \rightarrow *E$ where action expressions E, F, G are such that $F' = F_t, G' = G_t, E' = E_t$.

By the transition relation definition (7), rule **Act**, we have

$$((c \rightarrow d \rightarrow F_t \parallel G_t) \rightarrow *E_t) \xrightarrow{c} (d \rightarrow F_t \rightarrow *E_t)$$

By rule **Act**, we have

$$(d \rightarrow F_t \rightarrow *E_t) \xrightarrow{d} (F_t \rightarrow *E_t)$$

Hence,

$$(\mathbf{c} \rightarrow d \rightarrow F_t \parallel G_t) \rightarrow *E_t \xrightarrow{cd} (F_t \rightarrow *E_t)$$

In general, Q_i can have more than one *cd*-derivative, since G_t itself could have the form $c \rightarrow d \rightarrow F'' \parallel G''$. However, there is no loss of generality in assuming that the execution of the c and then d in $c \rightarrow d \rightarrow F_t$ leads to Q_i'. Hence we have $\beta Q_i' = (F_t \rightarrow *E_t)$.

Now $\beta P_i = (c \rightarrow d \rightarrow F \parallel G) \rightarrow *E$. By the transition relation definition (7), rules **Act, Ch, Seq**, we have

$$((c \rightarrow F \parallel G) \rightarrow *E) \xrightarrow{c} (F \rightarrow *E)$$

Since $\beta P_i = (c \rightarrow F \parallel G) \rightarrow *E$, there exists a P_i' such that $P_i \xrightarrow{c} P_i'$ and $\beta P_i' = F \rightarrow *E$. Since $\beta Q_i' = F_t \rightarrow *E_t$, we have $Q_i' = P_i'[c/c \rightarrow d]$ by the sequence introduction transformation (11).

Case 2: $i \in PA_Q(c) - PA_Q(d)$.

Consider P_i, Q_i, Q_i' for an arbitrary $i \in PA_Q(c) - PA_Q(d)$. By $Q = P[c/c \rightarrow d]$ and the sequence introduction transformation definition (11), we have $Q_i = P_i$. Letting P_i' be Q_i', we get $P \xrightarrow{c} P'$ since $Q \xrightarrow{c} Q'$.

Case 3: $i \in \varphi - PA_Q(c)$.

Consider Q_i, P_i, Q_i' for an arbitrary $i \in \varphi - PA_Q(c)$. By $Q = P[c/c \rightarrow d]$ and the sequence introduction transformation definition (11), we have $Q_i = P_i$. Now $Q_i = Q_i'$ in this case as explained above. Letting $P_i' = P_i$ we get $P_i' = Q_i'$.

If we now consider $P' = (\| \ i \in \varphi : P'_i)$, where the P'_i are as given by the preceding case analysis, we see that $P \xrightarrow{c} P'$ by the transition relation definition (7), and also that $Q' = P'[c/c \rightarrow d]$, by the sequence introduction transformation definition (11), since the above three cases cover the entire indexing set φ. Since $\mathcal{S} = \{(P, Q) \mid Q = P[c/c \rightarrow d]\}$, we have $P' \, \mathcal{S} \, Q'$ as required.

We have shown that all five clauses of definition 12 hold, hence $\mathcal{S}$ is a cd-bisimulation. □

Proposition 4 *Let $Q = P[c/c \rightarrow d]$. If Q' is an arbitrary derivative of Q, then either $Q' \xrightarrow{d}$, or there exists a derivative P' of P such that $P' \sim Q'$.*

Proof: Let $\mathcal{S} = \{(P, Q) \mid Q = P[c/c \rightarrow d]\}$. By proposition 3, $\mathcal{S}$ is a *cd*-bisimulation. Hence $\mathcal{S} \subseteq \sim$ by definition 13. Since $P \, \mathcal{S} \, Q$ by assumption, we conclude $P \sim Q$. Since Q' is a derivative of Q, we have $Q \xrightarrow{a_1} \cdots \xrightarrow{a_n} Q'$ for some path $\pi = a_1, \ldots, a_n$ of length n. Now π will contain some number (possibly 0) of occurrences of c. Furthermore, from the syntactic form of Q and the transition relation (definition 7), we have that every pair of successive occurrences of c along π have exactly one occurrence of d between them, and likewise every pair of successive occurrences of d along π have exactly one occurrence of c between them. Furthermore, the first occurrence of c in π (if any) is not preceded by an occurrence of d.

Suppose that d does not occur in the suffix of π starting with the last occurrence of c in π. Then, from the syntactic form of Q, the transition relation (definition 7), and $Q \xrightarrow{\pi} Q'$, we conclude $Q' \xrightarrow{d}$, and we are done. Hence we can assume in the rest of the proof that there is exactly one occurrence of d in the suffix of π starting with the last occurrence of c in π. In other words, every occurrence of c can be matched with the subsequent occurrence of d. Consider a segment of π starting with some

arbitrary occurrence of c in π and ending with the matching occurrence of d. This segment has the form $c, b_1, \ldots, b_m, d$ where none of $b_1, \ldots, b_n$ is either c or d.

From $Q = P[c/c \rightarrow d]$, we have that any $Q_i \in PA_d(Q)$ which executes c can only execute d as its next action; it has no other choice. Thus we conclude that no $Q_i \in PA_d(Q)$ participates in any of the actions $b_2, b_2, \ldots, b_m$. Thus, by definition 8, d and b_j are independent, for $1 \leq j \leq m$. Hence, by definition 10, the paths $c, b_1, b_2, \ldots, b_m, d$ and $c, d, b_1, b_2, \ldots, b_m$ are equivalent paths. Let ρ be the path obtained from π by replacing every segment of the form $c, b_1, \ldots, b_m, d$ by the segment $c, d, b_1, b_2, \ldots, b_m$. From the definition of equivalence (def. 10), we easily see that equivalence is preserved by path-concatenation. Hence ρ is equivalent to π. Thus, by proposition 1 and $Q \xrightarrow{\pi} Q'$, we have $Q \xrightarrow{\rho} Q'$. We now prove:

there exists a derivative P' of P such that $P' \sim Q'$ (P1)

Proof of P1.

The proof is by induction of the length ℓ of ρ.

Base Case, $\ell = 0$

Hence ρ is the empty path, and so $Q' = Q$. Let P' be P. By assumption, $Q = P[c/c \rightarrow d]$. Thus by proposition 3, $P \sim Q$. Hence $P' \sim Q'$.

Induction Step, $\ell = n + 1$

We assume the inductive hypothesis for $\ell \leq n$ $(n \geq 0)$ and establish P1 for $\ell = n + 1$. Since ρ has length $n + 1$, it can be written as $a_1, \ldots, a_n, a_{n+1}$. Since every c in ρ is immediately followed by a d in ρ, a_{n+1} cannot be c. Thus we have two cases.

Case 1: $a_{n+1} = d$.

Hence $a_n = c$. Thus $\rho = a_1, \ldots, c, d$. From this and $Q \xrightarrow{\rho} Q'$, there is a Q'' be such that $Q \xrightarrow{a_1} \cdots \xrightarrow{a_{n-1}} Q''$ and $Q'' \xrightarrow{cd} Q'$. Assuming the inductive hypothesis for $\ell = n - 1$,

we have that there exists a derivative P'' of P such that $P'' \sim Q''$. From this, $Q'' \xrightarrow{cd} Q'$, and proposition 2, we have $P'' \xrightarrow{c} P'$ for some P' such that $P' \sim Q'$. Since P' is a derivative of P, P1 is established in this case.

Case 2: $a_{n+1} \neq d$.

From $\rho = a_1, \ldots, a_n, a_{n+1}$ and $Q \xrightarrow{\rho} Q'$, there is a Q'' be such that $Q \xrightarrow{a_1} \cdots \xrightarrow{a_n} Q''$ and $Q'' \xrightarrow{a_{n+1}} Q'$. Assuming the inductive hypothesis for $\ell = n$, we have that there exists a derivative P'' of P such that $P'' \sim Q''$. From this, $Q'' \xrightarrow{a_{n+1}} Q'$, and proposition 2, we have $P'' \xrightarrow{a_{n+1}} P'$ for some P' such that $P' \sim Q'$. Since P' is a derivative of P, P1 is established in this case, which completes the induction step.

(end proof of P1)

Since P1 implies the proposition, we are done. □

As stated in the introduction, our aim is to design syntactic transformations which automatically preserve desirable program properties. We now show that the right-sequence introduction transformation preserves deadlock-freedom.

Definition 14 (*Deadlock-Freedom*)

If for every derivative P' of P, there is some action a such that $P' \xrightarrow{a}$, then P is deadlock-free.

Theorem 1 *Let $Q = P[c/c \rightarrow d]$. If P is deadlock-free, then so is Q.*

Proof: Let Q' be an arbitrary derivative of Q. By definition 14, it suffices to show $Q' \xrightarrow{a}$ for some action a. From proposition 3 and $Q = P[c/c \rightarrow d]$, we have $P \sim Q$. Thus, by proposition 4, we have

either $Q' \xrightarrow{d}$, or there exists a derivative P' of P such that $P' \sim Q'$

If $Q' \xrightarrow{d}$ then we are done. Otherwise, there exists a derivative P' of P such that $P' \sim Q'$. By assumption, P is deadlock-free. Hence, by definition 14, $P' \xrightarrow{a'}$ for some

a'. If $a' \neq c$, then $Q' \xrightarrow{a'}$ by proposition 2. If $a' = c$, then $Q' \xrightarrow{cd}$, again by proposition 2. Hence in all cases we have $Q' \xrightarrow{a}$ for some a. □

3.3 The Left-Sequence Introduction Transformation

Our second transformation allows us to introduce a new action, d, in sequence with, and immediately before, an already-present action, c. Intuitively, we are again using the transformation to refine c.

Definition 15 (*Left-sequence Introduction Transformation*)

We define the left-sequence introduction transformation $[c/d \rightarrow c]$ *in a bottom-up manner as follows. Let* a *be an arbitrary action, and* E, F *be arbitrary action expressions. Then, we have*

$$\varepsilon[c/d \rightarrow c] = \varepsilon$$

$$0[c/d \rightarrow c] = 0$$

$$a[c/d \rightarrow c] = a \text{ if } a \neq c$$

$$c[c/d \rightarrow c] = d \rightarrow c$$

$$(E \parallel F)[c/d \rightarrow c] = ((E[c/d \rightarrow c]) \parallel (F[c/d \rightarrow c]))$$

$$(E \rightarrow F)[c/d \rightarrow c] = ((E[c/d \rightarrow c]) \rightarrow (F[c/d \rightarrow c]))$$

In the sequel, we will use the abbreviation E_t *for* $E[c/d \rightarrow c]$ *for an arbitrary action expression* E.

For an arbitrary process P_i *such that* $c \in \alpha P_i$, *and* $\beta P_i = F \rightarrow *E$ *for some action expressions* F, E, *define* $P_i[c/d \rightarrow c] = Q_i$, *where* $\alpha Q_i = \alpha P_i \cup \{d\}$, $\beta Q_i = F_t \rightarrow *E_t$. *In addition, if* $\beta P_i = c \rightarrow F \rightarrow *E$, *for some action expressions* F, E, *we define* $P_i[c/d \rightarrow c, nl] = Q_i$, *where* $\alpha Q_i = \alpha P_i \cup \{d\}$, $\beta Q_i = c \rightarrow F_t \rightarrow *E_t$, *i.e., the*

leading c action is not preceded by a d action (this is needed for technical reasons).

Let $P = (\ \|\ i \in \varphi : P_i)$ *be an arbitrary program such that* $c \in \alpha P$, $d \notin \alpha P$ *for actions* c, d. *Let* ψ *be an arbitrary non-empty subset of* $PA_P(c)$ *such that for all* $i \in \psi$ *no occurrence of* c *in* P_i *is the operand of* $[\!]$. *We define* $P[c/d \rightarrow c] = (\ \|\ i \in \psi : P_i[c/d \rightarrow c])\ \|\ (\ \|\ i \in \varphi - \psi : P_i)$. *If* $P \xrightarrow{c}$, *then we also define* $P[c/d \rightarrow c, nl] = (\ \|\ i \in \psi : P_i[c/d \rightarrow c, nl])\ \|\ (\ \|\ i \in \varphi - \psi : P_i)$.

If $Q = P[c/d \rightarrow c]$ or $Q = P[c/d \rightarrow c, nl]$, then we say that Q results from P by means of a *left-sequence introduction transformation.* The left-sequence introduction transformation $[c/d \rightarrow c]$ takes a program P containing an action c such that c is not involved in a choice, and introduces a new action d before c and in sequence with c. In the sequel, whenever we write $Q = P[c/d \rightarrow c]$ or $Q = P[c/d \rightarrow c, nl]$, we shall implicitly assume that the conditions $\alpha P \cup \{d\} = \alpha Q$, $d \notin \alpha P$, $c \in \alpha P$, $c \in \alpha Q$ are all true.

3.3.1 Discussion and Preliminary Technical Definitions

From the transition rules we conclude that if a left-sequence introduction transformation were applied to a process body where c is in a choice with other actions, it could lead to a deadlock. For example consider the following program:

$$*(e \rightarrow c)\ \|\ *(e\ [\!]\ c)$$

The above program is deadlock-free. Applying a left-sequence introduction transformation to this program, we obtain:

$$*(e \rightarrow c)\ \|\ *(e\ [\!]\ d \rightarrow c)$$

Any path of the form $(edc)^*d$ ends in a deadlock state. This is why we do not allow occurrences of c to be operands of $[\!]$ in the definition of the left-sequence introduction

transformation. We can therefore assume the following form for the body of a process which is ready to execute c: $(c \rightarrow F) \rightarrow *E$.

However, if a process is ready to execute an action a other than c, then a can be in a choice with other actions and therefore the same analysis as in page 9 follows. Thus if a is one of the next possible actions and $a \neq c$ then the general form for the body of a process is $(a \rightarrow F \parallel G) \rightarrow *E$.

We now introduce the notion of dc-bisimulation, which will be used to relate programs P, Q such that $Q = P[c/d \rightarrow c]$ in much the same way that cd-bisimulation was used to relate programs P, Q such that $Q = P[c/c \rightarrow d]$ in the previous subsection.

Definition 16 *Let $\mathcal{S}$ be a binary relation over programs. Then $\mathcal{S}$ is a* dc-bisimulation *iff $P \mathcal{S} Q$ implies:*

1. $\alpha P \cup \{d\} = \alpha Q$, $d \notin \alpha P$, $c \in \alpha P$, $c \in \alpha Q$
2. *for all* $a \in \alpha P - \{c\}$, *if* $P \xrightarrow{a} P'$, *then* $Q \xrightarrow{a} Q'$ *for some* Q' *such that* $P' \mathcal{S} Q'$
3. *if* $P \xrightarrow{c} P'$, *then*

   either $Q \xrightarrow{dc} Q'$ *for some* Q' *such that* $P' \mathcal{S} Q'$

   or $Q \xrightarrow{c} Q'$ *for some* Q' *such that* $P' \mathcal{S} Q'$
4. *for all* $a \in \alpha P - \{c\}$, *if* $Q \xrightarrow{a} Q'$, *then* $P \xrightarrow{a} P'$ *for some* P' *such that* $P' \mathcal{S} Q'$
5. *if* $Q \xrightarrow{d} Q'$, *then* $P \mathcal{S} Q'$, *and*

   if $Q \xrightarrow{c} Q'$, *then* $P \xrightarrow{c} P'$ *for some* P' *such that* $P' \mathcal{S} Q'$

Definition 17 *Following the treatment of strong bisimulation given in chapter 4 of [Mil89], we define*

$$\sim \stackrel{\text{df}}{=} \bigcup\{\mathcal{S} \mid \mathcal{S} \text{ is a } dc\text{-bisimulation}\}.$$

From this, we can establish

Proposition 5 *$P \sim Q$ iff*

1. *$\alpha P \cup \{d\} = \alpha Q$, $d \notin \alpha P$, $c \in \alpha P$, $c \in \alpha Q$*

2. *for all $a \in \alpha P - \{c\}$, if $P \xrightarrow{a} P'$, then $Q \xrightarrow{a} Q'$ for some Q' such that $P' \sim Q'$*

3. *if $P \xrightarrow{c} P'$, then*

   either $Q \xrightarrow{dc} Q'$ for some Q' such that $P' \sim Q'$

   or $Q \xrightarrow{c} Q'$ for some Q' such that $P' \sim Q'$

4. *for all $a \in \alpha P - \{c\}$, if $Q \xrightarrow{a} Q'$, then $P \xrightarrow{a} P'$ for some P' such that $P' \sim Q'$*

5. *if $Q \xrightarrow{d} Q'$, then $P \sim Q'$, and*

   if $Q \xrightarrow{c} Q'$, then $P \xrightarrow{c} P'$ for some P' such that $P' \sim Q'$

Proof: The proof proceeds in the same way as the proof of proposition 4 in chapter 4 of [Mil89] (bearing in mind that the notions of bisimulation differ technically).

3.3.2 Proof of Correctness

Proposition 6 *Let $\mathcal{S} = \{(P,Q) \mid Q = P[c/d \rightarrow c]$ or $Q = P[c/d \rightarrow c,nl]\}$. Then $\mathcal{S}$ is a dc-bisimulation.*

Proof: Our approach to this proof shall be analogous to that for proposition 3. As before we shall use P_i to represent processes of program P and P'_i to represent processes of program P'. Similarly for Q_i and Q'_i. We observe that if ($Q = P[c/d \rightarrow c]$ or $Q = P[c/d \rightarrow c,nl]$) and no occurrence of c in P is an operand of $\|$, then, by the left sequence introduction transformation definition (15), no occurrence of c in Q will be an operand of $\|$.

Clause 1: $\alpha P \cup \{d\} = \alpha Q$, $d \notin \alpha P$, $c \in \alpha P$, $c \in \alpha Q$.

The assumption we make on the alphabets of P, Q whenever we write ($Q = P[c/d \rightarrow c]$ or $Q = P[c/d \rightarrow c, nl]$), (see page 29), is identical to this clause. Hence clause 1 holds by assumption.

Clause 2: for all $a \in \alpha P - \{c\}$, if $P \xrightarrow{a} P'$, then $Q \xrightarrow{a} Q'$ for some Q' such that $P' \mathcal{S} Q'$

Let $P \xrightarrow{a} P'$ for some $P' = (\| \ i \in \varphi : P_i')$ and $a \in \alpha P - \{c\}$. We show that there exists a $Q' = (\| \ i \in \varphi : Q_i')$ such that $Q \xrightarrow{a} Q'$ and $P' \mathcal{S} Q'$. By $P \xrightarrow{a} P'$ and the transition relation definition (7), $P_i \xrightarrow{a} P_i'$ for all $i \in PA_P(a)$, and $P_i = P_i'$ for all $i \in \varphi - PA_P(a)$. We have two cases.

Case 1: $i \in PA_Q(a)$.

Consider P_i, Q_i, P_i' for an arbitrary $i \in PA_Q(a)$. Since $P_i \xrightarrow{a}$, βP_i must have the form $(a \rightarrow F \ \| \ G) \rightarrow *E$ as discussed above (page 29), for some action expressions F, G, E. By the transition relation definition (7), rules **Act, Ch, Seq** applied in sequence, we have

$$((a \rightarrow F \ \| \ G) \rightarrow *E) \xrightarrow{a} (F \rightarrow *E)$$

In general, P_i can have more than one a-derivative, since G itself could have the form $a \rightarrow F' \ \| \ G'$. However, there is no loss of generality in assuming that the execution of the action a in $a \rightarrow F$ leads to P_i'. Hence we have $\beta P_i' = (F \rightarrow *E)$.

We have two subcases, $i \in PA_Q(a) \bigcap PA_Q(d)$ and $i \in PA_Q(a) - PA_Q(d)$.

Subcase 1.1: $i \in PA_Q(a) \bigcap PA_Q(d)$.

Now $P_i \overset{c}{\not\rightarrow}$ for all $i \in PA_P(d)$, since c is never the operand of $\|$ in $P_i, i \in PA_Q(d)$. From $P_i \overset{c}{\not\rightarrow}$ and definition 15, we get $Q_i \neq P_i[c/d \rightarrow c, nl]$. From ($Q = P[c/d \rightarrow c]$ or $Q = P[c/d \rightarrow c, nl]$), definition 15, and $i \in PA_Q(a) \bigcap PA_Q(d)$, we get ($Q_i = P_i[c/d \rightarrow c]$ or $Q_i = P_i[c/d \rightarrow c, nl]$). Hence $Q_i = P_i[c/d \rightarrow c]$. Since $\beta P_i = (a \rightarrow$

$F \parallel G) \rightarrow *E$, we get $\beta Q_i = (a \rightarrow F_t \parallel G_t) \rightarrow *E_t$, by the sequence introduction transformation (15). By the transition relation definition (7), rules **Act, Ch, Seq,** we have

$$((a \rightarrow F_t \parallel G_t) \rightarrow *E_t) \xrightarrow{a} (F_t \rightarrow *E_t)$$

Since $\beta Q_i = (a \rightarrow F_t \parallel G_t) \rightarrow *E_t$, there exists a Q_i' such that $Q_i \xrightarrow{a} Q_i'$, and $\beta Q_i' = F_t \rightarrow *E_t$. Since $\beta P_i' = F \rightarrow *E$, we have $Q_i' = P_i'[c/d \rightarrow c]$ by the sequence introduction transformation (15).

Subcase 1.2: $i \in PA_Q(a) - PA_Q(d)$.

From ($Q \in P[c/d \rightarrow c]$ or $Q \in P[c/d \rightarrow c, nl]$), the sequence introduction transformation definition (15), and the subcase condition, we have $Q_i = P_i$. Letting Q_i' be P_i', we get $Q_i \xrightarrow{a} Q_i'$ and $Q_i' = P_i'$, since $P_i \xrightarrow{a} P_i'$.

Case 2: $i \in \varphi - PA_Q(a)$.

Consider P_i, Q_i, P_i' for an arbitrary $i \in \varphi - PA_Q(a)$. We have two subcases, $i \in (\varphi - PA_Q(a)) - PA_Q(d)$ and $i \in (\varphi - PA_Q(a)) \bigcap PA_Q(d)$

Subcase 2.1: $i \in (\varphi - PA_Q(a)) - PA_Q(d)$.

From ($Q = P[c/d \rightarrow c]$ or $Q \in P[c/d \rightarrow c, nl]$), the sequence introduction transformation definition (15), and the subcase condition, we have $Q_i = P_i$. Therefore $P_i' = Q_i'$ since neither of P_i, Q_i participate in the action a (and so $P_i = P_i', Q_i = Q_i'$).

Subcase 2.2: $i \in (\varphi - PA_Q(a)) \bigcap PA_Q(d)$.

From ($Q = P[c/d \rightarrow c]$ or $Q = P[c/d \rightarrow c, nl]$), the sequence introduction transformation definition (15), and the subcase condition, we have ($Q_i = P_i[c/d \rightarrow c]$ or $Q_i = P_i[c/d \rightarrow c, nl]$). Again neither of P_i, Q_i participate in a, and so $P_i' = P_i$ and $Q_i' = Q_i$. Hence, we have ($Q_i' = P_i'[c/d \rightarrow c]$ or $Q_i' = P_i'[c/d \rightarrow c, nl]$).

(end of Case 2)

Now suppose $PA_Q(a) \bigcap PA_Q(d) \neq \emptyset$. Then, by subcase 1.1, $Q'_i = P'_i[c/d \rightarrow c]$ for some $i \in PA_Q(a) \bigcap PA_Q(d)$. Thus, by all the subcases above and definition 15, we conclude $Q' = P'[c/d \rightarrow c]$. On the other hand, if $PA_Q(a) \bigcap PA_Q(d) = \emptyset$, then $(\varphi - PA_Q(a)) \bigcap PA_Q(d) = PA_Q(d)$, and therefore, by subcases 1.2, 2.1, 2.2, and definition 15, we conclude $(Q' = P'[c/d \rightarrow c]$ or $Q' = P'[c/d \rightarrow c, nl])$. Thus we have $P \mathcal{S} Q$ in all cases.

If we now consider $Q' = (\| \ i \in \varphi : Q'_i)$, where the Q'_i are as given by the preceding case analysis, we see that $Q \xrightarrow{a} Q'$ by the transition relation definition (7). Hence clause 2 is satisfied.

Clause 3: if $P \xrightarrow{c} P'$, then either $Q \xrightarrow{dc} Q'$ for some Q' such that $P' \mathcal{S} Q'$ or $Q \xrightarrow{c} Q'$ for some Q' such that $P' \mathcal{S} Q'$

Let $P \xrightarrow{c} P'$ for some $P' = (\| \ i \in \varphi : P'_i)$. By $P \xrightarrow{c} P'$ and the transition relation definition (7), $P_i \xrightarrow{c} P'_i$ for all $i \in PA_Q(d)$, $P_i \xrightarrow{c} P'_i$ for all $i \in PA_Q(c) - PA_Q(d)$ and $P_i = P'_i$ for all $i \in \varphi - PA_Q(c)$. We have two cases.

Case 1: $Q \xrightarrow{d}$.

We show that there exists a $Q' = (\| \ i \in \varphi : Q'_i)$ such that $Q \xrightarrow{dc} Q'$ and $P' \mathcal{S} Q'$.

SubCase 1.1: $i \in PA_Q(d)$.

Consider P_i, Q_i, P'_i for an arbitrary $i \in PA_Q(d)$. By $(Q = P[c/d \rightarrow c]$ or $Q = P[c/d \rightarrow c, nl])$, the sequence introduction transformation definition (15), and $i \in PA_Q(d)$, we get $(Q_i = P_i[c/d \rightarrow c]$ or $Q_i = P_i[c/d \rightarrow \mathbf{c}, nl])$. From $Q \xrightarrow{d}$, we get $Q_i \xrightarrow{d}$. Hence $Q_i \neq P_i[c/d \rightarrow c, nl]$. Thus $Q_i = P_i[c/d \rightarrow c]$. Since $P_i \xrightarrow{c}$ and $i \in PA_Q(d)$, βP_i must have the form $(c \rightarrow F) \rightarrow *E$ as discussed above (page 29).

By the transition relation definition (7), rule **Act, Seq** , we have

$$((c \rightarrow F) \rightarrow *E) \xrightarrow{c} (F \rightarrow *E)$$

Hence we have

$$\beta P_i' = (F \rightarrow *E).$$

Now $Q_i = P_i[c/d \rightarrow c]$. Since $\beta P_i = (c \rightarrow F) \rightarrow *E$, we get $\beta Q_i = *(d \rightarrow c \rightarrow F_t) \rightarrow *E_t$, by the sequence introduction transformation (15). By the transition relation definition (7), rule **Act**, we have

$$((d \rightarrow c \rightarrow F_t) \rightarrow *E_t) \xrightarrow{d} ((c \rightarrow F_t) \rightarrow *E_t)$$

By rule **Act**, we have

$$(c \rightarrow F_t \rightarrow *E_t) \xrightarrow{c} (F_t \rightarrow *E_t)$$

Hence, we conclude

$$((d \rightarrow c \rightarrow F_t) \rightarrow *E_t) \xrightarrow{dc} (F_t \rightarrow *E_t)$$

Since $\beta Q_i = ((d \rightarrow c \rightarrow F_t) \rightarrow *E_t)$, there exists a Q_i' such that $Q_i \xrightarrow{dc} Q_i'$, and $\beta Q_i' = (F_t \rightarrow *E_t)$. Since $\beta P_i' = (F \rightarrow *E)$, we have $Q_i' = P_i'[c/d \rightarrow c]$ by the sequence introduction transformation (15).

SubCase 1.2: $i \in PA_Q(c) - PA_Q(d)$.

Consider P_i, Q_i, P_i' for an arbitrary $i \in PA_Q(c) - PA_Q(d)$. By ($Q = P[c/d \rightarrow c]$ or $Q = P[c/d \rightarrow c, nl]$) and the sequence introduction transformation definition (15), we have $Q_i = P_i$. Letting Q_i' be P_i', we get $Q_i \xrightarrow{c} Q_i'$ and $Q_i' = P_i'$.

SubCase 1.3: $i \in \varphi - PA_Q(c)$.

Consider P_i, Q_i, P_i' for an arbitrary $i \in \varphi - PA_Q(c)$. By ($Q = P[c/d \rightarrow c]$ or $Q = P[c/d \rightarrow c, nl]$) and the sequence introduction transformation definition (15), we have $Q_i = P_i$. Since $P_i = P_i'$ in this case, we let $Q_i' = Q_i$, and hence $Q_i' = P_i'$.

If we now consider $Q' = (\| \ i \in \varphi : Q_i')$, where the Q_i' are as given by the preceding case analysis, we see that $Q \xrightarrow{dc} Q'$ by the transition relation definition (7), and also that $Q' = P'[c/d \rightarrow c]$, by the sequence introduction transformation def-

inition (15), since the above three cases cover the entire indexing set φ. Since $\mathcal{S} = \{(P, Q) \mid Q = P[c/d \rightarrow c] \text{ or } Q = P[c/d \rightarrow c, nl]\}$, we have $P' \mathcal{S} Q'$ as required.

Case 2: $Q \overset{d}{\not\rightarrow}$.

We show that there exists a $Q' = (\| \ i \in \varphi : Q'_i)$ such that $Q \xrightarrow{c} Q'$ and $P' \mathcal{S} Q'$. By $P \xrightarrow{c} P'$ and the transition relation definition (7), $P_i \xrightarrow{c} P'_i$ for all $i \in PA_Q(d)$, $P_i \xrightarrow{c} P'_i$ for all $i \in PA_Q(c) - PA_Q(d)$, $P_i = P'_i$ for all $i \in \varphi - PA_Q(c)$. Furthermore, from $Q \overset{d}{\not\rightarrow}$, we have $Q_j \overset{d}{\not\rightarrow}$ for some $j \in PA_Q(d)$. We have three subcases.

SubCase 2.1: $i \in PA_Q(d)$.

Consider P_i, Q_i, P'_i for an arbitrary $i \in PA_Q(d)$. By ($Q = P[c/d \rightarrow c]$ or $Q = P[c/d \rightarrow c, nl]$), the sequence introduction transformation definition (15), $P_j \xrightarrow{c}$, and $Q_j \overset{d}{\not\rightarrow}$, we have $Q_j = P_j[c/d \rightarrow c, nl]$. Hence $Q \neq P[c/d \rightarrow c]$. Thus $Q = P[c/d \rightarrow c, nl]$. We therefore conclude $Q_i = P_i[c/d \rightarrow c, nl]$ (for all $i \in PA_Q(d)$).

Since $P_i \xrightarrow{c}$ and $i \in PA_Q(d)$, βP_i must have the form $(c \rightarrow F) \rightarrow *E$ as discussed above (page 29). By the transition relation definition (7), rule **Act, Seq**, we have

$$((c \rightarrow F) \rightarrow *E) \xrightarrow{c} (F \rightarrow *E)$$

Hence we have

$$\beta P'_i = (F \rightarrow *E).$$

Now $Q_i = P_i[c/d \rightarrow c, nl]$. Since $\beta P_i = (c \rightarrow F) \rightarrow *E$, we have $\beta Q_i = (c \rightarrow F_t) \rightarrow *E_t$, by the sequence introduction transformation (15). By the transition relation definition (7), rule **Act**, we have

$$(c \rightarrow F_t \rightarrow *E_t) \xrightarrow{c} (F_t \rightarrow *E_t)$$

Since $\beta Q_i = ((c \rightarrow F_t) \rightarrow *E_t)$, there exists a Q'_i such that $Q_i \xrightarrow{c} Q'_i$, and $\beta Q'_i = (F_t \rightarrow *E_t)$. Since $\beta P'_i = (F \rightarrow *E)$, we have $Q'_i = P'_i[c/d \rightarrow c]$ by the sequence introduction transformation (15).

SubCase 2.2: $i \in PA_Q(c) - PA_Q(d)$.

Consider P_i, Q_i, P_i' for an arbitrary $i \in PA_Q(d)$. By ($Q = P[c/d \rightarrow c]$ or $Q = P[c/d \rightarrow c, nl]$) and the sequence introduction transformation definition (15), we have $Q_i = P_i$. Letting Q_i' be P_i', we get $Q_i \xrightarrow{c} Q_i'$ and $Q_i' = P_i'$.

SubCase 2.3: $i \in \varphi - PA_Q(c)$.

Consider P_i, Q_i, P_i' for an arbitrary $i \in \varphi - PA_Q(c)$. By ($Q = P[c/d \rightarrow c]$ or $Q = P[c/d \rightarrow c, nl]$) and the sequence introduction transformation definition (15), we have $Q_i = P_i$. Since $P_i = P_i'$ in this case, we let $Q_i' = Q_i$, and hence $Q_i' = P_i'$.

If we now consider $Q' = (\| \ i \in \varphi : Q_i')$, where the Q_i' are as given by the preceding case analysis, we see that $Q \xrightarrow{c} Q'$ by the transition relation definition (7), and also that $Q' = P'[c/d \rightarrow c]$, by the sequence introduction transformation definition (15), since the above three cases cover the entire indexing set φ. Since $\mathcal{S} = \{(P, Q) \mid Q \in P[c/d \rightarrow c] \text{ or } Q \in P[c/d \rightarrow c, nl]\}$, we have $P' \mathcal{S} Q'$ as required.

Clause 4: for all $a \in \alpha P - \{c\}$, if $Q \xrightarrow{a} Q'$, then $P \xrightarrow{a} P'$ for some P' such that $P' \mathcal{S} Q'$

Let $Q \xrightarrow{a} Q'$ for some $Q' = (\| \ i \in \varphi : Q_i')$ and $a \in \alpha P - \{c\}$. We show that there exists a $P' = (\| \ i \in \varphi : P_i')$ such that $P \xrightarrow{a} P'$ and $P' \mathcal{S} Q'$. By $Q \xrightarrow{a} Q'$ and the transition relation definition (7), $Q_i \xrightarrow{a} Q_i'$ for all $i \in PA_Q(a)$, and $Q_i = Q_i'$ for all $i \in \varphi - PA_Q(a)$. We have two cases.

Case 1: $i \in PA_Q(a)$.

Consider P_i, Q_i, Q_i' for an arbitrary $i \in PA_Q(a)$. We have two subcases,

$i \in PA_Q(a) \bigcap PA_Q(d)$ and $i \in PA_Q(a) - PA_Q(d)$.

Subcase 1.1: $i \in PA_Q(a) \bigcap PA_Q(d)$.

From ($Q = P[c/d \rightarrow c]$ or $Q = P[c/d \rightarrow c, nl]$), the sequence introduction transformation definition (15), and the subcase condition $i \in PA_Q(a) \bigcap PA_Q(d)$, we have

$(Q_i = P_i[c/d \rightarrow c]$ or $Q_i = P_i[c/d \rightarrow c, nl])$. From $Q_i \xrightarrow{a}$, we get $Q_i \not\xrightarrow{c}$, since no occurrence of c in $Q_i, i \in PA_Q(d)$ is an operand of $[\!]$. Hence $Q_i \neq P_i[c/d \rightarrow c, nl]$. Thus $Q_i = P_i[c/d \rightarrow c]$. Since $Q_i \xrightarrow{a}$, we have $\beta Q_i = (a \rightarrow F' [\!] G') \rightarrow *E'$ for some action expressions F', G', E', as discussed above (page 29). From $Q_i = P_i[c/d \rightarrow c]$ and the sequence introduction transformation definition (15), we conclude that $\beta P_i = (a \rightarrow F [\!] G) \rightarrow *E$ where action expressions E, F, G are such that $F' = F_t, G' = G_t, E' = E_t$.

By the transition relation definition (7), rules **Act, Ch, Seq** applied in sequence, we have

$$((a \rightarrow F_t [\!] G_t) \rightarrow *E_t) \xrightarrow{a} (F_t \rightarrow *E_t)$$

In general, Q_i can have more than one a-derivative, since G_t itself could have the form $a \rightarrow F'_t [\!] G'_t$. However, there is no loss of generality in assuming that the execution of the action a in $a \rightarrow F_t$ leads to Q'_i. Hence we have $\beta Q'_i = (F_t \rightarrow *E_t)$.

Thus, we may assume, without loss of generality, that

$$\beta Q'_i = (F_t \rightarrow *E_t)$$

Now $Q_i = P_i[c/d \rightarrow c]$. Since $\beta Q_i = (a \rightarrow F_t [\!] G_t) \rightarrow *E_t$, we have $\beta P_i = (a \rightarrow F [\!] G) \rightarrow *E$, by the sequence introduction transformation (15). By the transition relation definition (7), rules **Act, Ch, Seq**, we have

$$((a \rightarrow F [\!] G) \rightarrow *E) \xrightarrow{a} (F \rightarrow *E)$$

Since $\beta P_i = (a \rightarrow F [\!] G) \rightarrow *E$, there exists a P'_i such that $P_i \xrightarrow{a} P'_i$ and $\beta P'_i = F \rightarrow *E$. Since $\beta Q'_i = F_t \rightarrow *E_t$, we have $Q'_i = P'_i[c/d \rightarrow c]$ by the sequence introduction transformation (15).

Subcase 1.2: $i \in PA_Q(a) - PA_Q(d)$.

From $(Q = P[c/d \rightarrow c]$ or $Q = P[c/d \rightarrow c, nl])$, the sequence introduction transfor-

mation definition (15), and the subcase condition, we have $Q_i = P_i$. Letting P_i' be Q_i' we get $P_i \xrightarrow{a} P_i'$ and $P_i' = Q_i'$.

Case 2: $i \in \varphi - PA_Q(a)$.

Consider P_i, Q_i, Q_i' for an arbitrary $i \in \varphi - PA_Q(a)$. We have two subcases, $i \in (\varphi - PA_Q(a)) \bigcap PA_Q(d)$ and $i \in (\varphi - PA_Q(a)) - PA_Q(d)$.

Subcase 2.1: $i \in (\varphi - PA_Q(a)) \bigcap PA_Q(d)$.

From ($Q = P[c/d \rightarrow c]$ or $Q = P[c/d \rightarrow c, nl]$), the sequence introduction transformation definition (15), and the subcase condition, we have ($Q_i = P_i[c/d \rightarrow c]$ or $Q_i = P_i[c/d \rightarrow c, nl]$). Since $i \in \varphi - PA_Q(a)$ these Q_i do not participate in action a. Therefore we have $Q_i' = Q_i$. We also have $P_i' = P_i$ since these P_i do not participate in action a either. Since ($Q_i = P_i[c/d \rightarrow c]$ or $Q_i = P_i[c/d \rightarrow c, nl]$), we have ($Q_i' = P_i'[c/d \rightarrow c]$ or $Q_i' = P_i'[c/d \rightarrow c, nl]$).

Subcase 2.2: $i \in (\varphi - PA_Q(a)) - PA_Q(d)$.

From ($Q = P[c/d \rightarrow c]$ or $Q = P[c/d \rightarrow c, nl]$), the sequence introduction transformation definition (15), and the subcase condition, we have $Q_i = P_i$. Since $i \in \varphi - PA_Q(a)$ these Q_i do not participate in action a. Letting P_i' be Q_i', we have $Q_i = P_i = Q_i' = P_i'$. (end of Case 2) Now suppose $PA_Q(a) \bigcap PA_Q(d) \neq \emptyset$. Then, by subcase 1.1, $Q_i' = P_i'[c/d \rightarrow c]$ for some $i \in PA_Q(a) \bigcap PA_Q(d)$. Thus, by all the subcases above and definition 15, we conclude $Q' = P'[c/d \rightarrow c]$. On the other hand, if $PA_Q(a) \bigcap PA_Q(d) = \emptyset$, then $(\varphi - PA_Q(a)) \bigcap PA_Q(d) = PA_Q(d)$, and therefore, by subcases 1.2, 2.1, 2.2, and definition 15, we conclude ($Q' = P'[c/d \rightarrow c]$ or $Q' = P'[c/d \rightarrow c, nl]$). Since $\mathcal{S} = \{(P, Q) \mid Q = P[c/d \rightarrow c]$ or $Q = P[c/d \rightarrow c, nl]\}$, we have $P' \, \mathcal{S} \, Q'$ in all cases.

If we now consider $P' = (\| \; i \in \varphi : P_i')$, where the P_i' are as given by the preceding case analysis, we see that $P \xrightarrow{a} P'$ by the transition relation definition (7).

Thus, clause 4 is satisfied.

Clause 5: if $Q \xrightarrow{d} Q'$, then $P \mathcal{S} Q'$, and if $Q \xrightarrow{c} Q'$, then $P \xrightarrow{c} P'$ for some P' such that $P' \mathcal{S} Q'$

Let us start with the first conjunct, namely "if $Q \xrightarrow{d} Q'$, then $P \mathcal{S} Q'$."

Let $Q \xrightarrow{d} Q'$ for some $Q' = (\| \ i \in \varphi : Q'_i)$. We show that $P \mathcal{S} Q'$. By $Q \xrightarrow{d} Q'$ and the transition relation definition (7), $Q_i \xrightarrow{d} Q'_i$ for all $i \in PA_Q(d)$ and $Q_i = Q'_i$ for all $i \in \varphi - PA_Q(d)$. We have two cases.

Case 1: $i \in PA_Q(d)$

Consider Q_i, P_i, Q'_i for an arbitrary $i \in PA_Q(d)$. From ($Q = P[c/d \rightarrow c]$ or $Q = P[c/d \rightarrow c, nl]$), the sequence introduction transformation definition (15), and the case condition $i \in PA_Q(d)$, we have ($Q_i = P_i[c/d \rightarrow c]$ or $Q_i = P_i[c/d \rightarrow c, nl]$). From $Q_i \xrightarrow{d}$, we have $Q_i \neq P_i[c/d \rightarrow c, nl]$. Hence $Q_i = P_i[c/d \rightarrow c]$. From this, $Q_i \xrightarrow{d}$, and $i \in PA_Q(d)$, we have $\beta Q_i = (d \rightarrow c \rightarrow F') \rightarrow *E'$ for some action expressions F', E', as discussed above (page 29). From $Q_i = P_i[c/d \rightarrow c]$ and the sequence introduction transformation definition (15), we conclude that $\beta P_i = (c \rightarrow F) \rightarrow *E$ where action expressions E, F are such that $F' = F_t, E' = E_t$.

By the transition relation definition (7), rule **Act**, we have

$$((d \rightarrow c \rightarrow F_t) \rightarrow *E_t) \xrightarrow{d} ((c \rightarrow F_t) \rightarrow *E_t)$$

Thus, we may assume, without loss of generality, that

$$\beta Q'_i = ((c \rightarrow F_t) \rightarrow *E_t).$$

Since $\beta P_i = (c \rightarrow F) \rightarrow *E$, we have $Q'_i = P_i[c/d \rightarrow c, nl]$ by the sequence introduction transformation (15).

Note that the assumption that no occurrence of c in $P_i, i \in PA_Q(d)$, is an operand of $\|$ is crucial in carrying through the proof in this case. Suppose that this

assumption is dropped. Then the form of βQ_i becomes $\beta Q_i = (d \rightarrow c \rightarrow F' \parallel G') \rightarrow *E'$. Also, $\beta P_i = (c \rightarrow F \parallel G) \rightarrow *E$. Now $((d \rightarrow c \rightarrow F_t \parallel G_t) \rightarrow *E_t) \xrightarrow{d} ((c \rightarrow F_t) \rightarrow *E_t)$, and so $\beta Q_i' = ((c \rightarrow F_t) \rightarrow *E_t)$. However, since $\beta P_i = (c \rightarrow F \parallel G) \rightarrow *E$, we get $P_i[c/d \rightarrow c, nl] = ((c \rightarrow F_t \parallel G_t) \rightarrow *E_t)$, and so we cannot conclude $Q_i' = P_i[c/d \rightarrow c, nl]$, due to the difference introduced by the presence of the action expression G in P, which can occur only if c is allowed to be the operand of $\parallel$.

Case 2: $i \in \varphi - PA_Q(d)$.

Consider P_i, Q_i, Q_i' for an arbitrary $i \in \varphi - PA_Q(d)$. By ($Q = P[c/d \rightarrow c]$ or $Q = P[c/d \rightarrow c, nl]$) and the sequence introduction transformation definition (15), we have $Q_i = P_i$. Since $Q_i = Q_i'$ in this case, we have $Q_i' = P_i$.

From the preceding case analysis, we see that $Q' = P[c/d \rightarrow c, nl]$, by the sequence introduction transformation definition (15), since the above two cases cover the entire indexing set φ. Since $\mathcal{S} = \{(P, Q) \mid Q = P[c/d \rightarrow c]$ or $Q = P[c/d \rightarrow c, nl]\}$, we have $P \mathcal{S} Q'$ as required. Thus the first conjunct is satisfied.

We next establish the second conjunct of the clause, namely "if $Q \xrightarrow{c} Q'$, then $P \xrightarrow{c} P'$ for some P' such that $P' \mathcal{S} Q'$."

Let $Q \xrightarrow{c} Q'$ for some $Q' = (\parallel i \in \varphi : Q_i')$. We show that there exists a $P' = (\parallel i \in \varphi : P_i')$ such that $P \xrightarrow{c} P'$ and $P' \mathcal{S} Q'$. By $Q \xrightarrow{c} Q'$ and the transition relation definition (7), $Q_i \xrightarrow{c} Q_i'$ for all $i \in PA_Q(d)$, $Q_i \xrightarrow{c} Q_i'$ for all $i \in PA_Q(c) - PA_Q(d)$, and $Q_i = Q_i'$ for all $i \in \varphi - PA_Q(c)$. We have three cases.

Case 1: $i \in PA_Q(d)$

From ($Q = P[c/d \rightarrow c]$ or $Q = P[c/d \rightarrow c, nl]$) the sequence introduction transformation definition (15), and $i \in PA_Q(d)$, we have ($Q_i = P_i[c/d \rightarrow c]$ or $Q_i = P_i[c/d \rightarrow c, nl]$). From $Q_i \xrightarrow{c} Q_i'$, we have $Q_i \neq P_i[c/d \rightarrow c]$. Hence $Q_i = P_i[c/d \rightarrow c, nl]$. Since

$Q_i \xrightarrow{c}$, we have $\beta Q_i = (c \rightarrow F' \parallel G') \rightarrow *E'$ for some action expressions F', G', E', as discussed above (page 29). Note that we can actually show $\beta Q_i = (c \rightarrow F') \rightarrow *E'$ by using the $i \in PA_Q(d)$ and our assumption that c does not occur in a choice in $P_i, i \in PA_Q(d)$. However, this is not needed to push through this case, and so we can use the more general form $\beta Q_i = (c \rightarrow F' \parallel G') \rightarrow *E'$. In reality, we will always have $G' = 0$. Thus $\beta P_i = (c \rightarrow F \parallel G) \rightarrow *E$ where action expressions E, F, G are such that $F' = F_t, G' = G_t, E' = E_t$.

By the transition relation definition (7), rules **Act, Ch, Seq** applied in sequence, we have

$$((c \rightarrow F_t \parallel G_t) \rightarrow *E_t) \xrightarrow{c} (F_t \rightarrow *E_t)$$

In general, Q_i can have more than one c-derivative, since G_t itself could have the form $c \rightarrow F_t' \parallel G_t'$. However, there is no loss of generality in assuming that the execution of the action c in $c \rightarrow F_t$ leads to Q_i'. Hence we have $\beta Q_i' = (F_t \rightarrow *E_t)$.

Thus, we may assume, without loss of generality, that

$$\beta Q_i' = (F_t \rightarrow *E_t).$$

Now $Q_i = P_i[c/d \rightarrow c, nl]$. Since $\beta Q_i = (c \rightarrow F_t \parallel G_t) \rightarrow *E_t$, we have $\beta P_i = (c \rightarrow F \parallel G) \rightarrow *E$, by the sequence introduction transformation (15). By the transition relation definition (7), rules **Act, Ch, Seq**, we have

$$((c \rightarrow F \parallel G) \rightarrow *E) \xrightarrow{c} (F \rightarrow *E)$$

Since $\beta P_i = (c \rightarrow F \parallel G) \rightarrow *E$, there exists a P_i' such that $P_i \xrightarrow{c} P_i'$ and $\beta P_i' = F \rightarrow *E$. Since $\beta Q_i' = F_t \rightarrow *E_t$, we have $Q_i' = P_i'[c/d \rightarrow c]$ by the sequence introduction transformation (15).

Case 2: $i \in PA_Q(c) - PA_Q(d)$.

Consider P_i, Q_i, Q_i' for an arbitrary $i \in PA_Q(c) - PA_Q(d)$. By ($Q = P[c/d \rightarrow c]$ or

$Q = P[c/d \rightarrow c, nl]$) and the sequence introduction transformation definition (15), we have $Q_i = P_i$. Letting P_i' be Q_i', we get $P_i \xrightarrow{c} P_i'$ and $Q_i' = P_i'$.

Case 3: $i \in \varphi - PA_Q(c)$.

Consider Q_i, P_i, Q_i' for an arbitrary $i \in \varphi - PA_Q(c)$. By ($Q = P[c/d \rightarrow c]$ or $Q = P[c/d \rightarrow c, nl]$) and the sequence introduction transformation definition (15), we have $Q_i = P_i$. Now $Q_i = Q_i'$ in this case as explained above. Letting $P_i' = P_i$ we get $P_i' = Q_i'$.

If we now consider $P' = (\| \; i \in \varphi : P_i')$, where the P_i' are as given by the preceding case analysis, we see that $P \xrightarrow{c} P'$ by the transition relation definition (7), and also that $Q' = P'[c/d \rightarrow c]$, by the sequence introduction transformation definition (15), since the above three cases cover the entire indexing set φ. Since $\mathcal{S} = \{(P, Q) \mid Q = P[c/d \rightarrow c]$ or $Q = P[c/d \rightarrow c, nl]\}$, we have $P' \, \mathcal{S} \, Q'$ as required.

We have shown that all five clauses of definition 16 hold, hence $\mathcal{S}$ is a *dc*-bisimulation. □

Proposition 7 *Let $Q = P[c/c \rightarrow d]$. If Q' is an arbitrary derivative of Q, then there exists a derivative P' of P such that $P' \sim Q'$.*

Proof: Let $\mathcal{S} = \{(P, Q) \mid Q = P[c/c \rightarrow d]\}$. By proposition 6, $\mathcal{S}$ is a *cd*-bisimulation. Hence $\mathcal{S} \subseteq \sim$ by definition 17. Since $P \, \mathcal{S} \, Q$ by assumption, we conclude $P \sim Q$. We now establish the proposition by induction on the number of steps of derivation of program Q. We let Q^i represent a derivative of Q which has been obtained after i events. Thus Q', the derivative of Q after n steps, is represented by Q^n etc.

Base Case.

We first prove the base case for one step of the derivation. Let $Q \xrightarrow{a_1} Q^1$. There

are three cases, depending on the value of a_1. First, if $a_1 = d$, from $P \sim Q$ and proposition 5 we conclude that $P \sim Q^1$. Second, if $a_1 = c$, from $P \sim Q$ and proposition 5 we conclude that there exists P^1, a derivative of P ($P \xrightarrow{c} P^1$) such that $P^1 \sim Q^1$. Third, if $a_1 \neq c$ and $a_1 \neq d$, then from $P \sim Q$ and proposition 5 we conclude that there exists P^1, a derivative of P ($P \xrightarrow{a_1} P^1$) such that $P^1 \sim Q^1$. Thus the base case is established.

Induction Hypothesis. Let us assume that our proposition holds **upto** n steps of the derivation.

Induction Step. Let $Q \xrightarrow{a_1} \cdots \xrightarrow{a_n} Q^n$ for some path $\rho = a_1, \ldots, a_n$ of actions. By our induction hypothesis , there exists a derivative P' of P such that $P' \sim Q^n$. Let $Q^n \xrightarrow{a_{n+1}} Q^{n+1}$. Using the same argument as the base case, we conclude that there exists a derivative P'' of P' such that $P'' \sim Q^{n+1}$. Since P'' is a derivative of P' which is a derivative of P, P'' is a derivative of P. Thus our induction step is established. □

As before we prove that our syntactic transformation preserves the property of Deadlock-freedom.

Theorem 2 *Let $Q = P[c/d \rightarrow c]$. If P is deadlock-free, then so is Q.*

Proof: Let Q' be an arbitrary derivative of Q. By definition 14, it suffices to show $Q' \xrightarrow{a}$ for some action a. By proposition 7, we have

If Q' is an arbitrary derivative of Q, then there exists a derivative P' of P such that $P' \sim Q'$

By assumption, P is deadlock-free. Hence, by definition 14, $P' \xrightarrow{a'}$ for some a'. If $a' \neq c$, then $Q' \xrightarrow{a'}$ by proposition 5. If $a' = c$, then either $Q' \xrightarrow{dc}$, or $Q' \xrightarrow{c}$ again by proposition 5. Hence in all cases we have $Q' \xrightarrow{a}$ for some a, and therefore Q is

deadlock-free. □

3.4 Example: The Elevator Problem

We will illustrate the transformations with the standard elevator control problem. Our version is defined as follows: An N elevator system is to be installed in a building with F floors. The internal mechanisms of the elevators are given. The problem is to design the logic control that moves elevators between floors according to the following constraints:

- Each elevator has a set of buttons, one for each floor. These illuminate when pressed and cause the elevator to visit the corresponding floor. The elevator-button illumination is canceled when the elevator stops at that floor.
- Each floor (except ground and top) has two buttons: one to request an up-elevator and one to request a down-elevator. These buttons also illuminate when pressed. A floor button is canceled when an elevator traveling in the desired direction visits the floor.
- All requests for elevators (from floors and within elevators) must be served within a finite amount of time.

Figure 3.1 gives an initial solution to this problem. The actions in figure 3.1 perform the following functions:

select Indicates that a particular floor button has been pressed, selects a particular elevator to service the floor-button request, and enters that request into the schedule of the elevator.

p–schedule Indicates that a particular panel button has been pressed and enters the panel-button request into the schedule of the elevator containing the panel button.

f–satisfy Indicates that a floor-button request has been satisfied; i.e., the elevator has visited the desired floor and was traveling in the desired direction.

p–satisfy Indicates that a panel button request has been satisfied, i.e., the elevator has visited the desired floor.

The *select* and *p–schedule* actions are very large-grain — they each represent several activities. To refine these actions, we apply the left-sequence introduction transformation to figure 3.1 twice. The first application has $d = f\text{–}press$ and $c = select$. The *f–press* action thus introduced models the pressing of the floor button. Hence, the granularity of *select* has been reduced since it now models only the selection of a specific elevator to service the floor-button request and the entering of that request into the schedule of the elevator. The second application has $d = p\text{–}press$ and $c = p\text{–}schedule$. Again, the granularity of *p–schedule* has been reduced since it now models only the insertion of the panel-button request into the schedule of the elevator containing the panel button. The resulting program is shown in figure 3.2.

We can refine the *p–schedule* action further by applying the right-sequence introduction transformation to figure 3.2, with $c = p\text{–}schedule$ and $d = p\text{–}insert$. *p–insert* models the actual insertion of the panel-button request into the schedule of the elevator containing the panel button. Afterward, *p–schedule* models only

the transmission of the request from the panel-button process (*p–button*) to the *elevator–controller* process. We also refine the *select* action, using four iterated applications of the right-sequence introduction transformation (in all of which we have $c = select$). The final result is shown in figure 3.3. Note that we have changed the name of *select* to *select–announce*, since it now only models the announcement of the floor-button request to all elevators. The other actions introduced have the following functions:

reply The reply from the elevators to the announcement of the floor-button request. This reply will contain information (i.e., its location and direction of travel) that the floor-button process will subsequently use to select a particular elevator to service its request.

choose The (local) action of the *f–button* process in which it selects the elevator to service its request.

inform The action in which *f–button* informs all the elevators of its choice.

f–insert The (local) action in which the chosen elevator adds the floor-button request to its schedule.

Although it is quite easy to verify by inspection that the program of figure 3.1 is deadlock-free, the same property is not so readily apparent for the program of figure 3.3. However, we know that the latter program *is* deadlock-free, because it was derived from the former using only transformations that preserve deadlock-freedom.

Of course, a relatively small additional effort would enable us to verify the deadlock-freedom of the program of figure 3.3, since this example is quite simple.

With a program of realistic size, however, the difference in complexity between the initial and final programs would often be substantial.

```
f–button :: *[ select → f–satisfy ]
||
p–button :: *[ p–schedule → p–satisfy ]
||
elevator–controller :: *[ select
                          ||
                          p–schedule
                          ||
                          f–satisfy
                          ||
                          p–satisfy
                          ]
```

Figure 3.1: Zero-Level Elevator System

```
f–button :: *[ f–press → select → f–satisfy ]
||
p–button :: *[ p–press → p–schedule → p–satisfy ]
||
elevator–controller :: *[ select → f–insert
                        []
                        p–schedule → p–insert
                        []
                        f–satisfy
                        []
                        p–satisfy
                        ]
```

Figure 3.2: First-Level Elevator System

```
f–button :: *[ f–press → select–announce → reply → choose →
                                          inform → f–satisfy ]
||
p–button :: *[ p–press → p–schedule → p–satisfy ]
||
elevator–controller :: *[ select–announce → reply → inform → f–insert
                        []
                        p–schedule → p–insert
                        []
                        f–satisfy
                        []
                        p–satisfy
                        ]
```

Figure 3.3: Second-Level Elevator System

Chapter 4

Merge Transformations

4.1 Introduction

This chapter focuses on syntactic transformation techniques for building programs which involve *merging* of deadlock free programs with constrained syntactic forms such that the merged program preserves the property of deadlock freedom. These merging techniques could be used to *build* large complex programs from smaller ones. Such tools, along with transformations which allow refinement of programs by action introduction, could form the basis of a promising methodology for the design and implementation of large, complex systems.

It is to be noted that the transformations described are mechanizable and therefore do not require an unreasonable amount of manual formal labour at every step, which is a drawback of most other similar methodologies. Manual verification can be done for the small programs which are used as the building blocks.

The program merging idea seems to be rather new in the domain of distributed programs. The only place where we have found a reference to similar techniques is in a paper by Sol M. Shatz [SS85]. The primary objective of his proposal is to find means to enhance the fault tolerance of a system. His technique uses process merging

to reconfigure a distributed program on detecting a faulty processing element. The failure element is removed from the system and the process which was being executed on this element is reallocated to a different element. If this element had already been executing a process, that process is merged with the newly migrated one to form a new sequential process.

In this chapter we shall use the notaion described in the introductory section. We next define the transition graph construct which we shall use in future.

4.2 Preliminary Technical Definitions

Corresponding to the execution of a distributed program, one can construct a transition graph such that each of its nodes corresponds to a state of the program. The states are encoded by tuples which are constructed by taking the indices of the processes constituting the program. The ordering of the indices in the tuple is *identical to the ordering of the corresponding processes in the syntactic representation of the program.* The "state-nodes" are connected by *directed edges* such that if the nodes S^i and S^j are connected by a directed edge labeled a, this implies that the program can go from state S^i to S^j on execution of action a.

Definition 18 (*trgraph*)

More formally, a transition graph G_P of program P is the least *graph such that :*

1) The initial state *P is a node in G_P.*

2) If P^1 is a node in G_P then (P^1, a, P^2) is a directed edge in G_P for every action a and program configuration P^2 such that $P^1 \xrightarrow{a} P^2$.

If the execution graphs of two programs are equal (upto a relabeling of the nodes), we will call them behaviorally equivalent. Two behaviorally equivalent programs are capable of exactly the same action sequences.

Given two programs P and Q consisting of sets of asynchronously executing and communicating processes, we will attempt to do a syntactic *merge* of the two programs to form a single new program R which embodies the functionality of both P and Q.

We will now present a set of transformations Unless otherwise stated we will assume that the programs being merged have equal numbers of processes. In the syntactic representation of the concurrent programs we have not explicitly included the $\|$ symbol to indicate parallel composition of the processes. *Parallel composition is always implied.* We refer to an arbitrary process of a program P as P_i.

4.3 Transformation 0

We shall start with an elementary merge transformation.

Definition 19 (*Transformation 0*)

Consider the programs P and Q of the following form and satisfying conditions $1 - 3$ *below.*

$$
\begin{array}{lll}
P = & & Q = \\
P_1 \quad *(c \rightarrow F_1) & \| & Q_1 \quad *(d \rightarrow G_1) \quad \| \\
P_2 \quad *(c \rightarrow F_2) & \| & Q_2 \quad *(d \rightarrow G_2) \quad \| \\
\vdots & & \\
P_n \quad *(c \rightarrow F_n) & & Q_n \quad *(d \rightarrow G_n)
\end{array}
$$

1. *For any action a such that $a \in \alpha P \cap \alpha Q$,*

$$\bigwedge i \in \{1 \ldots n\},\ a \in \alpha P_i \text{ iff } a \in \alpha Q_i.$$

2. $\bigwedge i \in \{1 \ldots n\},\ c \notin \alpha F_i \wedge d \notin \alpha G_i$

3. *P and Q are deadlock free.*

We merge the programs P and Q to produce the program R.

$R =$

$$\begin{array}{lll} R_1 :: & *((c \rightarrow F_1) \,[\!]\, (d \rightarrow G_1)) & \| \\ R_2 :: & *((c \rightarrow F_2) \,[\!]\, (d \rightarrow G_2)) & \| \\ \vdots & & \\ R_n :: & *((c \rightarrow F_n) \,[\!]\, (d \rightarrow G_n)) & \end{array}$$

It is easy to observe that the actions c and d act as top level guards for the process bodies from programs P and Q respectively. Suppose c is executed. Now the program R is *forced* to execute an action sequence of P till it gets back to the top level choice. During this period no action sequence of Q can be executed. We will present a formal correctness proof in the next sub-section.

4.3.1 Discussion

The first condition is designed to avoid the possibility of deadlock in R. Its use is illustrated by the following example which violates the condition. Let us merge the programs P and Q below. The program R is the result of the merge.

$$\begin{array}{lll} P = & & \\ P_1 :: & *(c \rightarrow a) & \| \\ P_2 :: & *(c \rightarrow b) & \| \\ P_3 :: & *(c \rightarrow a) & \end{array} \qquad \begin{array}{lll} Q = & & \\ Q_1 :: & *(d \rightarrow e) & \| \\ Q_2 :: & *(d \rightarrow a) & \| \\ Q_3 :: & *(d \rightarrow e) & \end{array}$$

R =
R_1 :: $*(c \rightarrow a \parallel d \rightarrow e)$ $\parallel$
R_2 :: $*(c \rightarrow b \parallel d \rightarrow a)$ $\parallel$
R_3 :: $*(c \rightarrow a \parallel d \rightarrow e)$

Note that the action a is such that $a \in \alpha P \cap \alpha Q$.

We also have $a \notin \alpha P_2$ and $a \in \alpha Q_2$ which violates the first condition. After executing c and b the program R is as follows and is clearly deadlocked.

R =
R_1 :: $a \rightarrow *(c \rightarrow a \parallel d \rightarrow e)$
R_2 :: $*(c \rightarrow b \parallel d \rightarrow a)$
R_3 :: $a \rightarrow *(c \rightarrow a \parallel d \rightarrow e)$

Enforcing the first condition enables us to avoid such deadlocks.

4.3.2 Proof of Correctness

Definition 20 *c–derivative*

Let R_i be a process of an arbitrary derivative of R. Of the actions $\{c, d\}$, if c was the last action executed by the process R_i, or if $R_i = *(c \rightarrow F_i \parallel d \rightarrow G_i)$, we call R_i a c–derivative. c–derivatives are denoted by R_i^c

Definition 21 *d–derivative*

Let R_i be a process of an arbitrary derivative of R. Of the actions $\{c, d\}$, if d was the last action executed by the process R_i, or if $R_i = *(c \rightarrow F_i \parallel d \rightarrow G_i)$, or if both the actions c and d are enabled in R_i, we call R_i a d–derivative. d–derivatives are denoted by R_i^d

We define a mapping ϕ_P between the processes of c derivatives and the processes of derivatives of program P.

$\phi_P(R_i) = \phi_P(\alpha R_i, \beta R_i) = (\alpha P_i, \phi_P(\beta R_i))$ where $\phi_P(\beta R_i)$ is defined as follows:

If $\beta R_i = C_i' \rightarrow *(c \rightarrow C_i \parallel d \rightarrow D_i)$ then $\phi_P(\beta R_i) = C_i' \rightarrow *(c \rightarrow C_i)$

If $\beta R_i = *(c \rightarrow C_i \parallel d \rightarrow D_i)$ then $\phi_P(\beta R_i) = *(c \rightarrow C_i)$

We define a mapping ϕ_Q between the processes of d derivatives and the processes of derivatives of the program Q.

$\phi_Q(R_i) = \phi_Q(\alpha R_i, \beta R_i) = (\alpha Q_i, \phi_Q(\beta R_i))$ where $\phi_Q(\beta R_i)$ is defined as follows:

If $\beta R_i = D_i' \rightarrow *(c \rightarrow C_i \parallel d \rightarrow D_i)$ then $\phi_Q(\beta R_i) = D_i' \rightarrow *(d \rightarrow D_i)$

If $\beta R_i = *(c \rightarrow C_i \parallel d \rightarrow D_i)$ then $\phi_Q(\beta R_i) = *(d \rightarrow D_i)$

We define ϕ_P' which maps derivatives of R consisting of processes which are $c-$ *derivatives* to derivatives of P.

$\phi_P'(< R_1^c, \ldots, R_n^c >) = < \phi(R_1^c), \ldots, \phi(R_n^c) >$

We define ϕ_Q' which maps derivatives of R consisting of processes which are $d-$ *derivatives* to derivatives of P.

$\phi_Q'(< R_1^d, \ldots, R_n^d >) = < \phi(R_1^d), \ldots, \phi(R_n^d) >$

We define the mapping ϕ_P'' between the edges of the transition graphs (see page 51) of R and P.

Let $< R^c, a, R^c >$ be an edge in G_R. Then,

$\phi_P''(< R^c, a, R^c >) = < \phi_P'(R^c), a, \phi_P'(R^c) >$

The mapping ϕ_Q'' between the edges of the transition graphs of R and Q is defined similarly.

Let $< R^d, a, R^d >$ be an edge in G_R. Then,

$\phi''_Q(< R^d, a, R^d >) = < \phi'_P(R^d), a, \phi'_P(R^d) >$

We define the mapping $\phi'''(G_R)$ between the transition graph of R and those of P and Q.

$\phi'''(G_R) = \{\phi''_P(t) | t \textit{ starts with a } c - \textit{derivative}\} \cup \{\phi''_Q(t) | t \textit{ starts with a } d - \textit{derivative}\}$

Lemma 1 *If $\phi'''(G)$ is the transition graph of a deadlock-free program, then G is also a transition graph of a deadlock-free program.*

Proof: We present a simple proof by contradiction. Let us assume that G is the transition graph of a program which deadlocks. Therefore there exists at least one node, say N, which has no out-going edge. N will map to some node, say N_1 in $\phi'''(G)$. From the definition of the mapping ϕ''' it is clear that N_1 cannot have an out-going edge. Therefore we infer that the transition graph $\phi'''(G)$ corresponds to a program with a reachable deadlock state. This, however, is contrary to the premises. Therefore our initial assumption must be false. Therefore G is the transition graph of a deadlock-free program. □

Proposition 8 $\phi'''(G_R) = G_P \cup G_Q$

Proof: Let $< R^1, a, R^2 >$ be an arbitrary edge in G_R. Therefore $R^1 \xrightarrow{a} R^2$. From the syntactic form of R we infer that this action could have resulted from any of the following transitions executed by processes of R.

1. $a = c$ and $\forall i \in PA_P(c)$:

   $R^1_i = *(c \rightarrow C_i \parallel d \rightarrow D_i)$, $R^2_i = C_i \rightarrow *(c \rightarrow C_i \parallel d \rightarrow D_i)$

2. $a = d$ and $\forall i \in PA_Q(d)$:

$$R_i^1 = *(c \to C_i \parallel d \to D_i),\ R_i^2 = D_i \to *(c \to C_i \parallel d \to D_i)$$

3. $\forall i \in PA_P(a)$:

$$R_i^1 = C_i' \to *(c \to C_i \parallel d \to D_i),\ R_i^2 = C_i'' \to *(c \to C_i \parallel d \to D_i)$$

4. $\forall i \in PA_Q(a)$:

$$R_i^1 = D_i' \to *(c \to C_i \parallel d \to D_i),\ R_i^2 = D_i'' \to *(c \to C_i \parallel d \to D_i)$$

Corresponding to each of these cases we have an action executed by processes of P or Q which have a corresponding edge in G_P or G_Q.

1. For case 1 above consider the processes $\phi_P(R_i) = *(c \to C_i)$. By the definition of the transition relation (see page 8), we have :

   $\forall i \in PA_P(c), *(c \to C_i) \xrightarrow{c} C_i \to *(c \to C_i)$

2. For case 2 above consider the processes $\phi_Q(R_i) = *(d \to D_i)$. By the definition of the transition relation (see page 8), we have :

   $\forall i \in PA_Q(d), *(d \to D_i) \xrightarrow{d} D_i \to *(d \to D_i)$

3. For case 3 above consider the processes $\phi_Q(R_i) = C_i' \to *(c \to C_i)$. By the definition of the transition relation (see page 8), we have :

   $\forall i \in PA_P(a), C_i' \to *(c \to C_i) \xrightarrow{a} C_i'' \to *(c \to C_i)$

4. For case 4 above consider the processes $\phi_Q(R_i) = D_i' \to *(d \to D_i)$. By the definition of the transition relation (see page 8), we have :

   $\forall i \in PA_Q(a), D_i' \to *(d \to D_i) \xrightarrow{a} D_i'' \to *(d \to D_i)$

Here it is also important to note that the alphabet restriction on the programs which are being merged (the first condition in the set) ensures that for all a, a an

action in R, the processes participating in a are exactly the same in terms of indices for P/Q and R. In other words we have :

1. if $a \in \alpha P \cap \alpha Q$ then $PA_P(a) = PA_Q(a) = PA_R(a)$.

2. if $a \notin \alpha P \cap \alpha Q$ and $a \in \alpha P$ then $PA_P(a) = PA_R(a)$.

3. if $a \notin \alpha P \cap \alpha Q$ and $a \in \alpha Q$ then $PA_Q(a) = PA_R(a)$.

Thus not only do we have a P_i (or Q_i) for every R_i which can execute an action a, we are also sure that these are the only processes in program P (or Q) which participate in the action a. This allows us to make the following claim.

Any edge of $\phi_2(G_R)$ has a corresponding edge in either of G_P or G_Q. (a)

In a similar way we can show :

Any edge of G_P or G_Q is also an edge of $\phi_2(G_R)$. (b)

From (a) and (b) we conclude $\phi_2(G_R) = G_P \bigcup G_Q$. □

Theorem 3 *If P, Q are deadlock free, so is R.*

Proof: Since by assumption G_P and G_Q are transition graphs of programs without reachable deadlock states, $\phi_2(G_R)$ is the transition graph of a deadlock free program. Therefore from lemma 1 (see page 56) we can conclude that G_R is the transition graph of a deadlock free program. Therefore R is deadlock free. □

4.4 Transformation 1

We will now go onto a merge transformation which uses the sequence operator for merging. We first define the single iteration constraint which is an essential condition for this transformation.

Definition 22 (*Single iteration constraint*)

The program P satisfies the single iteration constraint if its constituent processes P_i are such that any path of computation for P starting at P goes back to P in the transition graph with each body $\beta P_i = *S_i$ having executed exactly one iteration. This implies that none of the constituent processes may remain at their initial local states, i.e. not participate in any of the interactions of the path of computation. *A more rigorous definition of this constraint is :*

For all computations π of P, there exists a prefix ρ of π such that for every process P_i of the program P, ρ projected onto P_i is exactly one iteration of P_i. Projecting the path of computation of a program onto one of its constituent processes yields the sequence of those actions of the path of computation in which the process participated.

Definition 23 (*Transformation 1*)

Consider two programs P and Q of the following forms and satisfying the conditions below (where $0 \leq k \leq n$). k is the number of instantiations of processes of the form $*(b \,[\!]\, c)$. When $k = 0$, there are no processes of the form $*(b \,[\!]\, c)$.

$P =$		$Q =$	
$P_1 :: *S_1$	$\|$	$Q_1 :: *(b \,[\!]\, c)$	$\|$
$P_2 :: *S_2$	$\|$	$Q_2 :: *(b \,[\!]\, c)$	$\|$
$\vdots$			
$P_k :: *S_k$	$\|$	$Q_k :: *(b \,[\!]\, c)$	$\|$
$P_{k+1} :: *S_{k+1}$	$\|$	$Q_{k+1} :: *((b_2 \rightarrow B_{k+1} \rightarrow b) \,[\!]\, (c_2 \rightarrow C_{k+1} \rightarrow c))$	$\|$
$P_{k+2} :: *S_{k+2}$	$\|$	$Q_{k+2} :: *((b_2 \rightarrow B_{k+2} \rightarrow b) \,[\!]\, (c_2 \rightarrow C_{k+2} \rightarrow c))$	$\|$
$\vdots$			
$P_n :: *S_n$		$Q_n :: *((b_2 \rightarrow B_n \rightarrow b) \,[\!]\, (c_2 \rightarrow C_n \rightarrow c))$	

Conditions:

1. *P and Q are deadlock free.*

2. $\alpha P \cap \alpha Q = \varphi$.

3. *The program P satisfies the single iteration constraint (see page 59).*

4. *The program Q satisfies the single iteration constraint (see page 59).*

5. *The actions b_2, c_2, b, c do not occur in the action expressions B_i or C_i.*

(b)

The following is the program R which results from applying transformation 1 to the programs P and Q.

$R =$

$$
\begin{array}{lll}
R_1 & :: *(S_1 \rightarrow (b \,[\!]\, c)) & \| \\
R_2 & :: *(S_2 \rightarrow (b \,[\!]\, c)) & \| \\
\vdots & & \\
R_k & :: *(S_k \rightarrow (b \,[\!]\, c)) & \| \\
R_{k+1} & :: *(S_{k+1} \rightarrow ((b_2 \rightarrow B_{k+1} \rightarrow b) \,[\!]\, (c_2 \rightarrow C_{k+1} \rightarrow c))) & \| \\
R_{k+2} & :: *(S_{k+2} \rightarrow ((b_2 \rightarrow B_{k+2} \rightarrow b) \,[\!]\, (c_2 \rightarrow C_{k+2} \rightarrow c))) & \| \\
\vdots & & \\
R_n & :: *(S_n \rightarrow ((b_2 \rightarrow B_n \rightarrow b) \,[\!]\, (c_2 \rightarrow C_n \rightarrow c)))
\end{array}
$$

4.4.1 Discussion

It needs to be emphasized that *it is very easy to get into deadlocks if appropriate conditions are not specified.* For an illustration consider the merge of the following programs:

$$
\begin{array}{l}
*a \;\; \| \\
*a \;\; \| \\
*a
\end{array}
$$

and

$$
*(c \,[\!]\, d) \;\; \|
$$

$*c \quad \|$
$*d$

If we merge these programs into:

$*(a \rightarrow (c \,[\!]\, d)) \quad \|$
$*(a \rightarrow c) \quad \|$
$*(a \rightarrow d)$

We find that we have produced a program which deadlocks within two steps (consider the action sequence a, c or a, d).

The third and fourth constraints in the set ensure that pathological cases like the one described above are avoided. It is easy to observe that in that example the program corresponding to Q does not satisfy the second condition since there is no path of computation which will take it from Q back to Q with each process body having completed *exactly one* iteration. Only one path of computation where the condition does not hold is sufficient to disqualify the program. This does seem a little too restrictive but the ease with which merged programs tend to deadlock require strict conditions to be imposed. The second condition ensures that alphabet restrictions do not hamper the execution of the merged program. The fifth condition ensures that the new processes will be synchronized at the entry and exit of the process bodies belonging to the programs P and Q. This forces the execution of R to be composed of complete iterations of P and Q.

An inspection of the syntactic structure of R reveals that the action sequence of R's execution consists of alternating action sequences of P and Q. We have one iteration of P followed by an iteration of Q and so on. The processes are synchronized at the entry and exit of an iteration of Q. The actions b_2, c_2, b and c serve as the

synchronizers. They ensure that there is no interleaving between the iterations of P and Q.

4.4.2 Proof of Correctness

We now present a formal proof of correctness for the above transformation. We begin by defining the prefix program, which is the core construct of the proof.

Definition 24 (*prefix-process*)

The prefix process of a process M_i is the process created by taking the portion of the process body of M_i which precedes $*$. In case the process body of M_i starts with a $*$, the prefix process is constructed by taking the entire process body of M_i, excluding $*$.

$Pref(A_i' \rightarrow *A_i) = A_i'$

$Pref(*A_i) = A_i$

Definition 25 (*prefix-program*)

For a program M, the prefix program is the parallel composition of the prefix processes of the constituent processes of M. Henceforth we shall denote the prefix program of M as $Pref(M)$.

$Pref(M) = Pref(M_1) \parallel \ldots \parallel Pref(M_n)$

From our operational semantics and the definition of the prefix program it should be obvious that $Pref(M)$ simulates the execution of M for at least one transition. More formally we may state that $Pref(M)$ can execute the action a iff M can execute the action a.

For an arbitrary derivative R' of program R, if we could show that $Pref(R')$ can execute some action then it would immediately follow that the program R' can

execute the same action. This is sufficient to prove the deadlock freedom of R (see page 27). We will use the above approach for proving deadlock freedom preservation of this transformation.

Theorem 4 *The program R, the result of the above described merge of the deadlock free programs P and Q, is deadlock free.*

Proof: Consider the prefix programs of R and P.

$$
\begin{array}{l}
Pref(R) \ = \\
(S_1 \rightarrow (b \parallel c)) \quad \| \\
(S_2 \rightarrow (b \parallel c)) \quad \| \\
\vdots \\
(S_k \rightarrow (b \parallel c)) \quad \| \\
(S_{k+1} \rightarrow ((b_2 \rightarrow B_{k+1} \rightarrow b) \parallel (c_2 \rightarrow C_{k+1} \rightarrow c))) \quad \| \\
(S_{k+2} \rightarrow ((b_2 \rightarrow B_{k+2} \rightarrow b) \parallel (c_2 \rightarrow C_{k+2} \rightarrow c))) \quad \| \\
\vdots \\
(S_n \rightarrow ((b_2 \rightarrow B_n \rightarrow b) \parallel (c_2 \rightarrow C_n \rightarrow c)))
\end{array}
\tag{1.1}
$$

$$
\begin{array}{ll}
Pref(P) \ = & \\
S_1 & \| \\
S_2 & \| \\
\vdots & \\
S_k & \| \\
S_{k+1} & \| \\
S_{k+2} & \| \\
\vdots & \\
S_n &
\end{array}
\tag{1.2}
$$

Since P is deadlock free there exists an action which $Pref(P)$ can execute. This implies that $Pref(R)$, and therefore R can execute some action. Due to the single iteration constraint on program P, the deadlock freedom of P and the syntactic

structure of program Q, every path of computation of the program R leads to a derivative R' for which $Pref(R')$ has the syntactic form (1.3). We again observe from their respective syntactic forms that the execution of $Pref(R)$ is an exact simulation of the execution of P till it reaches the form (1.3).

$$
\begin{array}{ll}
(b \parallel c) & \| \\
(b \parallel c) & \| \\
\vdots & \\
(b \parallel c) & \| \\
((b_2 \rightarrow B_{k+1} \rightarrow b) \parallel (c_2 \rightarrow C_{k+1} \rightarrow c)) & \| \\
((b_2 \rightarrow B_{k+2} \rightarrow b) \parallel (c_2 \rightarrow C_{k+2} \rightarrow c)) & \| \\
\vdots & \\
((b_2 \rightarrow B_n \rightarrow b) \parallel (c_2 \rightarrow C_n \rightarrow c)) &
\end{array}
\tag{1.3}
$$

The above program (1.3) is identical to $Pref(Q)$. Therefore $Pref(R')$ and $Pref(Q)$ are able to execute the same set of actions. Since we know Q to be deadlock free, there exists at least one action a which Q can execute. Therefore there exists an action a which $Pref(Q)$ can execute. Therefore there exists an action a which $Pref(R')$ can execute from which it follows that R' can execute some action a.

From the deadlock freedom and single iteration constraints on Q we observe that $Pref(Q)$ executes to completion. Therefore $Pref(R')$ will also execute to completion. This completes one complete iteration of the program R.

The single iteration constraint on Q and the actions b and c ensure that at that point all the processes of the original program are lined up at the initial state, and a new unwinding takes place. The unwinding would take the prefix program back to form (1.1). Since P is deadlock free there exists an action which $Pref(P)$ can execute. Clearly this implies that $Pref(R)$ and R can execute some action. We then have another iteration which takes the prefix program through form (1.3) and back

to form (1.1). When our prefix program is between the forms (1.1) and (1.3), the deadlock freedom of P ensures that there is always some action which it can execute. When it is between the forms (1.3) and (1.1), deadlock freedom of Q ensures that there is always some action which it can execute. Thus the prefix program of R is deadlock free. Since $Pref(R)$ simulates R, R is deadlock free too. □

4.4.3 Straightforward Extensions

From the above proof it is easy to envisage a few trivial extensions to the syntactic forms of the processes being merged. These extensions can be demonstrated to preserve the deadlock freedom property in exactly the same way as above. We do not present them in detail to avoid repetition.

It is easy to see that the actions b and c need not be different. If they are represented by the common action bc, the processes of R of the form $*(b \parallel c)$ would be modeled as $*bc$. Proving the merged program deadlock free can be done in exactly the same way as above.

Another syntactic form we can easily accommodate is where the processes of the program Q have the following form ::

$$
\begin{array}{ll}
Q_1 :: & *(b \parallel c) \quad \| \\
Q_2 :: & *(b \parallel c) \quad \| \\
\vdots & \\
Q_k :: & *(b \parallel c) \quad \| \\
Q_{k+1} :: & *(\ (b_2 \rightarrow B_{k+1} \rightarrow ((B_{k+1,1} \rightarrow b) \parallel (B_{k+1,2} \rightarrow b) \parallel \ldots)) \\
 & \quad \parallel (c_2 \rightarrow C_{k+1} \rightarrow ((C_{k+1,1} \rightarrow c) \parallel (C_{k+1,2} \rightarrow c) \parallel \ldots))\ ) \quad \| \\
\vdots & \\
Q_n :: & *(\ (b_2 \rightarrow B_n \rightarrow ((B_{n,1} \rightarrow b) \parallel (B_{n,2} \rightarrow b) \parallel \ldots)) \\
 & \quad \parallel (c_2 \rightarrow C_n \rightarrow ((C_{n,1} \rightarrow c) \parallel (C_{n,2} \rightarrow c) \parallel \ldots))\ )
\end{array}
$$

The only way the above form differs from the one we have dealt with in detail

is that instead of single occurrences of the actions b or c, every path has the synchronizing actions b or c at the end. Deadlock freedom can be proved in the same way as above.

4.4.4 Comments

This transformation enables us to merge two programs satisfying the given set of conditions and executing different layers of functionality. The point of note is that the new program retains the deadlock freedom property. We believe that in many physical systems that we would like to model, functionality can be decomposed into different layers which interact only at well defined interfaces. Similar ideas have been elaborated in a paper by Elrad and Francez [EF82]. They group parts of processes which interact mutually and do not interact with other parts, into layers. Their proposal is essentially a divide and conquer approach which consists of developing a collection of layers from the specification of the distributed program and then composing these layers into the program while preserving their communication closedness. They have also proposed a language construct which does away with the requirement of having to take care of explicit synchronization to ensure a safe decomposition which does not result in inconsistent computations.

We could develop programs for the different layers separately and then merge them using syntactic gluing techniques on the lines of the one we described. What is important to ensure is that our merging techniques preserve all the properties we are interested in. Here we have only concerned ourselves with deadlock freedom. We emphasize that other properties and features such as safety and liveness are just as important. This work is a first step in devising property preserving transformations.

4.5 Transformation 2

We now present a simple merge transformation which uses the choice operator to merge programs.

Definition 26 (*Transformation 2*)

Consider the programs P and Q of the following forms and satisfying condition set (a).

P consists of a set of processes each of which have the form $*(a)$. Q is any program such that the number of processes in Q is greater than or equal to that of P. We shall denote the processes of Q as $*Q_1, *Q_2 \ldots$.

Conditions:

1. *Q is deadlock free.*
2. *$a \notin \alpha Q$.* *(a)*

We construct our merged program by putting some of the process bodies of Q in choice with the action a from the process bodies of program P. A different way of looking at the new program is to imagine that a new action a has been introduced in choice with some or all of the processes comprising the program Q. The program R looks as follows:

$$
\begin{array}{lll}
R_1 :: & *Q_1 & \| \\
R_2 :: & *(a \parallel Q_2) & \| \\
R_3 :: & *(Q_3) & \| \\
\vdots & &
\end{array}
$$

$R_n :: \quad *(a \parallel Q_n)$

The selection of Q_i which are put in choice with a is arbitrary.

An inspection of the syntactic form of R reveals that the execution of R consists of interleaved iterations of P and Q. As before the interleaving occurs at the level of iterations and not among individual actions from P and Q. Since the P layer consists of only one action (a), this is a trivial observation. We give a formal proof below for the sake of consistency.

4.5.1 Proof of Correctness

Definition 27 (*Enabled Set*)

The enabled set for a program P is the set of actions which the program can execute next. We shall denote it by $Enab(P)$. Formally, $Enab(P) = \{a \mid P \xrightarrow{a}\}$. If the program is deadlocked then $Enab(P) = \emptyset$.

Theorem 5 *The program R, the result of the above described merge of the deadlock free programs P and Q, is deadlock free.*

Proof: Let us set up a mapping ϕ from the processes of R (and its derivatives) to the processes of Q (and its derivatives).

$\phi(R_i) = \phi(\alpha R_i, \beta R_i) = (\alpha R_i - a, \phi(\beta R_i))$ where $\phi(\beta R_i)$ is defined as follows.

If $\beta R_i = *(a \parallel Q_2)$	then	$\phi(\beta R_i) = *(Q_2)$
If $\beta R_i = *(Q_2)$	then	$\phi(\beta R_i) = *(Q_2)$
If $\beta R_i = Q' \rightarrow *(a \parallel Q_2)$	then	$\phi(\beta R_i) = Q' \rightarrow *(Q_2)$
If $\beta R_i = Q' \rightarrow *(Q_2)$	then	$\phi(\beta R_i) = Q' \rightarrow *(Q_2)$

Consider the program R' which is an arbitrary derivative of R. From the definition of ϕ_1 it should be clear that $Enab(\phi_1(R')) \subseteq Enab(R')$. If all processes of R which contain a are back at the top level choice, then $Enab(R')$ has the action a in addition to the actions in $Enab(\phi_1(R'))$. In all other cases $Enab(\phi_1(R')) = Enab(R')$.

Now $\phi_1(R')$ is identical to some derivative Q^j of the program Q since ϕ_1 maps derivatives of R to those of Q. Since Q is deadlock free by assumption, there exists some action b such that $Q^j \xrightarrow{b}$ (see page 27), i.e. $Enab(Q^j) \neq \emptyset$. Therefore $Enab(\phi_1(R')) \neq \emptyset$. Therefore $Enab(R') \neq \emptyset$ (since $Enab(\phi_1(R')) \subseteq Enab(R')$). Thus R', an arbitrary derivative of R, has a non-empty enabled set. Therefore there exists some action b which R' can execute. Therefore R is deadlock free (see page 27). □

4.6 Transformation 3

We will extend the last transformation into a more general one. In the following paragraphs we explain some terms which are essential to our discussion.

Definition 28 (*Covering Actions*)

A covering action *of a program P is any action which it can fire when at the state P (i.e. all the processes of P are at their initial local states). The covering action set C_P of a program P is defined as follows :*

$C_P = \{\mathsf{c} \mid P \xrightarrow{\mathsf{c}}\}$.

Definition 29 (*Process Cover Set*)

The set $PA_P(a)$ of participant processes of action a is called a process cover set *iff a is a covering action.*

To distinguish process cover sets from the participant sets of actions that are not covering actions, we introduce the notation S_a^P for $PA_P(a)$ when a is a covering action. If P is understood from context, we omit it and write S_a.

Definition 30 (*Complete Cover Set*)

The union of the process cover sets of a program P is called the Complete Cover Set, CC_P.

$CC_P = \bigcup_a PA_P(a)$ *where a ranges over all the covering actions of P.*

Definition 31 (*An iteration*)

An iteration is an action sequence which can take a program P from the state P back to the state P without going through P.

Definition 32 (*Active Processes*)

The Active processes for an iteration of program P are defined to be those processes which participate in any of the actions of the iteration.

Definition 33 (*End Synchrony*)

End Synchrony is a property which a covering action (say a) may have with respect to a program (say P). Satisfaction of this property is determined by two criteria :

1. *For all computations π of program P which start with the covering action a, there exists a prefix ρ of π such that for every process P_i of the program P, ρ projected onto P_i is exactly one iteration of P_i if the result of the projection is non-empty. In other words if a process has participated in any of the actions of ρ, it must have completed an iteration. However, it has the option of not participating in any action at all. This is somewhat similar to the single iteration constraint we have discussed before.*

2. *For any particular path of execution, only one specific action is executed by all* Active processes *at the end of an iteration. This action must not appear anywhere else. Different paths of execution may have different actions appearing at the end of the iteration. The crucial point is that all processes which participate in the iteration execute the same action at the end of the iteration. We visualize these actions as marking the end of an iteration and refer to them in future as* end markers.

An example should make this clearer.

Consider the following program S :

$$
\begin{array}{lll}
S = & & \\
P_1 :: & *(c \rightarrow a) & \| \\
P_2 :: & *(c \rightarrow a) & \| \\
P_3 :: & *(c \ [\!] \ (d \rightarrow f \rightarrow e)) & \| \\
P_4 :: & *(d \rightarrow \mathrm{e}) & \| \\
P_5 :: & *a &
\end{array}
$$

The covering action set is $\{c, d\}$. The corresponding process cover sets are $\{1, 2, 3\}$ (for c) and $\{3, 4\}$ (for d). It is to be noted that a is not a covering action since it can never be executed by the program from state S. The complete cover set is $\{1, 2, 3, 4\}$. The action d has the property of end synchrony with respect to the program S. The action e serves to end-synchronize all the processes which participate in an iteration starting with the action d. Clearly the action c does not have the property of end synchrony, since P_1, P_2 execute a at the end, but P_3 executes c.

Definition 34 (*Transformation 3*)

The program R is the result of applying transformation 3 to the programs P and Q.

P =

$*P_1 \parallel$
$*P_2 \parallel$
$\vdots$
$*P_n$

Q =

$*Q_1 \parallel$
$*Q_2 \parallel$
$\vdots$
$*Q_n$

R =

$*[P_1 \; [\!] \; Q_1] \parallel$
$*[P_2 \; [\!] \; Q_2] \parallel$
$\vdots$
$*[P_n \; [\!] \; Q_n]$

Conditions satisfied by P and Q.

1. *Let $C_P = \{a_1, \ldots, a_k\}$ and $C_Q = \{b_1, \ldots, b_l\}$ be the sets of covering actions for the programs P and Q respectively. Let $S_{a_1} \ldots S_{a_k}$ and $S_{b_1} \ldots S_{b_l}$ be the process cover sets corresponding to the covering actions in C_P and C_Q.*

   (a) $C_P \bigcap \alpha Q = \phi$ and $C_Q \bigcap \alpha P = \phi$.

   (b) For any $1 \leq i \leq k$ and $1 \leq j \leq l$, $S_{a_i} \bigcap S_{b_j} \neq \phi$.

2. *All the actions of C_P have the property of end synchrony with respect to the program P. Similarly all the actions of C_Q have the property of end synchrony with respect to the program Q.*

3. *For any action a such that $a \in \alpha P \cap \alpha Q$,*

   $$a \in \alpha P_i \text{ iff } a \in \alpha Q_i.$$

4. *The programs P and Q are deadlock free.*

4.6.1 Discussion

The conditions 1(b) and 2 ensure that if any of the covering actions of P or Q get executed by R, no action from the other program body can execute until the program

is back to the initial state R. For example as soon as one of the covering actions of any of the program P executes, all the covering actions of Q are blocked off by the process bodies of P. Due to the end synchrony property, this blocking off persists till the program is back to the top level choice i.e., the initial state R. This enforces what we call a layered execution of P and Q. The execution history of R looks like an interleaving of complete iterations of the programs P and Q. This decreases the number of possible interleavings of actions from P and Q in the execution of R; while this restricts the power of our merged program, it enhances our ability to analyze the program and restrict its behavior.

The third condition is required to avoid deadlock. This has been explained before (see page 53). The condition 1 (a) ensures that the covering actions of the programs being merged are mutually exclusive and that no cross interactions between the programs which would make an analysis difficult, are allowed.

4.6.2 Proof of Correctness

We now give a proof for deadlock freedom using the idea of enabled sets (see page 68). Let us start with some observations about the programs involved in the merge.

Theorem 6 *The program R, the result of the above described merge of the deadlock free programs P and Q, is deadlock free.*

Proof: When R is in the initial state R, an examination of the syntactic structure of R lets us conclude that $Enab(R) = Enab(P) \bigcup Enab(Q)$. Since the program P is known to be deadlock free, $Enab(P) = C_P \neq \phi$ and $Enab(Q) = C_Q \neq \phi$. Obviously $Enab(R) \neq \phi$. Therefore R can execute some action. This action could belong to either C_P or C_Q. Let us assume that an action ac from the set C_P is executed. Since

we have $C_P \bigcap \alpha Q = \phi$ as one of our constraints, the portions of the process bodies which came from the program P participate in this action. Process bodies derived from Q cannot participate in this action. Thus the processes of R which participate in the covering action attain the following syntactic form :

$$P'_k \rightarrow *(P_k \parallel Q_k)$$

where $P_k \stackrel{ac}{\rightarrow} P'_k$.

Let S_{ac} be the process cover set (see page 69) corresponding to the action ac. Let $S_{Q_1} \ldots S_{Q_m}$ be the process cover sets corresponding to the covering actions in C_Q. From the constraint 1(b) we have for all j, $1 \leq j \leq m$, $S_{ac} \bigcap S_{Q_j} \neq \phi$. We also know that after execution of ac, each of the processes in the cover set S_{ac} has the syntactic form $P'_k \rightarrow *(P_k \parallel Q_k)$, as explained before. Since the cover set S_{ac} has an intersection with each of the cover sets of Q, all covering actions of Q are blocked. From the syntactic structure of the processes and our operational semantics, we infer that none of the covering actions of the program Q can execute till some of the fragments P'_k execute to completion and thus enable some covering action of Q.

Constraint 2 ensures that all actions in the Cover Set C_P have the property of end synchrony with respect to the program P. Obviously ac being a member of C_P, has the property of end synchrony with respect to the program P. Therefore in accordance with the end synchrony property (see page 70) the program P would return to the initial state P with all pocesses which executed any action at all having completed one complete iteration. All the Active processes participate in an end-marker action which comes at the end of the iteration. For the execution of the program R this implies that the program fragments of P block any further unfolding of the processes of the program R till the end-marker action has been executed.

Keeping our previous observations in view, let us set up a mapping ϕ from the process bodies of R (and its derivatives) to the process bodies of P (and its derivatives). We will see how this mapping clarifies our analysis of R's behavior *as it does an iteration which is initiated by a covering action of the program* P.

Let R_m be a process of the program R.

$\phi(R_m) = \phi(\alpha R_m, \beta R_m) = (\alpha P_m, \phi(\beta R_m))$ where $\phi(\beta R_m)$ is defined as follows:

1. If $\beta R_m = *(P_m \parallel Q_m)$ then $\phi(\beta R_m) = *P_m$

2. If $\beta R_m = P'_m \rightarrow *(P_m \parallel Q_m)$ then $\phi(\beta R_m) = P'_m \rightarrow *(P_m)$

We extend ϕ to map derivatives of R to those of P. Let R' be an arbitrary derivative of R.

$\phi(< R_1^i, \ldots, R_n^i >) = < \phi(R_1^i), \ldots, \phi(R_n^i) >$

From our operational semantics and the syntactic nature of the mapping we note that $Enab(\phi(R)) \subseteq Enab(R)$.

Let R' be an arbitrary derivative of the program R. Then $\phi(R')$ is identical to some derivative P_{derv} of the program P. (*)

Proof of (*) We present an inductive proof.

Since the program P is deadlock free, $Enab(P) \neq \phi$. Therefore $Enab(R) \neq \phi$ ($Enab(P) \subseteq Enab(R)$). Therefore there exists some action which R can execute. Let us consider the execution of the first action, a. We consider an action from the alphabet of program P since we are currently interested in an iteration where R simulates program P.

Let $R \xrightarrow{a} R^1$.

The action a is executed as a result of the following process transition:

$\forall i \in PA_P(a)$

$*(P_i \parallel Q_i) \xrightarrow{a} P_i' \rightarrow *(P_i \parallel Q_i)$

Corresponding to this transition, P has the following transition ::

$\forall i \in PA_P(a)$

$*(P_i) \xrightarrow{a} P_i' \rightarrow *(P_i)$

We have $P \xrightarrow{a} P^1$.

From the definition of ϕ we observe that $\phi(R^1) = P^1$. This establishes the base case of our inductive reasoning.

Consider an action a executed by R', an arbitrary derivative of R. By our **induction hypothesis (*)** there exists some derivative P^i of the program P which satisfies $\phi(R') = P^i$.

In this iteration the process bodies of Q do not get to execute, as explained in our previous discussions. Thus the possible syntactic forms of processes of R' are :

1. $\beta R_j' = *(P_i \parallel Q_i)$

2. $\beta R_j' = P_i' \rightarrow *(P_i \parallel Q_i)$

The action a could be executed as a result of any of the following transitions:

1. $\forall i \in PA_R(a),\ *(P_i \parallel Q_i) \xrightarrow{a} P_i' \rightarrow *(P_i \parallel Q_i)$

2. $\forall i \in PA_R(a),\ P_i' \rightarrow *(P_i \parallel Q_i) \xrightarrow{a} P_i'' \rightarrow *(P_i \parallel Q_i)$

Corresponding to each of the above cases we have an action a executed by $\phi(R') = P^i$.

1. $\phi(\ *(P_i \parallel Q_i)\ ) = *(P_i)$

   $*(P_i) \xrightarrow{a} P_i' \rightarrow *(P_i)$

2. $\phi(\ P_i' \rightarrow *(P_i \parallel Q_i)\ ) = P_i' \rightarrow *(P_i)$

   $P_i' \rightarrow *(P_i) \xrightarrow{a} P_i'' \rightarrow *(P_i)$

The alphabet restriction on the programs which are being merged (constraint 3) ensures that for all a, a an action in R, the processes participating in a are exactly the same in terms of indices for P and R. Thus we have :

1. if $a \in \alpha P \cap \alpha Q$ then $PA_P(a) = PA_Q(a) = PA_R(a)$.

2. if $a \notin \alpha P \cap \alpha Q$ and $a \in \alpha P$ then $PA_P(a) = PA_R(a)$.

3. if $a \notin \alpha P \cap \alpha Q$ and $a \in \alpha Q$ then $PA_Q(a) = PA_R(a)$.

The alphabet restriction ensures that all the processes which are participants in the action a are enabled. We do not have the situation where we have different sets of participants in an action for P and R. This avoids the deadlock situation described in page 53. We can now assert that the program P can execute the action a. Let $P^i \xrightarrow{a} P^{i+1}$.

We have $R' \xrightarrow{a} R^{i+1}$ and $P^i \xrightarrow{a} P^{i+1}$ where $\phi(R') = P^i$. It is easy to observe from the syntactic forms of the programs produced and the definition of the mapping that $\phi(R^{i+1}) = P^{i+1}$.

This establishes our induction step and since the base case has been established before, we may now claim that any arbitrary derivative $\phi(R')$ of program R is identical to some derivative P_{derv} of the program P.

(end proof of (*))

In our previous discussions we noted that $Enab(\phi_1(R')) \subseteq Enab(R')$. We have also demonstrated that $\phi_1(R')$ is identical to some derivative of program P. Let us call this derivative P^j. We have $Enab(\phi_1(R')) = Enab(P^j)$. Since the program P is deadlock free, all derivatives of it must have non-empty enabled sets. Therefore $Enab(P^j) \neq \phi$. Therefore $Enab(\phi(R')) \neq \phi$. Therefore $Enab(R')$, which is a superset of $Enab(\phi(R'))$ is also non-empty. Therefore R', an arbitrary derivative of program R, can execute some action.

If the iteration was initiated by a covering action of the program Q, using symmettric (w.r.t. P and Q) analysis we can show that an arbitrary derivative of the program R can execute some action.

Since the program R can be initiated by only the covering actions of the programs P or Q, these two are the only cases possible. Thus in all cases we find that arbitrary derivatives of the program R can execute some action. Therefore the program R is deadlock free. □

4.7 Transformation 3.1

We present an example of application of the last merge transformation.

Definition 35 (*Transformation 3.1*)

Consider the two programs P and Q of the following forms and which satisfy the conditions below:

$P =$ $\quad Q =$

$*c \quad \| \quad *d \quad \|$

$\vdots \quad \quad \vdots$

$*c \quad \| \quad *d \quad \|$

$\vdots \quad \quad \vdots$

$*(a \rightarrow C_1 \rightarrow c) \quad \| \quad *(b \rightarrow D_1 \rightarrow d) \quad \|$

$\vdots \quad \quad \vdots$

$*(a \rightarrow C_k \rightarrow c) \quad \| \quad *(b \rightarrow D_k \rightarrow d) \quad \|$

$\vdots \quad \quad \vdots$

Conditions :

1. *For any action a such that $a \in \alpha P \cap \alpha Q$,*

$$a \in \alpha P_i \text{ iff } a \in \alpha Q_i.$$

2. *P, Q are deadlock free.*

3. *$a \notin \alpha Q$ and $b \notin \alpha P$.*

4. *The action a has the property of end-synchrony with respect to program P.*

5. *The action b has the property of end-synchrony with respect to program Q.*

The following program R is formed by applying transformation 3.1 to the programs P and Q.

$R =$

$*(c \parallel d) \quad \|$

$\vdots$

$*(c \parallel d) \quad \|$

$\vdots$

$*((a \rightarrow C_1 \rightarrow c) \parallel (b \rightarrow D_1 \rightarrow d) \quad \|$

$$
\begin{array}{l}
\vdots \\
*((a \rightarrow C_k \rightarrow c) \parallel (b \rightarrow D_k \rightarrow d) \qquad\qquad \| \\
\vdots
\end{array}
$$

4.7.1 Discussion

Let us see if the last transformation technique (transformation 3) can be used to prove deadlock freedom of the program R under the given criteria.

From the syntactic structures of P and Q, we observe that their covering action sets C_P, C_Q are $\{a\}$ and $\{b\}$ respectively. If n processes are being merged, the process cover sets are $\{1, 2, \ldots, n\}$ and $\{1, 2, \ldots, n\}$.

We observe that $C_P \bigcap \alpha Q = \phi$ and $C_Q \bigcap \alpha P = \phi$. Also the intersection of the process cover sets is non-empty. Thus the first constraint for the merge is satisfied.

From the fourth condition we infer that all actions of C_P have the property of end-synchrony with respect to program P. Similarly, from the fifth condition we infer that all actions of C_Q have the property of end-synchrony with respect to program Q.

The third and fourth requirements are satisfied by the first two conditions satisfied by the programs P and Q.

We have shown that all four required conditions for transformation 3 are satisfied by the programs P and Q. Therefore the program R which is the result of merging P and Q is deadlock free. □

4.8 Transformation 1.1

Using the ideas of *covering actions* and *end synchrony* we shall extend transformation 1.

Definition 36 (*Transformation 1.1*)

Consider two programs P and Q of the following forms and satisfying the following conditions.

$$
\begin{array}{ll}
P = & Q = \\
P_1 :: \ *(cov \rightarrow S_1 \rightarrow covsynch) \ \| & Q_1 :: \ *M_1 \ \| \\
P_2 :: \ *(cov \rightarrow S_2 \rightarrow covsynch) \ \| & Q_2 :: \ *M_2 \ \| \\
\vdots & \vdots \\
P_n :: \ *(cov \rightarrow S_n \rightarrow covsynch) & Q_n :: \ *M_n
\end{array}
$$

Conditions:

1. *P and Q are deadlock free.*

2. *$\alpha P \cap \alpha Q = \phi$.*

3. *The single iteration constraint (see page 59) holds for program P.*

4. *The single iteration constraint (see page 59) holds for program Q.*

The following is the program R which results from applying transformation 1.1 to the programs P and Q.

$$
\begin{array}{l}
R = \\
R_1 :: \ *(\ (cov \rightarrow S_1 \rightarrow covsynch) \rightarrow M_1\ ) \ \| \\
R_2 :: \ *(\ (cov \rightarrow S_2 \rightarrow covsynch) \rightarrow M_2\ ) \ \| \\
\vdots \\
R_n :: \ *(\ (cov \rightarrow S_n \rightarrow covsynch) \rightarrow M_n\ )
\end{array}
$$

4.8.1 Discussion

The constraints are similar to the ones required for transformation 1. The syntactic structure of the program P ensures that all the processes of P participate in the covering action *cov*. The action *covsynch* along with the single iteration constraint ensures the end synchrony property of the action *cov* with respect to the program P.

The second condition eliminates the possibility of common actions which would destroy the well-defined layering of functionality that we are attempting to develop. The third condition is devised to eliminate pathological cases similar to the one described in page 60.

4.8.2 Proof of Correctness

We now present a formal proof of correctness for the above transformation. The proof is very similar to that of transformation 1.

Theorem 7 *The program R, the result of the above described merge of the deadlock free programs P and Q, is deadlock free.*

Proof: Consider the prefix programs (see page 62) of R and P. As explained before, for any program M, $Pref(M)$ simulates the execution of M. As in the proof of transformation 1, we prove the deadlock freedom of R by demonstrating that an arbitrary derivative R' of program R can execute some action.

$$
\begin{array}{l}
Pref(R) \ = \\
(cov \rightarrow S_1 \rightarrow covsynch) \rightarrow M_1 \quad \| \\
(cov \rightarrow S_2 \rightarrow covsynch) \rightarrow M_2 \quad \| \\
\vdots \\
(cov \rightarrow S_n \rightarrow covsynch) \rightarrow M_n
\end{array} \tag{1.1}
$$

$$
\begin{array}{l}
Pref(P) \ = \\
cov \rightarrow S_1 \rightarrow covsynch \quad \| \\
cov \rightarrow S_2 \rightarrow covsynch \quad \| \\
\vdots \\
cov \rightarrow S_n \rightarrow covsynch
\end{array} \tag{1.2}
$$

Let R' denote derivatives of the program R. Since P is deadlock free there exists an action which $Pref(P)$ can execute. This implies that $Pref(R)$, and therefore R

can execute some action. Due to the single iteration constraint on program P, the deadlock freedom of P and the end synchrony property of the covering action *cov*, for every path of computation of R, there will exist some derivative R' for which $Pref(R')$ has the syntactic form (1.3) below. We again observe from their respective syntactic forms that the execution of $Pref(R)$ is an exact simulation of the execution of P till it reaches the form (1.3).

$$\begin{array}{ll} M_1 & \| \\ M_2 & \| \\ \vdots & \\ M_n & \end{array} \tag{1.3}$$

The above program is identical to $Pref(Q)$. Therefore $Pref(R')$ and $Pref(Q)$ are able to execute the same set of actions. Since we know Q to be deadlock free, there exists at least one action a which Q can execute. Therefore there exists an action a which $Pref(Q)$ can execute. Therefore there exists an action a which $Pref(R')$ can execute from which it follows that R' can execute some action a.

From the deadlock freedom and single iteration constraints on Q we observe that $Pref(Q)$ and therefore $Pref(R')$, executes to completion. The covering action *cov* cannot be enabled until all the processes have finished executing the action expressions from the process bodies of program Q. This marks the end of an iteration. Thus the action *cov* serves to separate complete iterations of the program R.

A new unwinding of the program R can now take place. This takes the prefix program of R back to the form (1.1) and the cycle repeats. When the prefix program is between the forms (1.1) and (1.3), the deadlock freedom of P ensures that there is always some action which it can execute. When it is between the forms (1.3) and (1.1), the deadlock freedom of Q ensures that there is always some action which it

can execute. Thus the prefix program of R is deadlock free. Since $Pref(R)$ simulates R, R is deadlock free. □

4.9 Transformation 1.2

We present another extension of transformation 1.

Definition 37 (*Transformation 1.2*)

Let us consider the programs P, Q where P is made up of processes all of the form $*(a)$. The processes constituting Q are denoted as $*Q_1$, $*Q_2, \ldots$. The number of processes in P and Q need not be equal. The merged program looks as follows:

R =

$$
\begin{array}{ll}
*Q_1 & \| \\
*(a \rightarrow Q_2) & \| \\
*(Q_3) & \| \\
\vdots & \\
*(a \rightarrow Q_n) &
\end{array}
$$

An alternate way to look at program R is to regard it as the result of introducing action a before some of the processes of Q.

The criteria to be satisfied are as follows:

1. $a \notin \alpha Q = \phi$
2. *The action a is introduced in front of all the processes which have a covering action.*
3. *Each of the covering actions of the program Q have the property of end synchrony with respect to the program Q.*

4. *The program Q is deadlock free (It is easy to observe that P is deadlock free, and hence this does not need to be assumed).*

4.9.1 Discussion

The first condition is devised to ensure functional layering in composition. The rest of the conditions together ensure that the only change to the action sequences is that the action a is added at the beginning of each of the iterations involving the covering action (actions). Since every iteration begins with a covering action this implies that a is added to the beginning of every pre-existing iteration and this is the only change to the computation paths. Action a thus becomes the sole covering action for the merged program R.

The deadlock freedom proof is analogous to that for the transformations 1 and 1.1.

4.9.2 Proof of Correctness

Theorem 8 *The program R, the result of the above described merge of the deadlock free programs P and Q, is deadlock free.*

Proof: Consider the prefix programs (see page 62) of R. As before, we prove the deadlock freedom of R by demonstrating that an arbitrary derivative R' of program R can execute some action. Since $Pref(R)$ simulates the execution of R, it is sufficient to show that the prefix program can always execute some action.

$Pref(R) =$

$$
\begin{array}{ll}
Q_1 & \| \\
a \rightarrow Q_2 & \| \\
Q_3 & \| \\
\vdots & \\
a \rightarrow Q_n &
\end{array}
$$

(1.1)

All processes of R which participate in the action a have the action a enabled in $Pref(R)$. Therefore $Pref(R)$ can execute action a. After execution of the action a The prefix program attains the form (1.2).

$$
\begin{array}{l}
Pref(R') \ = \\
Q_1 \parallel \\
Q_2 \parallel \\
Q_3 \parallel \\
\vdots \\
Q_n
\end{array}
\tag{1.2}
$$

This program is identical to $Pref(Q)$. Therefore $Pref(R')$ and $Pref(Q)$ are able to execute the same set of actions. From the deadlock freedom and cover action end synchrony constraints on program Q we infer that $Pref(R')$ executes to completion. The action a cannot be enabled until all action expression residues of the previous iteration have executed to completion. Condition 2 ensures that the action a is the sole covering action for the program R. Thus the prefix program is forced to return to the configuration (1.1).

Execution of the action a demarcates the start of a new iteration. At every step we are ensured that the prefix program is capable of executing some action. Since the prefix program simulates the original program, this implies that the derivatives of R at each step are capable of executing some action. Therefore the program R is deadlock free. □

4.10 Transformation 4

We end our transformation set with a simple transformation which enables us to add a new process with a single action to a program.

Definition 38 (*Transformation 4*)

Consider a process $pr = *a$ *and a program* P *of the following form, and satisfying the condition set (a) below.*

$P =$

$*P_1 \|$
$*P_2 \|$
$*P_3 \|$
$\vdots$
$*P_n$

Conditions:

1. *The program* P *is deadlock free.* *(a)*

P *is any arbitrary deadlock free program whose alphabet does not contain the action* a*. The merged program* R *is constructed by making* pr *a process of* P*.*

$R =$

$*a \quad \|$
$*P_1 \quad \|$
$*P_2 \quad \|$
$*P_3 \quad \|$
$\vdots$
$*P_n$

4.10.1 Discussion

This is a very simple but also very useful transformation since it allows us to introduce new processes which can then be built up using the other transformations.

4.10.2 Proof of Correctness

Theorem 9 *The program* R*, the result of the above described merge of the deadlock free program* P *and the process* pr*, is deadlock free.*

Proof: We assume the convention that in the program R the first process is the one which was introduced. Let us set up a mapping ϕ from derivatives (R') of R to derivatives of P.

$\phi(\alpha R, R') = (\alpha P, \phi(R'))$.

$\phi(R')$ is the program which results from removing the first process of R. Since $a \notin \alpha P$, the first process is not a participant in any of the actions of $\alpha R - a = \alpha P$. Let us now consider the enabled sets (see page 68) of R' and $\phi(R')$. Now the process $*a$ is always ready to execute a, and therefore its introduction has no effect on the enablement of a or of any other action. Therefore $Enab(\phi(R')) \bigcup \{a\} = Enab(R')$.

Therefore $Enab(R') \neq \emptyset$. Thus R', an arbitrary derivative of the program R, has a non-empty enabled set. Therefore there exists some action which R' can execute. Therefore R is deadlock free (see page 27). □

Chapter 5

Extended Example: The Mobile Telephone System

5.1 Introduction

In this section we illustrate the application of the transformations to the development of a restricted version of a mobile telephone system. The problem description is taken from the EIA interim standard, "Cellular Radiotelecommunication Intersystem Operations: Intersystem Handoff"([EIA87]).

The system consists of a fixed number of mobile telephones and message switching centers. Each mobile has a radio link with one of the message switching centers (msc) which is called the mobile's manager. All calls to the mobile are routed by trunk lines to this msc and then radioed to the mobile. The movement of the mobile may take it away from the manager so that eventually the signal quality between the mobile and the manager deteriorates to an unacceptable level. This problem is handled by transferring the management of the mobile to another msc if it has a better signal. The transfer operation is called a handoff. The msc in control before the handoff is called the server (or, manager) msc. The msc to which control is transferred is called the target msc. Our example is a simplified version of this system,

consisting of a single mobile (mb) and two message switching centers ($mc1$ and $mc2$).

The manager (serving) msc repeatedly performs a signal-level check on the mobile telephone. If the level check indicates that the signal quality has deteriorated to an unacceptable level, the following events happen in sequence:

1. The manager msc synchronizes with the other msc.

2. Both mscs perform a signal level check with the mobile.

3. The msc with higher signal level is determined.

4. If the other msc has a higher signal level, then management of the mobile is assigned to it (handoff); otherwise there is no reassignment of mobile management.

The voice facilities consist of dedicated voice circuits between the two mscs for the purpose of continuing speech transmission after a handoff. These circuits may be digital, analog, or mixed (digital and analog), as agreed upon by the administrations of the networks.

A permanent dedicated data link is established between the mscs. All voice circuit control signalling (i.e., circuit seizure, release, etc.), will be performed by signalling on the data link. We will refer to the data link connections as trunk lines.

A number of timers are associated with the server and the target mscs. They enable time-out decisions to be made in case of failed connections, data corruption etc. The timers HOT (Handoff Order Timer) and MHOT (Mobile Handoff Order Timer) are associated with the current server msc. The target msc uses the timers HAT (Handoff Accepted Timer) and MAT (Mobile Arrival Timer).

5.2 Modelling the System

Each iteration of our programs corresponds to a signal level test followed (if signal level is below threshold) by an attempt to handoff. This attempt can have the following outcomes :

1. The handoff is successful and the control of the mobile is transferred from the server msc to the target msc.
2. The current server msc has the stronger signal and therefore retains control of the mobile.
3. The handoff attempt is aborted due to unavailability of trunk lines.
4. The handoff attempt is aborted due to an error condition in the target msc.
5. The handoff attempt is aborted due to expiry of the Handoff Order Timer (HOT).
6. The handoff attempt is aborted due to expiry of the Mobile Handoff Order Timer (MHOT).
7. The handoff attempt is aborted due to expiry of the Handoff Accepted Timer (HAT).
8. The handoff attempt is aborted due to expiry of the Mobile Arrival Timer (MAT).

We now introduce some actions to model the events in our system. Since similar events may be executed by either msc we use subscripts on the names to distinguish

between the cases where one or the other of the mscs execute the action. The letters **i,j** are used for these subscripts.

1. **alloctranid$_{ij}$** : This action allocates an identification number to a handoff transaction. The indices **i** and **j** are used to distinguish between the different situations where such allocations may be necessary. In all cases these are local actions, i.e. only a single process participates in the allocation.

2. **above–thresh$_{i}$** : This action occurs if the level of the signal found by the signal level check is acceptable. In that case handoff is not necessary. The subscript **i** is used to distinguish a signal level check between mb and $mc1$ from a signal level check between mb and $mc2$.

3. **canceltranid$_{ij}$** : This actions destroys the id entry for a transaction. It is used when a transaction has failed and the recovery sequence needs to be invoked. The subscripts **i** and **j** are used to distinguish between the different cases which arise. The subscript **i** has the value **1, 2** when the msc involved is $mc1, mc2$ (respectively). The second subscript **j** distinguishes between different uses of the action within the same process.

4. **complpath$_{ij}$** : This models invocation of the procedures which complete the path of communication. From the server msc the call path is connected to the Inter-msc trunk lines. At the target msc the path between the voice channel which has been allocated and the Inter-msc trunk is completed. The subscripts are used to distinguish between the different cases.

5. **CSSrecvd$_i$** : This models the event when the mobile is "sensed" on the allocated voice channel. The subscript **i** has the value **1, 2** when the msc involved is $mc1, mc2$ (respectively).

6. **election** : This action performs the election of a new msc on the basis of the signal level check which has been previously performed.

7. **exit$_{ij}$** : This exits the current task. It may be required when a transaction fails. The subscripts **i** and **j** are used to distinguish between the different cases which arise. The subscript **i** has the value **1, 2** when the msc involved is $mc1, mc2$ (respectively). The second subscript **j** distinguishes between different uses of the action within the same process.

8. **error–to–proc$_{ij}$** : This is used to communicate an error situation between the message switching centers. The subscripts **i** and **j** are used to distinguish between the different cases which arise. The subscript **i** has the value **1, 2** when the msc involved is $mc1, mc2$ (respectively). The second subscript **j** distinguishes between different uses of the action within the same process.

9. **enter–invoke$_i$** : This action models the synchronization between the mscs before entering the final handoff sequence where the communication paths will be completed via the inter-msc trunk lines. The subscript **i** has the value **1, 2** when the server msc is $mc2, mc1$ (respectively).

10. **handoff–synch$_i$** : This action synchronizes the message switching centers prior to actions related to a handoff. The subscript **i** is used to distinguish between the different cases which arise.

11. **handoff$_{ij}$** : This corresponds to the completion of a single iteration (see discussion at the beginning of this section, page 91) between the mscs. The subscripts **i** and **j** are used to indicate the participants in the handoff (i.e., the transfer of mobile control from the msc **i** to the msc **j**). If **i** and **j** are the same, it is implied that there has been no handoff (some book-keeping operations may have been performed).

12. **if–no–trunk$_i$** : This models the event when no inter-msc trunk is available for communication between the mscs. The subscript **i** has the value **1, 2** when the server msc is $mc1, mc2$ (respectively).

13. **if–trunk$_i$** : This models the event when an inter-msc trunk is available for communication between the mscs. The subscript **i** has the value **1, 2** when the server msc is $mc1, mc2$ (respectively).

14. **if–hot–exp$_i$** : This models the event when the Handoff Order Timer (HOT) in the server msc expires. The subscript **i** has the value **1, 2** when the msc involved is $mc1, mc2$ (respectively).

15. **if–mhot–exp$_i$** : This models the event when the Mobile Handoff Order Timer (MHOT) in the server msc expires. The subscript **i** has the value **1, 2** when the msc involved is $mc1, mc2$ (respectively).

16. **if–hat–exp$_i$** : This models the event when the Handoff Accepted Timer (HAT) in the target msc expires. The subscript **i** has the value **1, 2** when the msc involved is $mc1, mc2$ (respectively).

17. **if–mat–exp$_i$** : This models the event when the Mobile Arrival Timer (MAT) in the target msc expires. The subscript **i** has the value **1, 2** when the msc involved is $mc1, mc2$ (respectively).

18. **below–thresh$_i$** : This action is executed when the signal quality has deteriorated to an unacceptable level. This indicates that a handoff may be necessary. The subscript **i** has the value **1, 2** when the msc checking the signal level is $mc1, mc2$ (respectively).

19. **mob–inv–ch$_{ij}$** : This action models the sending of a MobileOnChannel message from msc **j** to msc **i**. It indicates that all activity related to the handoff has been completed by the target msc (msc **j**).

20. **msc–synch$_i$** : This synchronizes all the mscs before the election of a new msc can begin. The subscript **i** has the value **1, 2** when the server msc is $mc1, mc2$ (respectively).

21. **poll–sig–chk** : This does a signal level check with the competing mscs to produce the information which will be used in the election.

22. **recov$_{ij}$** : This models invocation of a recovery procedure, usually necessary after a failed transaction. The subscript **i** has the value **1, 2** when the msc involved is $mc1, mc2$ (respectively). The subscript **j** is used to distinguish between the different recovery procedures within the same process.

23. **relres$_{ij}$** : This action models procedures which release resources acquired during a transaction. The subscript **i** has the value **1, 2** when the msc involved is

$mc1, mc2$ (respectively). The subscript **j** is used to distinguish between the different procedures for release of resources within the same process.

24. **ret–err–type$_{ij}$** : This action is used to send an error type from the target to the serving msc. It would subsequently cause the exit or error recovery routines to be called. The subscripts **i** and **j** are used to denote the target and the server mscs respectively.

25. **ret–ok–type$_{ij}$** : This action models the communication between the mscs when a voice channel is available at the target msc and therefore a successful handoff can be initiated. The subscripts **i** and **j** are used to denote the target and the server mscs respectively.

26. **signal–chk$_i$** : This is the signal level check. The managing msc interacts with a mobile to determine the strength of the signal between them. The subscript **i** has the value **1, 2** when the msc involved is $mc1, mc2$ (respectively).

27. **sethat$_i$** : This action sets the Handoff Accepted Timer(HAT). The timer is set by the target msc after it receives an invoke (action $invoke_{xy}$) from the server msc. The subscript **i** has the value **1, 2** when the msc involved is $mc1, mc2$ (respectively).

28. **set–timer–hot$_i$** : This action sets the Handoff Order Timer (HOT) in the serving msc. The subscript **i** has the value **1, 2** when the msc involved is $mc1, mc2$ (respectively).

29. **setmat$_i$** : This sets the Mobile Arrival Timer (MAT) in the target msc. The subscript **i** has the value **1, 2** when the msc involved is $mc1, mc2$ (respectively).

30. **setmhot$_i$** : This acion sets the Mobile Handoff Order Timer (MHOT) in the serving msc. This timer is set before the msc starts the wait phase before the invoke response is received from the target msc. The subscript **i** has the value **1, 2** when the msc involved is $mc1, mc2$ (respectively).

31. **st$_{ij}$** : This corresponds to the setting up of trunk lines between the mscs **i** and **j**. In our example this action is used to encapsulate any procedures that may be executed in case the election algorithm results in the current server msc being chosen as the one to service the mobile.

32. **stphat$_{ij}$** : This action stops the Handoff Accepted Timer (HAT) in the target msc. The subscripts **i** and **j** are used to distinguish between the different cases which arise.

33. **stphot$_{ij}$** : This action stops the Handoff Order Timer (HOT) in the serving msc. The subscripts **i** and **j** are used to distinguish between the different cases which arise.

34. **stpmat$_i$** : This action stops the Mobile Arrival Timer (MAT) in the target msc. The subscript **i** has the value **1, 2** when the msc involved is $mc1, mc2$ (respectively).

35. **stpmhot$_i$** : This action stops the Mobile Handoff Order Timer (MHOT) in the serving msc. The subscript **i** has the value **1, 2** when the msc involved is $mc1, mc2$ (respectively).

5.3 Construction of the Solution using Transformations

We attempt to construct a solution to the problem which will be of the form $mb \parallel mc1 \parallel mc2$. The programs which are produced at each stage are guaranteed to be deadlock free by our transformations.

We start by constructing a program which models the handoff interaction between the parties involved when $mc1$ is the server of the mobile. We do not explicitly construct the program for the situation where $mc2$ is the server of the mobile since it is constructed in the same way and can be obtained by the program we construct from symmetry considerations.

Consider the following program which models the invocation sequence between the processes of the msc's. The $enter\text{–}invoke_2$ action is used to transmit a request from the serving msc $mc1$ to the target msc $mc2$.

$P =$

$P_1 :: *(enter\text{–}invoke_2 \rightarrow mob\text{–}inv\text{–}ch_{12} \rightarrow handoff_{12})$

$\parallel$

$P_2 :: *(enter\text{–}invoke_2 \rightarrow mob\text{–}inv\text{–}ch_{12} \rightarrow handoff_{12})$

It is trivial to verify that this program is deadlock free.

Next consider the program $R0$ which models the situation where the handoff timers MAT and MHOT expire in the target and server msc respectively. This would cause exit and recovery sequences to be called. For now we do not deal with the exit and recovery sequences. They will be introduced later.

Note: We name the processes of $R0$ $mc1$ and $mc2$ instead of the usual $R0_1$ and $R0_2$ to make the relation of the program to the example we are developing more clear.

$mc1$ and $mc2$ will be dveloped into the processes of the two message switching centers.

$R0 =$

$mc1 :: \ *(if\text{–}mhot\text{–}exp_1 \rightarrow handoff_{12})$
$\|$
$mc2 :: \ *(if\text{–}mat\text{–}exp_2 \rightarrow handoff_{12})$

It is trivial to verify that this program is deadlock free.

We now merge the programs P and $R0$ using Transformation 3 (see page 69) to produce the program $R1$ below. The conditions required for this transformation are satisfied, as explained below :

1. The covering action sets C_P and C_{R0} for the programs P and $R0$ are $\{enter\text{–}invoke_2\}$ and $\{if\text{–}mhot\text{–}exp_1, if\text{–}mat\text{–}exp_2\}$ respectively. We note that $C_P \bigcap \alpha R0 = \phi$ and $C_{R0} \bigcap \alpha P = \phi$. The process cover set for $enter\text{–}invoke_2$ is $\{1, 2\}$. The process cover sets for $if\text{–}mhot\text{–}exp_1$ and $if\text{–}mat\text{–}exp_2$ are $\{1\}$ and $\{2\}$ respectively. Clearly the first condition is satisfied.

   Note : The processes P_1 and P_2 are mapped to the numbers 1 and 2 respectively. Similarly the processes $mc1$ and $mc2$ are mapped to the numbers 1 and 2 respectively.

2. We observe that $enter\text{–}invoke_2$ has the property of end synchrony with respect to the program P. The actions $if\text{–}mhot\text{–}exp_1$ and $if\text{–}mat\text{–}exp_2$ have the property of end synchrony with respect to program $R0$.

3. $\alpha P \cap \alpha R0 = \{handoff_{12}\}$. $handoff_{12} \in \alpha P_i$ iff $handoff_{12} \in \alpha R0_i$ is satisfied for $i = 1, 2$.

4. The programs P and $R0$ are deadlock free.

$R1 =$

$$
\begin{array}{ll}
mc1 :: & *(enter\text{–}invoke_2 \rightarrow mob\text{–}inv\text{–}ch_{12} \rightarrow handoff_{12} \\
& \quad [\!] \\
& \quad if\text{–}mhot\text{–}exp_1 \rightarrow handoff_{12}) \\
\| & \\
mc2 :: & *(enter\text{–}invoke_2 \rightarrow mob\text{–}inv\text{–}ch_{12} \rightarrow handoff_{12} \\
& \quad [\!] \\
& \quad if\text{–}mat\text{–}exp_1 \rightarrow handoff_{12})
\end{array}
$$

From theorem 6, we have that $R1$ is deadlock free.

Next let us consider the program $P1$ which models the interaction between the mscs in the case that they proceed with the handoff. The action $ok\text{–}to\text{–}proc_{21}$ enables the participating mscs to agree to go ahead.

$P1 =$

$$
\begin{array}{lll}
P1_1 & :: & *(ok\text{–}to\text{–}proc_{21} \rightarrow ret\text{–}ok\text{–}type_{21}) \\
\| & & \\
P1_2 & :: & *(ok\text{–}to\text{–}proc_{21} \rightarrow ret\text{–}ok\text{–}type_{21})
\end{array}
$$

It is trivial to verify that this program is deadlock free.

We merge the program $P1$ with $R1$, the result of the last merge, using Transformation 1.1 (see page 80) to produce the program $R2$. The conditions required for this transformation are satisfied, as explained below :

1. $P1$ and $R1$ are deadlock free.

2. $\alpha P1 \cap \alpha R1 \ = \ \phi$.

3. The single iteration constraint (see page 59) holds for program $P1$.

4. The single iteration constraint holds for program $R1$.

5. The actions $ok\text{–}to\text{–}proc_{21}$ $(= cov)$ and $ret\text{–}ok\text{–}type_{21}$ $(= covsynch)$ occur only in the appropriate positions and so $P1$ has the correct form.

$R2 =$

$$
\begin{array}{lll}
mc1 :: & *(ok\text{–}to\text{–}proc_{21} \rightarrow & ret\text{–}ok\text{–}type_{21} \rightarrow \\
 & & (\ (\ enter\text{–}invoke_2 \rightarrow mob\text{–}inv\text{–}ch_{12} \rightarrow handoff_{12} \\
 & & \quad \| \\
 & & \quad if\text{–}mhot\text{–}exp_1 \rightarrow handoff_{12}))) \\
\| & & \\
mc2 :: & *(ok\text{–}to\text{–}proc_{21} \rightarrow & ret\text{–}ok\text{–}type_{21} \rightarrow \\
 & & (\ (\ enter\text{–}invoke_2 \rightarrow mob\text{–}inv\text{–}ch_{12} \rightarrow handoff_{12} \\
 & & \quad \| \\
 & & \quad if\text{–}mat\text{–}exp_1 \rightarrow handoff_{12})))
\end{array}
$$

From theorem 7, we have that $R2$ is deadlock free.

Consider the program $P2$ which models the situation when proceeding with the handoff may cause an error situation and recovery routines may need to be called. As before we concern ourselves with the program skeleton for now, and we introduce the necessary refinement details later.

$P2 =$

$$
\begin{array}{lll}
P2_1 & :: & *(error\text{–}to\text{–}proc_{21} \rightarrow ret\text{–}err\text{–}type_{21} \rightarrow handoff_{12}) \\
\| & & \\
P2_2 & :: & *(error\text{–}to\text{–}proc_{21} \rightarrow ret\text{–}err\text{–}type_{21} \rightarrow handoff_{12})
\end{array}
$$

It is trivial to verify that this program is deadlock free.

We merge the program $P2$ with $R2$, the result of the last merge, using Transformation 3 (see page 69) to produce the program $R3$. The conditions required for this transformation are satisfied, as explained below :

1. The covering action sets C_{P2} and C_{R2} for the programs $P2$ and $R2$ are $\{error\text{–}to\text{–}proc_{21}\}$ and $\{ok\text{–}to\text{–}proc_{21}\}$ respectively. We note that $C_{P2} \bigcap \alpha R2 = \phi$ and $C_{R2} \bigcap \alpha P2 = \phi$. The process cover set for $error\text{–}to\text{–}proc_{21}$ is $\{1, 2\}$. The

process cover set for $ok\text{–}to\text{–}proc_{21}$ is $\{1,2\}$. Clearly the first condition is satisfied.

2. We observe that $error\text{–}to\text{–}proc_{21}$ has the property of end synchrony with respect to the program $P2$. The action $ok\text{–}to\text{–}proc_{21}$ has the property of end synchrony with respect to program $R2$.

3. $\alpha P2 \cap \alpha R2 = \{handoff_{12}\}$. $handoff_{12} \in \alpha P2_i$ iff $handoff_{12} \in \alpha mci$ is satisfied, for $i = 1, 2$.

4. The programs $P2$ and $R2$ are deadlock free.

$R3 =$

$$
\begin{array}{lll}
mc1 :: & *((error\text{–}to\text{–}proc_{21} \rightarrow ret\text{–}err\text{–}type_{21} \rightarrow handoff_{12}) & \\
& [\!] \ (ok\text{–}to\text{–}proc_{21} \rightarrow ret\text{–}ok\text{–}type_{21} \rightarrow & \\
& \qquad ((enter\text{–}invoke_2 \rightarrow mob\text{–}inv\text{–}ch_{12} \rightarrow handoff_{12}) & \\
& \qquad\quad [\!] & \\
& \qquad\quad (if\text{–}mhot\text{–}exp_1 \rightarrow handoff_{12})))) & \\
\| & & \\
mc2 :: & *((error\text{–}to\text{–}proc_{21} \rightarrow ret\text{–}err\text{–}type_{21} \rightarrow handoff_{12}) & \\
& [\!] \ (ok\text{–}to\text{–}proc_{21} \rightarrow ret\text{–}ok\text{–}type_{21} \rightarrow & \\
& \qquad ((enter\text{–}invoke_2 \rightarrow mob\text{–}inv\text{–}ch_{12} \rightarrow handoff_{12}) & \\
& \qquad\quad [\!] & \\
& \qquad\quad (if\text{–}mat\text{–}exp_1 \rightarrow handoff_{12})))) &
\end{array}
$$

From theorem 6, we have that $R3$ is deadlock free.

Consider the program $P3$ which models the situation when either of the timers HOT or HAT time out in the serving and target mscs respectively. We will introduce the necessary recovery and exit sequences later.

$P3 =$

$$
\begin{array}{lll}
P3_1 & :: & *(if\text{–}hot\text{–}exp_1 \rightarrow handoff_{12}) \\
\| & & \\
P3_2 & :: & *(if\text{–}hat\text{–}exp_2 \rightarrow handoff_{12})
\end{array}
$$

It is trivial to verify that this program is deadlock free.

We merge the program $P3$ with $R3$, the result of the last merge, using Transformation 3 (see page 69) to produce the program $R4$. The conditions required for this transformation are satisfied, as explained below :

1. The covering action sets C_{P3} and C_{R3} for the programs $P3$ and $R3$ are $\{if\text{–}hot\text{–}exp_1, if\text{–}hat\text{–}exp_2\}$ and $\{error\text{–}to\text{–}proc_{21}, ok\text{–}to\text{–}proc_{21}\}$. We note that $C_{P3} \bigcap \alpha R3 = \phi$ and $C_{R3} \bigcap \alpha P3 = \phi$. The process cover sets for the actions $if\text{–}hot\text{–}exp_1$ and $if\text{–}hat\text{–}exp_2$ are $\{1\}$ and $\{2\}$ respectively. The process cover sets for the actions $error\text{–}to\text{–}proc_{21}$ and $ok\text{–}to\text{–}proc_{21}$ are $\{1,2\}$ and $\{1,2\}$ respectively. Clearly the first constraint is satisfied.

2. The covering actions $if\text{–}hot\text{–}exp_1$, $if\text{–}hat\text{–}exp_2$, $error\text{–}to\text{–}proc_{21}$ and $ok\text{–}to\text{–}proc_{21}$ have the end synchrony property.

3. $\alpha P3 \cap \alpha R3 = handoff_{12}$. The requirement $handoff_{12} \in \alpha P3_i$ iff $handoff_{12} \in \alpha mci$ is satisfied for $i = 1, 2$.

4. The programs $P3$ and $R3$ are deadlock free.

$R4 =$

$$
\begin{array}{lll}
mc1 :: & *(& (if\text{–}hot\text{–}exp_1 \rightarrow handoff_{12}) \\
 & [] & (error\text{–}to\text{–}proc_{21} \rightarrow ret\text{–}err\text{–}type_{21} \rightarrow handoff_{12}) \\
 & [] & (ok\text{–}to\text{–}proc_{21} \rightarrow ret\text{–}ok\text{–}type_{21} \rightarrow \\
 & & \quad ((enter\text{–}invoke_2 \rightarrow mob\text{–}inv\text{–}ch_{12} \rightarrow handoff_{12}) \\
 & & \quad\quad [] \\
 & & \quad (if\text{–}mhot\text{–}exp_1 \rightarrow handoff_{12}))) \\
[] & & \\
mc2 :: & *(& (if\text{–}hat\text{–}exp_2 \rightarrow handoff_{12}) \\
 & [] & (error\text{–}to\text{–}proc_{21} \rightarrow ret\text{–}err\text{–}type_{21} \rightarrow handoff_{12} \\
 & [] & (ok\text{–}to\text{–}proc_{21} \rightarrow ret\text{–}ok\text{–}type_{21} \rightarrow \\
 & & \quad ((enter\text{–}invoke_2 \rightarrow mob\text{–}inv\text{–}ch_{12} \rightarrow handoff_{12}) \\
 & & \quad\quad []
\end{array}
$$

$(if\text{–}mat\text{–}exp_1 \rightarrow handoff_{12})))$

From theorem 6, we have that $R4$ is deadlock free.

Consider the program $P4$ which models the interaction between the mscs for setting up of the communication trunk lines (assuming the lines are available; we will deal with the case where no lines are available in the next step).

$P4 =$

$P4_1 \quad :: \quad *(if\text{–}trunk_1 \rightarrow set\text{–}timer\text{–}hot_1 \rightarrow invoke_{12})$
$\|$
$P4_2 \quad :: \quad *(if\text{–}trunk_1 \rightarrow invoke_{12})$

It is trivial to verify that this program is deadlock free.

We merge the program $P4$ with $R4$, the result of the last merge, using Transformation 1.1 (see page 80) to produce the program $R5$. The conditions required for this transformation are satisfied, as explained below :

1. $P4$ and $R4$ are deadlock free.

2. $\alpha P4 \cap \alpha R4 \; = \; \phi$.

3. The single iteration constraint (see page 59) holds for program $P4$.

4. The single iteration constraint holds for program $R4$.

5. The actions $if\text{–}trunk_1$ $(= cov)$ and $invoke_{12}$ $(= covsynch)$ occur only in the appropriate positions, and so $P4$ has the correct form.

$R5 =$

$$
\begin{aligned}
mc1 :: \; *(\; & if\text{–}trunk_1 \rightarrow set\text{–}timer\text{–}hot_1 \rightarrow invoke_{12} \rightarrow \\
& (\; (if\text{–}hot\text{–}exp_1 \rightarrow handoff_{12}) \\
& \; [\!] \; (error\text{–}to\text{–}proc_{21} \rightarrow ret\text{–}err\text{–}type_{21} \rightarrow handoff_{12}) \\
& \; [\!] \; (ok\text{–}to\text{–}proc_{21} \rightarrow \; ret\text{–}ok\text{–}type_{21} \rightarrow
\end{aligned}
$$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\quad (enter\text{–}invoke_2 \rightarrow \ mob\text{–}inv\text{–}ch_{12} \rightarrow handoff_{12}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \| \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (if\text{–}mhot\text{–}exp_1 \rightarrow handoff_{12}))))) \\
\| \\
mc2 :: \ *(\ if\text{–}trunk_1 \rightarrow invoke_{12} \rightarrow \\
\qquad\qquad (\ (if\text{–}hat\text{–}exp_2 \rightarrow handoff_{12}) \\
\qquad\qquad \| \ (error\text{–}to\text{–}proc_{21} \rightarrow ret\text{–}err\text{–}type_{21} \rightarrow handoff_{12}) \\
\qquad\qquad \| \ (ok\text{–}to\text{–}proc_{21} \rightarrow ret\text{–}ok\text{–}type_{21} \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad (\quad (enter\text{–}invoke_2 \rightarrow \ mob\text{–}inv\text{–}ch_{12} \rightarrow handoff_{12}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \| \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (if\text{–}mat\text{–}exp_1 \rightarrow handoff_{12})))))
\end{array}
$$

From theorem 7, we have that $R5$ is deadlock free.

Consider the program $P5$ which models the interaction between the mscs when the trunk lines are not available. Here we have the program skeleton which will be refined later.

$$
\begin{array}{l}
P5 = \\
P5_1 \ :: \ *(if\text{–}no\text{–}trunk_1 \rightarrow cancel\text{–}tranid_{1a} \rightarrow exit_{1a} \rightarrow handoff_{12}) \\
\| \\
P5_2 \ :: \ *(if\text{–}no\text{–}trunk_1 \rightarrow handoff_{12})
\end{array}
$$

It is trivial to verify that this program is deadlock free.

We merge the program $P5$ with $R5$, the the result of the last merge, using Transformation 3 (see page 69) to produce the program $R6$. The conditions required for this transformation are satisfied, as explained below :

1. The covering action sets C_{P5} and C_{R5} for the programs $P5$ and $R5$ are $\{if\text{–}no\text{–}trunk_1\}$ and $\{if\text{–}trunk_1\}$ respectively. We note that $C_{P5} \bigcap \alpha R5 = \phi$ and $C_{R5} \bigcap \alpha P5 = \phi$. The process cover set for $if\text{–}no\text{–}trunk_1$ is $\{1, 2\}$. The process cover set for $if\text{–}trunk_1$ is $\{1, 2\}$. Clearly the first condition is satisfied.

2. We observe that $if\text{–}no\text{–}trunk_1$ has the property of end synchrony with respect

to the program $P5$. The action $if\text{–}trunk_1$ has the property of end synchrony with respect to the program $R5$.

3. $\alpha P5 \cap \alpha R5 = \{handoff_{12}\}$.

   $handoff_{12} \in \alpha P5_i$ iff $handoff_{12} \in \alpha mci$ is satisfied for $i = 1, 2$.

4. The programs $P5$ and $R5$ are deadlock free.

$R6 =$

$$
\begin{array}{l}
mc1 :: \ *(\ (if\text{–}no\text{–}trunk_1 \rightarrow cancel\text{–}tranid_{1a} \rightarrow exit_{1a} \rightarrow handoff_{12}) \\
\quad [\!] \ (if\text{–}\ trunk_1 \rightarrow set\text{–}timer\text{–}hot_1 \rightarrow invoke_{12} \rightarrow \\
\qquad (\ \ (if\text{–}hot\text{–}exp_1 \rightarrow handoff_{12}) \\
\qquad\ [\!] \ (error\text{–}to\text{–}proc_{21} \rightarrow ret\text{–}err\text{–}type_{21} \rightarrow handoff_{12}) \\
\qquad\ [\!] \ (ok\text{–}to\ \text{–}proc_{21} \rightarrow ret\text{–}ok\text{–}type_{21} \rightarrow \\
\qquad\qquad (\ \ (enter\text{–}invoke_2 \rightarrow mob\text{–}inv\text{–}ch_{12} \rightarrow handoff_{12}) \\
\qquad\qquad\ [\!] \ (if\text{–}mhot\text{–}exp_1 \rightarrow handoff_{12}))))))
\end{array}
$$

$[\!]$

$$
\begin{array}{l}
mc2 :: \ *(\ (if\text{–}no\text{–}trunk_1 \rightarrow handoff_{12}) \\
\quad [\!] \ (if\text{–}\ trunk_1 \rightarrow invoke_{12} \rightarrow \\
\qquad (\ \ (if\text{–}hat\text{–}exp_2 \rightarrow handoff_{12}) \\
\qquad\ [\!] \ (error\text{–}to\text{–}proc_{21} \rightarrow ret\text{–}err\text{–}type_{21} \rightarrow handoff_{12}) \\
\qquad\ [\!] \ (ok\text{–}to\ \text{–}proc_{21} \rightarrow ret\text{–}ok\text{–}type_{21} \rightarrow \\
\qquad\qquad (\ \ (enter\text{–}invoke_2 \rightarrow mob\text{–}inv\text{–}ch_{12} \rightarrow handoff_{12}) \\
\qquad\qquad\ [\!] \ (if\text{–}mat\text{–}exp_1 \rightarrow handoff_{12}))))))
\end{array}
$$

From theorem 6, we have that $R6$ is deadlock free.

Consider the program $P6$ which models the synchronization between the message switching centers prior to the actions related to a handoff. The action $handoff\text{–}synch_2$ provides synchronization. We have also introduced the actions which allocate an id to the current transaction and attempt to allocate a trunk line for communication.

$P6 =$

$P6_1 \quad :: \quad *(handoff\text{–}synch_2 \rightarrow alloc\text{–}tranid_{1a} \rightarrow alloc\text{–}trunk_{12})$
$\|$
$P6_2 \quad :: \quad *(handoff\text{–}synch_2)$

It is trivial to verify that this program is deadlock free.

We merge the program $P6$ with $R6$, the result of the last merge, using the generalized version of Transformation 1 (see page 65) to produce the program $R7$. The conditions required for this transformation are satisfied, as explained below :

1. The programs $P6$ and $R6$ are deadlock free.

2. $\alpha P6 \cap \alpha R6 \; = \; \phi$.

3. The single iteration constraint holds for program $P6$.

4. The single iteration constraint holds for program $R6$.

5. The actions $if\text{–}no\text{–}trunk_1$ and $if\text{–}trunk_1$ appear only in the appropriate positions, as explained in condition 5 of the conditions for transformation 1.

$R7 =$

$$
\begin{array}{l}
mc1 :: \; *(handoff\ \text{–}synch_2 \rightarrow alloc\text{–}tranid_{1a} \rightarrow alloc\text{–}trunk_{12} \rightarrow \\
\qquad (\;\; (if\text{–}no\text{–}trunk_1 \rightarrow \; cancel\text{–}tranid_{1a} \rightarrow \\
\qquad\qquad\qquad\qquad exit_{1a} \rightarrow handoff_{12}) \\
\qquad [\!] \;\; (if\text{–}\; trunk_1 \rightarrow set\text{–}timer\text{–}hot_1 \rightarrow invoke_{12} \rightarrow \\
\qquad\qquad (\;\; (if\text{–}hot\text{–}exp_1 \rightarrow handoff_{12}) \\
\qquad\qquad [\!] \;\; (error\text{–}to\ \text{–}proc_{21} \rightarrow \\
\qquad\qquad\qquad\qquad ret\text{–}err\text{–}type_{21} \rightarrow handoff_{12}) \\
\qquad\qquad [\!] \;\; (ok\text{–}to\ \text{–}proc_{21} \rightarrow ret\text{–}ok\text{–}type_{21} \rightarrow \\
\qquad\qquad\qquad\qquad (\;\; (enter\text{–}\; invoke_2 \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad mob\text{–}inv\text{–}ch_{12} \rightarrow handoff_{12}) \\
\qquad\qquad\qquad\qquad [\!] \;\; (if\text{–}mhot\text{–}exp_1 \rightarrow handoff_{12})))))))
\end{array}
$$

$\|$

$$
\begin{array}{l}
mc2 :: \; *(handoff\text{–}synch_2 \rightarrow \\
\qquad\qquad\qquad\qquad (\;\; (if\text{–}no\text{–}trunk_1 \rightarrow handoff_{12})
\end{array}
$$

$$
\begin{array}{l}
[\!] \ (if\text{–}\ trunk_1 \rightarrow invoke_{12} \rightarrow \\
\quad (\ \ (if\text{–}hat\text{–}exp_2 \rightarrow handoff_{12}) \\
\quad [\!] \ (error\text{–}to\ \text{–}proc_{21} \rightarrow \\
\qquad\qquad ret\text{–}err\text{–}type_{21} \rightarrow handoff_{12}) \\
\quad [\!] \ (ok\text{–}to\ \text{–}proc_{21} \rightarrow ret\text{–}ok\text{–}type_{21} \rightarrow \\
\qquad\qquad (\ \ (enter\ \text{–}invoke_2 \rightarrow \\
\qquad\qquad\qquad mob\text{–}inv\text{–}ch_{12} \rightarrow handoff_{12}) \\
\qquad\qquad [\!] \ (if\text{–}mat\text{–}exp_1 \rightarrow handoff_{12}))))))
\end{array}
$$

From theorem 4, we have that $R7$ is deadlock free.

Using transformation 4 (see page 86) we now add a process $mb = *(handoff_{12})$ to the program $R7$. This process represents the mobile. The necessary conditions are satisfied as explained below :

1. The program $R7$ is deadlock free.

The merged program $R8$ is as follows :

$$
\begin{array}{l}
R8 = \\
mb \ :: \ *(handoff_{12}) \\
\| \\
\\
mc1 \ :: \ *(handoff\ \text{–}synch_2 \rightarrow alloc\text{–}tranid_{1a} \rightarrow alloc\text{–}trunk_{12} \rightarrow \\
\qquad\qquad (\ \ (if\text{–}no\text{–}trunk_1 \rightarrow \ cancel\text{–}tranid_{1a} \rightarrow \\
\qquad\qquad\qquad\qquad exit_{1a} \rightarrow handoff_{12}) \\
\qquad\qquad [\!] \ (if\text{–}\ trunk_1 \rightarrow set\text{–}timer\text{–}hot_1 \rightarrow invoke_{12} \rightarrow \\
\qquad\qquad\qquad (\ \ (if\text{–}hot\text{–}exp_1 \rightarrow handoff_{12}) \\
\qquad\qquad\qquad [\!] \ (error\text{–}to\ \text{–}proc_{21} \rightarrow \\
\qquad\qquad\qquad\qquad ret\text{–}err\text{–}type_{21} \rightarrow handoff_{12}) \\
\qquad\qquad\qquad [\!] \ (ok\text{–}to\ \text{–}proc_{21} \rightarrow ret\text{–}ok\text{–}type_{21} \rightarrow \\
\qquad\qquad\qquad\qquad (\ \ (enter\text{–}\ invoke_2 \rightarrow \\
\qquad\qquad\qquad\qquad\qquad mob\text{–}inv\text{–}ch_{12} \rightarrow handoff_{12}) \\
\qquad\qquad\qquad\qquad [\!] \ (if\text{–}mhot\text{–}exp_1 \rightarrow handoff_{12})))))))) \\
\| \\
\\
mc2 \ :: \ *(handoff\text{–}synch_2 \ \rightarrow \\
\qquad\qquad (\ \ (if\text{–}no\text{–}trunk_1 \rightarrow handoff_{12})
\end{array}
$$

$$
\begin{array}{l}
[\!] \ (if\text{–}\ trunk_1 \rightarrow invoke_{12} \rightarrow \\
\qquad (\ \ (if\text{–}hat\text{–}exp_2 \rightarrow handoff_{12}) \\
\qquad\ [\!] \ (error\text{–}to\ \text{–}proc_{21} \rightarrow \\
\qquad\qquad\qquad ret\text{–}err\text{–}type_{21} \rightarrow handoff_{12}) \\
\qquad\ [\!] \ (ok\text{–}to\ \text{–}proc_{21} \rightarrow ret\text{–}ok\text{–}type_{21} \rightarrow \\
\qquad\qquad\qquad (\ \ (enter\ \text{–}invoke_2 \rightarrow \\
\qquad\qquad\qquad\qquad\qquad mob\text{–}inv\text{–}ch_{12} \rightarrow handoff_{12}) \\
\qquad\qquad\qquad\ [\!] \ (if\text{–}mat\text{–}exp_1 \rightarrow handoff_{12}))))))
\end{array}
$$

From theorem 9, we have that $R8$ is deadlock free.

Note: The addition of a new process alters the mapping of processes to numbers. We now map the processes mb, $mc1$ and $mc2$ to the numbers 1, 2 and 3 respectively.

We are now in a position to use a series of Right-sequence introduction transformations (see page 13) to introduce all the local actions that we need to fill in the details in our design. The first set of introductions are those which are made in the process $mc1$. The subset in which the introductions take place is $\{2\}$ (process $mc1$). We list the Right-sequence introduction transformations that are to be applied :

1. $[if\text{–}hot\text{–}exp_1 / if\text{–}hot\text{–}exp_1 \rightarrow relres_{1a}]$
2. $[relres_{1a} / relres_{1a} \rightarrow canctranid_{1b}]$
3. $[canctranid_{1b} / canctranid_{1b} \rightarrow recov_{1a}]$
4. $[recov_{1a} / recov_{1a} \rightarrow exit_{1b}]$
5. $[ret\text{–}err\text{–}type_{21} / ret\text{–}err\text{–}type_{21} \rightarrow stphot_{1a}]$
6. $[stphot_{1a} / stphot_{1a} \rightarrow canctranid_{1c}]$
7. $[canctranid_{1c} / canctranid_{1c} \rightarrow recov_{1b}]$
8. $[recov_{1b} / recov_{1b} \rightarrow exit_{1c}]$

9. $[ret\text{–}ok\text{–}type_{21}/ret\text{–}ok\text{–}type_{21} \rightarrow stphot_{1b}]$

10. $[stphot_{1b}/stphot_{1b} \rightarrow setmhot_1]$

11. $[mob\text{–}inv\text{–}ch_{12}/mob\text{–}inv\text{–}ch_{12} \rightarrow stpmhot_1]$

12. $[stpmhot_1/stpmhot_1 \rightarrow complpath_{1a}]$

13. $[if\text{–}mhot\text{–}exp_1/if\text{–}mhot\text{–}exp_1 \rightarrow relres_{1b}]$

14. $[relres_{1b}/relres_{1b} \rightarrow canctranid_{1d}]$

15. $[canctranid_{1d}/canctranid_{1d} \rightarrow recov_{1c}]$

16. $[recov_{1c}/recov_{1c} \rightarrow exit_{1d}]$

The second set of Right sequence introduction transformations to be applied affect only process $mc2$. The subset in which the introductions take place is $\{3\}$ (process $mc2$):

1. $[invoke_{12}/invoke_{12} \rightarrow sethat_2]$

2. $[if\text{–}hat\text{–}exp_2/if\text{–}hat\text{–}exp_2 \rightarrow relres_{2b}]$

3. $[relres_{2b}/relres_{2b} \rightarrow exit_{2b}]$

4. $[error\text{–}to\text{–}proc_{21}/error\text{–}to\text{–}proc_{21} \rightarrow stphat_{2a}]$

5. $[ret\text{–}err\text{–}type_{21}/ret\text{–}err\text{–}type_{21} \rightarrow exit_{2a}]$

6. $[ok\text{–}to\text{–}proc_{21}/ok\text{–}to\text{–}proc_{21} \rightarrow stphat_{2b}]$

7. $[stphat_{2b}/stphat_{2b} \rightarrow alloctranid_{2b}]$

8. $[ret\text{–}ok\text{–}type_{21}/ret\text{–}ok\text{–}type_{21} \rightarrow setmat_2]$

9. $[enter\text{–}invoke_2/enter\text{–}invoke_2 \rightarrow CSSrecvd_2]$

10. $[CSSrecvd_2/CSSrecvd_2 \rightarrow stpmat_2]$

11. $[stpmat_2/stpmat_2 \rightarrow complpath_{2b}]$

12. $[if\text{–}mat\text{–}exp_1/if\text{–}mat\text{–}exp_1 \rightarrow relres_{2c}]$

13. $[relres_{2c}/relres_{2c} \rightarrow canctranid_{2c}]$

14. $[canctranid_{2c}/canctranid_{2c} \rightarrow exit_{2c}]$

Note that all the actions introduced correspond to local actions and do not have any impact on inter-process interaction.

After application of the above transformations, the program looks as follows::

$$
\begin{array}{l}
R9 = \\
mb \;\; :: \;\; *(handoff_{12}) \\
\| \\
\\
mc1 \;\; :: \;\; *\,(\; handoff\text{–}synch_2 \rightarrow alloc\text{–}tranid_{1a} \rightarrow alloc\text{–}trunk_{12} \rightarrow \\
\qquad (\;\; (if\text{–}no\text{–}trunk_1 \rightarrow cancel\text{–}tranid_{1a} \rightarrow exit_{1a} \rightarrow handoff_{12}) \\
\qquad [\!] \;\; (if\text{–}trunk_1 \rightarrow set\text{–}timer\text{–}hot_1 \rightarrow invoke_{12} \rightarrow \\
\qquad\qquad (\;\; (if\text{–}hot\text{–}exp_1 \rightarrow \\
\qquad\qquad\quad relres_{1a} \rightarrow canctranid_{1b} \rightarrow recov_{1a} \rightarrow \\
\qquad\qquad\qquad\qquad exit_{1b} \rightarrow handoff_{12}) \\
\qquad\qquad [\!] \;\; (error\text{–}to\text{–}proc_{21} \rightarrow \\
\qquad\qquad\quad ret\text{–}err\text{–}type_{21} \rightarrow stphot_{1a} \rightarrow \\
\qquad\qquad\qquad\quad canctranid_{1c} \rightarrow recov_{1b} \rightarrow \\
\qquad\qquad\qquad\qquad exit_{1c} \rightarrow handoff_{12}) \\
\qquad\qquad [\!] \;\; (ok\text{–}to\text{–}proc_{21} \;\rightarrow \\
\qquad\qquad\quad ret\text{–}ok\text{–}type_{21} \rightarrow stphot_{1b} \rightarrow setmhot_1 \rightarrow \\
\qquad\qquad\qquad (\;\; (enter\text{–}invoke_2 \rightarrow \\
\qquad\qquad\qquad\qquad mob\text{–}inv\text{–}ch_{12} \rightarrow stpmhot_1 \rightarrow \\
\qquad\qquad\qquad\qquad\quad complpath_{1a} \rightarrow handoff_{12}) \\
\qquad\qquad\qquad\quad [\!] \;\; (if\text{–}mhot\text{–}exp_1 \rightarrow relres_{1b} \rightarrow \\
\qquad\qquad\qquad\qquad canctranid_{1d} \rightarrow recov_{1c} \rightarrow \\
\qquad\qquad\qquad\qquad\quad exit_{1d} \rightarrow handoff_{12}))))))) \\
\\
\| \\
\\
mc2 \;\; :: \;\; *\,(handoff\text{–}synch_2 \rightarrow \\
\qquad (\;\; (if\text{–}no\text{–}trunk_1 \rightarrow handoff_{12}) \\
\qquad\quad [\!] \\
\qquad\quad (if\text{–}trunk_1 \rightarrow invoke_{12} \rightarrow sethat_2 \rightarrow \\
\qquad\qquad (\;\; (if\text{–}hat\text{–}exp_2 \rightarrow relres_{2b} \rightarrow exit_{2b} \rightarrow handoff_{12}) \\
\qquad\qquad [\!] \;\; (error\text{–}to\text{–}proc_{21} \rightarrow stphat_{2a} \rightarrow \\
\qquad\qquad\qquad ret\text{–}err\text{–}type_{21} \rightarrow exit_{2a} \rightarrow handoff_{12}) \\
\qquad\qquad [\!] \;\; (ok\text{–}to\text{–}proc_{21} \rightarrow stphat_{2b} \rightarrow \\
\qquad\qquad\qquad alloctranid_{2a} \rightarrow ret\text{–}ok\text{–}type_{21} \rightarrow setmat_2 \rightarrow \\
\qquad\qquad\qquad (\;\; (enter\text{–}invoke_2 \rightarrow CSSrecvd_2 \rightarrow \\
\qquad\qquad\qquad\quad stpmat_2 \rightarrow complpath_{2b} \rightarrow \\
\qquad\qquad\qquad\qquad mob\text{–}inv\text{–}ch_{12} \rightarrow handoff_{12})
\end{array}
$$

$$\| \ (if\text{–}mat\text{–}exp_1 \rightarrow relres_{2c} \rightarrow canctranid_{2a} \rightarrow exit_{2c} \rightarrow handoff_{12})))))))$$

From theorem 1, we have that $R9$ is deadlock free.

We are now ready to start developing the other functionalities related to a handoff. We also need to consider the case where $mc1$ has the stronger signal and therefore no handoff occurs. The program for switching center synchronization when the election results in the same manager (switching center $mc1$) being chosen is trivial since no handoff is attempted. We name this program $P9$.

$P9 =$
$P9_1 \quad :: \quad *(handoff_{11})$
$\|$
$P9_2 \quad :: \quad *(handoff\text{–}synch_1 \rightarrow st_{11} \rightarrow handoff_{11})$
$\|$
$P9_3 \quad :: \quad *(handoff\text{–}synch_1 \rightarrow st_{11} \rightarrow handoff_{11})$

From the problem description we know that the overall functionality we aim to model requires a program which can be composed by combining the processes of the above two programs $P9$ and $R9$ with the choice operator. This provides us with the motivation to merge the programs. We do our merge using the transformation 3 (see page 69). The subprograms satisfy the necessary conditions as explained below :

1. The covering action sets C_{R9} and C_{P9} for the programs $R9$ and $P9$ are $\{handoff\text{–}synch_2\}$ and $\{handoff\text{–}synch_1\}$. We note that $C_{P9} \bigcap \alpha R9 = \phi$ and $C_{R9} \bigcap \alpha P9 = \phi$. The respective process cover sets are $\{2,3\}$ and $\{2,3\}$. Clearly the first condition is satisfied.

2. The actions $\{handoff\text{–}synch_2\}$ and $\{handoff\text{–}synch_1\}$ have the property of end synchrony.

3. $\alpha P9 \cap \alpha R9 = \phi$.

4. The programs $R9$ and $P9$ are deadlock free.

The merged program, which we name $R10$, looks as follows:

$R10 =$

$$
\begin{array}{l}
mb \;\; :: \;\; *(handoff_{11} \\
\qquad [\!] \; handoff_{12}) \\
\\
\| \\
\\
mc1 \;\; :: \;\; *(\; handoff\text{–}synch_1 \rightarrow st_{11} \rightarrow handoff_{11} \\
\qquad\qquad [\!] \\
\qquad\qquad (\; handoff\text{–}synch_2 \rightarrow alloc\text{–}tranid_{1a} \rightarrow alloc\text{–}trunk_{12} \rightarrow \\
\qquad\qquad\qquad (\quad (if\text{–}no\text{–}trunk_1 \rightarrow cancel\text{–}tranid_{1a} \rightarrow exit_{1a} \rightarrow handoff_{12}) \\
\qquad\qquad\qquad\quad [\!] \; (if\text{–}trunk_1 \rightarrow set\text{–}timer\text{–}hot_1 \rightarrow invoke_{12} \rightarrow \\
\qquad\qquad\qquad\qquad (\quad (if\text{–}hot\text{–}exp_1 \rightarrow \\
\qquad\qquad\qquad\qquad\qquad relres_{1a} \rightarrow canctranid_{1b} \rightarrow recov_{1a} \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad exit_{1b} \rightarrow handoff_{12}) \\
\qquad\qquad\qquad\qquad [\!] \quad (error\text{–}to\text{–}proc_{21} \rightarrow \\
\qquad\qquad\qquad\qquad\qquad ret\text{–}err\text{–}type_{21} \rightarrow stphot_{1a} \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad canctranid_{1c} \rightarrow recov_{1b} \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad exit_{1c} \rightarrow handoff_{12}) \\
\qquad\qquad\qquad\qquad [\!] \quad (ok\text{–}to\text{–}proc_{21} \rightarrow \\
\qquad\qquad\qquad\qquad\qquad ret\text{–}ok\text{–}type_{21} \rightarrow stphot_{1b} \rightarrow setmhot_1 \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\quad (\quad (enter\text{–}invoke_2 \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad mob\text{–}inv\text{–}ch_{12} \rightarrow stpmhot_1 \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad complpath_{1a} \rightarrow handoff_{12}) \\
\qquad\qquad\qquad\qquad\qquad\quad [\!] \; (if\text{–}mhot\text{–}exp_1 \rightarrow relres_{1b} \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad canctranid_{1d} \rightarrow recov_{1c} \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad exit_{1d} \rightarrow handoff_{12}))))))) \\
\\
\| \\
\end{array}
$$

$$
\begin{array}{l}
mc2 \;::\; *(\; handoff\text{–}synch_1 \rightarrow st_{11} \rightarrow handoff_{11} \\
\quad \| \\
\quad (handoff\text{–}synch_2 \rightarrow \\
\qquad (\;\; (if\text{–}no\text{–}trunk_1 \rightarrow handoff_{12}) \\
\qquad\quad \| \\
\qquad\quad (if\text{–}trunk_1 \rightarrow invoke_{12} \rightarrow sethat_2 \rightarrow \\
\qquad\qquad (\;\; (if\text{–}hat\text{–}exp_2 \rightarrow relres_{2b} \rightarrow exit_{2b} \rightarrow handoff_{12}) \\
\qquad\qquad\;\; \| \;\; (error\text{–}to\text{–}proc_{21} \rightarrow stphat_{2a} \rightarrow \\
\qquad\qquad\qquad\quad ret\text{–}err\text{–}type_{21} \rightarrow exit_{2a} \rightarrow handoff_{12}) \\
\qquad\qquad\;\; \| \;\; (ok\text{–}to\text{–}proc_{21} \rightarrow stphat_{2b} \rightarrow \\
\qquad\qquad\qquad\quad alloctranid_{2a} \rightarrow ret\text{–}ok\text{–}type_{21} \rightarrow setmat_2 \rightarrow \\
\qquad\qquad\qquad\quad (\;\; (enter\text{–}invoke_2 \rightarrow CSSrecvd_2 \rightarrow \\
\qquad\qquad\qquad\qquad stpmat_2 \rightarrow complpath_{2b} \rightarrow \\
\qquad\qquad\qquad\qquad\quad mob\text{–}inv\text{–}ch_{12} \rightarrow handoff_{12}) \\
\qquad\qquad\qquad\quad\;\; \| \;\; (if\text{–}mat\text{–}exp_1 \rightarrow relres_{2c} \rightarrow \\
\qquad\qquad\qquad\qquad canctranid_{2a} \rightarrow exit_{2c} \rightarrow handoff_{12}))))))
\end{array}
$$

From theorem 6, we have that $R10$ is deadlock free.

We next construct a small program $P10$ which does some of the sequence of actions described on page 90 when the signal quality is discovered to be too low. The program for performing the handoff has already been built (it is $R10$). We therefore stop after the election. As before, we observe that this simple program is easy to check manually (in fact, its only computation is $below\text{–}thresh_1, msc\text{–}synch_1, poll\text{–}sig\text{–}chk$, $election, below\text{–}thresh_1 \ldots$).

$$
\begin{array}{l}
P10 = \\
P10_1 \quad :: \quad *(below\text{–}thresh_1 \rightarrow poll\text{–}sig\text{–}chk) \\
\| \\
P10_2 \quad :: \quad *(below\text{–}thresh_1 \rightarrow msc\text{–}synch_1 \rightarrow poll\text{–}sig\text{–}chk \rightarrow election) \\
\| \\
P10_3 \quad :: \quad *(msc\text{–}synch_1 \rightarrow poll\text{–}sig\text{–}chk \rightarrow election)
\end{array}
$$

Next we shall apply the generalized version of transformation 1 (see page 65) to the programs $P10$ and $R10$. The conditions required for this transformation are satisfied as explained below:

1. $P10$ and $R10$ are deadlock free.

2. $\alpha P10 \cap \alpha R10 = \phi$.

3. The single iteration constraint holds for $P10$ and $R10$.

4. The actions $handoff\text{–}synch_1$, $handoff\text{–}synch_2$, $handoff_{12}$ and $handoff_{11}$ appear in the appropriate positions in $R10$, as described in condition 5 of the conditions for transformation 1.

The program $R11$ which is produced by the merge follows:

$R11 =$

$$
\begin{array}{l}
mb \;\; :: \;\; *(below\text{–}thresh_1 \rightarrow poll\text{–}sig\text{–}chk \rightarrow \\
\qquad\qquad\qquad\qquad\qquad (handoff_{11} \\
\qquad\qquad\qquad\qquad\qquad \;[\!] \; handoff_{12}) \\
\qquad\qquad)
\end{array}
$$

$[\!]$

$$
\begin{array}{l}
mc1 \;\; :: \;\; *(below\text{–}thresh_1 \rightarrow msc\text{–}synch_1 \rightarrow poll\text{–}sig\text{–}chk \rightarrow election \rightarrow \\
\qquad\qquad (\; handoff\text{–}synch_1 \rightarrow st_{11} \rightarrow handoff_{11} \\
\qquad\qquad\qquad [\!] \\
\qquad\qquad\qquad (\; handoff\text{–}synch_2 \rightarrow alloc\text{–}tranid_{1a} \rightarrow alloc\text{–}trunk_{12} \rightarrow \\
\qquad\qquad\qquad\qquad (\quad (if\text{–}no\text{–}trunk_1 \rightarrow cancel\text{–}tranid_{1a} \rightarrow exit_{1a} \rightarrow handoff_{12}) \\
\qquad\qquad\qquad\qquad\quad [\!] \; (if\text{–}trunk_1 \rightarrow set\text{–}timer\text{–}hot_1 \rightarrow invoke_{12} \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\quad (\qquad (if\text{–}hot\text{–}exp_1 \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad relres_{1a} \rightarrow canctranid_{1b} \rightarrow recov_{1a} \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad exit_{1b} \rightarrow handoff_{12}) \\
\qquad\qquad\qquad\qquad\qquad\quad [\!] \qquad (error\text{–}to\text{–}proc_{21} \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad ret\text{–}err\text{–}type_{21} \rightarrow stphot_{1a} \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad canctranid_{1c} \rightarrow recov_{1b} \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad exit_{1c} \rightarrow handoff_{12}) \\
\qquad\qquad\qquad\qquad\qquad\quad [\!] \qquad (ok\text{–}to\text{–}proc_{21} \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad ret\text{–}ok\text{–}type_{21} \rightarrow stphot_{1b} \rightarrow setmhot_1 \rightarrow
\end{array}
$$

$$
\begin{array}{l}
(\quad (enter\text{--}invoke_2 \rightarrow \\
\quad\quad mob\text{--}inv\text{--}ch_{12} \rightarrow stpmhot_1 \rightarrow \\
\quad\quad\quad complpath_{1a} \rightarrow handoff_{12}) \\
\; [] \quad (if\text{--}mhot\text{--}exp_1 \rightarrow relres_{1b} \rightarrow \\
\quad\quad canctranid_{1d} \rightarrow recov_{1c} \rightarrow \\
\quad\quad\quad exit_{1d} \rightarrow handoff_{12})))))))
\end{array}
$$

)

$\|$

$$
\begin{array}{l}
mc2 :: *(msc\text{--}synch_1 \rightarrow poll\text{--}sig\text{--}chk \rightarrow election \rightarrow \\
\quad (\; handoff\text{--}synch_1 \rightarrow st_{11} \rightarrow handoff_{11} \\
\quad\quad [] \\
\quad\quad (handoff\text{--}synch_2 \rightarrow \\
\quad\quad\quad (\; (if\text{--}no\text{--}trunk_1 \rightarrow handoff_{12}) \\
\quad\quad\quad\quad [] \\
\quad\quad\quad\quad (if\text{--}trunk_1 \rightarrow invoke_{12} \rightarrow sethat_2 \rightarrow \\
\quad\quad\quad\quad\quad (\; (if\text{--}hat\text{--}exp_2 \rightarrow relres_{2b} \rightarrow exit_{2b} \rightarrow handoff_{12}) \\
\quad\quad\quad\quad\quad [] \; (error\text{--}to\text{--}proc_{21} \rightarrow stphat_{2a} \rightarrow \\
\quad\quad\quad\quad\quad\quad\quad ret\text{--}err\text{--}type_{21} \rightarrow exit_{2a} \rightarrow handoff_{12}) \\
\quad\quad\quad\quad\quad [] \; (ok\text{--}to\text{--}proc_{21} \rightarrow stphat_{2b} \rightarrow \\
\quad\quad\quad\quad\quad\quad\quad alloctranid_{2a} \rightarrow ret\text{--}ok\text{--}type_{21} \rightarrow setmat_2 \rightarrow \\
\quad\quad\quad\quad\quad\quad\quad (\; (enter\text{--}invoke_2 \rightarrow CSSrecvd_2 \rightarrow \\
\quad\quad\quad\quad\quad\quad\quad\quad stpmat_2 \rightarrow complpath_{2b} \rightarrow \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad mob\text{--}inv\text{--}ch_{12} \rightarrow handoff_{12}) \\
\quad\quad\quad\quad\quad\quad\quad [] \; (if\text{--}mat\text{--}exp_1 \rightarrow relres_{2c} \rightarrow \\
\quad\quad\quad\quad\quad\quad\quad\quad canctranid_{2a} \rightarrow exit_{2c} \rightarrow handoff_{12}))))))) \\
)
\end{array}
$$

From theorem 4, we have that $R11$ is deadlock free.

$R11$ models the complete set of actions when the signal is below the threshold level. In case the level of the signal is above threshold, we do not need to perform any of these actions. This is modeled by the following program $P11$:

$P11 =$

$P11_1$ :: $*(above\text{–}thresh_1)$
$\|$
$P11_2$ :: $*(above\text{–}thresh_1)$

We merge this small program with $R11$ using transformation 2 (see page 67). The necessary conditions are satisfied as explained below:

1. $R11$ is deadlock free.

2. $above\text{–}thresh_1 \notin \alpha R11$.

The merging gives us the following program, which we name $R12$.

$R12 =$

mb :: $*($ $above\text{–}thresh_1$
$[\!]$
$(below\text{–}thresh_1 \rightarrow poll\text{–}sig\text{–}chk \rightarrow$
$(handoff_{11}$
$[\!]\ handoff_{12})$
$)$
$)$

$\|$

$mc1$ :: $*($ $above\text{–}thresh_1$
$[\!]$
$(below\text{–}thresh_1 \rightarrow msc\text{–}synch_1 \rightarrow poll\text{–}sig\text{–}chk \rightarrow election \rightarrow$
$(\ handoff\text{–}synch_1 \rightarrow st_{11} \rightarrow handoff_{11})$
$[\!]$
$(\ handoff\text{–}synch_2 \rightarrow alloc\text{–}tranid_{1a} \rightarrow alloc\text{–}trunk_{12} \rightarrow$
$($ $(if\text{–}no\text{–}trunk_1 \rightarrow cancel\text{–}tranid_{1a} \rightarrow exit_{1a} \rightarrow handoff_{12})$
$[\!]$ $(if\text{–}trunk_1 \rightarrow set\text{–}timer\text{–}hot_1 \rightarrow invoke_{12} \rightarrow$
$($ $(if\text{–}hot\text{–}exp_1 \rightarrow$
$relres_{1a} \rightarrow canctranid_{1b} \rightarrow recov_{1a} \rightarrow$
$exit_{1b} \rightarrow handoff_{12})$
$[\!]$ $(error\text{–}to\text{–}proc_{21} \rightarrow$
$ret\text{–}err\text{–}type_{21} \rightarrow stphot_{1a} \rightarrow$
$canctranid_{1c} \rightarrow recov_{1b} \rightarrow$
$exit_{1c} \rightarrow handoff_{12})$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\qquad [\!] \ (ok\text{–}to\text{–}proc_{21} \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad ret\text{–}ok\text{–}type_{21} \rightarrow stphot_{1b} \rightarrow setmhot_1 \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\quad (enter\text{–}invoke_2 \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad mob\text{–}inv\text{–}ch_{12} \rightarrow stpmhot_1 \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad complpath_{1a} \rightarrow handoff_{12}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad [\!] \quad (if\text{–}mhot\text{–}exp_1 \rightarrow relres_{1b} \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad canctranid_{1d} \rightarrow recov_{1c} \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad exit_{1d} \rightarrow handoff_{12})))))))
\end{array}
$$

$$
\begin{array}{l}
\qquad\qquad\qquad\quad) \\
\qquad\qquad\quad)
\end{array}
$$

$$
\|
$$

$$
\begin{array}{l}
mc2 \ :: \ {}^*(msc\text{–}synch_1 \rightarrow poll\text{–}sig\text{–}chk \rightarrow election \rightarrow \\
\qquad (\ handoff\text{–}synch_1 \rightarrow st_{11} \rightarrow handoff_{11}) \\
\qquad\quad [\!] \\
\qquad (handoff\text{–}synch_2 \rightarrow \\
\qquad\quad (\ (if\text{–}no\text{–}trunk_1 \rightarrow handoff_{12}) \\
\qquad\qquad [\!] \\
\qquad\qquad (if\text{–}trunk_1 \rightarrow invoke_{12} \rightarrow sethat_2 \rightarrow \\
\qquad\qquad\qquad\quad (\ (if\text{–}hat\text{–}exp_2 \rightarrow relres_{2b} \rightarrow exit_{2b} \rightarrow handoff_{12}) \\
\qquad\qquad\qquad\quad [\!] \ (error\text{–}to\text{–}proc_{21} \rightarrow stphat_{2a} \rightarrow \\
\qquad\qquad\qquad\qquad ret\text{–}err\text{–}type_{21} \rightarrow exit_{2a} \rightarrow handoff_{12}) \\
\qquad\qquad\qquad\quad [\!] \ (ok\text{–}to\text{–}proc_{21} \rightarrow stphat_{2b} \rightarrow \\
\qquad\qquad\qquad\qquad alloctranid_{2a} \rightarrow ret\text{–}ok\text{–}type_{21} \rightarrow setmat_2 \rightarrow \\
\qquad\qquad\qquad\qquad (\ (enter\text{–}invoke_2 \rightarrow CSSrecvd_2 \rightarrow \\
\qquad\qquad\qquad\qquad\qquad stpmat_2 \rightarrow complpath_{2b} \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad mob\text{–}inv\text{–}ch_{12} \rightarrow handoff_{12}) \\
\qquad\qquad\qquad\qquad\quad [\!] \ (if\text{–}mat\text{–}exp_1 \rightarrow relres_{2c} \rightarrow \\
\qquad\qquad\qquad\qquad\qquad canctranid_{2a} \rightarrow exit_{2c} \rightarrow handoff_{12})))))))
\end{array}
$$

$$
\qquad)
$$

From theorem 5, we have that $R12$ is deadlock free.

The signal level check action $signal\text{–}chk_i$ precedes the actions taken in this program. It makes the managing msc interact with a mobile to determine the strength of the signal between them. The level check between the serving switching center

$(mc1)$ and the mobile is modeled by the following program $P12$:

$P12 =$

$P12_1$ :: $*(signal\text{–}chk_1)$

$\|$

$P12_2$ :: $*(signal\text{–}chk_1)$

We will merge this program with $R12$ using transformation 1.2 (see page 84) to obtain the program $R13$. The necessary conditions are satisfied as explained below :

1. $signal\text{–}chk_1 \bigcap \alpha R12 = \phi$

2. The covering actions for the program $R12$ are $\{above\text{–}thresh_1, below\text{–}thresh_1\}$. The corresponding process cover sets are $\{1,2\}$ and $\{1,2\}$. We introduce $signal\text{–}chk_1$ in front of both the processes mb and $mc1$.

3. Both $above\text{–}thresh_1$ and $below\text{–}thresh_1$ have the property of end synchrony. The processes in the cover set of the program $R12$ make exactly one iteration each for every path of computation from initial state $R12$ back to state $R12$ The process which does not have a covering action, $mc2$, either does a complete iteration or does not participate in any action of the path (this happens when $above\text{–}thresh_1$ is executed).

4. The program $R12$ is deadlock free.

The resulting program is $R13$:

$R13 =$

mb :: * $(signal\text{–}chk_1 \rightarrow$

$$(\; above\text{–}thresh_1$$
$$[\!]$$
$$(below\text{–}thresh_1 \rightarrow poll\text{–}sig\text{–}chk \rightarrow$$
$$(handoff_{11}$$
$$[\!] \; handoff_{12})$$
$$))$$
$$)$$

$$\|$$

$$mc1 \;\; :: \; *(signal\text{–}chk_1 \rightarrow$$
$$(\;\; above\text{–}thresh_1$$
$$[\!]$$
$$(below\text{–}thresh_1 \rightarrow msc\text{–}synch_1 \rightarrow poll\text{–}sig\text{–}chk \rightarrow election \rightarrow$$
$$(\;\; handoff\text{–}synch_1 \rightarrow st_{11} \rightarrow handoff_{11}$$
$$[\!]$$
$$(\; handoff\text{–}synch_2 \rightarrow alloc\text{–}tranid_{1a} \rightarrow alloc\text{–}trunk_{12} \rightarrow$$
$$(\;\; (if\text{–}no\text{–}trunk_1 \rightarrow cancel\text{–}tranid_{1a} \rightarrow exit_{1a} \rightarrow handoff_{12})$$
$$[\!] \; (if\text{–}trunk_1 \rightarrow set\text{–}timer\text{–}hot_1 \rightarrow invoke_{12} \rightarrow$$
$$(\;\;\; (if\text{–}hot\text{–}exp_1 \rightarrow$$
$$relres_{1a} \rightarrow canctranid_{1b} \rightarrow recov_{1a} \rightarrow$$
$$exit_{1b} \rightarrow handoff_{12})$$
$$[\!] \;\;\; (error\text{–}to\text{–}proc_{21} \rightarrow$$
$$ret\text{–}err\text{–}type_{21} \rightarrow stphot_{1a} \rightarrow$$
$$canctranid_{1c} \rightarrow recov_{1b} \rightarrow$$
$$exit_{1c} \rightarrow handoff_{12})$$
$$[\!] \;\;\; (ok\text{–}to\text{–}proc_{21} \rightarrow$$
$$ret\text{–}ok\text{–}type_{21} \rightarrow stphot_{1b} \rightarrow setmhot_1 \rightarrow$$
$$(\;\; (enter\text{–}invoke_2 \rightarrow$$
$$mob\text{–}inv\text{–}ch_{12} \rightarrow stpmhot_1 \rightarrow$$
$$complpath_{1a} \rightarrow handoff_{12})$$
$$[\!] \;\; (if\text{–}mhot\text{–}exp_1 \rightarrow relres_{1b} \rightarrow$$
$$canctranid_{1d} \rightarrow recov_{1c} \rightarrow$$
$$exit_{1d} \rightarrow handoff_{12})))))))$$

$$)$$
$$)$$
$$)$$
$$\|$$

$$mc2 \; :: \; *(msc\text{–}synch_1 \rightarrow poll\text{–}sig\text{–}chk \rightarrow election \rightarrow$$

$$
\begin{array}{l}
(\quad handoff\text{–}synch_1 \rightarrow st_{11} \rightarrow handoff_{11} \\
\quad \| \\
\quad (handoff\text{–}synch_2 \rightarrow \\
\quad\quad (\ (if\text{–}no\text{–}trunk_1 \rightarrow handoff_{12}) \\
\quad\quad\quad \| \\
\quad\quad\quad (if\text{–}trunk_1 \rightarrow invoke_{12} \rightarrow sethat_2 \rightarrow \\
\quad\quad\quad\quad (\quad (if\text{–}hat\text{–}exp_2 \rightarrow relres_{2b} \rightarrow exit_{2b} \rightarrow handoff_{12}) \\
\quad\quad\quad\quad\ \| \ (error\text{–}to\text{–}proc_{21} \rightarrow stphat_{2a} \rightarrow \\
\quad\quad\quad\quad\quad\quad ret\text{–}err\text{–}type_{21} \rightarrow exit_{2a} \rightarrow handoff_{12}) \\
\quad\quad\quad\quad\ \| \ (ok\text{–}to\text{–}proc_{21} \rightarrow stphat_{2b} \rightarrow \\
\quad\quad\quad\quad\quad\quad alloctranid_{2a} \rightarrow ret\text{–}ok\text{–}type_{21} \rightarrow setmat_2 \rightarrow \\
\quad\quad\quad\quad\quad\quad (\quad (enter\text{–}invoke_2 \rightarrow CSSrecvd_2 \rightarrow \\
\quad\quad\quad\quad\quad\quad\quad\quad stpmat_2 \rightarrow complpath_{2b} \rightarrow \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad mob\text{–}inv\text{–}ch_{12} \rightarrow handoff_{12}) \\
\quad\quad\quad\quad\quad\quad\quad\ \| \ (if\text{–}mat\text{–}exp_1 \rightarrow relres_{2c} \rightarrow \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad canctranid_{2a} \rightarrow exit_{2c} \rightarrow handoff_{12}))))))) \quad)
\end{array}
$$

From theorem 8, we have that $R13$ is deadlock free.

Proceeding in a symmettrical manner, we can build the deadlock free program $R14$ which models the situation where $mc2$ is the manager of the mobile and the signal check interaction is done between it and the mobile.

$R14 =$

$$
\begin{array}{l}
mb \quad : \quad *(signal\text{–}chk_2 \rightarrow \\
\qquad\qquad\qquad\qquad (\quad above\text{–}thresh_2 \\
\qquad\qquad\qquad\qquad\quad \| \\
\qquad\qquad\qquad\qquad\quad (below\text{–}thresh_2 \rightarrow poll\text{–}sig\text{–}chk \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (handoff_{21} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \| \ handoff_{22}) \\
\qquad\qquad\qquad\qquad\quad) \\
\qquad\qquad\qquad\quad) \\
\qquad\quad)
\end{array}
$$

$\|$

$$
\begin{array}{l}
mc1 \quad : \quad * \ (msc\text{–}synch_2 \rightarrow poll\text{–}sig\text{–}chk \rightarrow election \rightarrow \\
\qquad\quad (\ handoff\text{–}synch_2 \rightarrow st_{22} \rightarrow handoff_{22}
\end{array}
$$

$$
\begin{array}{l}
\quad [\!] \\
\quad (handoff\text{–}synch_1 \rightarrow \\
\quad\ (\ \ (if\text{–}no\text{–}trunk_2 \rightarrow handoff_{21}) \\
\qquad [\!] \\
\qquad (if\text{–}trunk_2 \rightarrow invoke_{21} \rightarrow sethat_1 \rightarrow \\
\qquad\qquad (\ \ (if\text{–}hat\text{–}exp_1 \rightarrow relres_{1y} \rightarrow exit_{1y} \rightarrow handoff_{21}) \\
\qquad\qquad\ [\!]\ (error\text{–}to\text{–}proc_{12} \rightarrow stphat_{1x} \rightarrow \\
\qquad\qquad\qquad ret\text{–}err\text{–}type_{12} \rightarrow exit_{1x} \rightarrow handoff_{21}) \\
\qquad\qquad\ [\!]\ (ok\text{–}to\text{–}proc_{12} \rightarrow stphat_{1y} \rightarrow \\
\qquad\qquad\qquad alloctranid_{1x} \rightarrow ret\text{–}ok\text{–}type_{12} \rightarrow setmat_1 \rightarrow \\
\qquad\qquad\qquad (\ \ (enter\text{–}invoke_1 \rightarrow CSSrecvd_1 \rightarrow \\
\qquad\qquad\qquad\qquad stpmat_1 \rightarrow complpath_{1y} \rightarrow \\
\qquad\qquad\qquad\qquad\qquad mob\text{–}inv\text{–}ch_{21} \rightarrow handoff_{21}) \\
\qquad\qquad\qquad\ [\!]\ (if\text{–}mat\text{–}exp_2 \rightarrow relres_{1z} \rightarrow \\
\qquad\qquad\qquad\qquad canctranid_{1x} \rightarrow exit_{1z} \rightarrow handoff_{21}))))))) \\
\quad)
\end{array}
$$

$$
[\!]
$$

$$
\begin{array}{l}
mc2\ \ :\ *\ (signal\text{–}chk_2 \rightarrow \\
\qquad (\ \ above\text{–}thresh_2 \\
\qquad\quad [\!] \\
\qquad\quad (below\text{–}thresh_2 \rightarrow msc\text{–}synch_2 \rightarrow poll\text{–}sig\text{–}chk \rightarrow election \rightarrow \\
\qquad\qquad (\ \ handoff\text{–}synch_2 \rightarrow st_{22} \rightarrow handoff_{22}) \\
\qquad\qquad\quad [\!] \\
\qquad\qquad\quad (\ handoff\text{–}synch_1 \rightarrow alloc\text{–}tranid_{2x} \rightarrow alloc\text{–}trunk_{21} \rightarrow \\
\qquad\qquad\qquad (\ \ (if\text{–}no\text{–}trunk_2 \rightarrow cancel\text{–}tranid_{2x} \rightarrow exit_{2x} \rightarrow handoff_{21}) \\
\qquad\qquad\qquad\ [\!]\ (if\text{–}trunk_2 \rightarrow set\text{–}timer\text{–}hot_2 \rightarrow invoke_{21} \rightarrow \\
\qquad\qquad\qquad\qquad (\ \ \ (if\text{–}hot\text{–}exp_2 \rightarrow \\
\qquad\qquad\qquad\qquad\qquad relres_{2x} \rightarrow canctranid_{2y} \rightarrow recov_{2x} \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad exit_{2y} \rightarrow handoff_{21}) \\
\qquad\qquad\qquad\qquad\ [\!]\ \ (error\text{–}to\text{–}proc_{12} \rightarrow \\
\qquad\qquad\qquad\qquad\qquad ret\text{–}err\text{–}type_{12} \rightarrow stphot_{2x} \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad canctranid_{2z} \rightarrow recov_{2y} \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad exit_{2z} \rightarrow handoff_{21}) \\
\qquad\qquad\qquad\qquad\ [\!]\ \ (ok\text{–}to\text{–}proc_{12}\ \rightarrow \\
\qquad\qquad\qquad\qquad\qquad ret\text{–}ok\text{–}type_{12} \rightarrow stphot_{2y} \rightarrow setmhot_2 \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\quad (\ \ (enter\text{–}invoke_1 \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad mob\text{–}inv\text{–}ch_{21} \rightarrow stpmhot_2 \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad complpath_{2x} \rightarrow handoff_{21}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\ [\!]\ (if\text{–}mhot\text{–}exp_2 \rightarrow relres_{2y} \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad canctranid_{2w} \rightarrow recov_{2z} \rightarrow
\end{array}
$$

$exit_{2w} \rightarrow handoff_{21})))))))$

)

)

)

We are one step away to giving a final shape to our program. We still require to merge the programs $R13$ and $R14$ since we may have either of $mc1$ or $mc2$ as the manager of the mobile. This suggests use of transformation 3 (see page 69). Let us see if $R13$ and $R14$ satisfy the necessary conditions.

1. The covering action sets of $R13$ and $R14$ are $\{signal\text{–}chk_1\}$ and $\{signal\text{–}chk_2\}$ respectively. The corresponding process cover sets are $\{1,2\}$ (for $signal\text{–}chk_1$) and $\{1,3\}$ (for $signal\text{–}chk_2$). Clearly their intersection is nonempty. Also $signal\text{–}chk_1 \notin \alpha R13$ and $signal\text{–}chk_2 \notin \alpha R14$. Thus conditions 1(a) and 1(b) are satisfied.

2. The action $signal\text{–}chk_1$ has the property of end synchrony with respect to the program $R13$ (the last action of an iteration starting with $signal\text{–}chk_1$ could be any of $\{above\text{–}thresh_1, handoff_{11}, handoff_{12}\}$). Similarly $signal\text{–}chk_2$ has the property of end synchrony with respect to the program $R14$.

3. We observe from inspection that for all actions a such that $a \in \alpha R13 \cap \alpha R14$, if a is in the alphabet of a process of $R13$, then it is also in the alphabet of the process with the same index in $R14$ with which it will be merged.
   ($\alpha R13 \cap \alpha R14 = \{handoff_{11}, handoff_{12}, poll\text{–}sig\text{–}chk, election,$
   $handoff\text{–}synch_1, handoff\text{–}synch_2\}$)

4. The programs $R13$ and $R14$ are deadlock free.

Since all the necessary conditions are satisfied, we can merge the programs $R13$ and $R14$ using transformation 3 to obtain the final form of our program ($R15$):

$R15 =$

$mb :: *(signal\text{–}chk_1 \rightarrow (above\text{–}thresh_1$
$\quad [\!] (below\text{–}thresh_1 \rightarrow poll\text{–}sig\text{–}chk \rightarrow (handoff_{11}$
$\quad\quad [\!] handoff_{12}))$
$\quad) [\!]$
$\quad (signal\text{–}chk_2 \rightarrow (above\text{–}thresh_2$
$\quad [\!] (below\text{–}thresh_2 \rightarrow poll\text{–}sig\text{–}chk \rightarrow (handoff_{21}$
$\quad\quad [\!] handoff_{22}))$
$\quad)$

$\|$

$mc1 :: *(signal\text{–}chk_1 \rightarrow$
$\quad (above\text{–}thresh_1$
$\quad\quad [\!]$
$\quad\quad (below\text{–}thresh_1 \rightarrow msc\text{–}synch_1 \rightarrow poll\text{–}sig\text{–}chk \rightarrow election \rightarrow$
$\quad\quad\quad (handoff\text{–}synch_1 \rightarrow st_{11} \rightarrow handoff_{11}$
$\quad\quad\quad\quad [\!]$
$\quad\quad\quad\quad (handoff\text{–}synch_2 \rightarrow alloc\text{–}tranid_{1a} \rightarrow alloc\text{–}trunk_{12} \rightarrow$
$\quad\quad\quad\quad\quad ((if\text{–}no\text{–}trunk_1 \rightarrow cancel\text{–}tranid_{1a} \rightarrow exit_{1a} \rightarrow handoff_{12})$
$\quad\quad\quad\quad\quad\quad [\!] (if\text{–}trunk_1 \rightarrow set\text{–}timer\text{–}hot_1 \rightarrow invoke_{12} \rightarrow$
$\quad\quad\quad\quad\quad\quad\quad ((if\text{–}hot\text{–}exp_1 \rightarrow$
$\quad\quad\quad\quad\quad\quad\quad\quad relres_{1a} \rightarrow canctranid_{1b} \rightarrow recov_{1a} \rightarrow$
$\quad\quad\quad\quad\quad\quad\quad\quad\quad exit_{1b} \rightarrow handoff_{12})$
$\quad\quad\quad\quad\quad\quad\quad\quad [\!] (error\text{–}to\text{–}proc_{21} \rightarrow$
$\quad\quad\quad\quad\quad\quad\quad\quad\quad ret\text{–}err\text{–}type_{21} \rightarrow stphot_{1a} \rightarrow$
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad canctranid_{1c} \rightarrow recov_{1b} \rightarrow$
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad exit_{1c} \rightarrow handoff_{12})$
$\quad\quad\quad\quad\quad\quad\quad\quad [\!] (ok\text{–}to\text{–}proc_{21} \rightarrow$
$\quad\quad\quad\quad\quad\quad\quad\quad\quad ret\text{–}ok\text{–}type_{21} \rightarrow stphot_{1b} \rightarrow setmhot_1 \rightarrow$
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad ((enter\text{–}invoke_2 \rightarrow$
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad mob\text{–}inv\text{–}ch_{12} \rightarrow stpmhot_1 \rightarrow$
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad complpath_{1a} \rightarrow handoff_{12})$
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad [\!] (if\text{–}mhot\text{–}exp_1 \rightarrow relres_{1b} \rightarrow$
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad canctranid_{1d} \rightarrow recov_{1c} \rightarrow$
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad exit_{1d} \rightarrow handoff_{12})))))))$

$))$

$\square$

$(msc\text{--}synch_2 \rightarrow poll\text{--}sig\text{--}chk \rightarrow election \rightarrow$
$(\; handoff\text{--}synch_2 \rightarrow st_{22} \rightarrow handoff_{22}$
$\square$
$(handoff\text{--}synch_1 \rightarrow$
$(\; (if\text{--}no\text{--}trunk_2 \rightarrow handoff_{21})$
$\square$
$(if\text{--}trunk_2 \rightarrow invoke_{21} \rightarrow sethat_1 \rightarrow$
$(\; (if\text{--}hat\text{--}exp_1 \rightarrow relres_{1y} \rightarrow exit_{1y} \rightarrow handoff_{21})$
$\square \; (error\text{--}to\text{--}proc_{12} \rightarrow stphat_{1x} \rightarrow$
$ret\text{--}err\text{--}type_{12} \rightarrow exit_{1x} \rightarrow handoff_{21})$
$\square \; (ok\text{--}to\text{--}proc_{12} \rightarrow stphat_{1y} \rightarrow$
$alloctranid_{1x} \rightarrow ret\text{--}ok\text{--}type_{12} \rightarrow setmat_1 \rightarrow$
$(\; (enter\text{--}invoke_1 \rightarrow CSSrecvd_1 \rightarrow$
$stpmat_1 \rightarrow complpath_{1y} \rightarrow$
$mob\text{--}inv\text{--}ch_{21} \rightarrow handoff_{21})$
$\square \; (if\text{--}mat\text{--}exp_2 \rightarrow relres_{1z} \rightarrow$
$canctranid_{1x} \rightarrow exit_{1z} \rightarrow handoff_{21})))))))$
$)$

$\|$

$mc2 \; :: \; *(msc\text{--}synch_1 \rightarrow poll\text{--}sig\text{--}chk \rightarrow election \rightarrow$
$(\; handoff\text{--}synch_1 \rightarrow st_{11} \rightarrow handoff_{11}$
$\square$
$(handoff\text{--}synch_2 \rightarrow$
$(\; (if\text{--}no\text{--}trunk_1 \rightarrow handoff_{12})$
$\square$
$(if\text{--}trunk_1 \rightarrow invoke_{12} \rightarrow sethat_2 \rightarrow$
$(\; (if\text{--}hat\text{--}exp_2 \rightarrow relres_{2b} \rightarrow exit_{2b} \rightarrow handoff_{12})$
$\square \; (error\text{--}to\text{--}proc_{21} \rightarrow stphat_{2a} \rightarrow$
$ret\text{--}err\text{--}type_{21} \rightarrow exit_{2a} \rightarrow handoff_{12})$
$\square \; (ok\text{--}to\text{--}proc_{21} \rightarrow stphat_{2b} \rightarrow$
$alloctranid_{2a} \rightarrow ret\text{--}ok\text{--}type_{21} \rightarrow setmat_2 \rightarrow$
$(\; (enter\text{--}invoke_2 \rightarrow CSSrecvd_2 \rightarrow$
$stpmat_2 \rightarrow complpath_{2b} \rightarrow$
$mob\text{--}inv\text{--}ch_{12} \rightarrow handoff_{12})$
$\square \; (if\text{--}mat\text{--}exp_1 \rightarrow relres_{2c} \rightarrow$

$$canctranid_{2a} \rightarrow exit_{2c} \rightarrow handoff_{12})))))))\ )$$

$$\|$$

$$(signal\text{–}chk_2 \rightarrow$$
$$(\ above\text{–}thresh_2$$
$$\|$$
$$(below\text{–}thresh_2 \rightarrow msc\text{–}synch_2 \rightarrow poll\text{–}sig\text{–}chk \rightarrow election \rightarrow$$
$$(\ handoff\text{–}synch_2 \rightarrow st_{22} \rightarrow handoff_{22})$$
$$\|$$
$$(\ handoff\text{–}synch_1 \rightarrow alloc\text{–}tranid_{2x} \rightarrow alloc\text{–}trunk_{21} \rightarrow$$
$$(\ (if\text{–}no\text{–}trunk_2 \rightarrow cancel\text{–}tranid_{2x} \rightarrow exit_{2x} \rightarrow handoff_{21})$$
$$\|\ (if\text{–}trunk_2 \rightarrow set\text{–}timer\text{–}hot_2 \rightarrow invoke_{21} \rightarrow$$
$$(\ (if\text{–}hot\text{–}exp_2 \rightarrow$$
$$relres_{2x} \rightarrow canctranid_{2y} \rightarrow recov_{2x} \rightarrow$$
$$exit_{2y} \rightarrow handoff_{21})$$
$$\|\ (error\text{–}to\text{–}proc_{12} \rightarrow$$
$$ret\text{–}err\text{–}type_{12} \rightarrow stphot_{2x} \rightarrow$$
$$canctranid_{2z} \rightarrow recov_{2y} \rightarrow$$
$$exit_{2z} \rightarrow handoff_{21})$$
$$\|\ (ok\text{–}to\text{–}proc_{12} \rightarrow$$
$$ret\text{–}ok\text{–}type_{12} \rightarrow stphot_{2y} \rightarrow setmhot_2 \rightarrow$$
$$(\ (enter\text{–}invoke_1 \rightarrow$$
$$mob\text{–}inv\text{–}ch_{21} \rightarrow stpmhot_2 \rightarrow$$
$$complpath_{2x} \rightarrow handoff_{21})$$
$$\|\ (if\text{–}mhot\text{–}exp_2 \rightarrow relres_{2y} \rightarrow$$
$$canctranid_{2w} \rightarrow recov_{2z} \rightarrow$$
$$exit_{2w} \rightarrow handoff_{21})))))))$$

$$)))$$

From theorem 6, we have that $R15$ is deadlock free.

Chapter 6

Conclusions

In this work we have described syntactic transformations for program refinement as well as merging. Thus we have support for both top-down (refinement) as well as bottom-up (merging) program development. We proved formally that our transformations preserve deadlock-freedom. The example we presented illustrates how a combination of the refinement and merging methodologies may be used to develop programs.

The essence of the ideas presented is to identify sets of conditions the enforcement of which enables property preserving process/program merging or refinement to be done. *It needs to be emphasized that every time we check a program to ensure that it satisfies a particular condition, we need not start from scratch.* Since the programs participating in the transformations are presumably built the same way we *know* a lot about their structure, and this knowledge can be effectively used to reduce the effort spent on checking whether the prescribed conditions hold for the given program.

Attempting a verification oriented approach would require extremely laborious proofs. A proof technique proposed in the paper [NF89] uses an extension of co-operating proofs to handle the verification of distributed programs with multiparty

interactions. Cooperating proofs are a two level proof system. In the first stage local proofs for the distinct processes are designed using certain assumptions about the environment's (i.e., the other processes) behavior. In the second stage the local proofs are checked for mutual consistency of all the assumptions made.

In this proof system global properties such as deadlock freedom of a program would be verified by constructing a global invariant to be preserved by the program. Establishing preservation of the invariant involves use of the global interaction axiom [NF89] and formation rules discussed in the paper [KNW80]. As is evident from the examples presented in this latter paper, this involves a very complicated and laborious process. It is necessary to compose the local proofs of processes (sets of pre and post conditions) with the invariant. Each action in the program requires a formulation of a set of pre and post conditions. In the worst case the number of criteria to be validated at the second level of the proof (i.e., when the local proofs are being composed with the global invariant) may be on the order of the product of the length of the processes being verified for a global property.

The number of applications of transformations in our methodology is linear in the length of the program. In the worst case we have sequences of refinement transformations, each of which add a single new action to the program. However, for the most part we expect to have introductions which apply simultaneously at more than one point. Merge transformations enable us to tackle complicated constructs and add the choice or the sequence operators to the program. Also our methodology is easier to automate and avoids the pitfalls of post-development verification of programs.

Admittedly the transformations are restricted in scope. At present only the deadlock freedom property is guaranteed to be preserved. The next goal is to verify

that the same transformations preserve other correctness properties, such as general safety and liveness. The underlying language also needs to be enhanced with constructs such as variables, expressions, and guards. Our transformations should also be shown to preserve the single iteration condition in order to reduce the effort spent in using them.

Correctness-preserving transformations for distributed systems are, in principle, a foundation for the eventual goal of compiling abstract specifications into architecturally adequate code. Those who find that objective too distant should, nevertheless, be interested in the medium-term goal of automating certain laborious and error-prone parts of the development process. An interactive compiler that handles much of the labor — and is guaranteed not to introduce the deadlocks and other errors that plague concurrent systems — would be valuable, even if it still depends heavily on human design creativity. This research is designed to support both the medium- and the long-term goals.

Bibliography

[BCMD90] J.R. Burch, E.M. Clarke, K.L. McMillan, and D.L. Dill. Sequential circuit verification using symbolic model checking . Proceedings of the 27th ACM/IEEE Design Automation Conference. IEEE Computer Society Press, June 1990.

[BKS83] R.J.R. Back, R. Kurki-Suonio, "Decentralization of Process Nets With Centralized Control," Distributed Computing vol. 3, pp. 73-87, 1989.

[BKS85] R.J.R. Back, R. Kurki-Suonio, "Serializability in Distributed Systems With Handshaking", TR 85-109, CMU, 1985.

[CBGM91] E.M. Clarke, J.R. Burch, O. Grumberg, D. E. Long, and K.L. McMillan. Royal Society, London October 3-4, 1991.

[CLM89] E.M. Clarke, D. E. Long, and K.L. McMillan. Compositional model checking . Proceedings of the 10th Annual ACM Symposium on Principles of Programming Languages, January 1983.

[CM81] K.M. Chandy and J. Misra. "Proofs of Networks of Processes", IEEE Trans. Software Engrg. 7(4), 1981

[CM88] K.M. Chandy, J. Misra, "Parallel Program Design," Addison-Wesley, 1988.

[DM90] M. Dyer and A. Kouchakdjian: "Correctness Verification: Alternative to Software Testing", Information and Software Technology, V. 32, Jan/Feb 1990, p53-9.

[DB89] D. John and B. Randell: "Program Verification: public image and private reality", Communications of the ACM, v.32, April 1989, p420-2.

[FJ89] Fetzer, James H. : "Program Verification: the very idea"

[EIA87] Electronic Industries Association Interim Standard, "Cellular Radiotelecommunications Intersystem Operations: Intersystem Handoff," (document EIA/IS-41.2), Electronic Industries Association, November, 1987.

[EF82] Tzilla Elrad, Nissim Francez: "Decomposition of Distributed Programs into Communication-Closed Layers", Science of Computer Programming 2 :155-173, 1982

[Fo87] I.R. Forman: "The Lift Problem Revisited", TR STP-269-87, MCC, Austin, TX, Sep. 1987.

[Fo89] I.R. Forman: "Design by Decomposition of Multiparty Interactions in Raddle87", MCC Austin, Texas, 1989 ACM.

[Fr92] N. Francez, "Program Verification," Addison-Wesley, 1992.

[Ga91] Gregory R. Andrews: "Concurrent Programming Principles and Practice", Benjamin Cummings, 1991.

[GCR94] S. Gerhart, D. Craigen and T. Ralston: "Experience with Formal Methods in Critical Systems", IEEE Software, January 1994.

[Ho69] C.A.R. Hoare: "An Axiomatic Basis for Computer Programming", Communications of the ACM, vol. 12, no. 10, pp. 576–580, 583, 1969.

[Ho78] C.A.R. Hoare: "Communicating Sequential Processes" , CACM 21,8, pp. 666-678, August 1978.

[Ho85] C.A.R. Hoare, "Communicating Sequential Processes," Prentice-Hall, 1985.

[KM90] Richard A. Kemmerer: "Integrating Formal Methods into the Development Process", IEEE Software, September 1990.

[KNW80] Krzysztof R. Apt, Nissim Francez, Willem P. De Roever: "A Proof System for Communicating Sequential Processes", ACM Transactions on Programming Languages and Systems, Vol. 2, No. 3, pp. 359–385, July 1980. "ACM Transactions on Programming Languages and Systems"

[La80] L. Lamport, "The 'Hoare Logic' of Concurrent Programs," Acta Informatica, vol. 14, pp. 21–37, 1980.

[Mil80] R. Milner: " A Calculus for Communicating Processes", Lecture Notes in Compute r Science 92(Springer, Berlin, 1980).

[Mil89] R. Milner: Communication and Concurrency, Prentice-Hall, 1989.

[NF89] Nissim Francez: "Cooperating Proofs for Distributed Programs With Multiparty Interactions", Information Processing Letters, Vol. 32, No. 5, pp. 235-242, Sept 1989.

[Pn85] A. Pneuli, "In Transition From Global to Modular Temporal Reasoning about Programs", in Logics and Models of Concurrent Systems, Springer-Verlag Berlin 1985.

[RM87] S. Ramesh, H. Mehndiratta: "A methodology for developing distributed programs", IEEE-TSE vol. SE-13, 8: 967-976, August 1987.

[SS85] Sol M. Shatz: "Post-Failure Reconfiguration of CSP Programs", IEEE-TSE vol. SE-11, No 10, October 1985.

[V87] M.Y. Vardi, "Verification of Concurrent Programs, The Automata Theoretic Framework," *Logic in Computer Science,* 1987

Submitted By : CHAMPAK DAS

Student Number :

Mailing Address:

Miami FL 33174

Phone no: (305)