SYMBOLIC SEMANTICS FOR CSP

BY

LIYI LI

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2014

Urbana, Illinois

Adviser:

Professor Elsa Gunter

Abstract

Communicating Sequential Processes (CSP) is a well-known formal language for describing concurrent systems, for which a transition semantics has been given by Brookes, Hoare and Roscoe [1]. In this thesis, we present a generalized transition semantics of CSP, which we call HCSP, that merges the original transition system with ideas from Floyd-Hoare logic and symbolic computation. This generalized semantics is shown to be sound and complete with respect to the original trace semantics. Traces in our system are symbolic representations of trace families given in the original semantics. This more compact representation allows us to expand the original CSP systems to effectively and efficiently analyze some CSP programs that are difficult or impossible for other CSP systems to analyze. In particular, our system can handle certain classes of non-deterministic choices as a single transition, while the original semantics would treat each choice separately, possibly leading to large or unbounded case analyses. All the work described in this thesis, carried out in the theorem prover Isabelle [2], provides us with a framework for automated and interactive analyses of CSP processes. It also gives us the ability to extract Ocaml code for an HCSP-based simulator directly from Isabelle.

Acknowledgments

This material is based upon work supported in part by NASA Contract NNA10DE79C and NSF Grant 0917218. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NASA or NSF.

Table of Contents

Chapter 1

Introduction

Communicating Sequential Processes (CSP) is a process algebra for describing the behavior and interactions of concurrent systems. With its expressive features of external and internal choice and its parallel composition, it has been used practically in industry to specify and verify concurrent features in various systems, especially ones combining human operators and automations, such as the medical mediator system in Gunter *et al.* [3], the airline ticket reservation system in Wong and Gibbons' paper [4] and the interactive systems with human error tolerance in Wright *et al.* [5].

In the traditional semantics of CSP, processes are given semantics via the set of traces they may generate, i.e. the set of sequences of individual actions the processes may execute. For example, the CSP process $c?x : B \rightarrow P$ is generally modeled as receiving a single value from the set $\{x|B\}$ and proceeding as P with that value. The set of possible traces will depend on the size of that set. Previous CSP analysis tools have followed this semantics by enumerating all traces individually. In practice, if $\{x|B\}$ is an infinite set, the current CSP analysis tools actually create an endless number of similar processes and wait for the other parts of the program to stop these processes. This diminishes efficiency and decreases the scope of analyzable problems.

In this thesis, we present a simulator HSim to effectively generate the behaviors of CSP programs based on Holistic CSP (HCSP) semantics, a new semantics for CSP processes that uses a symbolic representation of actions to capture a group of properties simultaneously instead of considering only a single element with a single property. The approach we take in this work is to represent families of transitions in CSP by a single transition in HCSP. This allows us to view a set of actions as a whole in some contexts, but also divide it based on various properties in other contexts.

The differences between the original CSP semantics-based tools and the HCSP ones can be demonstrated by some very simple examples. Four such examples are shown in Figure 1.1. For each process, the problem is to show the traces of a given CSP process. The CSP simulator ProBE [6] could handle case A easily, but it fails to terminate on cases B and C. It begins to run on B, but eventually generates a stack overflow. When process C is directly input into the CSPM-based simulator ProBE, the whole program crashes. However,

$$A.\quad \Big(\prod_{x:x>0\wedge x<10} \mathsf{A}.x \to \mathsf{SKIP}\Big)[\![\{k.x|k = A \wedge x > 0 \wedge x < 100\}]\!] \; (\mathsf{A}?x : x > 0 \wedge x < 150 \to \mathsf{SKIP})$$

$$B.\quad \Big(\prod_{x:x>0\wedge x<10^5} \mathsf{A}.x \to \mathsf{SKIP}\Big)[\![\{k.x|k = A \wedge x > 0 \wedge x < 10^6\}]\!] \; (\mathsf{A}?x : x > 0 \wedge x < 15 * 10^5 \to \mathsf{SKIP})$$

$$C.\quad \Big(\prod_{x:x>0\wedge x<10} \mathsf{A}.x \to \mathsf{SKIP}\Big)[\![\{k.x|k = A \wedge x > 0 \wedge x < 100\}]\!] \; \Big(\square_{y:y=1-1} \mathsf{A}?x : x > y \to \mathsf{SKIP}\Big)$$

$$D.\quad \Big(\prod_{x:x>0\wedge x<10} \mathsf{A}.x \to \mathsf{SKIP}\Big)[\![\{k.x|k = A \wedge x > 0 \wedge x < 100\}]\!] \; (\mathsf{A}?x : x > 0 \wedge x < 100 \to \mathsf{SKIP}) \square (\mathsf{A}?x : x > 99 \to \mathsf{SKIP})$$

Figure 1.1: Example

the HCSP simulator HSim we have derived from the semantics given in this thesis easily shows all traces of processes A - D in the same amount of time. We show more details of the experiments in Section 6, but from these examples we can clearly see that the running time of ProBE depends on the size of the sets bounding choice and parallel composition in each process.

These facts reflect that the original CSP and Machine-Readable CSP (CSPM) semantics views replicated operators (Replicated Internal Choice, Replicated External Choice, etc) as macros of their binary versions over sets. This means that the original CSP and CSPM semantics cannot express a replicated operator if the set of the replicated operator is infinite, such as the second Replicated Internal Choice operator in process C. Even if the set is finite, the cost is very high for CSP semantics-based tools to run a small replicated process in a large macro, such as in process B. On the other hand, HCSP semantics-based tools can overcome this problem and run CSP processes regardless of the size of the sets bounding the replicated operators. By using HCSP semantics, the three processes in the example will have the same number of next possible moves. This property allows the HCSP semantics-based tools to run faster than the traditional CSP ones in some cases. In addition, HCSP semantics-based tools can expand the set of possible CSP processes to analyze, of which processes B and C above are examples. We will see that this fact is useful in some real applications such as the medical mediator system in Gunter *et al.* [3].

This thesis's contribution is a new formalization of a symbolic semantics based on CSP. The HCSP semantics has been proved in Isabelle to be consistent with the original semantics. Consequently, we obtain an HCSP simulator HSim that is directly extracted from the HCSP semantics in Isabelle. We show families

of examples that our HCSP-semantics tools can analyze quite efficiently that previous tools could not analyze. Our symbolic semantics has additional benefits over other symbolic semantics. Firstly, the data set in HCSP choice operators can be calculated at runtime, allowing us to analyze some processes similar to process C in Figure 1.1. The HSIM is able to decide the value of variable y at runtime and determine the set $\{x|x > y\}$ by using the value of y. Secondly, HCSP is designed to model the broadcast message passing mechanism, allowing us to define point-to-group communication. This mechanism is particularly useful in describing interactive communication between humans and computers. One such example is the medical mediator system in Gunter *et al.* [3].

Chapter 2

Syntax

The syntax of HCSP and its informal meaning is given in Figure 2.1. For the remainder of this thesis, the following name conventions will be used. We will use P and Q for processes. Lower case p refers to an HCSP process name. The letter c represents an HCSP channel, while the letter a represents an HCSP action. The letter B is a proposition describing the property of a set. In HCSP, we include both *variables* and *parameters*, which are distinct types. We use k for variables ranging over HCSP channels and x for variables over actions. We use U and V to refer to parameters ranging over channels and actions. Variables and parameters serve similar functions, but differ as follows: variables may occur free or bound in HCSP processes and may be replaced by actions, channels or parameters, while parameters occur essentially as local constants not subject to binding or substitution. We will return to the reasons for having both *variables* or *parameters* in Section 3. In the rest of the thesis, we will use *freeParams* to refer to a function returning all free parameters in an expression of arbitrary type. We will use l to represent a transition label. Finally, the Greek letter ρ refers to a function that assigns values to variables or parameters.

One remark must be made here concerning the scope of variables. In the processes $c?x : B \to P$, $\prod_{x:B} P$ and $\bigsqcup_{x:B} P$, the scope of variable x is both the proposition B and the process P, while the scope of variables k and x is only the proposition B in the processes $P[\![\{k.x|B\}]\!]Q$ and $P \setminus \{k.x|B\}$.

To facilitate the application of HCSP to specific examples, it is parameterized into four user-defined types: a type of expression for actions and channels (acts), a type of proposition, a type of process name and a type of value to be assigned to the acts. The language of boolean expressions must minimally support conjunction, negation, equality, and a special operator Wf ranging over the boolean expressions, while the language of the acts must minimally support a function injecting the disjoint sum of the variables and parameters into the acts. The Wf operator checks whether a given proposition is well-formed in a given environment, in the sense that it can be evaluated to a boolean value in that environment. The operator Wf is necessary because in the CSP semantics, there is a certain ambiguity in dealing with ill-formed expressions. We address the ambiguity in CSP in Section 3.

The syntax of HCSP differs from that of CSPM by Bryan Scattergood [7] in

$P =$		
	Ω	Successful termination
$\mid$	SKIP	Awaiting successful termination
$\mid$	STOP	Unexpected termination
$\mid$	$c.a \rightarrow P$	Prefix by action a on channel c
$\mid$	$\$p$	Process name p as prcoess
$\mid$	if b then P else Q	If statement
$\mid$	$P \Box Q$	Binary external choice
$\mid$	$P; Q$	Sequential execution
$\mid$	$P \sqcap Q$	Binary internal choice
$\mid$	$\prod_{x:B} P$	Replicated internal choice
$\mid$	$\Box_{x:B} P$	Replicated external choice
$\mid$	$c?x : B \rightarrow P$	External set prefix
$\mid$	$P [\![\{k.x \mid B\}]\!] Q$	Parallel composition
$\mid$	$P \setminus \{k.x \mid B\}$	Hiding over a set of actions
$\mid$	let $p = P$ in Q	Local process name binding

Figure 2.1: HCSP Syntax

three ways. Firstly, the actions of CSP are explicitly divided into channels and actions (written $c.a$) in HCSP syntax. Secondly, for sets used in constructs such as the parallel composition of two processes or a replicated internal choice, we use a set comprehension notation. This decomposition of sets into variables and predicates facilitates the statement of the transition rules of HCSP semantics. Finally, HCSP currently lacks a CSP Renaming operator.

The Replicated Interleaving operator, $|||_{m} P$, in the System process uses the CSP macro: $|||_{x:m} P = P[m_1/x] [\![\{\}]\!] P[m_2/x] [\![\{\}]\!] ...$, where $m_1, m_2, ... \in \{x|m\}$. This means that all of its subprocesses work individually and do not communicate at all. We keep the same macro in HCSP semantics because it allows infinite parallel processes to occur in a program if we allow infinite sets in the Replicated Interleaving operator, which is a fundamental change in the meaning of the CSP language. In the original CSP definition, the External Set Prefix operator, $c?x : B \rightarrow P$, is defined as a macro for $\Box_{x:B} c.x \rightarrow P$. In HCSP, we provide the operator with a semantic interpretation because the operator is useful in representing "receiving" in CSP and it generates fewer vacuous rule applications than the specialized instance of the Replicated External Choice operator. Also, the $c!a \rightarrow P$ operator is the same as $c.a \rightarrow P$, as stated in the CSP book [8]. It should be noted to know that in the CSP world, there is no notion of sending and receiving messages; instead, internal choice and external choice represent sending and receiving. For example, in the Nurse process in Figure 2.2, the operator $\mathsf{HCI}_m^n!\mathsf{RFIDChan}_p^{n,m} \rightarrow P$ means that we internally commit an action $\mathsf{RFIDChan}_p^{n,m}$ on channel HCI_m^n, where both action and channel are previously internally selected. On the other hand, in the Med process in Figure 2.2, the operator $\mathsf{HCI}_m^n?x_1 \rightarrow \mathsf{P}$ means that we externally wait to synchronize on every

$$
\begin{aligned}
\mathsf{System} = \quad & ((\|_m (\sqcap_n(\mathsf{Nurse}(n,m)\ [\![\{\{\mathsf{HCI}^n_m.x|\mathsf{True}\}\}]\!] \\
& \mathsf{Med}(n,m))\ [\![\{\mathsf{EHRCh}_m.x|\mathsf{True}\}\}]\!] \\
& \mathsf{EHRInterface})) \setminus \{y|\forall n\ m\ p\ d\ x. \\
& y \neq \mathsf{RFIDChan}^n_p.x \wedge y \neq \mathsf{EHRBECh}^m.x \wedge \\
& y \neq \mathsf{BTAddr}^{n,m}_d.x\ \ \wedge y \neq \mathsf{BTScan}_m.x\}); \\
& \mathsf{System} \\
\mathsf{BTDevs} = \quad & \Box_m(\sqcap_{X \subseteq \mathsf{BTDevices}} \mathsf{BTScan}_m!X \rightarrow \mathsf{BTDevs}) \\
\mathsf{Nurse}(n,m) = & \\
& \mathsf{HCI}^n_m!\mathsf{GetID} \rightarrow \sqcap_p(\mathsf{HCI}^n_m!(\mathsf{RFIDChan}^{n,m}_p) \rightarrow \\
& \sqcap_d(\mathsf{HCI}^n_m!(\mathsf{RFIDChan}^{n,m}_d) \rightarrow \mathsf{HCI}^n_m?x \rightarrow \\
& \ \mathsf{if}\ x = (\mathsf{Name}(p), \mathsf{Name}(d)) \\
& \ \mathsf{then}\ (\mathsf{HCI}^n_m!\mathsf{Yes} \rightarrow \mathsf{TakeCkReading}(n,m,p,d)) \\
& \ \mathsf{else}\ \ \mathsf{if}\ \ x = \mathsf{Error}\ \ \mathsf{then}\ (\mathsf{HCI}^n_m!\mathsf{OK} \rightarrow \mathsf{Skip}) \\
& \qquad \mathsf{else}\ (\mathsf{HCI}^n_m!\mathsf{No} \rightarrow \mathsf{Skip}))) \\
\mathsf{Med}(n,m) = \mathsf{HCI}^n_m?\mathsf{GetID} \rightarrow \mathsf{HCI}^n_m?x_1 \rightarrow x_1?y_1 \rightarrow & \\
\mathsf{HCI}^n_m?x_2 \rightarrow x_2?y_2 \rightarrow \mathsf{EHRCh}^m!(y_1,y_2) \rightarrow \mathsf{EHRCh}^m?n_1 & \\
\rightarrow\ \mathsf{if}\ \ n_1 = \mathsf{Error}\ \ \mathsf{then}\ \ \mathsf{HCI}^n_m!n_1 \rightarrow \mathsf{HCI}^n_m?\mathsf{OK} \rightarrow \mathsf{Skip} & \\
\mathsf{else}\ \ \mathsf{EHRCh}^m?n_2 \rightarrow \mathsf{HCI}^n_m!(n_1,n_2) \rightarrow \mathsf{HCI}^n_m?z \rightarrow & \\
\mathsf{if}\ z = \mathsf{Yes}\ \ \mathsf{then}\ \ \mathsf{MedRead}(n,m)\ \ \mathsf{else}\ \mathsf{Skip} &
\end{aligned}
$$

Figure 2.2: Part of Medical Mediator Project Code [3]

possible value through the HCI^n_m channel. The operator $\mathsf{HCI}^n_m?x_1 \rightarrow \mathsf{P}$ uses an abbreviation of x_1 as $x_1 : \mathsf{True}$, while we omit the proposition True here.

In the following development, we will use the medical mediator project [3] example to highlight the functionality of CSP. This project provides a formal model of the use of a device for Automated Identification and Data Capture (AIDC) for vital signs measurements in hospitals. To demonstrate our HCSP semantics, we provide a small piece of the CSP code for the medical mediator project in Figure 2.2 and describe here the operators occurring in the code. The Nurse process models all of the behaviors that nurses can do with the mediator, while the Med process models the identification system. The BTDevs process models the behavior of announcing Bluetooth channels to the mediators scanning for them. The MedRead, TakeCKReading and EHRInterface process are not listed here. They are CSP processes which model certain behavior in the medical mediator model such as the backend system for delivering information displayed on the mediator and collecting data transmitted from the mediator. Finally, the System process combines the Nurse process, the Med process, and the EHRInterface process together using the Parallel operator, and limits the set of actions that these three processes can communicate to the set in the middle of the Parallel operator.

In the System process, we select a mediator from the Replicated Interleaving operator and a nurse from the Replicated Internal Choice, then select the corresponding channel from the HCI channels. This channel will be used in the Nurse and Med processes as the HCI^n_m channel. Then, these two processes, Nurse and Med, can communicate via the Parallel operator in the System process because

the set of the Parallel operator includes all communication along all the HCI_m^n channel. The Parallel operator $\mathsf{Nurse}(n,m)[\![\{\mathsf{HCI}_m^n.x|True\}]\!]\mathsf{Med}(n,m)$ allows interleaving actions to happen if the actions from $\mathsf{Nurse}(n,m)$ and $\mathsf{Med}(n,m)$ do not belong to the middle set, while it allows communication between processes $\mathsf{Nurse}(n,m)$ and $\mathsf{Med}(n,m)$ through the same action if the action belongs to the middle set. Finally, the Hiding operator is used to make a transition become local, i.e., invisible to the outside world. It provides security for the process involved. For example, the System process hides a transition if it belongs to either of the HCI or EHRCh channels. This means that the System process does not want outside processes to see and affect the actions along the HCI and EHRCh channels.

Chapter 3

Semantics

In order to describe the HCSP semantics, there are some functions that need to be supplied to evaluate the user-defined types. A process environment S is a function that interprets the process name p, where the process name p in the Proc_name operator (denoted by \$ in the concrete syntax in Figure 2.1) is user-defined, and can be arbitrarily complicated. A family of substitution functions $T[a/x]$ is needed for the replacement of a variable x by an act a in each term T, where T is one of an act, a boolean expression or a process name. Using these, we define the substitution function for processes. Moreover, the function $T[U_0, \ldots, U_n/x_0, \ldots, x_n]$ substitutes the parameters $U_0, \ldots, U_n$ for the variables $x_0, \ldots, x_n$ in T, respectively. The function $var(T)$ collects all of the free variables in T, while the function $funName(p)$ gets the function name of the process name p. For example, if a user defines a process name as $\$p = x(a_1, \ldots, a_n)$, the function name of the process name is x. The symbol $S(p)$ represents a user-defined function that provides the corresponding process for a given process name p and process environment S. We want this to be a user-defined function because we want the user to be free to deal with the structure of a given process name p, since we allow the user to have arbitrary structures for their process names. There also needs to be a family of user-defined evaluation functions for acts and boolean expressions and a "models" function, $\models$, for checking whether a boolean expression is true under a given assignment function.

The labels of the HCSP semantics are as follows:

$$l = \surd \mid \tau \mid (U.V)$$

The label $\surd$ represents process completion; the label τ represents a process performing an invisible action; and the label $(U.V)$ represents a pair of parameters, one for a channel and one for a real action. In any execution of a process in accordance with this semantics not labeled $\surd$ or τ, the sequence of transitions is labeled with mutually distinct pairs of parameters $(U.V)$.

The semantics for HCSP is given in Figures 3.1 to 3.4. It merges the original CSP semantics given by Roscoe *et al.* [9] with ideas from Floyd-Hoare logic [10] and symbolic computation. We present a labeled transition system for HCSP over quadruples of the form (α, γ, S, P) or (β, ϕ, T, Q), called configurations in

$$(\alpha,\gamma,S,\Omega[\![\{k.x|B\}]\!]\Omega) \xrightarrow{\surd} (\alpha,\gamma,S,\Omega) \quad \textsf{Par\_omega}$$

$$(\alpha,\gamma,S,\textsf{SKIP}) \xrightarrow{\surd} (\alpha,\gamma,S,\Omega) \quad \textsf{Skip}$$

$$(\alpha,\gamma,S,\$p) \xrightarrow{\tau} (\alpha,\gamma,S,S(p)) \quad \textsf{Proc\_name}$$

$$(\alpha,\gamma,S,P \sqcap Q) \xrightarrow{\tau} (\alpha,\gamma,S,P) \quad \textsf{Int\_choice1}$$

$$(\alpha,\gamma,S,P \sqcap Q) \xrightarrow{\tau} (\alpha,\gamma,S,Q) \quad \textsf{Int\_choice2}$$

$$\frac{(\alpha,\gamma,S,P) \xrightarrow{\tau} (\alpha',\gamma',S',P')}{(\alpha,\gamma,P\Box Q) \xrightarrow{\tau} (\alpha',\gamma',S',P'\Box Q)} \textsf{Ext\_choice\_tau1}$$

$$\frac{(\alpha,\gamma,S,Q) \xrightarrow{\tau} (\alpha',\gamma',S',Q')}{(\alpha,\gamma,S,P\Box Q) \xrightarrow{\tau} (\alpha',\gamma',S',P\Box Q')} \textsf{Ext\_choice\_tau2}$$

$$\frac{l \neq \tau \quad (\alpha,\gamma,S,P) \xrightarrow{l} (\alpha',\gamma',S',P')}{(\alpha,\gamma,S,P\Box Q) \xrightarrow{l} (\alpha',\gamma',S',P')} \textsf{Ext\_choice1}$$

$$\frac{l \neq \tau \quad (\alpha,\gamma,S,Q) \xrightarrow{l} (\alpha',\gamma',S',Q')}{(\alpha,\gamma,S,P\Box Q) \xrightarrow{l} (\alpha',\gamma',S',Q')} \textsf{Ext\_choice2}$$

Figure 3.1: HCSP semantics (category 1st (first part))

this thesis, where P and Q are HCSP processes; γ and ϕ are environment condition propositions in HCSP that are intended to state the current requirements for parameters "in scope", including those occurring free in P, and α and β are sets of parameters large enough to contain all parameters occurring free in P or γ. The tuples (l,α,γ,S,P) and (l,β,ϕ,T,Q) are called moves in this thesis. We carry α (or β) with us to allow for the choice of fresh parameter names guaranteed not to clash with a potentially bigger scope than the one presented locally by P (or Q) and γ (or ϕ). S and T are the interpretation functions for interpreting a process name p in a given HCSP process. The environment conditions γ (or ϕ) play the role of providing the pre- and post-conditions for each transition. The values potentially represented by labels of the form $(U.V)$ are progressively restricted by the conditions in each of the subsequent quadruples from each transition in the execution. In this way, a single execution in the transition semantics of HCSP potentially represents a parameterized family of executions from the original CSP semantics.

Variables and parameters are separated to solve variable-capturing problems in HCSP. We will use an example to highlight the reason for separating identifiers into variables and parameters in HCSP. Suppose we have the following processes: $\prod_{y:y=1} P$ and $P = C.y \rightarrow \textsf{SKIP}[\![\{k.x|k = C \wedge x = y\}]\!]C.2 \rightarrow \textsf{SKIP}$. We can easily see that process P leads to a deadlock. Recall that we use an identifier set α to keep track of existing free parameters and to enable the selection of new identifiers in Isabelle. When we use the set α to evaluate the process $\prod_{y:y=1} P$, if we do not separate identifiers into variables, a variable x might be generated by

$$\frac{l \neq \surd \quad (\alpha,\gamma,S,P) \xrightarrow{l} (\alpha',\gamma',S',P')}{(\alpha,\gamma,S,P;Q) \xrightarrow{l} (\alpha',\gamma',S',P';Q)}\ \mathsf{Seq\_step}$$

$$\frac{(\alpha,\gamma,S,P) \xrightarrow{\surd} (\alpha',\gamma',S',P')}{(\alpha,\gamma,S,P;Q) \xrightarrow{\tau} (\alpha',\gamma',S',Q)}\ \mathsf{Seq\_check}$$

$$\frac{(\alpha,\gamma,S,P) \xrightarrow{\tau} (\alpha',\gamma',S',P')}{(\alpha,\gamma,S,P \setminus \{k.x|B\}) \xrightarrow{\tau} (\alpha',\gamma',S',P' \setminus \{k.x|B\})}\ \mathsf{Hid\_tau}$$

$$\frac{(\alpha,\gamma,S,P) \xrightarrow{\surd} (\alpha',\gamma',S',P')}{(\alpha,\gamma,S,P \setminus \{k.x|B\}) \xrightarrow{\surd} (\alpha',\gamma',S',\Omega)}\ \mathsf{Hid\_omega}$$

$$\frac{(\alpha,\gamma,S,P) \xrightarrow{\surd} (\alpha',\gamma',S',P')}{(\alpha,\gamma,S,P[\![\{k.x|B\}]\!]Q) \xrightarrow{\tau} (\alpha',\gamma',S',\Omega[\![\{k.x|B\}]\!]Q)}\ \mathsf{Par\_check1}$$

$$\frac{(\alpha,\gamma,S,Q) \xrightarrow{\surd} (\alpha',\gamma',S',Q')}{(\alpha,\gamma,S,P[\![\{k.x|B\}]\!]Q) \xrightarrow{\tau} (\alpha',\gamma',S',P[\![\{k.x|B\}]\!]\Omega)}\ \mathsf{Par\_check2}$$

$$\frac{(\alpha,\gamma,S,P) \xrightarrow{\tau} (\alpha',\gamma',S',P')}{(\alpha,\gamma,S,P[\![\{k.x|B\}]\!]Q) \xrightarrow{\tau} (\alpha',\gamma',S',P'[\![\{k.x|B\}]\!]Q)}\ \mathsf{Par\_tau1}$$

$$\frac{(\alpha,\gamma,S,Q) \xrightarrow{\tau} (\alpha',\gamma',S',Q')}{(\alpha,\gamma,S,P[\![\{k.x|B\}]\!]Q) \xrightarrow{\tau} (\alpha',\gamma',S',P[\![\{k.x|B\}]\!]Q')}\ \mathsf{Par\_tau2}$$

$$\frac{U_0 \notin \alpha \quad \forall i \in [1,n]\,.\, U_i \notin \alpha \cup \{U_0,\ldots,U_{i-1}\}}{\begin{array}{l}(\alpha,\gamma,S,\mathsf{let}\ p = P\ \mathsf{in}\ Q) \xrightarrow{\tau} \\ \quad (\alpha \cup \{U_0,\ldots,U_n\},\gamma, \\ \quad\quad S + [p[U_0,\ldots,U_n/vars(p)] \\ \quad\quad\quad \mapsto P[U_0,\ldots,U_n/vars(p)]], Q[U_0/funName(p)])\end{array}}\ \mathsf{Let\_rec}$$

Figure 3.2: HCSP semantics (category 1st (second part))

the evaluation, and when we substitute x for y in process P, we get a new process $C.x \rightarrow \mathsf{SKIP}[\![\{k.x|k = C \wedge x = x\}]\!]C.2 \rightarrow \mathsf{SKIP}$. The deadlock in the new process disappears. We can see that distinguishing between identifiers with variables and parameters is one trivial solution. Otherwise, we need to keep track of all identifiers in α. A direct consequence is that we need to also keep track of all variables in the range of S. then we have to know all identifiers in the process name p and its corresponding process $S(p)$ (assume that S is the process environment) before the process $\$p$ starts to be evaluated. This lead to the fact that either we cannot have bound variables in the image of S or we try very hard to get all identifiers from the image of S for all cases, which restricts the use of the process $\$p$, which is hard to implement. The reason to avoid using the existing nominal logic tool in Isabelle to solve this problem is that we want to extract an HCSP simulator directly from our Isabelle code, but nominal logic does not allow us to do that [11].

$$\frac{\exists \rho \,.\, \rho \models (b \wedge \gamma)}{(\alpha, \gamma, S, \mathsf{if}\ b\ \mathsf{then}\ P\ \mathsf{else}\ Q) \xrightarrow{\tau} (\alpha, b \wedge \gamma, S, P)}\ \mathsf{Ifthenelse1}$$

$$\frac{\exists \rho \,.\, \rho \models (\neg b \wedge \mathsf{Wf(b)} \wedge \gamma)}{(\alpha, \gamma, S, \mathsf{if}\ b\ \mathsf{then}\ P\ \mathsf{else}\ Q) \xrightarrow{\tau} (\alpha, \neg b \wedge \mathsf{Wf(b)} \wedge \gamma, S, Q)}\ \mathsf{Ifthenelse2}$$

$$\frac{(\alpha, \gamma, S, P) \xrightarrow{(U.V)} (\alpha', \gamma', S', P') \quad \exists \rho \,.\, \rho \models (\neg B[U/k][V/x] \wedge \gamma')}{(\alpha, \gamma, S, P \setminus \{k.x|B\}) \xrightarrow{(U.V)} (\alpha', \neg B[U/k][V/x] \wedge \gamma', S', P' \setminus \{k.x|B\})}\ \mathsf{Hid\_neg}$$

$$\frac{(\alpha, \gamma, S, P) \xrightarrow{(U.V)} (\alpha', \gamma', S', P') \quad \exists \rho \,.\, \rho \models (B[U/k][V/x] \wedge \gamma')}{(\alpha, \gamma, S, P \setminus \{k.x|B\}) \xrightarrow{\tau} (\alpha', B[U/k][V/x] \wedge \gamma', S', P' \setminus \{k.x|B\})}\ \mathsf{Hid\_pos}$$

$$\frac{(\alpha, \gamma, S, P) \xrightarrow{(U.V)} (\alpha', \gamma', S', P') \quad \exists \rho \,.\, \rho \models (\neg B[U/k][V/x] \wedge \gamma')}{(\alpha, \gamma, S, P[\![\{k.x|B\}]\!]Q) \xrightarrow{(U.V)} (\alpha', \neg B[U/k][V/x] \wedge \gamma', S', P'[\![\{k.x|B\}]\!]Q)}\ \mathsf{Par\_out1}$$

$$\frac{(\alpha, \gamma, S, Q) \xrightarrow{(U.V)} (\alpha', \gamma', S', Q') \quad \exists \rho \,.\, \rho \models (\neg B[U/k][V/x] \wedge \gamma')}{(\alpha, \gamma, S, P[\![\{k.x|B\}]\!]Q) \xrightarrow{(U.V)} (\alpha', \neg B[U/k][V/x] \wedge \gamma', S', P[\![\{k.x|B\}]\!]Q')}\ \mathsf{Par\_out2}$$

$$\frac{\begin{array}{c}(\alpha, \gamma, S, P) \xrightarrow{(U.V)} (\alpha', \gamma', S', P') \\ (\alpha', \gamma', S', Q) \xrightarrow{(U'.V')} (\alpha'', \gamma'', S'', Q') \\ \exists \rho \,.\, \rho \models (U = U' \wedge V = V' \wedge B[U/k][V/x] \wedge \gamma'')\end{array}}{\begin{array}{l}(\alpha, \gamma, S, P[\![\{k.x|B\}]\!]Q) \xrightarrow{(U.V)} \\ \quad (\alpha'', U = U' \wedge V = V' \wedge B[U/k][V/x] \wedge \gamma'', S'', P'[\![\{k.x|B\}]\!]Q')\end{array}}\ \mathsf{Par\_in}$$

Figure 3.3: HCSP semantics (categories 2nd)

The main contribution of this thesis is to provide a basic framework for translating transition semantics into symbolic semantics by merging state information from Floyd-Hoare logic [10] with the existing semantics. The basis of our semantics is to associate a state environment, whose structure is implemented as a predicate, with the evaluation of a CSP process. When we are going to evaluate a statement in a programming language, instead of evaluating the statement directly, we view each transition as a constraint that can be merged into the current environment. After updating the environment, we can reason about the evaluation by asking about the satisfiability of the environment predicate. For example, if we translate the traditional semantics of CSP into our framework, then the new semantics tells users the range of values for a particular next possible move in CSP when the environment predicate is satisfiable, or that a next move is impossible because the environment predicate associated with it is unsatisfiable. A trace in this semantics is an execution pattern corresponding to a possibly large or even infinite set of individual execution traces in the original CSP semantics. For instance, the HCSP semantics allows a potentially infinite collection of data in the set of replicated choice operators to execute as one single transition, which is a nice feature that traditional CSP-based analysis tools

$$\frac{U \notin \alpha \quad V \notin \{U\} \cup \alpha \quad \exists \rho \,.\, \rho \models (U = c \wedge V = a \wedge \gamma)}{(\alpha, \gamma, S, c.a \to P) \xrightarrow{(U.V)} (\{U, V\} \cup \alpha, U = c \wedge V = a \wedge \gamma, S, P)}\ \textsf{Act\_prefix}$$

$$\frac{U \notin \alpha \quad V \notin \{U\} \cup \alpha \quad \exists \rho \,.\, \rho \models (U = c \wedge B[V/x] \wedge \gamma)}{\begin{array}{c}(\alpha, \gamma, S, c?x : B \to P) \xrightarrow{(U.V)} \\ (\{U, V\} \cup \alpha, U = c \wedge B[V/x] \wedge \gamma, S, P[V/x])\end{array}}\ \textsf{Ext\_prefix}$$

$$\frac{U \notin \alpha \quad \exists \rho \,.\, \rho \models (B[U/x] \wedge \gamma)}{(\alpha, \gamma, S, \prod_{x:B} P) \xrightarrow{\tau} (\{u\} \cup \alpha, B[U/x] \wedge \gamma, S, P[U/x])}\ \textsf{Rep\_int\_choice}$$

$$\frac{\begin{array}{c}U \notin \alpha \quad \exists \rho \,.\, \rho \models \gamma' \\ (\{u\} \cup \alpha, B[U/x] \wedge \gamma, S, P[U/x]) \xrightarrow{\tau} (\alpha', \gamma', S', P')\end{array}}{(\alpha, \gamma, S, \square_{x:B} P) \xrightarrow{\tau} (\alpha', \gamma', S', (\square_{x:B \wedge x \neq U} P) \square P')}\ \textsf{Rep\_ext\_tau}$$

$$\frac{\begin{array}{c}U \notin \alpha \quad l \neq \tau \quad \exists \rho \,.\, \rho \models \gamma' \\ (\{u\} \cup \alpha, B[U/x] \wedge \gamma, S, P[U/x]) \xrightarrow{l} (\alpha', \gamma', S', P')\end{array}}{(\alpha, \gamma, S, \square_{x:B} P) \xrightarrow{l} (\alpha', \gamma', S', P')}\ \textsf{Rep\_ext\_nor}$$

Figure 3.4: HCSP semantics (category 3rd)

cannot provide.

When translating the original CSP semantics into HCSP semantics, the main task is to merge information about actions and channels into the environment condition γ, and use this condition when we are trying to evaluate a CSP process. To do so, we will divide the transition rules in CSP into three categories: rules for basic operators having no side conditions other than $\surd$-τ label constraints, rules with side conditions needing to be translated into the HCSP framework, and rules for operators that were treated as macros in CSP.

The rules without side conditions are those for SKIP, STOP, Internal Choice, External Choice and Sequential Composition operators. For these, we just need to add the set of free parameters and environment conditions to the processes involved, propagating constraints from hypotheses to conclusions in the original CSP rules. For example, the External Choice operator is translated from the original CSP semantics as follows:

$$\frac{l \neq \tau \quad P \xrightarrow{l} P'}{P \square Q \xrightarrow{l} P'} \quad \text{becomes} \quad \frac{l \neq \tau \quad (\alpha, \gamma, S, P) \xrightarrow{l} (\alpha', \gamma', S', P')}{(\alpha, \gamma, S, P \square Q) \xrightarrow{l} (\alpha', \gamma', S', P')}\ \textsf{Ext\_choice1}$$

The second category contains the operators with set or boolean guard infor-

mation, namely, Parallel, Hiding and If-then-else. The general idea for translating the rules of CSP into HCSP is to treat the set information and boolean guards as new restrictions on the environment and merge them into the conditions as post-conditions of the transitions after we do the same steps to translate the rules as were done with the first category. This requires us to translate all set information into set comprehension notation. For example, for the Parallel operator, we make the following translation:

$$\frac{l \in X \quad P \xrightarrow{l} P' \quad Q \xrightarrow{l} Q'}{P|[X]|Q \xrightarrow{l} P'|[X]|Q'}$$

becomes

$$\frac{\begin{array}{c}(\alpha, \gamma, S, P) \xrightarrow{(U.V)} (\alpha', \gamma', S', P') \\ (\alpha', \gamma', S', Q) \xrightarrow{(U'.V')} (\alpha'', \gamma'', S'', Q') \\ \exists \rho \,.\, \rho \models (U = U' \wedge V = V' \wedge B[U/k][V/x] \wedge \gamma'')\end{array}}{\begin{array}{l}(\alpha, \gamma, S, P[\![\{k.x|B\}]\!]Q) \xrightarrow{(U.V)} \\ \quad (\alpha'', U = U' \wedge V = V' \wedge B[U/k][V/x] \wedge \gamma'', S'', P'[\![\{k.x|B\}]\!]Q')\end{array}}$$

This rule means that a Parallel process can communicate between the left-hand-side and right-hand-side subprocesses, if they can produce the same action and the action is valid in the middle set of the Parallel process (the Par_in rule). Other rules allow each subprocess to progress independently. For rules in this category, we replace labels by parameter pairs $(U.V)$, giving a parameter pair for each transition hypothesis. This allows for separation of the constraints for each of the hypotheses. In the Par_in rule, since we need to make sure that the actions produced by the left-hand-side and right-hand-side subprocesses are equal, we also need to put into the final environment condition the condition that the two actions are equal. Also, we must add a requirement that the label (in its two parts, U and V) satisfies the set constraint of the Parallel operator. After we finish constructing the new condition, we need to verify it and see whether or not there is an assignment function ρ that models the new condition. The translation of the Parallel rules displays the main difference between the CSP and HCSP semantics. In CSP, the Parallel rules specify individual transitions that are allowed, while in HCSP, whole families of transitions are specified, hence the "Holistic" in HCSP.

In the CSP semantics, there is a certain ambiguity in dealing with ill-formed expressions. The boolean guard in the If-then-else operator is an example of this. Therefore, special treatment is needed when merging the boolean guard into the existing condition. We introduce a condition that includes the Wf operator applied to the boolean guard b of the If-then-else process to resolve the ambiguity. This means that the If-then-else process can transition if and only if the boolean guard is fully evaluated and is actually boolean. In a given environment, the

expression for the boolean guard might or might not be capable of evaluating to a boolean value, and it is important to distinguish the case of non-evaluation from evaluating to either true (enabling transition to the then process) or false (enabling transition to the else process). For example, in the HCSP process if $x < 1$ then P else Q, the else branch should be taken when $x \geq 1$, not simply when $x < 1$ does not hold, since the latter includes the case when x is a list or something other than a number. We do not want to transition to Q in the case where x is, for example, a string. However, the semantics of the parallel operator of HCSP instead uses the "satisfies" versus "fails to satisfy" meaning of boolean expressions. For instance, in the process $P[\![\{k.x|k = c \wedge x < 1\}]\!]Q$, if the process P commits a string action of $c."hi"$, the whole process can actually take a further step by a single non-communicating move in P by the original CSP transition semantics.

The third category includes all replicated operators, which in CSP are considered to be macros based on other rules. The ones we have included are the Replicated External Choice, Replicated Internal Choice and External Set Prefix operators. For example, the Replicated Internal Choice operator has the following macro definition in CSP: $\prod_{x:S} P = P[a_1/x] \sqcap P[a_2/x] \sqcap ...$, where $a_1, a_2, ... \in S$. We will translate the rules of these operators from CSP into HCSP by their semantic meanings. This means that we will create new rules that are semantically equivalent to the macros for these operators. For example, we create new rules for Replicated External Choice for the original CSP as follows:

$$\frac{a \in S \quad P[a/x] \xrightarrow{\tau} P'}{\Box x : S@P \xrightarrow{\tau} (\Box x : (S - \{a\})@P)\Box P'}$$

becomes

$$\frac{l \neq \tau \quad a \in S \quad P[a/x] \xrightarrow{l} P'}{\Box x : S@P \xrightarrow{l} P'}$$

Relying upon the associativity and commutativity of Replicated External Choice in the original CSP, these rules can be seen as a strict generalization of the rules for the corresponding binary operators. Having added these rules to CSP, when translating them into HCSP rules, we follow the same procedure as for the rules in the second category except that we add new hypotheses to indicate that the parameters, such as U and V in the Ext_prefix rule, are not in the existing parameter set α.

Chapter 4

Correctness of HCSP with Respect to CSP

In this section, two main theorems are proved to show that the HCSP semantics is sound and complete with respect to the original CSP semantics. Our reference semantics is the CSP transition semantics introduced by Roscoe, Brookes and Walker [9], with the updated syntax of CSP-Prover [12]. All of the work described here has been formally carried out in the interactive theorem prover Isabelle/HOL [2]. Our Isabelle code may be found at `http://www.cs.illinois.edu/~egunter/fms/HCSP/hcsp.tar.gz`. The proofs of the soundness and completeness of the HCSP semantics are parameterized by user-defined acts, which correspond to user-defined values in the original CSP semantics. These proofs are also parameterized by a user-defined set of process names, together with their associated processes as given by *procEnv*, and a notion of *freeParams* satisfying

$$\mathit{freeParams}(\mathit{procEnv}(p)) \subseteq \mathit{freeParams}(p))$$

for every process name p. We define the functions $sem(\rho, P)$ and $sem(\rho, l)$ as interpretation functions to interpret a given HCSP process or label as a CSP process or label with respect to the valuation of ρ.

In the Isabelle code for HCSP, values and acts are two different types, but in this thesis, by common abuse of notation, we will assume that values and acts have the same type. We use m and n to refer to values below. In the Isabelle code, we must also have a separate function for each type of construct we need to translate from HCSP to CSP. By and large, these functions are just the obvious translations. Here we will abuse notation and uniformly refer to them all as *sem*. We also use the following definition in our theorems:

Definition 1 (Respect)**.** We say that S *respects* α if for all process name p, we have $\mathit{freeParams}(S(p)) \in \alpha$.

Before stating the soundness and completeness theorems, a very important observation is that transitions properly track the parameters introduced:

Theorem 4.0.1 (Well-tracked Parameters)**.** *Assume we have a transition* $(\alpha, \gamma, S, P) \xrightarrow{l} (\alpha', \gamma', S', P')$ *in HSCP such that* $\mathit{freeParams}(P) \cup \mathit{freeParams}(\gamma) \subseteq \alpha$*, and* S *respects* α*. Then* $\mathit{freeParams}(P') \cup \mathit{freeParams}(\gamma') \subseteq \alpha'$ *and* S' *respects* α'*.*

Proof. The main idea is to proceed by induction on the HCSP semantics rules. For all HCSP rules, we can see that the resultant α' always contains both α and any new parameters. The replicated operators use that $\mathit{freeParams}(P) \subseteq \alpha$ implies $\mathit{freeParams}(P[U/x]) \subseteq \alpha \cup \{U\}$. To prove the statement S respects α implies S' respects α', we rely on the assumption that for any process name p, any process environment S as well as any process P, the update function $S + [p \mapsto P]$ will not create new parameters other than the existing parameters in p, P and the range of S. When users instantiate our HCSP theory, they need to implement their process names in the way that all their implementations should satisfy our assumption. □

The proofs of soundness and completeness require the following facts describing how parameters, assignments, substitution, and translation interact:

Lemma 4.0.2. *If* $\rho(U) = n$, *then* $sem(\rho, P[U/x]) = sem(\rho, P[n/x])$.

Lemma 4.0.3. *If* $\rho \models B[U/x][V/y]$ *and* $sem(\rho, U)$ *and* $sem(\rho, V)$ *are well defined, then* $(\rho(U).\rho(V))$ *is in the set* $sem(\rho, \{x.y|B\})$.

Lemma 4.0.4. *If* $\mathit{freeParams}(P) \subseteq \alpha$, $\rho'|_\alpha = \rho|_\alpha$, *then* $sem(\rho, P) = sem(\rho', P)$.

With these, we can show that every interpretation of every transition we can take in HCSP is valid in CSP.

Theorem 4.0.5 (Soundness)**.** *For all HCSP processes P, P', assignments ρ, environment conditions γ, γ' and process environments S, S' such that $\rho \models \gamma$ and $\rho \models \gamma'$, and parameter set α such that $\mathit{freeParams}(P) \cup \mathit{freeParams}(\gamma) \subseteq \alpha$ and S respects α, if $(\alpha, \gamma, S, P) \xrightarrow{l} (\alpha', \gamma', S', P')$, then $\mathit{freeParams}(P') \cup \mathit{freeParams}(\gamma') \subseteq \alpha'$, S' respects α' and $sem(\rho, P) \overset{sem(\rho,l)}{\longrightarrow} sem(\rho, P')$.*

Proof. By induction on the HCSP semantics rules. It is worth noting that each assignment function ρ is a total function, and thus gives a value to every parameter, regardless of whether it has been included in the parameter set α.

Each rule in HCSP has a corresponding rule in CSP. When we prove soundness, we must show, that for each rule in HCSP, if the hypotheses of the rule are valid in HCSP, then the corresponding hypotheses of the corresponding CSP rule are also valid. This follows from Theorem 4.0.1 and the fact that the environment condition γ only becomes logically stronger with each transition. Handling the rules for the replicated operators requires use of Lemma 4.0.2. The hypotheses in the CSP rules include side conditions that have become incorporated in the environment condition in the derived HCSP rules. It is therefore necessary to prove these side conditions from the assumption of the validity of the initial environment condition and the specifics of the transition in HCSP. For this, we need to make use of Lemma 4.0.3. □

As yet, HCPS does not support all of the processes in CSP. In particular, there is no support for the Renaming operator. As a result, our completeness theorem is restricted to that portion of CSP supported by HCSP.

Theorem 4.0.6 (Relative Completeness)**.** *Let P be an HCSP process, ρ be an assignment, γ be an environment condition, S be a process environment such that $\rho \models \gamma$, and α be a parameter set such that $freeParams(P) \cup freeParams(\gamma) \subseteq \alpha$, and S respects α, such that $sem(\rho, P) \xrightarrow{i} T$ in CSP semantics. Then there exist an HCSP process P', an assignment ρ', an environment condition γ', a parameter set α', a process environment S', and a label l such that $freeParams(P') \cup freeParams(\gamma') \subseteq \alpha'$, and S' respects α' ,and $i = sem(\rho', l)$, and $\rho'|_\alpha = \rho|_\alpha$, and $T = sem(\rho', P')$, $\rho' \models \gamma'$, and $(\alpha, \gamma, S, P) \xrightarrow{l} (\alpha', \gamma', S', P')$.*

Proof. By induction on the CSP semantics rules. For each rule in HCSP, the first thing we need to do is use Theorem 4.0.1 and Lemma 4.0.4 to prove the theorem hypotheses for the inductive instances. Then we will interpret l, P' α', γ' and ρ' according to the corresponding HCSP rules and show that the interpretation is correct. For each rule, since the only parameters generated from the evaluation of an HCSP process are not in set α, we can always make a new assignment function ρ' based on the current ρ and $\rho'|_\alpha = \rho|_\alpha$. To show that $i = sem(\rho', l)$, we need to do a case analysis on all possible labels an HCSP process can produce, since there is always an interpretation for a given free parameter U. Finally, in some rules, we need to show that the ρ' we are selecting can properly model the post-condition of the environment. To prove this, we must use the same technique as we use in reasoning about the post-condition in the Soundness theorem using Lemmas 4.0.3. □

Chapter 5

HCSP Simulator

('a, 'c)procName =	ProcName p (('a, 'c)act list, P list)	process name
('a, 'c)act =		
	BasicAct 'a	User defined data
|	Id x	Identifier
|	Num n	Integer number
|	act + act	Plus operators
|	act − act	Minus operators
|	act ∗ act	Times operators
|	Prop propt	Proposition act
|	[]	Empty list
|	act#act	List Cons
|	act@act	Append
|	Hd act	Head
|	Tl act	Tail
|	Nth act	Nth element
|	Last act	Last element
|	Chan 'c	Channel
|	(act, act)	Pair
|	Fst act	First element
|	Snd act	Second element

Figure 5.1: Practical Action and Process Name Syntax(first part)

To put the theory of HCSP into practice, we have implemented an HCSP simulator with a rich mutually recursive action and proposition datatype in Isabelle. We limit the propositions to quantifier-free first-order theory Presburger arithmetic in order to maintain decidability, because the original purpose of HCSP was to overcome the shortcomings of current CSP model checkers and provide a useful tool for examples such as the medical mediator project. The datatype includes integers, lists, tuples, sets, etc. In this datatype, a proposition can be an action in HCSP. We define the process name as a constructor with three arguments: a process variable, a list of actions and a list of processes. The process variable is the name of the process, while the lists of actions and processes are two lists of input arguments.

In this simulator, we need to divide the semantics into two functions:

and ('a, 'c)propt =		
	True	Logic true
|	False	Logic false
|	propt ∧ propt	Logic and
|	propt ∨ propt	Logic or
|	¬ propt	Logic not
|	Wf(propt)	Well-formed
|	act = act	Equal operator
|	act < act	Less than
|	act ∈ actSet	Set menbership
|	actSet ⊆ actSet	Subset equal
|	Men act act	List menbership
|	act ⊑ act	Sub-list equal
and ('a, 'c)actSet =		
	{x|propt}	A descriptive set
|	ActionSet (act list)	A enumerated set
|	ActListSet act	Set with act list
|	actSet ∩ actSet	Set intersection
|	actSet ∪ actSet	Set union

Figure 5.2: Practical Action and Process Name Syntax(second part)

oneStepEval and eval. The job of the oneStepEval function is to get all of the possible immediate moves for a process. It collects next possible moves by exploring all of the possible ways a given process can behave recursively with a trivial logic check on the types of the labels of the next possible moves. For example, if a process is of the form $P[\![\{x.y|B\}]\!]Q$ and either P or Q returns a label τ, we do not need to check the other branch to see that the only possible rules are Par_tau1 and Par_tau2.

The eval function returns a list of traces with the very last event as the first element of the list and all other possible traces stored in the unevaluated function data body by using the special datatype below:

'a stream = Stream of ('a * (unit → 'a stream)) | Bottom

This datatype takes advantage of the delayed-evaluation feature of ML-based languages. It allows users to examine a single trace at a time and delay exploring the other possibilities for a given process by a depth-first search strategy. The main task of eval is to use the decision procedure, i.e. Presburger arithmetic, to determine if a process is a possible move by exploring the post-condition of the action environment. Running the decision procedure can be very time-intensive. For instance, as shown by Viktor Kuncak [13], deciding the truth of a quantifier-free boolean formula with Presburger arithmetic is NP-complete. Therefore, it is worth moving the decision procedure into the eval function and only running it once per generated process. Finally, the eval function is interactive, allowing users to browse through the generated traces as necessary.

Chapter 6

Examples and Experiment

$$\mathsf{Clicker}(c,r) = \bigsqcap_{\substack{s:s>0\wedge\\ s\leq N}} K.r.c.s \to \mathsf{Clicker}(c,r)$$

$$\mathsf{Broadcast}(r) = \bigsqcap_{c:true} K.r.c?s : s>0 \wedge s \leq N \to \mathsf{Out}.r.s \to \mathsf{Broadcast}(r)$$

$$\mathsf{Room}(r) = \begin{matrix}\mathsf{Clicker}(c1_r,r)\\ [\![\{\}]\!]\\ \mathsf{Clicker}(c2_r,r)\end{matrix} \quad [\![\{k.x|\exists c\ s.\ \mathsf{hd}(k)=K\}]\!] \mathsf{Broadcast}(r)$$

$$\mathsf{Center} = \mathsf{Room}(1)[\![\{\}]\!]\mathsf{Room}(2)$$

$$\mathsf{ATM1} = \mathsf{Incard}?c : (M<c<N) \to \mathsf{PIN}.c \to \mathsf{Req}?n : (99<n) \to \bigsqcap_{x:x=n\wedge bx<2000} \mathsf{Dispense}.x \to \mathsf{Outcard}.c \to \mathsf{ATM1}$$

$$\mathsf{ATM2} = \mathsf{Incard}?c : (M<c<N) \to \mathsf{PIN}.c \to \mathsf{Req}?n : (99<n) \to \bigsqcap_{x:x=n\wedge bx<2000} \mathsf{Dispense}.x \to \begin{matrix}(\mathsf{Outcard}.c \to \mathsf{ATM2})\\ \sqcap(\mathsf{Refuse}.1 \to \mathsf{ATM2} \sqcap \mathsf{Outcard}.c \to \mathsf{ATM2})\end{matrix}$$

Figure 6.1: Examples

Besides the small examples in Fig. 6.1 and the medical mediator example from Gunter *et al.* [3], there are many other real implementations that can benefit from modeling in the HCSP system. Generally speaking, every real model with several users trying to access one or more copies of a very large database can benefit from the HCSP system. A song broadcasting system and an ATM are two systems which cannot be modeled in current CSP tools and can benefit from HSim.

Song broadcasting systems are used in entertainment businesses such as discos and karaokes to allow people to select songs from a large database. Such systems typically have a large collection of songs; a collection in excess of 200,000 would not be uncommon. A typical karaoke bar has more than twenty separate rooms for entertainment. Typically, each room has two remote clickers for selecting the next song to be played. After a user selects a song, the remote clicker will send the song selection to the broadcasting system, which will play it

in the room. Since only one song can be broadcast at a time, if two people send selections simultaneously, only one signal will be honored immediately, while the other will be delayed for later action. We model the karaoke center in CSP in Figure 6.1. For simplicity, we assume our karaoke center only has two rooms.

In Fig. 6.1, the capital letter N refers to an arbitrary number to represent the size of the database that contains all of the songs in the song broadcasting system. Typically, we know that the number N is a large number, but we do not know exactly how large it is. In order to verify properties in the system, such as safety and deadlock-freedom, it is better to leave the N unspecified. We will set it at 500 and 500,000 for test cases in the experiment.

Likewise, we implement two ATMs in Fig. 6.1. The two ATM processes are to describe the procedures of a machine that is receiving commands from humans and responding to them. In these ATMs, the numbers N and M specify the range of the debit and credit cards that can be read. Typically, a debit or credit card will have sixteen digits. We test the cases when the numbers N and M are one, four and sixteen digits long.

Programs	ProBE	HSim
Process A	$<$ 2 secs	$<$ 2 secs
Process B	N/A	$<$ 2 secs
Process C	N/A	$<$ 2 secs
Process C	N/A	$<$ 2 secs
ATM1 one digit	25 secs	$<$ 2 secs
ATM1 four digits	$>$ 12 hours	$<$ 2 secs
ATM1 16 digits	N/A	$<$ 2 secs
ATM2 one digit	$<$ 2 secs	$<$ 2 sec
ATM2 four digits	$>$ 12 hours	$<$ 2 sec
ATM2 16 digits	N/A	$<$ 2 secs
Karaoke 5	$<$ 2 secs	15 secs
Karaoke 500	$>$ 12 hours	15 secs
Karaoke 500,000	N/A	15 secs
Medical one nurse	50 mins	1.5 hours
Medical two nurses	N/A	1.7 hours

Figure 6.2: Experiment Results

We compared the efficiencies of ProBE and HSim on some programs. The experiment was run on an Intel core i7 machine with eight gigabytes of memory and a Ubuntu 13.04 system. The programs tested were processes A, B, C and D from Section 1 and the ATMs with N and M being one, four and sixteen digits. In addition, we tested the karaoke center examples with N equal to 500 and 500,000, and medical mediator examples in which there were two mediators, one nurse, three patients and three devices, and when there were two mediators, two nurses, three patients and three devices. The results can be found in Figure 6.2.

From the table, we can see that HSim finished all of the jobs, while ProBE failed to execute some programs. In most cases, HSim was more efficient at

showing the traces of the programs than ProBE. In addition, ProBE was very sensitive to the size of the input data, and it could not recognize similarities in different programs. It succeeded in showing some programs, but failed when we change them a little bit. For example, ProBE executed process A completely, but failed to even read processes C or D. The medical mediator was a more representative example. Because of the sensitivity of ProBE with respect to the input data, it could finish a job when there was only one nurse, but not when there were two. On the other hand, even though HSim needed a longer time to finish a job when the size of the set governing a replicated choice operator, it showed the traces successfully no matter how big the size of the governing set is.

Chapter 7

Related Work

Currently, there are several existing CSP simulators and model checkers based on the original CSP transition semantics. CSPM [7] gives a standard CSP syntax and semantics in machine readable form, introduced by Bryan Scattergood, which is based on the transition semantics introduced by Brookes and Roscoe [9]. It provides the standard for many CSP tools, including FDR2 by Formal Systems (Europe) Ltd., the industry standard for CSP model checkers [14]. ProBE [6], a simulator created by the same group, simulates a CSP process by listing all of the actions and states one by one as a tree structure [6]. Jun Sun *et al.* [15] merged partial order reduction with trace refinement model checking in the tool called PAT. CSP-Prover is a theorem proving tool built on top of Isabelle [12]. It provides a denotational semantics of CSP in higher order logic. CSPsim [16] is another simulator based on the CSPM standard. Its major innovation is the use of "lazy evaluation". The basic idea of CSPsim is to keep track of all of the current actions, compare them with the actions of the outside world and only select the possible executable actions for the very next step [16]. The phrase "lazy evaluation" refers to a pre-processing step in which CSPsim selects some processes that contain fewer actions and generates some conditions in advance. After that, CSPsim evaluates the whole program based on these conditions.

These tools use the traditional view of actions as single elements, and tend to generate a large number of states when comparing multiple possibilities for actions. Additionally they treat some operators, especially replicated operators, as macros. As a result, while it is possible for some tools to analyze complicated programs such as medical mediator, it is impossible for these tools to generate traces when the replicated set is infinite. The medical mediator project by Gunter *et al.* [3] provides an example of the advantages of HCSP over CSP-semantics based tools. The main goal of the medical mediator project is to prove that the set of traces of the process $System|[Vis]|Given$ is a subset of the traces of $Safety|[Vis]|Given$, where Vis is a set defined as $\{y.(\exists n\ d\ m\ x.\ y = RFIDChan_d^{n,m}.x) \vee (\exists m\ z.\ y = EHRBECh_m.z)\}$. This requires exploring all possible traces generated by the System process. Tools based on the original CSP semantics, such as FDR2, fail when dealing with large or unbounded sets. For example, the Med process as given does not put any restrictions on the sets

of values that may be received over various channels. The simulator and model checker we have built based on HCSP semantics has the benefit of being able to handle large or unbounded sets uniformly as single actions, thus avoiding state explosion problems.

While writting this thesis, FDR3 came out [17], providing a parallelized algorithm for model checking the trace refinement property over FDR2. When we used FDR3 to test our programs, the performance was better than that of FDR2. However, it still could not capture the behavior of infinite sets of replicated choice operators. For example, when we tested the processes C and D in Figure 1.1, FDR3 failed to terminate.

There are several existing symbolic semantics for process algebras, mainly serving process algebras with structures similar to Π-calculus. The early work of Hennessy and Lin [18] provided a framework of symbolic semantics for value-passing process algebras. Later, Sangiorgi applied this symbolic semantics idea to Π-calculus [19]. Bonchi and Montanari revisited the symbolic semantics of Π-calculus [20], providing a symbolic transition semantics for it by including the predicate environment condition as part of a label in a transition system. In their symbolic transition semantics, they only discovered the relation in the parallel operator in Π-calculus.

LOTOS is a kind of process algebra that contains features from both Π-calculus and CSP. Its parallel operator is similar to that of CSP. The parallel operator contains a middle set to restrict the communication actions between the left and right processes. Calder and Shankland provided a symbolic semantics for LOTOS [21]. As in the work done by Bonchi and Montanari, Calder and Shankland both represented their condition in the label position instead of dividing it into pre- and post-conditions. Pugliese, Tiezzi and Yoshida proposed a symbolic semantics for the service-oriented computing COWS that is similar to the Π-calculus [22]. For the works above that provide symbolic transition semantics, the condition is placed in a label instead of dividing it into pre- and post-conditions, which makes their symbolic semantics fail to answer differently for different input environments for the same program. In many cases, it is necessary to consider different initial environment conditions and these lead to different results in CSP programs.

mCRL2 is a process algebra designed to execute symbolically [23]. It is a well-known π-calculus-like generic language with symbolic transition semantics to model point-to-point communication. It claims to catch the behavior of infinite sets in its choice summation operator. However, the sets need to be determined statically; thus, it cannot catch behavior similar to process C in Figure 1.1. In addition, even though mCRL2 claims its language is generic, and that other process algebras can be translated into it, it is very hard for mCRL2 to model a broadcast communication system with point-to-group communication, because it requires the total number of processes in the universe, and sends a number of messages to each individual process in the group. However, knowing

the total number of processes is very hard in some cases. For example, it is almost impossible for mCRL2 to model a group of people reaching a consensus in a conference. On the other hand, we can model this procedure easily by using a replicated choice operator to select the total number of people in the conference, then using a replicated parallel operator with a consensus value to model the fact that all of the people communicate with each other by the consensus value.

Chapter 8

Conclusion and Future Work

In this thesis we have presented a new semantics for CSP, the HCSP semantics. HCSP provides an alternative way to model CSP processes by viewing transitions as bundles of the original transitions, where all transitions in the bundle can be described by a uniform property derived from the process. By this translation, we can allow HCSP-based tools to run some CSP programs which cannot be run in the original CSP-based tools. We have shown the HCSP semantics to be equivalent to the original CSP transition semantics. We have also presented an HCSP-based simulator, extracted directly from the Isabelle code for the HCSP semantics. We use an HCSP-based simulator to show traces of CSP programs and demonstrate by experiment that it is very efficient in dealing with some CSP programs. By using several examples in the experiment, we show that the HCSP semantics-based trace simulator can overcome some difficulties that traditional CSP semantics-based tools cannot handle.

For further study, we are interested in adding semantics for dealing with the replicated parallel operators in HCSP and use it in the simulator. We also propose to create a trace simulation model checker to check the trace refinement, trace failure and failure-divergence refinement properties of CSP programs. We believe that answering the trace refinement problem will significantly increase the efficiency of the model checker. We also want to generalize our framework to deal with other kinds of transition semantics.

Appendix A

HCSP theorem proof by Isabelle

In this section, we show the Isabelle proofs for the theories that we prove in Section 4. We will show the soundness and completeness proofs of HCSP with respect to original CSP. The original CSP syntax and semantics are defined in Isabelle theory file `lang`. The theory file `cspp_data` defines the generic CSP action datatype that we will be used in the proofs. The datatype can be instantiated to an executable datatype. For example, the simulator we show in Appendix B uses the instantiation procedure to get its action datatype. The theory `cspp_syntax` defines the syntax of HCSP, while the theory `cspp_semantic` defines the semantics of HCSP. The theory `cspp_procnamelocale` is a helper theory file to build the connection between HCSP and original CSP. The theory `cspp_noillformness` proves the Theorem 4.0.1 in Section 4. The Isabelle theory `cspp_soundness` proves the Theorem 4.0.5 in Section 4, while the Isabelle theory `cspp_completeness` proves the Theorem 4.0.6 in Section 4.

```
theory lang
imports cspp_data Main
begin


(*
the following section states the syntax and semantics of the original
    CSP which
we want to prove the soundness and completeness of HCSP semantics with
    respect to.
*)
datatype
('p,'a) proc
=
  OMEGA
| STOP
| SKIP
| DIV
| Act_prefix "'a" "('p,'a) proc" ("(1_ /-> _)" [150,80] 80)
| Ext_pre_choice "'a set" "'a => ('p,'a) proc"("(1? :_ /-> _)" [900,80]
    80)
| Ext_choice "('p,'a) proc" "('p,'a) proc" ("(1_ /[+] _)" [72,73] 72)
```

```
| Int_choice "('p,'a) proc" "('p,'a) proc" ("(1_ /|~| _)" [64,65] 64)
| Rep_int_choice "'a set" "'a => ('p,'a) proc" ("(1!! :_ /->_)" [900,68]
    68)
| IF "bool" "('p,'a) proc" "('p,'a) proc" ("(0IF _ /THEN _ /ELSE _)"
    [900,88,88] 88)
| Parallel "('p,'a) proc" "'a set" "('p,'a) proc" ("(1_ /|[_]| _)"
    [76,0,77] 76)
| Seq_compo "('p,'a) proc" "('p,'a) proc" ("(1_ /;; _)" [79,78] 78)
| Proc_name "'p" ("$_" [900] 90)
| Hid "('p, 'a) proc" "'a set"


locale TransSem =
 fixes PNfun :: "'p   ('p,'a)proc"
begin

inductive trans where
skip: "trans SKIP Check OMEGA" |
div: "trans DIV  DIV" |
act_prefix: "trans (a -> P) (Action a) P" |
set_prefix: "a   X    trans (? :X -> P) (Action a) (P a)" |
rep_prefix: "a   X    trans (!! :X -> P)  (P a)" |
int_choice1: "trans (P |~| Q)  P" |
int_choice2: "trans (P |~| Q)  Q" |
ext_choice_tau1: "trans P  P'   trans (P [+] Q)  (P' [+] Q)" |
ext_choice_tau2: "trans Q  Q'   trans (P [+] Q)  (P [+] Q')" |
ext_choice1: "a         trans P a P'   trans (P [+] Q) a P'" |
ext_choice2: "a         trans Q a Q'   trans (P [+] Q) a Q'" |
ifthenelse: "trans (IF b THEN P ELSE Q)  (if b then P else Q)" |
seq_step: "x   Check    trans P x P'   trans (P ;; Q) x (P' ;; Q)" |
seq_check: " P '. trans P Check P'   trans (P ;; Q)  Q" |
par_tau1: "trans P  P'    trans (P |[X]| Q)  (P' |[X]| Q)" |
par_tau2: "trans Q  Q'    trans (P |[X]| Q)  (P |[X]| Q')" |
par_out1: "a   X    trans P (Action a) P'
trans (P |[X]| Q) (Action a) (P' |[X]| Q)" |
par_out2: "a   X    trans Q (Action a) Q'
trans (P |[X]| Q) (Action a) (P |[X]| Q')" |
par_in: "a   X    trans P (Action a) P'  trans Q (Action a) Q'
trans (P |[X]| Q) (Action a) (P' |[X]| Q')" |
par_check1: "trans P Check P'  trans (P |[X]| Q)  (OMEGA |[X]| Q)" |
par_check2: "trans Q Check Q'  trans (P |[X]| Q)  (P |[X]| OMEGA)" |
par_omega: "trans (OMEGA |[X]| OMEGA) Check OMEGA" |
proc_name_rule: "trans ($p) Tau (PNfun p)"|
hid_omega: "trans P Check Q  trans (Hid P X) Check OMEGA"|
hid_tau: "trans P Tau Q  trans (Hid P X) Tau (Hid Q X)"|
hid_neg: "a   X  trans  P (Action a) Q   trans (Hid P X) (Action a)
    (Hid Q X)"|
```

```
hid_pos: "a    X     trans P (Action a) Q   trans (Hid P X) Tau (Hid Q X)"
end

end
```

```
theory cspp_data
imports Main
begin


(* the label used in the CSP and HCSP transition *)
datatype 'a label = Tau ("") | Check | Action 'a


(* type for variable *)
datatype 'a var = Var 'a


(* type for parameter *)

datatype 'a par = Par 'a

(*
a locale to provide the user the chance to implement their own data,
    bool expression and how they want to evaluate their data with some
    constraints. Since the assignment function defined here is a total
    function, we have a Null value to assign all the undefined
    parameters to this value.
*)
locale data =
  fixes  IdV :: "'var var   'act"
         and IdP :: "'var par   'act"
         and Equal :: "'act   'act    'boolexp"
         and Not :: "'boolexp   'boolexp"
         and True_p :: "'boolexp"
         and False_p :: "'boolexp"
         and Wf :: "'boolexp   'boolexp"
          and And :: "'boolexp   'boolexp    'boolexp"
        and model :: "('var par   'value)    'boolexp    bool"
        and replaceVarInAct :: "'act   'var    'act    'act"
        and replaceVarInBool :: "'boolexp   'var    'act    'boolexp"
        and evalInAct :: "('var par   'value)    'act    'value option"
        and evalInBool :: "('var par   'value)    'boolexp    bool option"
        and ActVal :: "'value   'act"
        and Null :: "'value"
 assumes
```

```
          True_not_False : "True_p  False_p"
         and evalPro: "evalInAct  (IdP U) = Some ( U)"
         and MTrue: "model  True_p = True"
        and MFalse: "model  False_p = False"
       and MNot: "model  (Not b) = ( (model   b))"
       and MEqual: "model  (Equal a c) = (case evalInAct  a of None
           False | Some e   case (evalInAct  c) of None   False | Some
           f   (e = f))"
       and MAnd: "model  (And b d) = (model  b    model   d)"
       and MWf: "model  (Wf b) = (case (evalInBool  b) of None  False |
           Some t   True)"
       and evalBoolPro: "(model  b) = (evalInBool  b = Some True)"
and actValProInBool: "(evalInBool  (replaceVarInBool b s (IdP y)) =
    evalInBool  (replaceVarInBool b s (ActVal ( y))))"
and actValProInAct: "(evalInAct  (replaceVarInAct a s (IdP y)) =
    evalInAct  (replaceVarInAct a s (ActVal ( y))))"
and singleSubProInValue:" svar    replaceVarInBool (replaceVarInBool b
    s (ActVal n)) var (ActVal av) = replaceVarInBool (replaceVarInBool
    b var (ActVal av)) s (ActVal n)"
and singleSubProInId:"svar    replaceVarInBool (replaceVarInBool b s
    (IdP y)) var (ActVal av) = replaceVarInBool (replaceVarInBool b var
    (ActVal av)) s (IdP y)"
and singleSubActInValue:" svar    replaceVarInAct (replaceVarInAct a s
    (ActVal n)) var (ActVal av) = replaceVarInAct (replaceVarInAct a
    var (ActVal av)) s (ActVal n)"
and singleSubActInId:"svar    replaceVarInAct (replaceVarInAct a s (IdP
    y)) var (ActVal av) = replaceVarInAct (replaceVarInAct a var
    (ActVal av)) s (IdP y)"

end
```

```
theory cspp_syntax
imports Main cspp_data
begin


(*
this function will provide the syntax of HCSP
*)
datatype ('var, 'act, 'bool , 'pname) cspp =
   omega| LSKIP | LSTOP | LDIV
 (**)| LProc_name 'pname(**)

(* c.a    P *)
 | LAct_prefix "'act" "'act" "('var, 'act,'bool , 'pname) cspp"
```

```
(*P [] Q*)
 | LExt_choice "('var, 'act,'bool , 'pname) cspp" "('var, 'act,'bool ,
     'pname) cspp"

(*P |~| Q *)
| LInt_choice "('var, 'act,'bool, 'pname) cspp" "('var, 'act,'bool,
    'pname) cspp"

(* IF B then P else Q *)
 | LIF "'bool" "('var, 'act,'bool , 'pname) cspp" "('var, 'act,'bool ,
     'pname) cspp"

(* P |[{k.x|B}]| Q *)
| LParallel "('var, 'act,'bool , 'pname) cspp" 'var 'var "'bool" "('var,
    'act,'bool , 'pname) cspp"

(* P ; Q*)
 | LSeq_compo "('var, 'act,'bool , 'pname) cspp" "('var, 'act,'bool ,
     'pname) cspp"

(* c?x:B    P *)
 | LExt_pre_choice "'act" 'var "'bool" "('var, 'act,'bool, 'pname) cspp"

(*|~|x:B @ P *)
 | LRep_int_choice 'var "'bool" "('var, 'act,'bool , 'pname) cspp"

(* P \ {x|B} *)
 | LHid "('var, 'act,'bool , 'pname) cspp" 'var 'var "'bool"

end
```

```
theory cspp_semantic
imports cspp_data cspp_syntax Main
begin

(*
the locale mainly define the process name and process environment for
    the HCSP semantics
the function replaceVarInDoubleBool is just an abbreviation of function
    T[U/k][V/x]. Providing this
function is because we want to make the evaluation of the transition
    rules more faster, ie, people can
substitute two variables at a time instead of applying the substitution
    function twice
*)
```

```
locale procEnvLocale = data IdV IdP Equal Not True_p False_p Wf And
    model replaceVarInAct replaceVarInBool evalInAct evalInBool ActVal
    Null
for          IdV :: "'var var   'act"
         and IdP :: "'var par   'act"
         and Equal :: "'act   'act    'boolexp"
         and Not :: "'boolexp   'boolexp"
         and True_p :: "'boolexp"
         and False_p :: "'boolexp"
         and Wf :: "'boolexp   'boolexp"
          and And :: "'boolexp   'boolexp    'boolexp"
        and model :: "('var par   'value)    'boolexp    bool"
        and replaceVarInAct :: "'act   'var    'act    'act"
        and replaceVarInBool :: "'boolexp   'var    'act    'boolexp"
        and evalInAct :: "('var par   'value)    'act    'value option"
        and evalInBool :: "('var par   'value)    'boolexp    bool option"
        and ActVal :: "'value   'act"
        and Null :: "'value"
+
fixes procEnv :: "'pname   ('var,'act, 'boolexp,'pname) cspp"
and replaceVarInProcName :: "('act   'var    'act    'act)    'var
    'act    'pname    'pname"
and getNewVar :: "('var par) list   'var par"
and replaceVarInDoubleBool :: "'boolexp   'var   'act    'var    'act
    'boolexp"
and procNameSem :: "(('var par   'value)    'pname    'pname)"
assumes
freshVar : " varSet . ((getNewVar varSet)) (set varSet)"
and procNameSemProperty: "    a . (procNameSem   (replaceVarInProcName
    replaceVarInAct s (IdP a) P)=procNameSem   (replaceVarInProcName
    replaceVarInAct s (ActVal ( a)) P))"
and twoSubProInId: "svar1 & s    var2    replaceVarInDoubleBool
    (replaceVarInBool bool s (IdP y))
         var1 (ActVal av) var2 (ActVal cv) =
        replaceVarInBool
         (replaceVarInDoubleBool bool var1 (ActVal av) var2 (ActVal cv))
         s (IdP y)"
and twoSubProInValue: "svar1 & s    var2    replaceVarInDoubleBool
    (replaceVarInBool bool s (ActVal n))
         var1 (ActVal av) var2 (ActVal cv) =
        replaceVarInBool
         (replaceVarInDoubleBool bool var1 (ActVal av) var2 (ActVal cv))
         s (ActVal n)"
 and procNameSubst1: "ss'
      replaceVarInProcName replaceVarInAct s' (ActVal cv)
       (replaceVarInProcName replaceVarInAct s (ActVal n) pname) =
      replaceVarInProcName replaceVarInAct s (ActVal n)
```

```
       (replaceVarInProcName replaceVarInAct s' (ActVal cv) pname)"
and procNameSubst2: "s s'
      replaceVarInProcName replaceVarInAct s' (ActVal cv)
       (replaceVarInProcName replaceVarInAct s (IdP y) pname) =
      replaceVarInProcName replaceVarInAct s (IdP y)
       (replaceVarInProcName replaceVarInAct s' (ActVal cv) pname)"
and actValProInDoubleBool:"(evalInBool  (replaceVarInDoubleBool b s (IdP
    x) s' (IdP y))=evalInBool  (replaceVarInDoubleBool b s (ActVal (
    x)) s' (ActVal ( y))))"



(*
a function to tell how to substitute a variable with a user difiend act
    in a process in HCSP
*)
primrec (in procEnvLocale) replaceVarInCspp :: "(('act  'var  'act
    'act)    'var    'act    'pname    'pname)    ('var,'act,
    'boolexp,'pname) cspp  'var   'act    ('var,'act, 'boolexp,'pname)
    cspp" where
"replaceVarInCspp pf LSKIP s a = LSKIP"
|"replaceVarInCspp pf omega s a=omega"
| "replaceVarInCspp pf LSTOP s a = LSTOP"
| "replaceVarInCspp pf LDIV s a = LDIV"
| "replaceVarInCspp pf (LProc_name p) s a = LProc_name (pf
    replaceVarInAct s a p)"
| "replaceVarInCspp pf (LAct_prefix c aa P) s a = LAct_prefix
    (replaceVarInAct c s a) (replaceVarInAct aa s a) (replaceVarInCspp
    pf P s a)"
| "replaceVarInCspp pf (LExt_choice P Q) s a = LExt_choice
    (replaceVarInCspp pf P s a) (replaceVarInCspp pf Q s a)"
| "replaceVarInCspp pf (LInt_choice P Q) s a = LInt_choice
    (replaceVarInCspp pf P s a) (replaceVarInCspp pf Q s a)"
| "replaceVarInCspp pf (LSeq_compo P Q) s a = LSeq_compo
    (replaceVarInCspp pf P s a) (replaceVarInCspp pf Q s a)"
| "replaceVarInCspp pf (LIF b P Q) s a = LIF (replaceVarInBool b s a)
    (replaceVarInCspp pf P s a) (replaceVarInCspp pf Q s a)"
| "replaceVarInCspp pf (LParallel P c s us Q) s' a = LParallel
    (replaceVarInCspp pf P s' a) c s (if ((c=s') | (s=s')) then us else
    (replaceVarInBool us s' a)) (replaceVarInCspp pf Q s' a)"
| "replaceVarInCspp pf (LExt_pre_choice c ss aset P) s a =(if (s = ss)
    then LExt_pre_choice (replaceVarInAct c s a) ss aset P else
    LExt_pre_choice (replaceVarInAct c s a) ss (replaceVarInBool aset s
    a) (replaceVarInCspp pf P s a))"
| "replaceVarInCspp pf (LRep_int_choice ss aset P) s a =(if (s = ss)
    then LRep_int_choice ss aset P else LRep_int_choice ss
```

```
    (replaceVarInBool aset s a) (replaceVarInCspp pf P s a))"
| "replaceVarInCspp pf (LHid P c s us) s' a= LHid (replaceVarInCspp pf P
    s' a) c s (if ((c=s') | (s=s')) then us else (replaceVarInBool us
    s' a))"



(*
This is the actually HCSP semantics.
*)
inductive (in procEnvLocale) oneStepEval where
(*
(  ,  ,SKIP) -- Check  oemga
*)
lskip[intro!]: "oneStepEval
    (varSet,replaceVarInProcName,procEnv,varEnv,LSKIP) Check
    (varSet,replaceVarInProcName,procEnv,varEnv,omega)"
(*
DIV --  DIV
*)
|ldiv[intro!]: "oneStepEval
    (varSet,replaceVarInProcName,procEnv,varEnv,LDIV) Tau
    (varSet,replaceVarInProcName,procEnv,varEnv,LDIV)"
(*
act_prefix rule: c.a  P --(U.V) P  by given varEnv   (U=c)    (V=a)
*)
|lact_prefix: " U =getNewVar varSet ; V = getNewVar (U#varSet)
    oneStepEval
    (varSet,replaceVarInProcName,procEnv,varEnv,(LAct_prefix c a P))
    (Action (U,V)) ((U# (V#varSet)),replaceVarInProcName,procEnv, And
    (Equal (IdP U) c) (And (Equal (IdP V) a) varEnv),P)"
(*
internal choice rules
*)
| lint_choice1: "oneStepEval
    (varSet,replaceVarInProcName,procEnv,varEnv,(LInt_choice P Q)) Tau
    (varSet,replaceVarInProcName,procEnv,varEnv,P)"
| lint_choice2: "oneStepEval
    (varSet,replaceVarInProcName,procEnv,varEnv,(LInt_choice P Q)) Tau
    (varSet,replaceVarInProcName,procEnv,varEnv,Q)"
(*
external choice with tau label
*)
| lext_choice_tau1: "oneStepEval
    (varSet,replaceVarInProcName,procEnv,varEnv,P) Tau
    (varSet',replaceVarInProcName',procEnv',varEnv',P')
    oneStepEval(varSet,replaceVarInProcName,procEnv,varEnv,(LExt_choice
```

```
    P Q)) Tau
    (varSet',replaceVarInProcName',procEnv',varEnv',(LExt_choice P' Q))"
| lext_choice_tau2: "oneStepEval
    (varSet,replaceVarInProcName,procEnv,varEnv,Q) Tau
    (varSet',replaceVarInProcName',procEnv',varEnv',Q')
    oneStepEval(varSet,replaceVarInProcName,procEnv,varEnv,(LExt_choice
    P Q)) Tau
    (varSet',replaceVarInProcName',procEnv',varEnv',(LExt_choice P Q'))"
(*
external choice with non-tau label
*)
| lext_choice1: "  a     Tau ; oneStepEval
    (varSet,replaceVarInProcName,procEnv,varEnv,P) a
    (varSet',replaceVarInProcName',procEnv',varEnv',P')  oneStepEval
    (varSet,replaceVarInProcName,procEnv,varEnv,(LExt_choice P Q)) a
    (varSet',replaceVarInProcName',procEnv',varEnv',P')"
| lext_choice2: " a     Tau ; oneStepEval
    (varSet,replaceVarInProcName,procEnv,varEnv,Q) a
    (varSet',replaceVarInProcName',procEnv',varEnv',Q')  oneStepEval
    (varSet,replaceVarInProcName,procEnv,varEnv,(LExt_choice P Q)) a
    (varSet',replaceVarInProcName',procEnv',varEnv',Q')"
(*
sequential step
*)
| lseq_step: "  xCheck   ; oneStepEval
    (varSet,replaceVarInProcName,procEnv,varEnv,P) x
    (varSet',replaceVarInProcName',procEnv',varEnv',P')  oneStepEval
    (varSet,replaceVarInProcName,procEnv,varEnv,(LSeq_compo P Q)) x
    (varSet',replaceVarInProcName',procEnv',varEnv',(LSeq_compo P' Q))"
(*
sequential rule if label is check
*)
| lseq_check: " P '.(oneStepEval
    (varSet,replaceVarInProcName,procEnv,varEnv,P) Check
    (varSet',replaceVarInProcName',procEnv',varEnv',P'))  oneStepEval
    (varSet,replaceVarInProcName,procEnv,varEnv,(LSeq_compo P Q)) Tau
    (varSet',replaceVarInProcName',procEnv',varEnv',Q)"
(*
If statement
*)
| lifthenelse1: "oneStepEval
    (varSet,replaceVarInProcName,procEnv,varEnv,LIF b P Q) Tau
    (varSet,replaceVarInProcName,procEnv,(And b varEnv),P)"
| lifthenelse2: "oneStepEval
    (varSet,replaceVarInProcName,procEnv,varEnv,LIF b P Q) Tau
    (varSet,replaceVarInProcName,procEnv,(And (And (Not b) (Wf b))
    varEnv),Q)"
```

```
(*
external pre-choice rule
*)
| lext_prefix: " U  = getNewVar varSet ; V = getNewVar (U#varSet)
    oneStepEval
        (varSet,replaceVarInProcName,procEnv,varEnv,LExt_pre_choice c s
        apropt P)
    (Action (U,V))
(U#(V# varSet),replaceVarInProcName,procEnv,And (Equal (IdP U) c) (And
    varEnv (replaceVarInBool apropt s (IdP V))),replaceVarInCspp
    replaceVarInProcName P s (IdP V))"
(*
replicated internal choice
*)
| lint_prefix: " U= getNewVar varSet
   oneStepEval
       (varSet,replaceVarInProcName,procEnv,varEnv,LRep_int_choice s
       apropt P)
   Tau
   (U # varSet,replaceVarInProcName,procEnv,(And varEnv
       (replaceVarInBool apropt s (IdP U))),(replaceVarInCspp
       replaceVarInProcName P s (IdP U)))"
(*
parallel rules with check label
*)
| lpar_check1: "oneStepEval
    (varSet,replaceVarInProcName,procEnv,varEnv,P) Check
    (varSet',replaceVarInProcName',procEnv',varEnv',P')  oneStepEval
    (varSet,replaceVarInProcName,procEnv,varEnv,LParallel P s s' X Q)
    Tau (varSet',replaceVarInProcName',procEnv',varEnv',LParallel omega
    s s' X Q)"
| lpar_check2: "oneStepEval
    (varSet,replaceVarInProcName,procEnv,varEnv,Q) Check
    (varSet',replaceVarInProcName',procEnv',varEnv',Q')  oneStepEval
    (varSet,replaceVarInProcName,procEnv,varEnv,LParallel P s s' X Q)
    Tau (varSet',replaceVarInProcName',procEnv',varEnv',LParallel P s
    s' X omega)"
(*
parallel rules with tau label
*)
| lpar_omega: "oneStepEval
    (varSet,replaceVarInProcName,procEnv,varEnv,LParallel omega s s' X
    omega) Check (varSet,replaceVarInProcName,procEnv,varEnv,omega)"
| lpar_tau1: "oneStepEval (varSet,replaceVarInProcName,procEnv,varEnv,P)
    Tau (varSet',replaceVarInProcName',procEnv',varEnv',P')
    oneStepEval (varSet,replaceVarInProcName,procEnv,varEnv,LParallel P
    s s' X Q) Tau
```

```
    (varSet',replaceVarInProcName',procEnv',varEnv',LParallel P' s s' X
    Q)"
| lpar_tau2: "oneStepEval (varSet,replaceVarInProcName,procEnv,varEnv,Q)
    Tau (varSet',replaceVarInProcName',procEnv',varEnv',Q')
    oneStepEval (varSet,replaceVarInProcName,procEnv,varEnv,LParallel P
    s s' X Q) Tau
    (varSet',replaceVarInProcName',procEnv',varEnv',LParallel P s s' X
    Q')"
(*
parallel rules without communication between two processes
*)
| lpar_out1: "oneStepEval (varSet,replaceVarInProcName,procEnv,varEnv,P)
    (Action (U,V)) (varSet',replaceVarInProcName',procEnv',varEnv',P')
    oneStepEval (varSet,replaceVarInProcName,procEnv,varEnv,LParallel P
    s s' X Q) (Action (U,V)) (varSet',replaceVarInProcName',procEnv',
(And varEnv' (Not (replaceVarInDoubleBool X s (IdP U) s' (IdP V)))),
LParallel P' s s' X Q)"
| lpar_out2: "oneStepEval (varSet,replaceVarInProcName,procEnv,varEnv,Q)
    (Action (U,V)) (varSet',replaceVarInProcName',procEnv',varEnv',Q')
    oneStepEval (varSet,replaceVarInProcName,procEnv,varEnv,LParallel P
    s s' X Q) (Action (U,V)) (varSet',replaceVarInProcName',procEnv',
(And varEnv' (Not (replaceVarInDoubleBool X s (IdP U) s' (IdP V)))),
LParallel P s s' X Q')"
(*
parallel communication rule:
(  ,  ,P) --(c.a)    (  ',  ',P')   (  ',  ',Q) --(c'.a')    (  '',  '',Q')
-------------------------------------------------------------------------
(  ,  ,P|[k.x|B]|Q) --(c.a)
     (  '',(c=c')  (a=a')  (B[c/k][a/x])    '',P'|[k.x|B]|Q')
*)
| lpar_in: "oneStepEval (varSet,replaceVarInProcName,procEnv,varEnv,P)
    (Action (U,V)) (varSet',replaceVarInProcName',procEnv',varEnv',P')
    ; oneStepEval (varSet',replaceVarInProcName',procEnv',varEnv',Q)
    (Action (U',V'))
    (varSet'',replaceVarInProcName'',procEnv'',varEnv'',Q')
                 oneStepEval
                  (varSet,replaceVarInProcName,procEnv,varEnv,LParallel
                  P s s' X Q)
                           (Action (U,V))
                            (varSet'',replaceVarInProcName'',procEnv'',
                                   And (And (Equal (IdP U) (IdP U'))
                                        (And varEnv''
                                        (replaceVarInDoubleBool X s
                                        (IdP U) s' (IdP V)))) (Equal
                                        (IdP V) (IdP V'))
                                      ,LParallel P' s s' X Q')"
(*
```

```
hiding rule with check label
*)
| lhid_omega: "oneStepEval
    (varSet,replaceVarInProcName,procEnv,varEnv,P) Check
    (varSet',replaceVarInProcName',procEnv',varEnv',P')  oneStepEval
    (varSet,replaceVarInProcName,procEnv,varEnv,LHid P s s' X) Check
    (varSet',replaceVarInProcName',procEnv',varEnv',omega)"
(*
hiding rule with tau label
*)
| lhid_tau: "oneStepEval (varSet,replaceVarInProcName,procEnv,varEnv,P)
    Tau (varSet',replaceVarInProcName',procEnv',varEnv',P')
    oneStepEval (varSet,replaceVarInProcName,procEnv,varEnv,LHid P s s'
    X) Tau (varSet',replaceVarInProcName',procEnv',varEnv',LHid P' s s'
    X)"
(*
hiding rule with the action is not the one we want to hide
*)
| lhid_neg: "oneStepEval (varSet,replaceVarInProcName,procEnv,varEnv,P)
    (Action (U,V)) (varSet',replaceVarInProcName',procEnv',varEnv',P')
    oneStepEval (varSet,replaceVarInProcName,procEnv,varEnv,LHid P s s'
    X) (Action (U,V)) (varSet',replaceVarInProcName',procEnv',(And
    varEnv' (Not (replaceVarInDoubleBool X s (IdP U) s' (IdP V)))),LHid
    P' s s' X)"
(*
hiding rule with the action is the one we want to hide
*)
|lhid_pos: "oneStepEval (varSet,replaceVarInProcName,procEnv,varEnv,P)
    (Action (U,V)) (varSet',replaceVarInProcName',procEnv',varEnv',P')
    oneStepEval (varSet,replaceVarInProcName,procEnv,varEnv,LHid P s s'
    X) Tau (varSet',replaceVarInProcName',procEnv',(And varEnv'
    (replaceVarInDoubleBool X s (IdP U) s' (IdP V))),LHid P' s s' X)"
(*
process name rule
*)
| lproc_name: "oneStepEval
    (varSet,replaceVarInProcName,procEnv,varEnv,LProc_name p) Tau
    (varSet,replaceVarInProcName,procEnv,varEnv,(procEnv p))"



end
```

```
theory cspp_procnamelocale
imports cspp_data cspp_syntax cspp_semantic lang Main
```

```
begin



(*
the semantics function for the label, and translate a label in HCSP into
    a label in the original CSP.
*)
primrec (in procEnvLocale) labelSem :: "('var par  'value)  (('var par)
    * ('var par)) label  ('value * 'value) label" where
"labelSem env Check= Check"|
"labelSem env Tau = Tau"|
"labelSem env (Action x) = (case x of (c,a)  (Action (env c,env a)))"



(*
the semantics function will translate a HCSP process into a process in
    the original CSP
the function needs to be proved termination in the Isabelle by using a
    simple counter function
*)
function (in procEnvLocale) csppSem :: "(('act  'var  'act    'act)
    'var    'act    'pname    'pname)   ('var par    'value)
      ('var,'act, 'boolexp,'pname) cspp ('pname,('value * 'value))
    proc" where
"csppSem pf env LSKIP = SKIP"|
"csppSem pf env omega = OMEGA"|
"csppSem pf env LSTOP = STOP"|
"csppSem pf env LDIV = DIV"|
"csppSem pf env (LProc_name p) = Proc_name (procNameSem env p)"|
"csppSem pf env (LIF p P Q) = (case (evalInBool env p) of None  STOP |
    Some b  (IF b THEN (csppSem pf env P) ELSE (csppSem pf env Q)))"|
"csppSem pf env (LAct_prefix c a P) = (case (evalInAct env c) of None
    STOP | Some c' (case (evalInAct env a) of None  STOP | Some a'
    Act_prefix (c',a') (csppSem pf env P)))" |
"csppSem pf env (LExt_choice P Q) = (Ext_choice (csppSem pf env P)
    (csppSem pf env Q))"|
"csppSem pf env (LInt_choice P Q) = (Int_choice (csppSem pf env P)
    (csppSem pf env Q))"|
"csppSem pf env (LParallel P s s' uset Q) = Parallel (csppSem pf env P)
{(cv,av). (model env (replaceVarInDoubleBool uset s (ActVal cv) s'
    (ActVal av)))}
(csppSem pf env Q)"|
"csppSem pf env (LSeq_compo P Q) = Seq_compo (csppSem pf env P) (csppSem
    pf env Q)"|
"csppSem pf env (LExt_pre_choice c s as P) = (Ext_pre_choice
```

```
{(c',a). (evalInAct env c = Some c')&(model env (replaceVarInBool as s
    (ActVal a)))}
(  (c,n). csppSem pf env (replaceVarInCspp pf P s (ActVal n))))"|
"csppSem pf env (LRep_int_choice s as P) =(Rep_int_choice
{(c',a). (model env (replaceVarInBool as s (ActVal a)))}
(  (c,n). csppSem pf env (replaceVarInCspp pf P s (ActVal n))))"|
"csppSem pf env (LHid P s s' uset)= Hid (csppSem pf env P) {(cv,av).
    (model env (replaceVarInDoubleBool uset s (ActVal cv) s' (ActVal
    av)))}"
by pat_completeness auto


primrec counter :: "('var,'act, 'boolexp,'pname) cspp  nat" where
"counter LSKIP = 0"
|"counter LDIV = 0"
| "counter omega = 0"
| "counter LSTOP = 0"
| "counter (LProc_name p)= 0"
| "counter (LIF p P Q) = 1+ (counter P) + (counter Q)"
| "counter (LAct_prefix c a P) = 1+ (counter P)"
| "counter (LExt_choice P Q) = 1+ (counter P) + (counter Q)"
| "counter (LInt_choice P Q) = 1+ (counter P) + (counter Q)"
| "counter (LParallel P s s' uset Q) = 1+(counter P) + (counter Q)"
| "counter (LSeq_compo P Q) = 1+(counter P)+(counter Q)"
| "counter (LExt_pre_choice c s as P) = 1+(counter P)"
| "counter (LRep_int_choice s as P) = 1+(counter P)"
| "counter (LHid P s s' uset) = 1+(counter P)"
(*| "counter (LRep_ext_choice s as P) = 1+(counter P)"*)


(*
this two lemmas prove the termination of function csppSem
*)
lemma (in procEnvLocale) termination_aux1: "counter (replaceVarInCspp pf
    P s (ActVal xa)) < Suc (counter P)"
apply (induct P)
apply auto
done


termination (in procEnvLocale) csppSem
apply (relation "measure ((pf,env,p). counter p)")
apply auto
apply (simp add:termination_aux1)
apply (simp add:termination_aux1)
done
```

```
(*
the locale provides the freeVars function and useful assumptions to
    prove soundness and completeness
*)
locale procNameLocale = procEnvLocale IdV IdP Equal Not True_p False_p
    Wf And model replaceVarInAct replaceVarInBool evalInAct evalInBool
    ActVal Null procEnv replaceVarInProcName getNewVar
    replaceVarInDoubleBool procNameSem
for          IdV :: "'var var   'act"
         and IdP :: "'var par   'act"
         and Equal :: "'act   'act    'boolexp"
         and Not :: "'boolexp   'boolexp"
         and True_p :: "'boolexp"
         and False_p :: "'boolexp"
         and Wf :: "'boolexp   'boolexp"
          and And :: "'boolexp   'boolexp    'boolexp"
        and model :: "('var par   'value)    'boolexp    bool"
        and replaceVarInAct :: "'act  'var    'act    'act"
        and replaceVarInBool :: "'boolexp  'var    'act    'boolexp"
        and evalInAct :: "('var par  'value)    'act    'value option"
        and evalInBool :: "('var par  'value)    'boolexp    bool option"
        and ActVal :: "'value   'act"
        and Null :: "'value"
        and procEnv :: "'pname   ('var,'act, 'boolexp,'pname) cspp"
        and replaceVarInProcName :: "('act  'var    'act    'act)
            'var    'act    'pname    'pname"
        and getNewVar :: "('var par) list  'var par"
        and replaceVarInDoubleBool :: "'boolexp  'var   'act    'var
            'act    'boolexp"
        and procNameSem :: "(('var par  'value)    'pname    'pname)"
+
fixes
getFreeParsInBool :: "'boolexp  ('var par) set"
and getFreeParsInAct :: "'act  ('var par) set"
and getFreeParsInProcName :: "'pname  ('var par) set"
assumes
varsIdP: "getFreeParsInAct (IdP U) = {U}"
and varsIdV: "getFreeParsInAct (IdV x) = {}"
and varsAnd: "getFreeParsInBool (And b d) = getFreeParsInBool b Un
    getFreeParsInBool d"
and varsNot: "getFreeParsInBool (Not b) = getFreeParsInBool b"
and varsEqual: "getFreeParsInBool (Equal a c) = getFreeParsInAct a Un
    getFreeParsInAct c"
and varsWf: "getFreeParsInBool (Wf b) = getFreeParsInBool b"
and getFreeParsInProcNameFeature: "varSet . getFreeParsInProcName pname
    varSet  getFreeParsInProcName (replaceVarInProcName replaceVarInAct
```

```
    s (IdP y) pname)  insert y varSet"
and getFreeParsInProcNameFeatureX: "varSet . getFreeParsInProcName pname
     varSet  getFreeParsInProcName (replaceVarInProcName
    replaceVarInAct s (ActVal av) pname)  varSet"
and getFreeParsInActFeature: "varSet . getFreeParsInAct a  varSet
    getFreeParsInAct (replaceVarInAct a s (IdP y))  insert y varSet"
and getFreeParsInActFeatureX: "varSet . getFreeParsInAct a  varSet
    getFreeParsInAct (replaceVarInAct a s (ActVal av))  varSet"
and getFreeParsInBoolFeature: "varSet . getFreeParsInBool b  varSet
    getFreeParsInBool (replaceVarInBool b s (IdP y))  insert y varSet"
and getFreeParsInBoolFeatureX: "varSet . getFreeParsInBool b  varSet
    getFreeParsInBool (replaceVarInBool b s (ActVal av))  varSet"
and getFreeParsInDoubleBoolFeature: "varSet varSet' . getFreeParsInBool
    b varSet & U  varSet' & V   varSet' & varSet   varSet'
    getFreeParsInBool (replaceVarInDoubleBool b s (IdP U) s' (IdP V))
    varSet'"
and getFreeParsInDoubleBoolFeatureX: "varSet. (getFreeParsInBool b
    varSet  getFreeParsInBool (replaceVarInDoubleBool b s (ActVal av)
    s' (ActVal cv))  varSet)"
and  procNameSemFeature : "    p. csppSem replaceVarInProcName  (procEnv
    p) = csppSem replaceVarInProcName ( x . Null) (procEnv (procNameSem
     p))"
and getFreeParsProperty : "     '. getFreeParsInProcName pname  varSet &
    ( x . x     varSet       x =   ' x)     procNameSem   pname =
    procNameSem ' pname"
and  evalProOfSameEnv: "      '. getFreeParsInAct a  varSet & ( x . x
    varSet       x =   ' x)     evalInAct   a = evalInAct ' a"
and  evalBoolOfSameEnv: "      '. getFreeParsInBool b  varSet & ( x . x
       varSet       x =   ' x)     evalInBool   b = evalInBool ' b"



(*
a function to get all free paramters in HCSP process, this corresponds
    to the actual freeVars function in paper. Since all free
    identifiers generated in the evaluation of a HCSP process is
    parameters.
*)
primrec (in procNameLocale) getFreeParsInCspp :: "('var,'act,
    'boolexp,'pname) cspp ('var par) set" where
"getFreeParsInCspp LSKIP = {}"
|"getFreeParsInCspp omega ={}"
| "getFreeParsInCspp LSTOP ={}"
| "getFreeParsInCspp LDIV ={}"
| "getFreeParsInCspp (LProc_name p) = getFreeParsInProcName p"
| "getFreeParsInCspp (LAct_prefix c aa P) =(getFreeParsInAct c) Un
    (getFreeParsInAct aa)Un (getFreeParsInCspp P)"
```

```
| "getFreeParsInCspp (LExt_choice P Q) = (getFreeParsInCspp P)Un
    (getFreeParsInCspp Q)"
| "getFreeParsInCspp (LInt_choice P Q) =(getFreeParsInCspp P)Un
    (getFreeParsInCspp Q)"
| "getFreeParsInCspp (LSeq_compo P Q) = (getFreeParsInCspp P)Un
    (getFreeParsInCspp Q)"
| "getFreeParsInCspp (LIF b P Q) = (getFreeParsInBool b)Un
    (getFreeParsInCspp P)Un (getFreeParsInCspp Q)"
| "getFreeParsInCspp (LParallel P s s' us Q) = (getFreeParsInCspp P) Un
    ((getFreeParsInBool us)) Un (getFreeParsInCspp Q)"
| "getFreeParsInCspp (LExt_pre_choice c ss aset P)=(getFreeParsInAct c)
    Un (((getFreeParsInBool aset) Un(getFreeParsInCspp P)))"
| "getFreeParsInCspp (LRep_int_choice ss aset P) = (((getFreeParsInBool
    aset) Un (getFreeParsInCspp P)))"
| "getFreeParsInCspp (LHid P s s' us)= (getFreeParsInCspp P) Un
    ((getFreeParsInBool us))"



(*
sublocale to merge PNfun locale and procNameLocale and we will be ready
    to prove soundness and completeness
*)
sublocale procNameLocale  TransSem
where PNfun = " p.  (csppSem replaceVarInProcName ( x . Null) (procEnv
    p))"
done

end
```

```
theory cspp_noillformness
imports cspp_data cspp_syntax cspp_semantic lang Main cspp_procnamelocale
begin

(*show the noillformness *)

(*
The lemma suggests that the variables set will always grow by the
    evaluation through the HCSP semantics.
*)
lemma (in procEnvLocale) setGrowing: "oneStepEval x a y  (set (fst
    x))(set (fst y))"
apply (erule oneStepEval.induct)
apply auto
done
```

```
(*
This following four lemmas suggest that if all the free variables of the
    pre evaluated process belongs to the pre existing variables set,
then the free variables in the label will belong to the post existing
    variables set.
*)
lemma (in procNameLocale) labelVarsBelonging_aux: "oneStepEval x a
     y (getFreeParsInCspp (snd (snd (snd (snd x)))) (set (fst x)) & (a=
    Action (c,acts))acts  (set (fst y)))"
apply (erule oneStepEval.induct)
apply clarsimp+
apply (frule setGrowing)
apply (frule_tac x="(varSet', replaceVarInProcName', procEnv', varEnv',
    Q)" and a="(Action (U', V'))" and y=" (varSet'',
    replaceVarInProcName'', procEnv'', varEnv'', Q')" in setGrowing)
apply clarsimp
apply blast
apply clarsimp+
done

lemma (in procNameLocale) labelVarsBelonging: "oneStepEval x a
    ygetFreeParsInCspp (snd (snd (snd (snd x)))) (set (fst x))    (a=
    Action (c,acts))acts    (set (fst y))"
apply (simp add:labelVarsBelonging_aux)
done


lemma (in procNameLocale) labelChanBelonging_aux: "oneStepEval x a
     y ( c  acts. (getFreeParsInCspp (snd (snd (snd (snd x)))) (set
    (fst x)) & (a= Action (c,acts))c  (set (fst y))))"
apply (erule oneStepEval.induct)
apply clarsimp+
apply (frule setGrowing)
apply (frule_tac x="(varSet', replaceVarInProcName', procEnv', varEnv',
    Q)" in setGrowing)
apply clarsimp
apply blast
apply clarsimp+
done

lemma (in procNameLocale) labelChanBelonging: "oneStepEval x a
    ygetFreeParsInCspp (snd (snd (snd (snd x)))) (set (fst x))    (a=
    Action (c,acts))  c    (set (fst y))"
apply (simp add:labelChanBelonging_aux)
done
```

```
(*
the following lemma suggests that if all free variables of process P
    belong to the set {x} U A, then all free vars of P[U/x] belong to
    {U} U A.
*)
lemma (in procNameLocale) freeVarSubsWithPar[rule_format]: "varSet .
    (getFreeParsInCspp P)  varSet  (getFreeParsInCspp (replaceVarInCspp
    replaceVarInProcName P s (IdP y)))  insert y varSet"
apply (induct P)
apply clarsimp+
apply (cut_tac s="s" and y = "y" and pname="pname" in
    getFreeParsInProcNameFeature)
apply (erule_tac x="varSet" in allE)
apply clarsimp
apply blast
apply clarsimp
apply (simp add:getFreeParsInActFeature)
apply clarsimp+
apply (cut_tac s="s" and y="y" and b="bool" in getFreeParsInBoolFeature)
apply (erule_tac x="varSet" in allE)+
apply clarsimp
apply blast
apply clarsimp
apply (case_tac "var1=s")
apply (case_tac "var2=s")
apply clarsimp
apply blast
apply clarsimp
apply blast
apply (case_tac "var2=s")
apply clarsimp
apply blast
apply clarsimp
apply (cut_tac s="s" and y="y" and b="bool" in getFreeParsInBoolFeature)
apply (erule_tac x="varSet" in allE)+
apply clarsimp
apply blast
apply clarsimp+
apply (case_tac "s=var")
apply clarsimp
apply (simp add: getFreeParsInActFeature)
apply (cut_tac s="s" and y="y" and b="bool" in getFreeParsInBoolFeature)
apply (erule_tac x="varSet" in allE)+
apply force
apply clarsimp
apply (simp add: getFreeParsInActFeature)
```

```
apply (cut_tac s="s" and y="y" and b="bool" in getFreeParsInBoolFeature)
apply (erule_tac x="varSet" in allE)+
apply clarsimp
apply clarsimp
apply (case_tac "s=var")
apply clarsimp
apply (cut_tac s="s" and y="y" and b="bool" in getFreeParsInBoolFeature)
apply (erule_tac x="varSet" in allE)+
apply force
apply clarsimp
apply (cut_tac s="s" and y="y" and b="bool" in getFreeParsInBoolFeature)
apply (erule_tac x="varSet" in allE)+
apply clarsimp
apply blast
apply clarsimp
apply (case_tac "var1=s")
apply (case_tac "var2=s")
apply clarsimp
apply blast
apply clarsimp
apply blast
apply (case_tac "var2=s")
apply clarsimp
apply blast
apply clarsimp
apply (cut_tac s="s" and y="y" and b="bool" in getFreeParsInBoolFeature)
apply (erule_tac x="varSet" in allE)+
apply clarsimp
apply blast
done


(*
The lemma suggests: A   B and B    C, then A    C, it is the transitivity
    of set.
*)
lemma setTrans: "  xvarSet   ; varSet    varSet'       xvarSet '"
apply auto
done


(*
these two lemmas state the fact that when we evaluate a process, the
    procEnv and replaceVarInProcName function will not change.
*)
lemma (in procEnvLocale) fix_replace : "oneStepEval x a y  (fst (snd
    y))= (fst (snd x))"
```

```
apply (erule oneStepEval.induct)
apply auto
done

lemma (in procEnvLocale) fix_procEnv : "oneStepEval x a y  (fst (snd
    (snd y)))= (fst (snd (snd x)))"
apply (erule oneStepEval.induct)
apply auto
done

(*
The noillformness suggests the fact that if all free variables of
    current process and the pre-condition belong to the pre-existing
    variables set,
and free variables set in any process names p will be larger than the
    free variables set in its corresponding process in procEnv, then
    all free
variables of the evaluated process and the post-condition will belong to
    the post-variable set.
*)
lemma (in procNameLocale) noIllFormedNess: " oneStepEval x a y     ((
    ((getFreeParsInCspp (snd (snd(snd(snd x))))) Un(getFreeParsInBool
    (fst (snd (snd (snd x))))))(set (fst x))) & ( p . getFreeParsInCspp
    ((fst (snd (snd x))) p)  getFreeParsInProcName p)
    (getFreeParsInCspp (snd (snd(snd(snd y))))) Un(getFreeParsInBool
    (fst (snd (snd (snd y))))) (set (fst y))))"
apply (erule oneStepEval.induct)
apply clarsimp
apply clarsimp
apply clarsimp
apply (subgoal_tac "set varSet insert (getNewVar varSet) (insert
    (getNewVar (getNewVar varSet # varSet)) (set varSet))")
apply (frule_tac x="getFreeParsInCspp P" and varSet="set varSet" and
    varSet'="insert (getNewVar varSet) (insert (getNewVar (getNewVar
    varSet # varSet)) (set varSet))" in setTrans)
apply assumption
apply simp
apply (simp add:varsAnd varsEqual varsIdP)
apply blast
apply blast
apply clarsimp+
apply (frule setGrowing)
apply clarsimp
apply blast
apply clarsimp
apply (frule setGrowing)
apply clarsimp
```

```
apply blast
apply clarsimp+
apply (frule setGrowing)
apply clarsimp
apply blast
apply clarsimp+
apply (frule setGrowing)
apply clarsimp
apply blast
apply clarsimp
apply (simp add:varsAnd)
apply (case_tac "x   getFreeParsInBool b")
apply clarsimp
apply blast
apply clarsimp
apply blast
apply clarsimp
apply (simp add:varsAnd varsWf varsNot)
apply (case_tac "x   getFreeParsInBool b")
apply clarsimp
apply blast
apply clarsimp
apply blast
apply clarsimp
apply (cut_tac P="P" and s="s" and varSet="set varSet" and y="getNewVar
    (getNewVar varSet # varSet)" in freeVarSubsWithPar)
apply clarsimp
apply (rule conjI)
apply blast
apply clarsimp
apply (simp add:varsAnd varsEqual varsIdP)
apply (case_tac "x   getFreeParsInAct c")
apply clarsimp
apply blast
apply (case_tac "x   getFreeParsInBool varEnv")
apply clarsimp
apply blast
apply clarsimp
apply (cut_tac s="s" and y="getNewVar (getNewVar varSet # varSet)" and b
    = "apropt" in getFreeParsInBoolFeature)
apply (erule_tac x="set varSet" in allE)
apply clarsimp
apply blast
apply clarsimp
apply (cut_tac P="P" and s="s" and varSet="set varSet" and y="getNewVar
    varSet" in freeVarSubsWithPar)
apply clarsimp
```

```
apply (rule conjI)
apply blast
apply clarsimp
apply (simp add:varsAnd varsEqual varsIdP)
apply (case_tac "x   getFreeParsInBool varEnv")
apply clarsimp
apply blast
apply clarsimp
apply (cut_tac s="s" and y="(getNewVar varSet)" and b = "apropt" in
    getFreeParsInBoolFeature)
apply (erule_tac x="set varSet" in allE)
apply clarsimp
apply blast
apply clarsimp
apply (frule setGrowing)
apply clarsimp
apply blast
apply clarsimp
apply (frule setGrowing)
apply clarsimp
apply blast
apply clarsimp
apply (frule setGrowing)
apply clarsimp
apply blast
apply clarsimp
apply (frule setGrowing)
apply clarsimp
apply blast
apply clarsimp
apply (frule setGrowing)
apply clarsimp
apply (simp add:varsAnd varsNot)
apply (frule_tac x="(varSet, replaceVarInProcName, procEnv, varEnv, P)"
    and a="(Action (U, V))" and y="(varSet', replaceVarInProcName',
    procEnv', varEnv', P')" and c="U" and acts="V" in
    labelVarsBelonging)
apply clarsimp+
apply (frule_tac x="(varSet, replaceVarInProcName, procEnv, varEnv, P)"
    and a="(Action (U, V))" and y="(varSet', replaceVarInProcName',
    procEnv', varEnv', P')" and c="U" and acts="V" in
    labelChanBelonging)
apply clarsimp+
apply (rule conjI)
apply blast
apply (rule conjI)
apply blast
```

```
apply (cut_tac b="X" and s="s" and s'="s'" and U="U" and V="V" in
    getFreeParsInDoubleBoolFeature)
apply (erule_tac x="set varSet" in allE)
apply (erule_tac x="set varSet'" in allE)
apply clarsimp
apply clarsimp
apply (frule setGrowing)
apply clarsimp
apply (simp add:varsAnd varsNot)
apply (frule_tac x="(varSet, replaceVarInProcName, procEnv, varEnv, Q)"
    and a="(Action (U, V))" and y="(varSet', replaceVarInProcName',
    procEnv', varEnv', Q')" and c="U" and acts="V" in
    labelVarsBelonging)
apply clarsimp+
apply (frule_tac x="(varSet, replaceVarInProcName, procEnv, varEnv, Q)"
    and a="(Action (U, V))" and y="(varSet', replaceVarInProcName',
    procEnv', varEnv', Q')" and c="U" and acts="V" in
    labelChanBelonging)
apply clarsimp+
apply (rule conjI)
apply blast
apply (rule conjI)
apply blast
apply (cut_tac b="X" and s="s" and s'="s'" and U="U" and V="V" in
    getFreeParsInDoubleBoolFeature)
apply (erule_tac x="set varSet" in allE)
apply (erule_tac x="set varSet'" in allE)
apply clarsimp
apply clarsimp
apply (frule setGrowing)
apply clarsimp
apply (frule_tac x= "(varSet', replaceVarInProcName', procEnv', varEnv',
    Q)" and a= "(Action (U', V'))" and y="(varSet'',
    replaceVarInProcName'', procEnv'', varEnv'', Q')" in setGrowing)
apply clarsimp
apply (simp add:varsAnd varsEqual)
apply (frule_tac x="getFreeParsInCspp Q" and varSet="set varSet" and
    varSet'= "set varSet'" in setTrans)
apply clarsimp
apply (frule_tac x="(varSet, replaceVarInProcName, procEnv, varEnv, P)"
    and a="(Action (U, V))" and y="(varSet', replaceVarInProcName',
    procEnv', varEnv', P')" and c="U" and acts="V" in
    labelVarsBelonging)
apply clarsimp+
apply (frule_tac x="(varSet, replaceVarInProcName, procEnv, varEnv, P)"
    and a="(Action (U, V))" and y="(varSet', replaceVarInProcName',
    procEnv', varEnv', P')" and c="U" and acts="V" in
```

```
    labelChanBelonging)
apply clarsimp+
apply (frule_tac x="(varSet', replaceVarInProcName', procEnv', varEnv',
    Q)" and a="(Action (U', V'))" and y="(varSet'',
    replaceVarInProcName'', procEnv'', varEnv'', Q')" and c="U'" and
    acts="V'" in labelVarsBelonging)
apply clarsimp+
apply (frule_tac x="(varSet', replaceVarInProcName', procEnv', varEnv',
    Q)" and a="(Action (U', V'))" and y="(varSet'',
    replaceVarInProcName'', procEnv'', varEnv'', Q')" and c="U'" and
    acts="V'" in labelChanBelonging)
apply clarsimp+
apply (frule fix_procEnv)
apply clarsimp
apply (rule conjI)
apply blast
apply (rule conjI)
apply blast
apply (rule conjI)
apply (simp add:varsIdP)
apply force
apply (rule conjI)
apply (simp add:varsIdP)
apply (cut_tac b="X" and s="s" and s'="s'" and U="U" and V="V" in
    getFreeParsInDoubleBoolFeature)
apply (erule_tac x="set varSet" in allE)
apply (erule_tac x="set varSet'" in allE)
apply clarsimp
apply (frule_tac x="getFreeParsInBool (replaceVarInDoubleBool X s (IdP
    U) s' (IdP V))" and varSet="set varSet'" and varSet'= "set
    varSet''" in setTrans)
apply clarsimp
apply clarsimp
apply (simp add:varsIdP)
apply blast
apply clarsimp+
apply (frule setGrowing)
apply clarsimp
apply blast
apply clarsimp
apply (frule setGrowing)
apply clarsimp
apply (simp add:varsAnd varsNot)
apply (frule_tac x="(varSet, replaceVarInProcName, procEnv, varEnv, P)"
    and a="(Action (U, V))" and y="(varSet', replaceVarInProcName',
    procEnv', varEnv', P')" and c="U" and acts="V" in
    labelVarsBelonging)
```

```
apply clarsimp+
apply (frule_tac x="(varSet, replaceVarInProcName, procEnv, varEnv, P)"
    and a="(Action (U, V))" and y="(varSet', replaceVarInProcName',
    procEnv', varEnv', P')" and c="U" and acts="V" in
    labelChanBelonging)
apply clarsimp+
apply (rule conjI)
apply blast
apply (cut_tac b="X" and s="s" and s'="s'" and U="U" and V="V" in
    getFreeParsInDoubleBoolFeature)
apply (erule_tac x="set varSet" in allE)
apply (erule_tac x="set varSet'" in allE)
apply clarsimp
apply clarsimp
apply (frule setGrowing)
apply clarsimp
apply (simp add:varsAnd)
apply (frule_tac x="(varSet, replaceVarInProcName, procEnv, varEnv, P)"
    and a="(Action (U, V))" and y="(varSet', replaceVarInProcName',
    procEnv', varEnv', P')" and c="U" and acts="V" in
    labelVarsBelonging)
apply clarsimp+
apply (frule_tac x="(varSet, replaceVarInProcName, procEnv, varEnv, P)"
    and a="(Action (U, V))" and y="(varSet', replaceVarInProcName',
    procEnv', varEnv', P')" and c="U" and acts="V" in
    labelChanBelonging)
apply clarsimp+
apply (rule conjI)
apply blast
apply (cut_tac b="X" and s="s" and s'="s'" and U="U" and V="V" in
    getFreeParsInDoubleBoolFeature)
apply (erule_tac x="set varSet" in allE)
apply (erule_tac x="set varSet'" in allE)
apply clarsimp
apply clarsimp
apply blast
done


end
```

```
theory cspp_soundness
imports cspp_data cspp_syntax cspp_semantic lang cspp_noillformness lang
    Main
begin
```

```
(*
This two lemmas suggest that the label semantics function will preserve
    the structure of the label
*)
lemma (in procNameLocale) labelSameTau: "aTau   (labelSem   a)    Tau"
apply (induct a)
apply simp
apply simp
apply (case_tac a)
apply clarsimp
done


lemma (in procNameLocale) labelSameCheck: "aCheck   (labelSem   a) Check "
apply (induct a)
apply simp
apply simp
apply (case_tac a)
apply clarsimp
done



(*
These two lemmas is a extra step for the proof in the if-else statement
    in soundness to use auto in Isabelle.
*)
lemma  (in procNameLocale) ifTrueTrans: "trans (IF True THEN P ELSE Q)
    Tau (if True then P else Q) trans (IF True THEN P ELSE Q) Tau P"
apply auto
done

lemma (in procNameLocale) ifFalseTrans: "trans (IF False THEN P ELSE Q)
    Tau (if False then P else Q) trans (IF False THEN P ELSE Q) Tau Q"
apply auto
done


(*
These four lemma suggests that if some assignment functions  models (not
    models) the proposition B[U/x], and if U will be evaluated to
some value n, then n will (will not) in the set formed by the   B[n/x].
    They are using in different context, some will take one variables,
```

```
and is used in the External and Internal pre choice, some will take two
    variables and is used in the par_out and par_in rule.
*)
lemma (in procNameLocale) singleParInSetChan: "model  (replaceVarInBool
    apropt s (IdP z))  (  y,  z)    ({  y}   {a. model
    (replaceVarInBool apropt s (ActVal a))})"
apply clarsimp
apply (simp add:evalBoolPro)
apply (simp add:actValProInBool)
done


lemma (in procNameLocale) singleParInSetNoChan: "model
    (replaceVarInBool b s (IdP y))  ( c . (c,  y)    ({(c,a). model
    (replaceVarInBool b s (ActVal a))}))"
apply clarsimp
apply (simp add:evalBoolPro)
apply (simp add:actValProInBool)
done


lemma (in procNameLocale) doubleParsNotInSet : "  (model
    (replaceVarInDoubleBool X s (IdP u) s' (IdP v)))
        (  u,  v)   {(cv, av).
              model
               (replaceVarInDoubleBool X s (ActVal cv) s'
                 (ActVal av))}"
apply clarsimp
apply (simp add:evalBoolPro)
apply (cut_tac s="s" and s'="s'" and x="u" and y="v" and  = "" and b="X"
    in actValProInDoubleBool)
apply clarsimp
done



lemma (in procNameLocale) doubleParsInSet : "model
    (replaceVarInDoubleBool X s (IdP u) s' (IdP v))
         (  u,  v)   {(cv, av).
              model
               (replaceVarInDoubleBool X s (ActVal cv) s'
                 (ActVal av))}"
apply clarsimp
apply (simp add:evalBoolPro)
apply (cut_tac s="s" and s'="s'" and x="u" and y="v" and  = "" and b="X"
    in actValProInDoubleBool)
apply clarsimp
```

```
done



(*
these two lemma are helper lemmas in the later lemma, it suggests the
    fact that P[n/x][m/y]=P[m/y][n/x] or P[U/x][V/y]=P[V/y][U/x] if x
    y.
It means that the substitution function will be evaluated to the same no
    matter the evaluation order
*)
lemma (in procNameLocale) subsNoOrderValue : "svarreplaceVarInCspp
    replaceVarInProcName (replaceVarInCspp replaceVarInProcName P s
    (ActVal n)) var (ActVal b)=replaceVarInCspp replaceVarInProcName
    (replaceVarInCspp replaceVarInProcName P var (ActVal b)) s (ActVal
    n)"
apply (induct P)
apply clarsimp+
apply (simp add:procNameSubst1)
apply auto
apply (simp add:singleSubActInValue)+
apply (simp add:singleSubProInValue)+
apply (simp add:singleSubActInValue)+
apply (simp add:singleSubProInValue)+
done

lemma (in procNameLocale) subsNoOrderId : "svarreplaceVarInCspp
    replaceVarInProcName (replaceVarInCspp replaceVarInProcName P s
    (IdP y)) var (ActVal n)=replaceVarInCspp replaceVarInProcName
    (replaceVarInCspp replaceVarInProcName P var (ActVal n)) s (IdP y)"
apply (induct P)
apply clarsimp+
apply (simp add:procNameSubst2)
apply auto
apply (simp add:singleSubActInId)+
apply (simp add:singleSubProInId)+
apply (simp add:singleSubActInId)+
apply (simp add:singleSubProInId)+
done


(*
These two lemmas suggest that it will have the same meaning if
    sem(,P[U/x])=sem(,P[n/x]),if  U = n. The lemma semSubSame_aux is
the helper version of the lemma semSubSame, and it is proved by
    induction on nat.
```

```
*)
lemma (in procNameLocale) semSubSame_aux: "P. (counter P =m
  (csppSem replaceVarInProcName
          (replaceVarInCspp replaceVarInProcName P s (ActVal ( y))))=
          (csppSem replaceVarInProcName
            (replaceVarInCspp replaceVarInProcName P s (IdP y)))) "
apply (induct m rule: nat_less_induct)
apply (case_tac n)
apply clarsimp
apply (case_tac P)
apply clarsimp+
apply (cut_tac s="s" and P="pname" in procNameSemProperty)
apply (erule_tac x="" in allE)
apply (erule_tac x="y" in allE)
apply clarsimp+
apply (case_tac P,clarsimp+)
apply (erule_tac x="counter cspp" in allE)
apply clarsimp
apply (erule_tac x="cspp" in allE)
apply clarsimp
apply (case_tac "evalInAct  (replaceVarInAct act1 s (ActVal ( y)))")
apply (simp add:actValProInAct)
apply clarsimp
apply (case_tac "evalInAct  (replaceVarInAct act1 s (IdP y))")
apply (simp add:actValProInAct)
apply (simp add:actValProInAct)
apply (case_tac "evalInAct  (replaceVarInAct act2 s (ActVal ( y)))")
apply (simp add:actValProInAct)
apply (simp add:actValProInAct)
apply clarsimp
apply (frule_tac x = "counter cspp1" in spec)
apply (erule_tac x = "counter cspp2" in allE)
apply clarsimp+
apply (frule_tac x = "counter cspp1" in spec)
apply (erule_tac x = "counter cspp2" in allE)
apply clarsimp+
apply (frule_tac x = "counter cspp1" in spec)
apply (erule_tac x = "counter cspp2" in allE)
apply clarsimp
apply (erule_tac x="cspp1" in allE)
apply (erule_tac x="cspp2" in allE)
apply clarsimp
apply (case_tac " evalInBool  (replaceVarInBool bool s (ActVal ( y)))")
apply (simp add:actValProInBool)
apply (case_tac "evalInBool  (replaceVarInBool bool s (IdP y))")
apply (simp add:actValProInBool)
apply (simp add:actValProInBool)
```

```
apply clarsimp
apply (case_tac "var1=s")
apply clarsimp
apply (case_tac "var2=s")
apply clarsimp
apply (frule_tac x = "counter cspp1" in spec)
apply (erule_tac x = "counter cspp2" in allE)
apply clarsimp
apply clarsimp
apply (frule_tac x = "counter cspp1" in spec)
apply (erule_tac x = "counter cspp2" in allE)
apply clarsimp
apply clarsimp
apply (case_tac "var2=s")
apply clarsimp
apply (frule_tac x = "counter cspp1" in spec)
apply (erule_tac x = "counter cspp2" in allE)
apply clarsimp+
apply (frule_tac x = "counter cspp1" in spec)
apply (erule_tac x = "counter cspp2" in allE)
apply clarsimp
apply (rule set_eqI)
apply clarsimp
apply (simp add:evalBoolPro)
apply (subgoal_tac " (replaceVarInDoubleBool (replaceVarInBool bool s
    (ActVal ( y))) var1
           (ActVal a) var2 (ActVal b))=(replaceVarInBool
               (replaceVarInDoubleBool bool var1 (ActVal a) var2
               (ActVal b)) s (ActVal ( y)))")
apply (subgoal_tac " (replaceVarInDoubleBool (replaceVarInBool bool s
    (IdP y)) var1
           (ActVal a) var2 (ActVal b))=(replaceVarInBool
               (replaceVarInDoubleBool bool var1 (ActVal a) var2
               (ActVal b)) s (IdP y))")
apply clarsimp
apply (simp add:actValProInBool)
apply (simp add:twoSubProInId)
apply (simp add:twoSubProInValue)
apply clarsimp
apply (frule_tac x = "counter cspp1" in spec)
apply (erule_tac x = "counter cspp2" in allE)
apply clarsimp+
apply (case_tac "s=var")
apply clarsimp
apply (rule set_eqI)
apply clarsimp
apply (simp add:evalBoolPro)
```

```
apply (simp add:actValProInAct)
apply clarsimp
apply (rule conjI)
apply (rule set_eqI)
apply clarsimp
apply (simp add:evalBoolPro)
apply (simp add:actValProInAct)
apply (simp add: singleSubProInValue singleSubProInId)
apply (simp add:actValProInBool)
apply (rule ext)
apply clarsimp
apply (simp add:subsNoOrderValue subsNoOrderId)
apply (erule_tac x="counter (replaceVarInCspp replaceVarInProcName cspp
    var (ActVal b))" in allE)
apply (simp add:termination_aux1)
apply clarsimp
apply (rule conjI)
apply (rule set_eqI)
apply clarsimp
apply (simp add:evalBoolPro)
apply (simp add:singleSubProInValue singleSubProInId)
apply (simp add:actValProInBool)
apply clarsimp
apply (rule ext)
apply clarsimp
apply (simp add:subsNoOrderValue subsNoOrderId)
apply (erule_tac x="counter (replaceVarInCspp replaceVarInProcName cspp
    var (ActVal b))" in allE)
apply (simp add:termination_aux1)
apply clarsimp
apply (case_tac "var1=s")
apply clarsimp
apply (case_tac "var2=s")
apply clarsimp
apply (erule_tac x = "counter cspp" in allE)
apply clarsimp
apply clarsimp
apply (erule_tac x = "counter cspp" in allE)
apply clarsimp
apply clarsimp
apply (case_tac "var2=s")
apply clarsimp
apply (erule_tac x = "counter cspp" in allE)
apply clarsimp+
apply (erule_tac x = "counter cspp" in allE)
apply clarsimp
apply (rule set_eqI)
```

```
apply clarsimp
apply (simp add:evalBoolPro)
apply (simp add: twoSubProInId twoSubProInValue)
apply (simp add: actValProInBool)
done


lemma (in procNameLocale) semSubSame: "(csppSem replaceVarInProcName
          (replaceVarInCspp replaceVarInProcName P s (ActVal ( y))))=
          (csppSem replaceVarInProcName
            (replaceVarInCspp replaceVarInProcName P s (IdP y)))"
apply (cut_tac m="counter P" and  ="" and y="y" and s="s" in
    semSubSame_aux)
apply (erule_tac x="P" in allE)
apply clarsimp
done



(*
This lemma suggests that if a     the post-condition in any HCSP
    semantics rules, it will also  the pre-condition,
in other words, the condition will always grow
*)
lemma (in procNameLocale) conditionGrow : "oneStepEval x a y  (  .(model
      (fst (snd (snd (snd y))))  model   (fst (snd (snd (snd x))))))"
apply (erule oneStepEval.induct)
apply clarsimp+
apply (simp add:MAnd MEqual)
apply clarsimp+
apply (simp add:MAnd)
apply clarsimp
apply (simp add:MAnd)
apply clarsimp
apply (simp add:MAnd)
apply clarsimp
apply (simp add:MAnd)
apply clarsimp+
apply (simp add:MAnd)
apply clarsimp
apply (simp add:MAnd)
apply clarsimp
apply (simp add:MAnd)
apply clarsimp+
apply (erule_tac x="" in allE)
apply (simp add:MAnd)
apply clarsimp
```

```
apply (erule_tac x="" in allE)
apply (simp add:MAnd)
apply clarsimp
done


(*
this is the soundness theorem, it has the following structure:

  oneStepEvalSoundness:
     oneStepEval  (?varSet, replaceVarInProcName, procEnv, ?varEnv, ?P)
        ?a
      (?varSet', ?replaceVarInProcName', ?procEnv', ?varEnv', ?P');
     getFreeParsInCspp ?P   set ?varSet
     getFreeParsInBool ?varEnv  set ?varSet
     ( p  . getFreeParsInCspp (procEnv p)  getFreeParsInProcName p)
     model ?  ?varEnv    model ?  ?varEnv'
        TransSem.trans
        ( p . csppSem replaceVarInProcName (x. Null) (procEnv p))
        (csppSem replaceVarInProcName ? ?P) (labelSem ? ?a)
        (csppSem ?replaceVarInProcName' ? ?P')
*)
lemma (in procNameLocale) soundness: " oneStepEval (varSet,
    replaceVarInProcName,procEnv, varEnv, P) a
    (varSet',replaceVarInProcName',procEnv', varEnv', P') (. (
 getFreeParsInCspp (snd (snd (snd (snd
     (varSet,replaceVarInProcName,procEnv, varEnv, P)))))
   getFreeParsInBool (fst (snd (snd (snd
        (varSet,replaceVarInProcName,procEnv, varEnv, P)))))
       (set (fst (varSet,replaceVarInProcName,procEnv, varEnv, P)))
 & ( p   . getFreeParsInCspp ((fst (snd (snd (varSet,
     replaceVarInProcName,procEnv, varEnv, P)))) p)
     getFreeParsInProcName p)
 &
model   (fst (snd (snd (snd (varSet,replaceVarInProcName,procEnv,
    varEnv, P)))))& ((*'.*) ( (model  (fst (snd (snd (snd
    (varSet',replaceVarInProcName',procEnv', varEnv', P')))))))
   ( x  . ((trans (csppSem (fst (snd (varSet,
    replaceVarInProcName,procEnv, varEnv, P)))  (snd (snd (snd (snd
    (varSet,replaceVarInProcName,procEnv, varEnv, P))))))(labelSem  a)
    (csppSem (fst (snd ((varSet',replaceVarInProcName',procEnv',
    varEnv', P'))))  (snd (snd (snd (snd
    (varSet',replaceVarInProcName',procEnv', varEnv', P')))))))))))"
apply (erule oneStepEval.induct)
apply simp_all
thm skip
apply clarsimp
```

```
apply (rule skip)
thm div
apply clarsimp
apply (rule div)
thm act_prefix
apply clarsimp
apply (simp add:MAnd MEqual evalPro)
apply (case_tac "evalInAct  c")
apply clarsimp
apply (case_tac "evalInAct  a")
apply clarsimp+
apply (simp add:act_prefix)
thm int_choice1
apply clarsimp
apply (rule int_choice1)
thm int_choice2
apply clarsimp
apply (rule int_choice2)
thm ext_choice_tau1
apply clarsimp
apply (erule_tac x = "" in allE)
apply clarsimp
apply (frule_tac x="(varSet, replaceVarInProcName, procEnv, varEnv, P)"
    in fix_replace)
apply clarsimp
apply (rule ext_choice_tau1)
apply assumption
thm ext_choice_tau2
apply clarsimp
apply (erule_tac x = "" in allE)
apply clarsimp
apply (frule_tac x="(varSet, replaceVarInProcName, procEnv, varEnv, Q)"
    in fix_replace)
apply clarsimp
apply (simp add:ext_choice_tau2)
thm ext_choice1
apply clarsimp
apply (erule_tac x = "" in allE)
apply clarsimp
apply (frule_tac x="(varSet, replaceVarInProcName, procEnv, varEnv, P)"
    in fix_replace)
apply clarsimp
apply (frule_tac a="a" and  = " " in labelSameTau)
apply (simp add:ext_choice1)
thm ext_choice2
apply clarsimp
apply (erule_tac x = "" in allE)
```

```
apply clarsimp
apply (frule_tac x="(varSet, replaceVarInProcName, procEnv, varEnv, Q)"
    in fix_replace)
apply clarsimp
apply (frule_tac a="a" and  = " " in labelSameTau)
apply (simp add:ext_choice2)
thm seq_step
apply clarsimp
apply (erule_tac x = "" in allE)
apply clarsimp
apply (frule_tac x="(varSet, replaceVarInProcName, procEnv, varEnv, P)"
    in fix_replace)
apply clarsimp
apply (frule_tac a="x" and  = " " in labelSameCheck)
apply (simp add:seq_step)
thm seq_check
apply clarsimp
apply (frule_tac x="(varSet, replaceVarInProcName, procEnv, varEnv, P)"
    in fix_replace)
apply clarsimp
apply (rule seq_check)
apply (rule_tac x="csppSem replaceVarInProcName  P'" in exI)
apply clarsimp
thm ifthenelse
apply clarsimp
apply (simp add:MAnd)
apply (case_tac "evalInBool  b")
apply (simp add:evalBoolPro)
apply clarsimp
apply (simp add:evalBoolPro)
apply (rule_tac P="csppSem replaceVarInProcName  P" and Q="csppSem
    replaceVarInProcName  Q" in ifTrueTrans)
apply (rule_tac b="True"in ifthenelse)
apply clarsimp
apply (simp add:MAnd MWf MNot)
apply (case_tac "evalInBool  b")
apply clarsimp
apply clarsimp
apply (simp add:evalBoolPro)
apply (rule_tac P="csppSem replaceVarInProcName  P" and Q="csppSem
    replaceVarInProcName  Q" in ifFalseTrans)
apply (rule_tac b="False" in ifthenelse)
thm set_prefix
apply clarsimp
apply (simp add:MAnd MEqual evalPro)
apply (case_tac "evalInAct  c")
apply clarsimp
```

```
apply clarsimp
apply (frule_tac  =" " and apropt="apropt" and y="getNewVar varSet" and
    z="getNewVar ((getNewVar varSet) # varSet)" in singleParInSetChan)
apply (frule_tac P =
"(  (c, n).
                 csppSem replaceVarInProcName
                  (replaceVarInCspp replaceVarInProcName P s (ActVal
                      n)))" in set_prefix)
apply clarsimp
apply (cut_tac s="s"and  =" " and P="P" and y="getNewVar ((getNewVar
    varSet) # varSet)" in semSubSame)
apply clarsimp
thm rep_prefix
apply clarsimp
apply (simp add:MAnd MEqual evalPro)
apply (frule_tac  =" " and b="apropt" and y="getNewVar varSet" in
    singleParInSetNoChan)
apply (cut_tac a="(c::('value), (getNewVar varSet))" and X="{(c', a).
    model  (replaceVarInBool apropt s (ActVal a))}" and P =
"(  (c, n).
                 csppSem replaceVarInProcName
                  (replaceVarInCspp replaceVarInProcName P s (ActVal
                      n)))" in rep_prefix)
apply clarsimp
apply clarsimp
apply (cut_tac s="s"and  =" " and P="P" and y="getNewVar varSet" in
    semSubSame)
apply clarsimp
thm par_check1
apply clarsimp
apply (erule_tac x = "" in allE)
apply clarsimp
apply (frule_tac x="(varSet, replaceVarInProcName, procEnv, varEnv, P)"
    in fix_replace)
apply clarsimp
apply (simp add:par_check1)
thm par_check2
apply clarsimp
apply (erule_tac x = "" in allE)
apply clarsimp
apply (frule_tac x="(varSet, replaceVarInProcName, procEnv, varEnv, Q)"
    in fix_replace)
apply clarsimp
apply (simp add:par_check2)
thm par_omega
apply clarsimp
apply (simp add:par_omega)
```

```
thm par_tau1
apply clarsimp
apply (erule_tac x = "" in allE)
apply clarsimp
apply (frule_tac x="(varSet, replaceVarInProcName, procEnv, varEnv, P)"
    in fix_replace)
apply clarsimp
apply (simp add:par_tau1)
thm par_tau2
apply clarsimp
apply (erule_tac x = "" in allE)
apply clarsimp
apply (frule_tac x="(varSet, replaceVarInProcName, procEnv, varEnv, Q)"
    in fix_replace)
apply clarsimp
apply (simp add:par_tau2)
thm par_out1
apply clarsimp
apply (erule_tac x = "" in allE)
apply clarsimp
apply (simp add:MAnd MNot)
apply clarsimp
apply (cut_tac s="s" and s'="s'" and X="X" and  = "" and u="U" and v="V"
    in doubleParsNotInSet)
apply assumption
apply (frule_tac x="(varSet, replaceVarInProcName, procEnv, varEnv, P)"
    in fix_replace)
apply clarsimp
apply (simp add:par_out1)
thm par_out2
apply clarsimp
apply (simp add:MAnd MNot)
apply (erule_tac x = "" in allE)
apply clarsimp
apply (cut_tac s="s" and s'="s'" and X="X" and  = "" and u="U" and v="V"
    in doubleParsNotInSet)
apply assumption
apply (frule_tac x="(varSet, replaceVarInProcName, procEnv, varEnv, Q)"
    in fix_replace)
apply clarsimp
apply (simp add:par_out2)
thm par_in
apply clarsimp
apply (simp add:MAnd MEqual)
apply (frule_tac x="(varSet',replaceVarInProcName',procEnv', varEnv',
    Q)" and a =" (Action (U', V'))" and
    y="(varSet'',replaceVarInProcName'',procEnv'', varEnv'', Q')" in
```

```
    conditionGrow)
apply clarsimp
apply (frule fix_replace)
apply (frule fix_procEnv)
apply (simp add:evalPro)
apply clarsimp
apply (frule_tac x="(varSet,replaceVarInProcName,procEnv, varEnv, P)"
    and a =" (Action (U, V))" and
    y="(varSet',replaceVarInProcName,procEnv, varEnv', P')" in
    setGrowing)
apply clarsimp
apply (frule_tac x="(varSet,replaceVarInProcName,procEnv, varEnv, P)"
    and y="(varSet',replaceVarInProcName,procEnv, varEnv', P')" in
    noIllFormedNess)
apply (erule_tac x = "" in allE)
apply clarsimp
apply (frule_tac x="getFreeParsInCspp Q" and varSet="set varSet" and
    varSet'="set varSet'" in setTrans)
apply assumption
apply clarsimp
apply (frule_tac s="s" and s'="s'" and X="X" and  ="" and u="U" and
    v="V" in doubleParsInSet)
apply (frule_tac x="(varSet', replaceVarInProcName, procEnv, varEnv',
    Q)" in fix_replace)
apply (rule par_in)
apply clarsimp
apply clarsimp
apply clarsimp
thm hid_omega
apply clarsimp
apply (erule_tac x="" in allE)
apply clarsimp
apply (simp add:hid_omega)
thm hid_tau
apply clarsimp
apply (erule_tac x="" in allE)
apply clarsimp
apply (simp add:hid_tau)
thm hid_neg
apply clarsimp
apply (erule_tac x="" in allE)
apply clarsimp
apply (simp add:MAnd MNot)
apply clarsimp
apply (frule_tac  ="  " and u="U" and v="V" and s="s" and X="X" and
    s'="s'" in doubleParsNotInSet)
apply (simp add:hid_neg)
```

```
thm hid_pos
apply clarsimp
apply (erule_tac x="" in allE)
apply clarsimp
apply (simp add:MAnd)
apply clarsimp
apply (frule_tac  ="  " and u="U" and v="V" and s="s" and X="X" and
    s'="s'" in doubleParsInSet)
apply (rule_tac a="( U,  V)" and P="(csppSem replaceVarInProcName  P)"
    and Q="(csppSem replaceVarInProcName'  P')" and X="{(cv, av).
              model
               (replaceVarInDoubleBool X s (ActVal cv) s' (ActVal
                   av))}" in hid_pos)
apply assumption
apply assumption
thm proc_name_rule
apply clarsimp
apply (cut_tac p= "procNameSem  p" in proc_name_rule)
apply (cut_tac procNameSemFeature)
apply (erule_tac x="" in allE)
apply (erule_tac x="p" in allE)
apply (erule_tac x="p" in allE)
apply clarsimp
done


lemmas (in procNameLocale) oneStepEvalSoundness =
    soundness[simplified,rule_format]

end
```

```
theory cspp_completeness
imports cspp_data lang cspp_syntax cspp_semantic cspp_noillformness
    cspp_soundness Main
begin


(*
a lemma suggests that if freeVars(P)  {x}UA, then freeVars(P[n/x])A
*)
lemma (in procNameLocale) freeVarsSub [rule_format]: " varSet .
    (getFreeParsInCspp P  varSet  getFreeParsInCspp (replaceVarInCspp
    replaceVarInProcName P s (ActVal b))  varSet)"
apply (induct P)
apply clarsimp+
```

```
apply (cut_tac av="b" and s="s" and pname="pname" in
    getFreeParsInProcNameFeatureX)
apply (erule_tac x="varSet" in allE)
apply blast
apply clarsimp
apply (cut_tac av="b" and s="s" and a="act1" in getFreeParsInActFeatureX)
apply (cut_tac av="b" and s="s" and a="act2" in getFreeParsInActFeatureX)
apply (erule_tac x="varSet" in allE)
apply (erule_tac x="varSet" in allE)
apply clarsimp+
apply (cut_tac av="b" and s="s" and b="bool" in
    getFreeParsInBoolFeatureX)
apply (erule_tac x="varSet" in allE)
apply blast
apply clarsimp
apply (case_tac "var1=s")
apply clarsimp
apply (cut_tac av="b" and s="s" and b="bool" in
    getFreeParsInBoolFeatureX)
apply (erule_tac x="varSet" in allE)
apply (erule_tac x="varSet" in allE)
apply (erule_tac x="varSet" in allE)
apply clarsimp
apply blast
apply clarsimp+
apply (case_tac "s=var")
apply clarsimp
apply (cut_tac a="act" and av="b" and s="s" in getFreeParsInActFeatureX)
apply (erule_tac x="varSet" in allE)
apply (erule_tac x="varSet" in allE)
apply clarsimp
apply blast
apply clarsimp
apply (rule conjI)
apply (cut_tac a="act" and av="b" and s="s" in getFreeParsInActFeatureX)
apply (erule_tac x="varSet" in allE)
apply (erule_tac x="varSet" in allE)
apply clarsimp
apply (cut_tac b="bool" and av="b" and s="s" in
    getFreeParsInBoolFeatureX)
apply (erule_tac x="varSet" in allE)
apply (erule_tac x="varSet" in allE)
apply clarsimp
apply clarsimp
apply (cut_tac b="bool" and av="b" and s="s" in
    getFreeParsInBoolFeatureX)
apply (erule_tac x="varSet" in allE)
```

```
apply (erule_tac x="varSet" in allE)
apply clarsimp
apply blast
apply clarsimp
apply (cut_tac b="bool" and s="s" and av="b" in
    getFreeParsInBoolFeatureX)
apply (erule_tac x="(varSet)" in allE)
apply (erule_tac x="(varSet)" in allE)
apply clarsimp
apply blast
done


(*
These two lemma suggest that if ' |dom() =  , then P will be
    interpreted the same in both  and ' if freeVars(P)  A, and dom()
    A.
semSameInGrownRho_aux is the helper version of the semSameInGrownRho
    lemma, it is existed because we need to have a special induction on
    the
semanatics function of HCSP process.
*)
lemma (in procNameLocale) semSameInGrownRho_aux: "P. (counter P = m
  & getFreeParsInCspp P  varSet & ( x . x    varSet      x =  ' x)
      csppSem replaceVarInProcName  P = csppSem replaceVarInProcName '
          P)"
apply (induct m rule: nat_less_induct)
apply (case_tac n)
apply clarsimp
apply (case_tac P)
apply clarsimp+
apply (cut_tac pname="pname" and varSet="varSet" in getFreeParsProperty)
apply (erule_tac x="" in allE)
apply (erule_tac x="'" in allE)
apply clarsimp+
apply (case_tac P, clarsimp+)
apply (erule_tac x="counter cspp" in allE)
apply clarsimp
apply (erule_tac x="cspp" in allE)
apply clarsimp
apply (cut_tac varSet="varSet" and a="act1" in evalProOfSameEnv)
apply (cut_tac varSet="varSet" and a="act2" in evalProOfSameEnv)
apply clarsimp
apply (erule_tac x="" in allE)
apply (erule_tac x="" in allE)
apply (erule_tac x="'" in allE)
apply (erule_tac x="'" in allE)
```

```
apply clarsimp
apply (case_tac "evalInAct ' act1")
apply clarsimp
apply clarsimp
apply (case_tac "evalInAct ' act2")
apply clarsimp+
apply (frule_tac x = "counter cspp1" in spec)
apply (erule_tac x = "counter cspp2" in allE)
apply clarsimp+
apply (frule_tac x = "counter cspp1" in spec)
apply (erule_tac x = "counter cspp2" in allE)
apply clarsimp+
apply (frule_tac x = "counter cspp1" in spec)
apply (erule_tac x = "counter cspp2" in allE)
apply clarsimp+
apply (erule_tac x="cspp1" in allE)
apply (erule_tac x="cspp2" in allE)
apply clarsimp
apply (cut_tac b="bool" and varSet="varSet" in evalBoolOfSameEnv)
apply clarsimp
apply (erule_tac x="" in allE)
apply (erule_tac x="'" in allE)
apply clarsimp
apply (case_tac "evalInBool ' bool")
apply clarsimp+
apply (frule_tac x = "counter cspp1" in spec)
apply (erule_tac x = "counter cspp2" in allE)
apply clarsimp+
apply (rule_tac set_eqI)
apply clarsimp
apply (simp add:evalBoolPro)
apply (cut_tac b="(replaceVarInDoubleBool bool var1 (ActVal a) var2
    (ActVal b))" and varSet="varSet" in evalBoolOfSameEnv)
apply (erule_tac x="" in allE)
apply (erule_tac x="'" in allE)
apply clarsimp
apply (cut_tac b="bool" and s="var1" and s'="var2" and av="a" and cv="b"
    in getFreeParsInDoubleBoolFeatureX)
apply (erule_tac x="varSet" in allE)
apply clarsimp
apply clarsimp
apply (frule_tac x = "counter cspp1" in spec)
apply (erule_tac x = "counter cspp2" in allE)
apply clarsimp+
apply (rule conjI)
apply (rule set_eqI)
apply clarsimp
```

```
apply (simp add:evalBoolPro)
apply (cut_tac b="replaceVarInBool bool var (ActVal b)" and
    varSet="varSet" in evalBoolOfSameEnv)
apply (cut_tac a="act" and varSet="varSet" in evalProOfSameEnv)
apply (erule_tac x="" in allE)
apply (erule_tac x="" in allE)
apply (erule_tac x="'" in allE)
apply (erule_tac x="'" in allE)
apply clarsimp
apply (cut_tac b="bool" and s="var" and av="b" in
    getFreeParsInBoolFeatureX)
apply (erule_tac x="varSet" in allE)
apply clarsimp
apply (rule ext)
apply clarsimp
apply (erule_tac x="counter (replaceVarInCspp replaceVarInProcName cspp
    var (ActVal b))" in allE)
apply (simp add:termination_aux1)
apply (erule_tac x="(replaceVarInCspp replaceVarInProcName cspp var
    (ActVal b))" in allE)
apply (cut_tac P="cspp" and s="var" and b="b" and varSet="varSet" in
    freeVarsSub)
apply clarsimp
apply blast
apply clarsimp
apply (rule conjI)
apply (rule set_eqI)
apply clarsimp
apply (simp add:evalBoolPro)
apply (cut_tac b="replaceVarInBool bool var (ActVal b)" and
    varSet="varSet" in evalBoolOfSameEnv)
apply (erule_tac x="" in allE)
apply (erule_tac x="'" in allE)
apply clarsimp
apply (cut_tac b="bool" and s="var" and av="b" in
    getFreeParsInBoolFeatureX)
apply (erule_tac x="varSet" in allE)
apply clarsimp
apply (rule ext)
apply clarsimp
apply (erule_tac x="counter (replaceVarInCspp replaceVarInProcName cspp
    var (ActVal b))" in allE)
apply (simp add:termination_aux1)
apply (erule_tac x="(replaceVarInCspp replaceVarInProcName cspp var
    (ActVal b))" in allE)
apply (cut_tac P="cspp" and s="var" and b="b" and varSet="varSet" in
    freeVarsSub)
```

```
apply clarsimp
apply blast
apply clarsimp
apply (erule_tac x = "counter cspp" in allE)
apply clarsimp
apply (rule set_eqI)
apply clarsimp
apply (simp add:evalBoolPro)
apply (cut_tac b="(replaceVarInDoubleBool bool var1 (ActVal a) var2
    (ActVal b))" and varSet="varSet" in evalBoolOfSameEnv)
apply (erule_tac x="" in allE)
apply (erule_tac x="'" in allE)
apply clarsimp
apply (cut_tac b="bool" and s="var1" and s'="var2" and av="a" and cv="b"
    in getFreeParsInDoubleBoolFeatureX)
apply (erule_tac x="varSet" in allE)
apply clarsimp
done

lemma (in procNameLocale) semSameInGrownRho: "getFreeParsInCspp P
    varSet; x. x    varSet      x =  '  x
   csppSem replaceVarInProcName  P =
        csppSem replaceVarInProcName ' P "
apply (cut_tac m="counter P" and ="" and varSet="varSet" and '=" '" in
    semSameInGrownRho_aux)
apply (erule_tac x="P" in allE)
apply clarsimp
done

(*
This two lemmas refer to the label semantics function preserve the label
    in a negation mode.
*)
lemma (in procNameLocale) labelSameCaseNotTau[rule_format]: "aTau
    ( aa . (labelSem   aa)=a     aaTau )"
apply (induct a)
apply auto
done

lemma (in procNameLocale) labelSameCaseNotCheck[rule_format]: "aCheck
    ( aa . (labelSem   aa)=a     aaCheck )"
apply (induct a)
apply auto
done
```

```
(*
The actual completeness theorem
*)
lemma (in procNameLocale) completeness: "trans R b  T
 ( P     varSet varEnv. model  varEnv
& (getFreeParsInBool varEnv Un getFreeParsInCspp P  set varSet)
& ( p  . getFreeParsInCspp (procEnv p)  getFreeParsInProcName p)
& R=csppSem replaceVarInProcName  P
   (    a P' varSet' varEnv' '.
(b = labelSem  ' a) &
( x .  x  set varSet      x =  ' x) &
T=csppSem replaceVarInProcName ' P' &
 model  ' varEnv' &
oneStepEval (varSet,replaceVarInProcName,procEnv,varEnv,P) a
    (varSet',replaceVarInProcName,procEnv,varEnv',P')))"
apply (erule trans.induct)
thm lskip
apply clarsimp
apply (case_tac P)
apply clarsimp
apply (rule_tac x="Check" in exI)
apply clarsimp
apply (rule_tac x="omega" in exI)
apply clarsimp
apply (rule_tac x="varSet" in exI)
apply (rule_tac x="varEnv" in exI)
apply (simp add:lskip)
apply (rule_tac x="" in exI)
apply clarsimp+
apply (case_tac "evalInAct  act1")
apply clarsimp
apply (case_tac "evalInAct  act2")
apply clarsimp+
apply (case_tac "evalInBool  bool")
apply clarsimp+
thm ldiv
apply (case_tac P)
apply clarsimp+
apply (rule_tac x="Tau" in exI)
apply (rule_tac x="LDIV" in exI)
apply (rule_tac x="varSet" in exI)
apply (rule_tac x="varEnv" in exI)
apply (rule_tac x="" in exI)
apply clarsimp+
apply (case_tac "evalInAct  act1")
apply clarsimp
apply (case_tac "evalInAct  act2")
```

```
apply clarsimp+
apply (case_tac "evalInBool  bool")
apply clarsimp+
thm lact_prefix
apply (case_tac "Pa")
apply clarsimp+
apply (case_tac "evalInAct  act1")
apply clarsimp
apply (case_tac "evalInAct  act2")
apply clarsimp+
apply (rule_tac x="Action (((getNewVar varSet)),((getNewVar ((getNewVar
    varSet) # varSet))))" in exI)
apply (rule_tac x="cspp" in exI)
apply (rule_tac x=" ((getNewVar varSet))#((getNewVar ((getNewVar varSet)
    # varSet))) # varSet" in exI)
apply (rule_tac x="And (Equal (IdP ((getNewVar varSet))) act1) (And
    (Equal (IdP ((getNewVar ((getNewVar varSet) # varSet)))) act2)
    varEnv)" in exI)
apply (rule_tac x="x. (if x = ((getNewVar varSet)) then (aa) else if x=
    ((getNewVar ((getNewVar varSet) # varSet))) then (ab) else  x)" in
    exI)
apply clarsimp
apply (subgoal_tac "(getNewVar ((getNewVar varSet) # varSet)  getNewVar
    varSet)")
apply clarsimp
apply (cut_tac varSet="varSet" and U ="getNewVar varSet" and
    V="getNewVar ((getNewVar varSet) # varSet)" and varEnv="varEnv" and
    c ="act1" and a="act2" and P="cspp" in lact_prefix)
apply clarsimp
apply clarsimp
apply (cut_tac freshVar)
apply (erule_tac x="varSet" in allE)
apply (cut_tac freshVar)
apply (erule_tac x="((getNewVar varSet))#varSet" in allE)
apply clarsimp
apply (cut_tac P="cspp" and varSet="set varSet" and  ="" and '="x. (if
    x = ((getNewVar varSet)) then (aa) else if x= ( (getNewVar (
    (getNewVar varSet) # varSet))) then (ab) else  x)" in
    semSameInGrownRho)
apply assumption
apply clarsimp
apply clarsimp
apply (simp add:MAnd MEqual)
apply (rule conjI)
apply (simp add:evalPro)
apply (cut_tac varSet="set varSet" and a="act1" in evalProOfSameEnv)
apply (erule_tac x="" in allE)
```

```
apply (erule_tac x="(x. if x = (getNewVar varSet) then aa
                  else if x =
                          (getNewVar
                             ( (getNewVar varSet) # varSet))
                       then ab else   x)" in allE)
apply clarsimp
apply (rule conjI)
apply (simp add:evalPro)
apply (cut_tac varSet="set varSet" and a="act2" in evalProOfSameEnv)
apply (erule_tac x="" in allE)
apply (erule_tac x="(x. if x = (getNewVar varSet) then aa
                  else if x =
                          (getNewVar
                             ( (getNewVar varSet) # varSet))
                       then ab else   x)" in allE)
apply clarsimp
apply (simp add:evalBoolPro)
apply (cut_tac varSet="set varSet" and b="varEnv" in evalBoolOfSameEnv)
apply (erule_tac x="" in allE)
apply (erule_tac x="(x. if x = (getNewVar varSet) then aa
                  else if x =
                          (getNewVar
                             ( (getNewVar varSet) # varSet))
                       then ab else   x)" in allE)
apply clarsimp
apply (cut_tac freshVar)
apply (erule_tac x="varSet" in allE)
apply (cut_tac freshVar)
apply (erule_tac x="( (getNewVar varSet))#varSet" in allE)
apply clarsimp+
apply (case_tac " evalInBool  bool")
apply clarsimp+
thm lext_prefix
apply (case_tac Pa)
apply clarsimp+
apply (case_tac "evalInAct  act1")
apply clarsimp
apply (case_tac "evalInAct  act2")
apply clarsimp+
apply (case_tac " evalInBool  bool")
apply clarsimp+
apply (rule_tac x="Action (( (getNewVar varSet)), ( (getNewVar (
    (getNewVar varSet) # varSet))))" in exI)
apply (rule_tac x="replaceVarInCspp replaceVarInProcName cspp var (IdP (
    (getNewVar ( (getNewVar varSet) # varSet))))" in exI)
apply (rule_tac x="( (getNewVar varSet))#( (getNewVar ( (getNewVar
    varSet) # varSet)))# varSet" in exI)
```

```
apply (rule_tac x="And (Equal (IdP ( (getNewVar varSet))) act) (And
    varEnv (replaceVarInBool bool var (IdP ( (getNewVar ( (getNewVar
    varSet) # varSet))))))" in exI)
apply (rule_tac x="x. (if x = ( (getNewVar varSet)) then ( a) else if x=
    ( (getNewVar ( (getNewVar varSet) # varSet))) then ( b) else  x)"
    in exI)
apply clarsimp
apply (subgoal_tac " (getNewVar ( (getNewVar varSet) # varSet))
    getNewVar varSet")
apply simp
apply (cut_tac varSet="varSet" and U ="getNewVar varSet" and
    V="(getNewVar ( (getNewVar varSet) # varSet))" and varEnv="varEnv"
    and c ="act" and s="var" and P="cspp" and apropt ="bool" in
    lext_prefix)
apply clarsimp
apply clarsimp
apply (cut_tac freshVar)
apply (cut_tac varSet="set varSet" and P="cspp" and b="b" and s="var" in
    freeVarsSub)
apply blast
apply (cut_tac P="(replaceVarInCspp replaceVarInProcName cspp var
    (ActVal b))" and varSet="set varSet" and  ="" and '="x. (if x = (
    (getNewVar varSet)) then ( a) else if x= ( (getNewVar ( (getNewVar
    varSet) # varSet))) then ( b) else  x)" in semSameInGrownRho)
apply assumption
apply clarsimp
apply (erule_tac x=" ( (getNewVar varSet) # varSet)" in allE)
apply (subgoal_tac "set varSet  set ( (getNewVar varSet) # varSet)")
apply blast
apply clarsimp
apply (erule_tac x="( (getNewVar varSet))#varSet" in allE)
apply clarsimp
apply (cut_tac  = "( x . if x = (getNewVar varSet) then a
            else if x = (getNewVar ( (getNewVar varSet) # varSet))
                 then b else  x)" and y = "getNewVar ( (getNewVar
                     varSet) # varSet)" and P="cspp" and s="var" in
                     semSubSame)
apply clarsimp
apply (cut_tac freshVar)
apply (erule_tac x="varSet" in allE)
apply clarsimp
apply (simp add:MAnd MEqual)
apply (rule conjI)
apply (simp add:evalPro)
apply (cut_tac varSet="set varSet" and a="act" in evalProOfSameEnv)
apply (erule_tac x="" in allE)
apply (erule_tac x="(x. if x = (getNewVar varSet) then a
```

```
                  else if x =
                          (getNewVar
                             ( (getNewVar varSet) # varSet))
                       then  b else   x)" in allE)
apply clarsimp
apply (rule conjI)
apply (simp add:evalBoolPro)
apply (cut_tac varSet="set varSet" and b="varEnv" in evalBoolOfSameEnv)
apply (erule_tac x="" in allE)
apply (erule_tac x="(x. if x = (getNewVar varSet) then a
                  else if x =
                          (getNewVar
                             ( (getNewVar varSet) # varSet))
                       then b else   x)" in allE)
apply clarsimp
apply (simp add:evalBoolPro)
apply (simp add:actValProInBool)
apply (cut_tac varSet="set varSet" and b="(replaceVarInBool bool var
    (ActVal b))" in evalBoolOfSameEnv)
apply (erule_tac x="" in allE)
apply (erule_tac x="(x. if x = (getNewVar varSet) then a
                  else if x =
                          (getNewVar
                             ( (getNewVar varSet) # varSet))
                       then b else   x)" in allE)
apply (cut_tac b="bool" and av="b" and s="var" in
    getFreeParsInBoolFeatureX)
apply (erule_tac x="set varSet" in allE)
apply clarsimp
apply (cut_tac freshVar)
apply (erule_tac x=" ( (getNewVar varSet))# varSet" in allE)
apply clarsimp+
thm lint_prefix
apply (case_tac Pa)
apply clarsimp+
apply (case_tac "evalInAct  act1")
apply clarsimp
apply (case_tac "evalInAct  act2")
apply clarsimp+
apply (case_tac " evalInBool  bool")
apply clarsimp+
apply (rule_tac x="Tau" in exI)
apply (rule_tac x="(replaceVarInCspp replaceVarInProcName cspp var (IdP
    ( (getNewVar varSet))))" in exI)
apply (rule_tac x="( (getNewVar varSet))# varSet" in exI)
apply (rule_tac x="(And varEnv (replaceVarInBool bool var (IdP (
    (getNewVar varSet)))))" in exI)
```

```
apply (rule_tac x="x. if x= (getNewVar varSet) then (b) else  x" in exI)
apply clarsimp
apply (cut_tac U="getNewVar varSet" and varSet="varSet" and
    varEnv="varEnv" and s="var" and apropt="bool" and P="cspp" in
    lint_prefix)
apply clarsimp
apply (cut_tac freshVar)
apply (cut_tac P="(replaceVarInCspp replaceVarInProcName cspp var
    (ActVal b))" and varSet="set varSet" and  ="" and '="x. (if x=
    (getNewVar varSet) then (b) else  x)" in semSameInGrownRho)
apply (simp add:freeVarsSub)
apply (erule_tac x="varSet" in allE)
apply clarsimp
apply (cut_tac  = "( x . if x = (getNewVar varSet) then b else  x)" and
    y = "getNewVar varSet" and P="cspp" and s="var" in semSubSame)
apply clarsimp
apply (simp add:MAnd MEqual)
apply (rule conjI)
apply (simp add:evalBoolPro)
apply (cut_tac varSet="set varSet" and b="varEnv" in evalBoolOfSameEnv)
apply (erule_tac x="" in allE)
apply (erule_tac x="(x. if x = (getNewVar varSet) then b else  x)" in
    allE)
apply clarsimp
apply (simp add:evalBoolPro)
apply (simp add:actValProInBool)
apply (cut_tac varSet="set varSet" and b="(replaceVarInBool bool var
    (ActVal b))" in evalBoolOfSameEnv)
apply (erule_tac x="" in allE)
apply (erule_tac x="(x. if x = (getNewVar varSet) then b else  x)" in
    allE)
apply (cut_tac b="bool" and av="b" and s="var" in
    getFreeParsInBoolFeatureX)
apply (erule_tac x="set varSet" in allE)
apply clarsimp+
thm lint_choice1
apply (case_tac Pa)
apply clarsimp+
apply (case_tac "evalInAct  act1")
apply clarsimp
apply (case_tac "evalInAct  act2")
apply clarsimp+
apply (rule_tac x="Tau" in exI)
apply (rule_tac x="cspp1" in exI)
apply (rule_tac x="varSet" in exI)
apply (rule_tac x="varEnv" in exI)
apply (rule_tac x="" in exI)
```

```
apply clarsimp
apply (cut_tac varSet="varSet" and varEnv="varEnv" and P="cspp1" and
    Q="cspp2" in lint_choice1)
apply simp
apply (case_tac " evalInBool  bool")
apply clarsimp+
thm lint_choice2
apply (case_tac Pa)
apply clarsimp+
apply (case_tac "evalInAct  act1")
apply clarsimp
apply (case_tac "evalInAct  act2")
apply clarsimp+
apply (rule_tac x="Tau" in exI)
apply (rule_tac x="cspp2" in exI)
apply (rule_tac x="varSet" in exI)
apply (rule_tac x="varEnv" in exI)
apply (rule_tac x="" in exI)
apply clarsimp
apply (cut_tac varSet="varSet" and varEnv="varEnv" and P="cspp1" and
    Q="cspp2" in lint_choice2)
apply simp
apply (case_tac " evalInBool  bool")
apply clarsimp+
thm lext_choice_tau1
apply (case_tac Pa)
apply clarsimp+
apply (case_tac "evalInAct  act1")
apply clarsimp
apply (case_tac "evalInAct  act2")
apply clarsimp+
apply (erule_tac x="cspp1" in allE)
apply (erule_tac x="" in allE)
apply (erule_tac x="varSet" in allE)
apply (erule_tac x="varEnv" in allE)
apply clarsimp
apply (subgoal_tac "Tau =labelSem ' a  a = Tau")
apply clarsimp
apply (rule_tac x="Tau" in exI)
apply (rule_tac x="LExt_choice P'a cspp2" in exI)
apply (rule_tac x="varSet'" in exI)
apply (rule_tac x="varEnv'" in exI)
apply (rule_tac x="'" in exI)
apply clarsimp
apply (cut_tac varSet="varSet" and varEnv="varEnv" and P="cspp1" and
    Q="cspp2" and P'="P'a" and varSet'="varSet'" and varEnv'="varEnv'"
    and replaceVarInProcName'="replaceVarInProcName" and
```

```
    procEnv'="procEnv" in lext_choice_tau1)
apply assumption
apply (simp add:semSameInGrownRho)
apply (case_tac a)
apply assumption
apply clarsimp
apply clarsimp+
apply (case_tac " evalInBool  bool")
apply clarsimp+
thm lext_choice_tau2
apply (case_tac Pa)
apply clarsimp+
apply (case_tac "evalInAct  act1")
apply clarsimp
apply (case_tac "evalInAct  act2")
apply clarsimp+
apply (erule_tac x="cspp2" in allE)
apply (erule_tac x="" in allE)
apply (erule_tac x="varSet" in allE)
apply (erule_tac x="varEnv" in allE)
apply clarsimp
apply (subgoal_tac "Tau =labelSem ' a  a = Tau")
apply clarsimp
apply (rule_tac x="Tau" in exI)
apply (rule_tac x="LExt_choice cspp1 P'" in exI)
apply (rule_tac x="varSet'" in exI)
apply (rule_tac x="varEnv'" in exI)
apply (rule_tac x="'" in exI)
apply clarsimp
apply (cut_tac varSet="varSet" and varEnv="varEnv" and P="cspp1" and
    Q="cspp2" and Q'="P'" and varSet'="varSet'" and varEnv'="varEnv'"
    and replaceVarInProcName'="replaceVarInProcName" and
    procEnv'="procEnv" in lext_choice_tau2)
apply assumption
apply (simp add:semSameInGrownRho)
apply (case_tac a)
apply assumption
apply clarsimp
apply clarsimp+
apply (case_tac " evalInBool  bool")
apply clarsimp+
thm lext_choice1
apply (case_tac Pa)
apply clarsimp+
apply (case_tac "evalInAct  act1")
apply clarsimp
apply (case_tac "evalInAct  act2")
```

```
apply clarsimp+
apply (erule_tac x="cspp1" in allE)
apply (erule_tac x="" in allE)
apply (erule_tac x="varSet" in allE)
apply (erule_tac x="varEnv" in allE)
apply clarsimp
apply (rule_tac x="aa" in exI)
apply (rule_tac x="P'a" in exI)
apply (rule_tac x="varSet'" in exI)
apply (rule_tac x="varEnv'" in exI)
apply (rule_tac x="'" in exI)
apply clarsimp
apply (cut_tac a="aa" and varSet="varSet" and varEnv="varEnv" and
    P="cspp1" and Q="cspp2" and P'="P'a" and
    replaceVarInProcName'="replaceVarInProcName" and procEnv'="procEnv"
    and varEnv'="varEnv'" and varSet'="varSet'" in lext_choice1)
apply (simp add:labelSameCaseNotTau)
apply assumption
apply assumption
apply clarsimp+
apply (case_tac " evalInBool  bool")
apply clarsimp+
thm lext_choice2
apply (case_tac Pa)
apply clarsimp+
apply (case_tac "evalInAct  act1")
apply clarsimp
apply (case_tac "evalInAct  act2")
apply clarsimp+
apply (erule_tac x="cspp2" in allE)
apply (erule_tac x="" in allE)
apply (erule_tac x="varSet" in allE)
apply (erule_tac x="varEnv" in allE)
apply clarsimp
apply (rule_tac x="aa" in exI)
apply (rule_tac x="P'" in exI)
apply (rule_tac x="varSet'" in exI)
apply (rule_tac x="varEnv'" in exI)
apply (rule_tac x="'" in exI)
apply clarsimp
apply (cut_tac a="aa" and varSet="varSet" and varEnv="varEnv" and
    P="cspp1" and Q="cspp2" and Q'="P'" and
    replaceVarInProcName'="replaceVarInProcName" and procEnv'="procEnv"
    and varEnv'="varEnv'" and varSet'="varSet'" in lext_choice2)
apply (simp add:labelSameCaseNotTau)
apply assumption
apply assumption
```

```
apply clarsimp+
apply (case_tac " evalInBool  bool")
apply clarsimp+
thm lifthenelse1
thm lifthenelse2
apply (case_tac b)
apply clarsimp
apply (case_tac Pa)
apply clarsimp+
apply (case_tac "evalInAct  act1")
apply clarsimp
apply (case_tac "evalInAct  act2")
apply clarsimp+
apply (case_tac "evalInBool  bool")
apply clarsimp
apply clarsimp
apply (rule_tac x="Tau" in exI)
apply (rule_tac x="cspp1" in exI)
apply (rule_tac x="varSet" in exI)
apply (rule_tac x="(And bool varEnv)" in exI)
apply (rule_tac x="" in exI)
apply clarsimp
apply (cut_tac varSet="varSet" and varEnv="varEnv" and P="cspp1" and
    Q="cspp2" and b="bool" in lifthenelse1)
apply (simp add:MAnd)
apply (simp add:evalBoolPro)
apply clarsimp+
apply (case_tac Pa)
apply clarsimp+
apply (case_tac "evalInAct  act1")
apply clarsimp
apply (case_tac "evalInAct  act2")
apply clarsimp+
apply (case_tac "evalInBool  bool")
apply clarsimp
apply clarsimp
apply (rule_tac x="Tau" in exI)
apply (rule_tac x="cspp2" in exI)
apply (rule_tac x="varSet" in exI)
apply (rule_tac x="And (And (Not bool) (Wf bool)) varEnv" in exI)
apply (rule_tac x="" in exI)
apply clarsimp
apply (cut_tac varSet="varSet" and varEnv="varEnv" and P="cspp1" and
    Q="cspp2" and b="bool" in lifthenelse2)
apply (simp add:MAnd MNot MWf)
apply (simp add:evalBoolPro)
apply clarsimp+
```

```
thm lseq_step
apply (case_tac Pa)
apply clarsimp+
apply (case_tac "evalInAct  act1")
apply clarsimp
apply (case_tac "evalInAct  act2")
apply clarsimp+
apply (case_tac "evalInBool  bool")
apply clarsimp+
apply (erule_tac x="cspp1" in allE)
apply (erule_tac x="" in allE)
apply (erule_tac x="varSet" in allE)
apply (erule_tac x="varEnv" in allE)
apply clarsimp
apply (rule_tac x="a" in exI)
apply (rule_tac x="LSeq_compo P'a cspp2" in exI)
apply (rule_tac x="varSet'" in exI)
apply (rule_tac x="varEnv'" in exI)
apply (rule_tac x="'" in exI)
apply clarsimp
apply (cut_tac varSet="varSet" and varEnv="varEnv" and P="cspp1" and
    Q="cspp2" and x="a" and varSet'="varSet'" and
    replaceVarInProcName'="replaceVarInProcName" and procEnv'="procEnv"
    and varEnv'="varEnv'" and P'="P'a" in lseq_step)
apply (simp add:labelSameCaseNotCheck)
apply assumption
apply (simp add:semSameInGrownRho)
apply clarsimp+
thm lseq_check
apply (case_tac Pa)
apply clarsimp+
apply (case_tac "evalInAct  act1")
apply clarsimp
apply (case_tac "evalInAct  act2")
apply clarsimp+
apply (case_tac "evalInBool  bool")
apply clarsimp+
apply (erule_tac x="cspp1" in allE)
apply (erule_tac x="" in allE)
apply (erule_tac x="varSet" in allE)
apply (erule_tac x="varEnv" in allE)
apply clarsimp
apply (subgoal_tac "Check = labelSem ' aa=Check")
apply clarsimp
apply (rule_tac x="Tau" in exI)
apply (rule_tac x="cspp2" in exI)
apply (rule_tac x="varSet'" in exI)
```

```
apply (rule_tac x="varEnv'" in exI)
apply (rule_tac x="'" in exI)
apply clarsimp
apply (cut_tac varSet="varSet" and varEnv="varEnv" and P="cspp1" and
    Q="cspp2" and varSet'="varSet'" and
    replaceVarInProcName'="replaceVarInProcName" and procEnv'="procEnv"
    and varEnv'="varEnv'" in lseq_check)
apply (rule_tac x="P'a" in exI)
apply assumption
apply (simp add:semSameInGrownRho)
apply (case_tac a)
apply clarsimp
apply assumption
apply clarsimp+
thm lpar_tau1
apply (case_tac Pa)
apply clarsimp+
apply (case_tac "evalInAct  act1")
apply clarsimp
apply (case_tac "evalInAct  act2")
apply clarsimp+
apply (case_tac "evalInBool  bool")
apply clarsimp+
apply (erule_tac x="cspp1" in allE)
apply (erule_tac x="" in allE)
apply (erule_tac x="varSet" in allE)
apply (erule_tac x="varEnv" in allE)
apply clarsimp
apply (subgoal_tac "Tau = labelSem ' aa=Tau")
apply clarsimp
apply (rule_tac x="Tau" in exI)
apply (rule_tac x="LParallel P'a var1 var2 bool cspp2" in exI)
apply (rule_tac x="varSet'" in exI)
apply (rule_tac x="varEnv'" in exI)
apply (rule_tac x="'" in exI)
apply clarsimp
apply (cut_tac varSet="varSet" and varEnv="varEnv" and P="cspp1" and
    Q="cspp2" and P'="P'a" and varSet'="varSet'" and
    replaceVarInProcName'="replaceVarInProcName" and procEnv'="procEnv"
    and varEnv'="varEnv'" and s="var1" and s'="var2" and X="bool" in
    lpar_tau1)
apply assumption
apply (simp add:semSameInGrownRho)
apply (rule set_eqI)
apply clarsimp
apply (simp add:evalBoolPro)
```

```
apply (cut_tac b="(replaceVarInDoubleBool bool var1 (ActVal a) var2
    (ActVal b))" and varSet="set varSet" in evalBoolOfSameEnv)
apply (erule_tac x="" in allE)
apply (erule_tac x="'" in allE)
apply clarsimp
apply (cut_tac b="bool" and s="var1" and s'="var2" and av="a" and cv="b"
    in getFreeParsInDoubleBoolFeatureX)
apply (erule_tac x="set varSet" in allE)
apply clarsimp
apply (case_tac a)
apply assumption
apply clarsimp
apply clarsimp+
thm lpar_tau2
apply (case_tac Pa)
apply clarsimp+
apply (case_tac "evalInAct  act1")
apply clarsimp
apply (case_tac "evalInAct  act2")
apply clarsimp+
apply (case_tac "evalInBool  bool")
apply clarsimp+
apply (erule_tac x="cspp2" in allE)
apply (erule_tac x="" in allE)
apply (erule_tac x="varSet" in allE)
apply (erule_tac x="varEnv" in allE)
apply clarsimp
apply (subgoal_tac "Tau = labelSem ' aa=Tau")
apply clarsimp
apply (rule_tac x="Tau" in exI)
apply (rule_tac x="LParallel cspp1 var1 var2 bool P'" in exI)
apply (rule_tac x="varSet'" in exI)
apply (rule_tac x="varEnv'" in exI)
apply (rule_tac x="'" in exI)
apply clarsimp
apply (cut_tac varSet="varSet" and varEnv="varEnv" and P="cspp1" and
    Q="cspp2" and Q'="P'" and varSet'="varSet'" and
    replaceVarInProcName'="replaceVarInProcName" and procEnv'="procEnv"
    and varEnv'="varEnv'" and s="var1" and s'="var2" and X="bool" in
    lpar_tau2)
apply assumption
apply (simp add:semSameInGrownRho)
apply (rule set_eqI)
apply clarsimp
apply (simp add:evalBoolPro)
apply (cut_tac b="(replaceVarInDoubleBool bool var1 (ActVal a) var2
    (ActVal b))" and varSet="set varSet" in evalBoolOfSameEnv)
```

```
apply (erule_tac x="" in allE)
apply (erule_tac x="'" in allE)
apply clarsimp
apply (cut_tac b="bool" and s="var1" and s'="var2" and av="a" and cv="b"
    in getFreeParsInDoubleBoolFeatureX)
apply (erule_tac x="set varSet" in allE)
apply clarsimp
apply (case_tac a)
apply assumption
apply clarsimp
apply clarsimp+
thm lpar_out1
apply (case_tac Pa)
apply clarsimp+
apply (case_tac "evalInAct  act1")
apply clarsimp
apply (case_tac "evalInAct  act2")
apply clarsimp+
apply (case_tac "evalInBool  bool")
apply clarsimp+
apply (erule_tac x="cspp1" in allE)
apply (erule_tac x="" in allE)
apply (erule_tac x="varSet" in allE)
apply (erule_tac x="varEnv" in allE)
apply clarsimp
apply (subgoal_tac "(Action (a, b)) = labelSem ' aa( a ' b'. aa=Action
    (a',b'))")
apply clarsimp
apply (rule_tac x="Action (a',b')" in exI)
apply (rule_tac x="LParallel P'a var1 var2 bool cspp2" in exI)
apply (rule_tac x="varSet'" in exI)
apply (rule_tac x="(And varEnv' (Not (replaceVarInDoubleBool bool var1
    (IdP a') var2 (IdP b'))))" in exI)
apply (rule_tac x="'" in exI)
apply clarsimp
apply (cut_tac varSet="varSet" and varEnv="varEnv" and P="cspp1" and
    s="var1" and s'="var2" and X="bool" and Q="cspp2" and U="a'" and
    V="b'" and varSet'="varSet'" and
    replaceVarInProcName'="replaceVarInProcName" and procEnv'="procEnv"
    and varEnv'="varEnv'" and P'="P'a" in lpar_out1)
apply assumption
apply (rule conjI)
apply (rule set_eqI)
apply clarsimp
apply (simp add:evalBoolPro)
apply (cut_tac b="(replaceVarInDoubleBool bool var1 (ActVal a) var2
    (ActVal b))" and varSet="set varSet" in evalBoolOfSameEnv)
```

```
apply (erule_tac x="" in allE)
apply (erule_tac x="'" in allE)
apply clarsimp
apply (cut_tac b="bool" and s="var1" and s'="var2" and av="a" and cv="b"
    in getFreeParsInDoubleBoolFeatureX)
apply (erule_tac x="set varSet" in allE)
apply clarsimp
apply (simp add:semSameInGrownRho)
apply (simp add:MAnd MNot)
apply (simp add:evalBoolPro)
apply clarsimp
apply (cut_tac b="(replaceVarInDoubleBool bool var1 (ActVal (' a')) var2
    (ActVal (' b')))" and varSet="set varSet" in evalBoolOfSameEnv)
apply (erule_tac x="" in allE)
apply (erule_tac x="'" in allE)
apply clarsimp
apply (cut_tac b="bool" and s="var1" and s'="var2" and av="' a'" and
    cv="' b'" in getFreeParsInDoubleBoolFeatureX)
apply clarsimp
apply (erule_tac x="set varSet" in allE)
apply (cut_tac  ="  '" and x="a'" and y="b'" and b="bool" and s="var1"
    and s'="var2" in actValProInDoubleBool)
apply clarsimp
apply clarsimp
apply (case_tac aa)
apply clarsimp
apply clarsimp
apply clarsimp+
thm lpar_out2
apply (case_tac Pa)
apply clarsimp+
apply (case_tac "evalInAct  act1")
apply clarsimp
apply (case_tac "evalInAct  act2")
apply clarsimp+
apply (case_tac "evalInBool  bool")
apply clarsimp+
apply (erule_tac x="cspp2" in allE)
apply (erule_tac x="" in allE)
apply (erule_tac x="varSet" in allE)
apply (erule_tac x="varEnv" in allE)
apply clarsimp
apply (subgoal_tac "(Action (a, b)) = labelSem ' aa( a ' b'. aa=Action
    (a',b'))")
apply clarsimp
apply (rule_tac x="Action (a',b')" in exI)
apply (rule_tac x="LParallel cspp1 var1 var2 bool P'" in exI)
```

```
apply (rule_tac x="varSet'" in exI)
apply (rule_tac x="(And varEnv' (Not (replaceVarInDoubleBool bool var1
    (IdP a') var2 (IdP b'))))" in exI)
apply (rule_tac x="'" in exI)
apply clarsimp
apply (cut_tac varSet="varSet" and varEnv="varEnv" and P="cspp1" and
    s="var1" and s'="var2" and X="bool" and Q="cspp2" and U="a'" and
    V="b'" and varSet'="varSet'" and
    replaceVarInProcName'="replaceVarInProcName" and procEnv'="procEnv"
    and varEnv'="varEnv'" and Q'="P'" in lpar_out2)
apply assumption
apply (simp add:semSameInGrownRho)
apply (rule conjI)
apply (rule set_eqI)
apply clarsimp
apply (simp add:evalBoolPro)
apply (cut_tac b="(replaceVarInDoubleBool bool var1 (ActVal a) var2
    (ActVal b))" and varSet="set varSet" in evalBoolOfSameEnv)
apply (erule_tac x="" in allE)
apply (erule_tac x="'" in allE)
apply clarsimp
apply (cut_tac b="bool" and s="var1" and s'="var2" and av="a" and cv="b"
    in getFreeParsInDoubleBoolFeatureX)
apply (erule_tac x="set varSet" in allE)
apply clarsimp
apply (simp add:MAnd MNot)
apply (simp add:evalBoolPro)
apply clarsimp
apply (cut_tac b="(replaceVarInDoubleBool bool var1 (ActVal (' a')) var2
    (ActVal (' b')))" and varSet="set varSet" in evalBoolOfSameEnv)
apply (erule_tac x="" in allE)
apply (erule_tac x="'" in allE)
apply clarsimp
apply (cut_tac b="bool" and s="var1" and s'="var2" and av="' a'" and
    cv="' b'" in getFreeParsInDoubleBoolFeatureX)
apply (erule_tac x="set varSet" in allE)
apply clarsimp
apply (cut_tac  ="  '" and x="a'" and y="b'" and b="bool" and s="var1"
    and s'="var2" in actValProInDoubleBool)
apply clarsimp
apply clarsimp
apply (case_tac aa)
apply clarsimp
apply clarsimp
apply clarsimp+
thm lpar_in
apply (case_tac Pa)
```

```
apply clarsimp+
apply (case_tac "evalInAct  act1")
apply clarsimp
apply (case_tac "evalInAct  act2")
apply clarsimp+
apply (case_tac " evalInBool  bool")
apply clarsimp+
apply (erule_tac x="cspp1" in allE)
apply (erule_tac x="" in allE)
apply (erule_tac x="varSet" in allE)
apply (erule_tac x="varEnv" in allE)
apply clarsimp
apply (erule_tac x="cspp2" in allE)
apply (erule_tac x="'" in allE)
apply (erule_tac x="varSet'" in allE)
apply (erule_tac x="varEnv'" in allE)
apply clarsimp
apply (subgoal_tac "(Action (a, b)) = labelSem ' aa( a ' b'. aa=Action
    (a',b'))")
apply clarsimp
apply (frule_tac x="(varSet, replaceVarInProcName, procEnv, varEnv,
    cspp1)" and a="Action (a',b')" and y="(varSet',
    replaceVarInProcName, procEnv, varEnv', P'a)" in setGrowing)
apply clarsimp
apply (frule_tac x="getFreeParsInCspp cspp2" and varSet="set varSet" and
    varSet'="set varSet'" in setTrans)
apply assumption
apply (frule noIllFormedNess)
apply clarsimp
apply (simp add:semSameInGrownRho)
apply clarsimp
apply (subgoal_tac "(Action (' a', ' b')) = labelSem 'a  a ( a ' b'.
    a=Action (a',b'))")
apply clarsimp
apply (rule_tac x="Action (a',b')" in exI)
apply (rule_tac x="LParallel P'a var1 var2 bool P'" in exI)
apply (rule_tac x="varSet'a" in exI)
apply (rule_tac x="And (And (Equal (IdP a') (IdP a'a)) (And varEnv'a
    (replaceVarInDoubleBool bool var1 (IdP a') var2 (IdP b')))) (Equal
    (IdP b') (IdP b'a))" in exI)
apply (rule_tac x="'a" in exI)
apply clarsimp
apply (cut_tac varSet="varSet" and varEnv="varEnv" and P="cspp1" and
    Q="cspp2" and P'="P'a" and Q'="P'" and U="a'" and V="b'" and
    U'="a'a" and V'="b'a" and
    replaceVarInProcName'="replaceVarInProcName" and procEnv'="procEnv"
    and replaceVarInProcName''="replaceVarInProcName" and
```

```
    procEnv''="procEnv" and varSet'="varSet'" and varEnv'="varEnv'" and
    varSet''="varSet'a" and varEnv''="varEnv'a" and s="var1" and
    s'="var2" and X="bool" in lpar_in)
apply assumption
apply assumption
apply (subgoal_tac "(x. x    set varSet     ' x =  'a x)")
apply clarsimp
apply (cut_tac a="IdP a'" and varSet="set varSet'" in evalProOfSameEnv)
apply (erule_tac x="'" in allE)
apply (erule_tac x="'a" in allE)
apply clarsimp
apply (cut_tac x="(varSet, replaceVarInProcName, procEnv, varEnv,
    cspp1)" and a=" (Action (a', b'))" and y="(varSet',
    replaceVarInProcName, procEnv, varEnv', P'a)" and c="a'" and
    acts="b'" in labelChanBelonging)
apply assumption
apply clarsimp
apply clarsimp
apply clarsimp
apply (cut_tac a="IdP b'" and varSet="set varSet'" in evalProOfSameEnv)
apply (erule_tac x="'" in allE)
apply (erule_tac x="'a" in allE)
apply clarsimp
apply (cut_tac x="(varSet, replaceVarInProcName, procEnv, varEnv,
    cspp1)" and a=" (Action (a', b'))" and y="(varSet',
    replaceVarInProcName, procEnv, varEnv', P'a)" and c="a'" and
    acts="b'" in labelVarsBelonging)
apply assumption
apply clarsimp
apply clarsimp
apply clarsimp
apply (cut_tac x="(varSet, replaceVarInProcName, procEnv, varEnv,
    cspp1)" and a="(Action (a', b'))" and y="(varSet',
    replaceVarInProcName, procEnv, varEnv', P'a)" in noIllFormedNess)
apply assumption
apply clarsimp
apply (simp add:semSameInGrownRho)
apply (rule conjI)
apply (rule set_eqI)
apply clarsimp
apply (simp add:evalBoolPro)
apply (cut_tac b="(replaceVarInDoubleBool bool var1 (ActVal a) var2
    (ActVal b))" and varSet="set varSet" in evalBoolOfSameEnv)
apply (erule_tac x="" in allE)
apply (erule_tac x="'" in allE)
apply clarsimp
```

```
apply (cut_tac b="bool" and s="var1" and s'="var2" and av="a" and cv="b"
    in getFreeParsInDoubleBoolFeatureX)
apply (erule_tac x="set varSet" in allE)
apply clarsimp
apply (cut_tac b="(replaceVarInDoubleBool bool var1 (ActVal a) var2
    (ActVal b))" and varSet="set varSet'" in evalBoolOfSameEnv)
apply (erule_tac x="'" in allE)
apply (erule_tac x="'a" in allE)
apply clarsimp
apply (simp add:MAnd MEqual evalPro)
apply (simp add:evalBoolPro)
apply (cut_tac b="(replaceVarInDoubleBool bool var1 (ActVal ('a a'a))
    var2 (ActVal ('a b'a)))" and varSet="set varSet" in
    evalBoolOfSameEnv)
apply (erule_tac x="" in allE)
apply (erule_tac x="'" in allE)
apply clarsimp
apply (cut_tac b="bool" and av="'a a'a" and cv="'a b'a" and s="var1" and
    s'="var2" in getFreeParsInDoubleBoolFeatureX)
apply (erule_tac x="set varSet" in allE)
apply clarsimp
apply (cut_tac b="(replaceVarInDoubleBool bool var1 (ActVal ('a a'a))
    var2 (ActVal ('a b'a)))" and varSet="set varSet'" in
    evalBoolOfSameEnv)
apply (erule_tac x="'" in allE)
apply (erule_tac x="'a" in allE)
apply clarsimp
apply (cut_tac  ="  'a" and x="a'" and y="b'" and b="bool" and s="var1"
    and s'="var2" in actValProInDoubleBool)
apply clarsimp
apply clarsimp
apply (erule_tac x="x" in allE)
apply (erule_tac x="x" in allE)
apply (frule setGrowing)
apply simp
apply (subgoal_tac "xset varSet & (set varSet) (set varSet')    xset
    varSet'")
apply simp
apply blast
apply (case_tac a)
apply clarsimp
apply clarsimp
apply clarsimp
apply (case_tac aa)
apply clarsimp
apply clarsimp
apply clarsimp+
```

```
thm lpar_check1
apply (case_tac Pa)
apply clarsimp+
apply (case_tac "evalInAct  act1")
apply clarsimp
apply (case_tac "evalInAct  act2")
apply clarsimp+
apply (case_tac "evalInBool  bool")
apply clarsimp+
apply (erule_tac x="cspp1" in allE)
apply (erule_tac x="" in allE)
apply (erule_tac x="varSet" in allE)
apply (erule_tac x="varEnv" in allE)
apply clarsimp
apply (subgoal_tac "Check= labelSem ' aa=Check")
apply clarsimp
apply (rule_tac x="Tau" in exI)
apply (rule_tac x="LParallel omega var1 var2 bool cspp2" in exI)
apply (rule_tac x="varSet'" in exI)
apply (rule_tac x="varEnv'" in exI)
apply (rule_tac x="'" in exI)
apply clarsimp
apply (cut_tac varSet="varSet" and varEnv="varEnv" and P="cspp1" and
    Q="cspp2" and P'="P'a" and varSet'="varSet'" and
    replaceVarInProcName'="replaceVarInProcName" and procEnv'="procEnv"
    and varEnv'="varEnv'" and s="var1" and s'="var2" and X="bool" in
    lpar_check1)
apply assumption
apply (simp add:semSameInGrownRho)
apply (rule set_eqI)
apply clarsimp
apply (simp add:evalBoolPro)
apply (cut_tac b="(replaceVarInDoubleBool bool var1 (ActVal a) var2
    (ActVal b))" and varSet="set varSet" in evalBoolOfSameEnv)
apply (erule_tac x="" in allE)
apply (erule_tac x="'" in allE)
apply clarsimp
apply (cut_tac b="bool" and s="var1" and s'="var2" and av="a" and cv="b"
    in getFreeParsInDoubleBoolFeatureX)
apply (erule_tac x="set varSet" in allE)
apply clarsimp
apply (case_tac a)
apply clarsimp
apply assumption
apply clarsimp+
thm lpar_check2
apply (case_tac Pa)
```

```
apply clarsimp+
apply (case_tac "evalInAct  act1")
apply clarsimp
apply (case_tac "evalInAct  act2")
apply clarsimp+
apply (case_tac "evalInBool  bool")
apply clarsimp+
apply (erule_tac x="cspp2" in allE)
apply (erule_tac x="" in allE)
apply (erule_tac x="varSet" in allE)
apply (erule_tac x="varEnv" in allE)
apply clarsimp
apply (subgoal_tac "Check= labelSem ' aa=Check")
apply clarsimp
apply (rule_tac x="Tau" in exI)
apply (rule_tac x="LParallel cspp1 var1 var2 bool omega" in exI)
apply (rule_tac x="varSet'" in exI)
apply (rule_tac x="varEnv'" in exI)
apply (rule_tac x="'" in exI)
apply clarsimp
apply (cut_tac varSet="varSet" and varEnv="varEnv" and P="cspp1" and
    Q="cspp2" and Q'="P'" and varSet'="varSet'" and
    replaceVarInProcName'="replaceVarInProcName" and procEnv'="procEnv"
    and varEnv'="varEnv'" and s="var1" and s'="var2" and X="bool" in
    lpar_check2)
apply assumption
apply (simp add:semSameInGrownRho)
apply (rule set_eqI)
apply clarsimp
apply (simp add:evalBoolPro)
apply (cut_tac b="(replaceVarInDoubleBool bool var1 (ActVal a) var2
    (ActVal b))" and varSet="set varSet" in evalBoolOfSameEnv)
apply (erule_tac x="" in allE)
apply (erule_tac x="'" in allE)
apply clarsimp
apply (cut_tac b="bool" and s="var1" and s'="var2" and av="a" and cv="b"
    in getFreeParsInDoubleBoolFeatureX)
apply (erule_tac x="set varSet" in allE)
apply clarsimp
apply (case_tac a)
apply clarsimp
apply assumption
apply clarsimp+
thm lpar_omega
apply (case_tac P)
apply clarsimp+
apply (case_tac "evalInAct  act1")
```

```
apply clarsimp
apply (case_tac "evalInAct  act2")
apply clarsimp+
apply (case_tac "evalInBool  bool")
apply clarsimp+
apply (case_tac cspp1)
apply (case_tac cspp2)
apply clarsimp
apply (rule_tac x="Check" in exI)
apply (rule_tac x="omega" in exI)
apply (rule_tac x="varSet" in exI)
apply (rule_tac x="varEnv" in exI)
apply (rule_tac x="" in exI)
apply clarsimp
apply (simp add:lpar_omega)
apply clarsimp+
apply (case_tac "evalInAct  act1")
apply clarsimp
apply (case_tac "evalInAct  act2")
apply clarsimp+
apply (case_tac "evalInBool  boola")
apply clarsimp+
apply (case_tac "evalInAct  act1")
apply clarsimp
apply (case_tac "evalInAct  act2")
apply clarsimp+
apply (case_tac "evalInBool  boola")
apply clarsimp+
thm lproc_name
apply (case_tac P)
apply clarsimp+
apply (rule_tac x="Tau" in exI)
apply (rule_tac x="procEnv pname" in exI)
apply (rule_tac x="varSet" in exI)
apply (rule_tac x="varEnv" in exI)
apply (rule_tac x="" in exI)
apply clarsimp
apply (cut_tac procNameSemFeature)
apply (erule_tac x="" in allE)
apply (erule_tac x="pname" in allE)
apply (erule_tac x="pname" in allE)
apply clarsimp
apply (rule lproc_name)
apply clarsimp
apply (case_tac "evalInAct  act1")
apply clarsimp
apply (case_tac "evalInAct  act2")
```

```
apply clarsimp+
apply (case_tac " evalInBool  bool")
apply clarsimp+
thm lhid_omega
apply (case_tac Pa)
apply clarsimp+
apply (case_tac "evalInAct  act1")
apply clarsimp
apply (case_tac "evalInAct  act2")
apply clarsimp+
apply (case_tac "evalInBool  bool")
apply clarsimp+
apply (erule_tac x="cspp" in allE)
apply (erule_tac x="" in allE)
apply (erule_tac x="varSet" in allE)
apply (erule_tac x="varEnv" in allE)
apply clarsimp
apply (subgoal_tac "Check= labelSem ' aa=Check")
apply clarsimp
apply (rule_tac x="Check" in exI)
apply (rule_tac x="omega" in exI)
apply (rule_tac x="varSet'" in exI)
apply (rule_tac x="varEnv'" in exI)
apply (rule_tac x="'" in exI)
apply clarsimp
apply (cut_tac varSet="varSet" and varEnv="varEnv" and P="cspp" and
    P'="P'" and varSet'="varSet'" and
    replaceVarInProcName'="replaceVarInProcName" and procEnv'="procEnv"
    and varEnv'="varEnv'" and s="var1" and s'="var2" and X="bool" in
    lhid_omega)
apply assumption
apply assumption
apply (case_tac a)
apply clarsimp
apply assumption
apply clarsimp+
thm lhid_tau
apply (case_tac Pa)
apply clarsimp+
apply (case_tac "evalInAct  act1")
apply clarsimp
apply (case_tac "evalInAct  act2")
apply clarsimp+
apply (case_tac "evalInBool  bool")
apply clarsimp+
apply (erule_tac x="cspp" in allE)
apply (erule_tac x="" in allE)
```

```
apply (erule_tac x="varSet" in allE)
apply (erule_tac x="varEnv" in allE)
apply clarsimp
apply (subgoal_tac "Tau= labelSem ' aa =Tau")
apply clarsimp
apply (rule_tac x="Tau" in exI)
apply (rule_tac x="LHid P' var1 var2 bool" in exI)
apply (rule_tac x="varSet'" in exI)
apply (rule_tac x="varEnv'" in exI)
apply (rule_tac x="'" in exI)
apply clarsimp
apply (cut_tac varSet="varSet" and varEnv="varEnv" and P="cspp" and
    P'="P'" and varSet'="varSet'" and
    replaceVarInProcName'="replaceVarInProcName" and procEnv'="procEnv"
    and varEnv'="varEnv'" and s="var1" and s'="var2" and X="bool" in
    lhid_tau)
apply assumption
apply clarsimp
apply (rule set_eqI)
apply clarsimp
apply (simp add:evalBoolPro)
apply (cut_tac b="(replaceVarInDoubleBool bool var1 (ActVal a) var2
    (ActVal b))" and varSet="set varSet" in evalBoolOfSameEnv)
apply (erule_tac x="" in allE)
apply (erule_tac x="'" in allE)
apply clarsimp
apply (cut_tac b="bool" and s="var1" and s'="var2" and av="a" and cv="b"
    in getFreeParsInDoubleBoolFeatureX)
apply (erule_tac x="set varSet" in allE)
apply clarsimp
apply (case_tac a)
apply assumption
apply clarsimp
apply clarsimp+
thm lhid_neg
apply (case_tac Pa)
apply clarsimp+
apply (case_tac "evalInAct  act1")
apply clarsimp
apply (case_tac "evalInAct  act2")
apply clarsimp+
apply (case_tac "evalInBool  bool")
apply clarsimp+
apply (erule_tac x="cspp" in allE)
apply (erule_tac x="" in allE)
apply (erule_tac x="varSet" in allE)
apply (erule_tac x="varEnv" in allE)
```

```
apply clarsimp
apply (subgoal_tac " (Action (a, b)) = labelSem ' aa( a ' b'. aa=Action
    (a',b'))")
apply clarsimp
apply (rule_tac x="Action (a',b')" in exI)
apply (rule_tac x="LHid P' var1 var2 bool" in exI)
apply (rule_tac x="varSet'" in exI)
apply (rule_tac x="(And varEnv' (Not (replaceVarInDoubleBool bool var1
    (IdP a') var2 (IdP b'))))" in exI)
apply (rule_tac x="'" in exI)
apply clarsimp
apply (cut_tac varSet="varSet" and varEnv="varEnv" and P="cspp" and
    s="var1" and s'="var2" and X="bool" and U="a'" and V="b'" and
    varSet'="varSet'" and replaceVarInProcName'="replaceVarInProcName"
    and procEnv'="procEnv" and varEnv'="varEnv'" and P'="P'" in
    lhid_neg)
apply assumption
apply clarsimp
apply (rule conjI)
apply (rule set_eqI)
apply clarsimp
apply (simp add:evalBoolPro)
apply (cut_tac b="(replaceVarInDoubleBool bool var1 (ActVal a) var2
    (ActVal b))" and varSet="set varSet" in evalBoolOfSameEnv)
apply (erule_tac x="" in allE)
apply (erule_tac x="'" in allE)
apply clarsimp
apply (cut_tac b="bool" and s="var1" and s'="var2" and av="a" and cv="b"
    in getFreeParsInDoubleBoolFeatureX)
apply (erule_tac x="set varSet" in allE)
apply clarsimp
apply (simp add:MAnd MNot)
apply (simp add:evalBoolPro)
apply (cut_tac b="(replaceVarInDoubleBool bool var1 (ActVal (' a')) var2
    (ActVal (' b')))" and varSet="set varSet" in evalBoolOfSameEnv)
apply (erule_tac x="" in allE)
apply (erule_tac x="'" in allE)
apply clarsimp
apply (cut_tac b="bool" and s="var1" and s'="var2" and av="' a'" and
    cv="' b'" in getFreeParsInDoubleBoolFeatureX)
apply (erule_tac x="set varSet" in allE)
apply clarsimp
apply (cut_tac  ="  '" and x="a'" and y="b'"and b="bool" and s="var1"
    and s'="var2" in actValProInDoubleBool)
apply clarsimp
apply clarsimp
apply (case_tac aa)
```

```
apply clarsimp
apply clarsimp
apply clarsimp+
thm lhid_pos
apply (case_tac Pa)
apply clarsimp+
apply (case_tac "evalInAct  act1")
apply clarsimp
apply (case_tac "evalInAct  act2")
apply clarsimp+
apply (case_tac "evalInBool  bool")
apply clarsimp+
apply (erule_tac x="cspp" in allE)
apply (erule_tac x="" in allE)
apply (erule_tac x="varSet" in allE)
apply (erule_tac x="varEnv" in allE)
apply clarsimp
apply (subgoal_tac "(Action (a, b)) = labelSem ' aa( a ' b'. aa=Action
    (a',b'))")
apply clarsimp
apply (rule_tac x="Tau" in exI)
apply (rule_tac x="LHid P' var1 var2 bool" in exI)
apply (rule_tac x="varSet'" in exI)
apply (rule_tac x="(And varEnv' (replaceVarInDoubleBool bool var1 (IdP
    a') var2 (IdP b')))" in exI)
apply (rule_tac x="'" in exI)
apply clarsimp
apply (cut_tac varSet="varSet" and varEnv="varEnv" and P="cspp" and
    s="var1" and s'="var2" and X="bool" and U="a'" and V="b'" and
    varSet'="varSet'" and replaceVarInProcName'="replaceVarInProcName"
    and procEnv'="procEnv" and varEnv'="varEnv'" and P'="P'" in
    lhid_pos)
apply assumption
apply clarsimp
apply (rule conjI)
apply (rule set_eqI)
apply clarsimp
apply (simp add:evalBoolPro)
apply (cut_tac b="(replaceVarInDoubleBool bool var1 (ActVal a) var2
    (ActVal b))" and varSet="set varSet" in evalBoolOfSameEnv)
apply (erule_tac x="" in allE)
apply (erule_tac x="'" in allE)
apply clarsimp
apply (cut_tac b="bool" and s="var1" and s'="var2" and av="a" and cv="b"
    in getFreeParsInDoubleBoolFeatureX)
apply (erule_tac x="set varSet" in allE)
apply clarsimp
```

```
apply (simp add:MAnd MNot)
apply (simp add:evalBoolPro)
apply (cut_tac b="(replaceVarInDoubleBool bool var1 (ActVal (' a')) var2
    (ActVal (' b')))" and varSet="set varSet" in evalBoolOfSameEnv)
apply (erule_tac x="" in allE)
apply (erule_tac x="'" in allE)
apply clarsimp
apply (cut_tac b="bool" and s="var1" and s'="var2" and av="' a'" and
    cv="' b'" in getFreeParsInDoubleBoolFeatureX)
apply (erule_tac x="set varSet" in allE)
apply clarsimp
apply (cut_tac  ="  '" and x="a'" and y="b'" and b="bool" and s="var1"
    and s'="var2" in actValProInDoubleBool)
apply clarsimp
apply clarsimp
apply (case_tac aa)
apply clarsimp
apply clarsimp
apply clarsimp+
done

end
```

Appendix B

Simulator extracted from Isabelle

In this section, we show how to extract our HCSP simulator from Isabelle. We define all our action datatype in the Isabelle theory file: `cspp_data_real`. Then we use the theory file `cspp_function` to instantiate the generic HCSP syntax and semantics with the action datatype we defined in the file `cspp_data_real`. The file `cspp_function` also defines the procedure to extract Isabelle code to an executable Ocaml code. The OneStepEval Ocaml module is the code that is directly extracted from our Isabelle theory files. It contains the core of the HCSP syntax and semantics in a step move. The content of the file `stream.ml` implements the technique that we mentioned in Section 5 to store the unevaluated function data body of a given HCSP process. Finally, th file `evaluator.ml` is the code that users actually use to see all traces step by step.

```
theory cspp_data_real
imports Main cspp_data
begin


(*
A datatype to implement the value data in data locale
*)
datatype ('basicData,'chanName) actVal = Null | BasicVal 'basicData |
    ChanVal 'chanName | PropVal bool | NumVal int | PairVal
    "(('basicData,'chanName) actVal) * (('basicData,'chanName) actVal)"
    | ListVal "('basicData,'chanName) actVal list" (*| SetVal
    "(nat'value actVal)"*)
(*cannot add sets here *)


(*
A recursive datatype to implement the act and boolexp data in data locale
*)
datatype ('var,'basicData,'chanName) act =
   BasicAct 'basicData

 | ActVal "('basicData,'chanName) actVal"
 | IdP "'var par"
```

```
 | IdV "'var var"

 | Num int
 | Plus "('var,'basicData,'chanName) act" "('var,'basicData,'chanName)
     act"
 | Minus "('var,'basicData,'chanName) act" "('var,'basicData,'chanName)
     act"
 | Neg "('var,'basicData,'chanName) act"
 | Times "('var,'basicData,'chanName) act" "('var,'basicData,'chanName)
     act"

 | Prop "('var,'basicData,'chanName) propt"

 | Nil
 | Cons "('var,'basicData,'chanName) act" "('var,'basicData,'chanName)
     act"
 | Hd  "('var,'basicData,'chanName) act"
 | Tl  "('var,'basicData,'chanName) act"
 | Append "('var,'basicData,'chanName) act" "('var,'basicData,'chanName)
     act"
 | Nth "('var,'basicData,'chanName) act" "('var,'basicData,'chanName)
     act"
 | Last "('var, 'basicData, 'chanName) act"

 | Chan 'chanName

 | Pair "('var,'basicData,'chanName) act" "('var,'basicData,'chanName)
     act"
 | Fst "('var,'basicData,'chanName) act"
 | Snd "('var,'basicData,'chanName) act"

and ('var,'basicData,'chanName) propt =
  True_p | False_p
 | And "('var,'basicData,'chanName) propt" "('var,'basicData,'chanName)
     propt"
 | Or "('var,'basicData,'chanName) propt" "('var,'basicData,'chanName)
     propt"
 | Not "('var,'basicData,'chanName) propt"
 | Wf "('var,'basicData,'chanName) propt"
 | Equal "('var,'basicData,'chanName) act" "('var,'basicData,'chanName)
     act"
 | Less "('var,'basicData,'chanName) act" "('var,'basicData,'chanName)
     act"
 | Elt "('var,'basicData,'chanName) act" "('var,'basicData,'chanName)
     actSet"
 | SubSetEq "('var,'basicData,'chanName) actSet"
     "('var,'basicData,'chanName) actSet"
```

```
 | Mem "('var,'basicData,'chanName) act" "('var,'basicData,'chanName)
     act"
 | SubListEq "('var,'basicData,'chanName) act"
     "('var,'basicData,'chanName) act"

and ('var,'basicData,'chanName) actSet = ProptSet "'var"
    "('var,'basicData,'chanName) propt"
 | ActionSet "(('var,'basicData,'chanName) act list)"
 | ActListSet "('var,'basicData,'chanName) act"
 | InterSet "('var,'basicData,'chanName) actSet"
     "('var,'basicData,'chanName) actSet"
 | UnionSet "('var,'basicData,'chanName) actSet"
     "('var,'basicData,'chanName) actSet"



(* group of functions to replace all the Bound act variable with an act
    in an action *)
primrec replaceVarInAct :: "('var,'basicData,'chanName) act  'var
    ('var,'basicData,'chanName) act  ('var,'basicData,'chanName) act"
    and replaceVarInActSet :: "('var,'basicData,'chanName) actSet  'var
     ('var,'basicData,'chanName) act  ('var,'basicData,'chanName)
    actSet" and replaceVarInList :: "('var,'basicData,'chanName) act
    list  'var  ('var,'basicData,'chanName) act
    ('var,'basicData,'chanName) act list" and replaceVarInBool ::
    "('var,'basicData,'chanName) propt  'var
    ('var,'basicData,'chanName) act  ('var,'basicData,'chanName) propt"
    where
"replaceVarInAct (BasicAct n) s a = (BasicAct n)"
| "replaceVarInAct (Chan n) s a = (Chan n)"
|"replaceVarInAct (Num n) s a= (Num n)"
| "replaceVarInAct (ActVal n) s a = (ActVal n)"
| "replaceVarInAct (Hd a) s a' = Hd (replaceVarInAct a s a')"
| "replaceVarInAct (Tl a) s a' = Tl (replaceVarInAct a s a')"
| "replaceVarInAct (Last a) s a' = Last (replaceVarInAct a s a')"
| "replaceVarInAct (IdP s') s a = (IdP s')"
| "replaceVarInAct (IdV s') s a = (if (s'= Var s) then a else IdV s')"
| "replaceVarInAct (Prop p) s a = Prop (replaceVarInBool p s a)"
| "replaceVarInAct (Pair x y) s a = Pair (replaceVarInAct x s a)
    (replaceVarInAct y s a)"
| "replaceVarInAct (Fst x) s a = Fst (replaceVarInAct x s a)"
| "replaceVarInAct (Snd x) s a = Snd (replaceVarInAct x s a)"
| "replaceVarInAct Nil s a = Nil"
| "replaceVarInAct (Cons x y) s a = Cons (replaceVarInAct x s a)
    (replaceVarInAct y s a)"
| "replaceVarInAct (Append x y) s a = Append (replaceVarInAct x s a)
    (replaceVarInAct y s a)"
```

```
| "replaceVarInAct (Nth x y) s a = Nth (replaceVarInAct x s a)
    (replaceVarInAct y s a)"
| "replaceVarInAct (Plus x y) s a = Plus (replaceVarInAct x s a)
    (replaceVarInAct y s a)"
| "replaceVarInAct (Minus x y) s a = Minus (replaceVarInAct x s a)
    (replaceVarInAct y s a)"
| "replaceVarInAct (Times x y) s a = Times (replaceVarInAct x s a)
    (replaceVarInAct y s a)"
| "replaceVarInAct (Neg x) s a = Neg (replaceVarInAct x s a)"
| "replaceVarInList [] s a = []"
| "replaceVarInList (x#xs) s a = ((replaceVarInAct x s
    a)#(replaceVarInList xs s a))"
|"replaceVarInActSet (ProptSet s' p) s a = (if s = s' then (ProptSet s'
    p) else (ProptSet s' (replaceVarInBool p s a)))"
| "replaceVarInActSet (ActionSet alist) s a = ActionSet
    (replaceVarInList alist s a)"
| "replaceVarInActSet (ActListSet a) s a' = ActListSet (replaceVarInAct
    a s a')"
| "replaceVarInActSet (InterSet x y) s a = InterSet (replaceVarInActSet
    x s a) (replaceVarInActSet y s a)"
| "replaceVarInActSet (UnionSet x y) s a = UnionSet (replaceVarInActSet
    x s a) (replaceVarInActSet y s a)"
| "replaceVarInBool True_p s a = True_p"
| "replaceVarInBool False_p s a = False_p"
| "replaceVarInBool (And x y) s a = And (replaceVarInBool x s a)
    (replaceVarInBool y s a)"
| "replaceVarInBool (Or x y) s a = Or (replaceVarInBool x s a)
    (replaceVarInBool y s a)"
| "replaceVarInBool (Not x) s a = Not (replaceVarInBool x s a)"
| "replaceVarInBool (Wf x) s a= Wf (replaceVarInBool x s a)"
| "replaceVarInBool (Equal x y) s a = Equal (replaceVarInAct x s a)
    (replaceVarInAct y s a)"
| "replaceVarInBool (Less x y) s a = Less (replaceVarInAct x s a)
    (replaceVarInAct y s a)"
| "replaceVarInBool (Elt x y) s a = Elt (replaceVarInAct x s a)
    (replaceVarInActSet y s a)"
| "replaceVarInBool (SubSetEq x y) s a = SubSetEq (replaceVarInActSet x
    s a) (replaceVarInActSet y s a)"
| "replaceVarInBool (Mem x y) s a =Mem (replaceVarInAct x s a)
    (replaceVarInAct y s a)"
| "replaceVarInBool (SubListEq x y) s a = SubListEq (replaceVarInAct x s
    a) (replaceVarInAct y s a)"


(*The implementation of eval function for act and boolexp*)
function evalInAct :: "('var par  ('basicData,'chanName) actVal)  ('var,
    'basicData, 'chanName) act  (('basicData,'chanName) actVal) option"
```

```
        and evalInBool :: "('var par  ('basicData,'chanName) actVal)
            ('var, 'basicData, 'chanName) propt  bool option"
       and evalInActSet :: "('var par  ('basicData,'chanName) actVal)
           ('var, 'basicData, 'chanName) actSet
           ((('basicData,'chanName) actVal) set) option"
        and evalInActList :: "('var par  ('basicData,'chanName) actVal)
             ('var, 'basicData, 'chanName) act list
            ((('basicData,'chanName) actVal) list) option"
 where
"evalInAct env (BasicAct a) = Some (BasicVal a)"
|"evalInAct env (Chan c) = Some (ChanVal c)"
| "evalInAct env (Num n) = Some (NumVal n)"
| "evalInAct env (ActVal a) = Some a"
| "evalInAct env (IdP v) = Some (env v)"
| "evalInAct evn (IdV v) = None"
| "evalInAct env (Pair a b) = (case (evalInAct env a) of None  None |
    Some av  (case (evalInAct env b) of None  None | Some bv  Some
    (PairVal (av,bv))))"
| "evalInAct env (Fst a) = (case (evalInAct env a) of Some (PairVal
    (av,bv))  Some av | _  None)"
| "evalInAct env (Snd a) = (case (evalInAct env a) of Some (PairVal
    (av,bv))  Some bv | _  None)"
| "evalInAct env (Hd a) = (case (evalInAct env a) of Some (ListVal
    (x#xs))  Some x | _  None)"
| "evalInAct env (Tl a) = (case (evalInAct env a) of Some (ListVal
    alist)  Some (ListVal (List.tl alist)) | _  None)"
| "evalInAct env (Last a) = (case (evalInAct env a) of Some (ListVal
    alist)  Some (List.last alist) | _  None)"
| "evalInAct env Nil = Some (ListVal [])"
| "evalInAct env (Cons a b) = (case (evalInAct env a) of None  None |
    Some av  (case (evalInAct env b) of Some (ListVal alist)  Some
    (ListVal (av#alist)) | _  None))"
| "evalInAct env (Append a b) = (case (evalInAct env a) of Some (ListVal
    alist)  (case (evalInAct env b) of Some (ListVal blist)  Some
    (ListVal (alist @ alist)) | _  None) | _  None )"
| "evalInAct env (Nth a b) = (case (evalInAct env a) of Some (NumVal av)
     (case (evalInAct env b) of Some (ListVal blist)  (if av 0 then
    Some (List.nth blist (nat av)) else None) | _  None) | _  None )"
| "evalInAct env (Plus a b) = (case (evalInAct env a) of Some (NumVal
    av)  (case (evalInAct env b) of Some (NumVal bv)  Some (NumVal
    (av+bv)) | _  None) | _  None )"
| "evalInAct env (Minus a b) = (case (evalInAct env a) of Some (NumVal
    av)  (case (evalInAct env b) of Some (NumVal bv)  Some (NumVal
    (av-bv)) | _  None) | _  None )"
| "evalInAct env (Times a b) = (case (evalInAct env a) of Some (NumVal
    av)  (case (evalInAct env b) of Some (NumVal bv)  Some (NumVal
    (av*bv)) | _  None) | _  None )"
```

```
| "evalInAct env (Neg a) = (case (evalInAct env a) of Some (NumVal av)
    Some (NumVal (-av)) | _  None)"
| "evalInAct env (Prop p) = (case (evalInBool env p) of None  None |
    Some b  Some (PropVal b))"

| "evalInBool env (True_p) = Some True"
| "evalInBool env (False_p) = Some False"
| "evalInBool env (And a b) = (case (evalInBool env a) of None  Some
    False | Some ab  (case (evalInBool env b) of None  Some False |
    Some bb  (Some (ab & bb))))"
| "evalInBool env (Or a b) = (case (evalInBool env a) of None  (case
    (evalInBool env b) of None  Some False | Some bb  Some bb) | Some
    ab  (case (evalInBool env b) of None  Some ab | Some bb(Some (ab |
    bb))))"
| "evalInBool env (Not a) = (case (evalInBool env a) of None  Some True
    | Some b  Some (~b))"
| "evalInBool env (Wf b) = (case (evalInBool env b) of None  Some False
    | Some b  Some True)"
| "evalInBool env (Equal a b) = (case (evalInAct env a) of None  None |
    Some aa  (case (evalInAct env b) of None  None | Some ba  Some
    (aa=ba)))"
| "evalInBool env (Less a b) = (case (evalInAct env a) of Some (NumVal
    av)  (case (evalInAct env b) of Some (NumVal bv)  (Some (av<bv)) |
    _  None) | _  None)"
| "evalInBool env (Elt a b) = (case (evalInAct env a) of None  None |
    Some aa  (case (evalInActSet env b) of None  None | Some aset  Some
    (aa  aset)))"
| "evalInBool env (SubSetEq a b) = (case (evalInActSet env a) of None
    None | Some aset  (case (evalInActSet env b) of None  None | Some
    bset Some (aset  bset)))"
| "evalInBool env (Mem a b) = (case (evalInAct env a) of None  None |
    Some aa  (case (evalInAct env b) of Some (ListVal blist)  Some (aa
    (set blist)) | _  None))"
| "evalInBool env (SubListEq a b) = (case (evalInAct env a) of Some
    (ListVal alist)  (case (evalInAct env b) of Some (ListVal blist)
    Some ((set alist)  (set blist)) | _  None) | _  None)"
| "evalInActSet env (UnionSet a b) = (case (evalInActSet env a) of None
    None | Some aset  (case (evalInActSet env b) of None  None | Some
    bset Some (aset Un bset)))"
| "evalInActSet env (InterSet a b) = (case (evalInActSet env a) of None
    None |Some aset  (case (evalInActSet env b) of None  None |Some
    bset  Some (aset Int bset)))"
| "evalInActSet env (ActionSet alist) = (case (evalInActList env alist)
    of None  None | Some alist  Some (set alist))"
| "evalInActSet env (ActListSet a) = (case (evalInAct env a) of Some
    (ListVal alist)  Some (set alist) | _  None)"
| "evalInActSet env (ProptSet s p) = Some {x. case (evalInBool env
```

```
    (replaceVarInBool p s (ActVal x))) of None  False | Some bb}"
| "evalInActList env [] =Some []"
| "evalInActList env (x#xs) = (case (evalInAct env x) of None  None |
    Some a  (case (evalInActList env xs) of None  None | Some alist
    Some (a#alist)))"
by pat_completeness auto

(*
A function to provide the termination order for the eval functions
*)
primrec evalInActCounter :: "('var par  ('basicData,'chanName) actVal)
    ('var,'basicData, 'chanName) act  nat"
         and evalInBoolCounter :: "('var par  ('basicData,'chanName)
             actVal)  ('var, 'basicData, 'chanName) propt  nat"
        and evalInActSetCounter :: "('var par  ('basicData,'chanName)
            actVal)  ('var, 'basicData, 'chanName) actSet  nat"
         and evalInActListCounter :: "('var par  ('basicData,'chanName)
             actVal)  ('var, 'basicData, 'chanName) act list  nat"
 where
"evalInActCounter env (BasicAct a) = 0"
|"evalInActCounter env (Chan c) = 0"
| "evalInActCounter env (Num n) = 0"
| "evalInActCounter env (IdP v) = 0"
| "evalInActCounter env (IdV v) = 0"
| "evalInActCounter env (ActVal a) = 0"
| "evalInActCounter env (Pair a b) = 1+(evalInActCounter env a)
    +(evalInActCounter env b)"
| "evalInActCounter env (Fst a) =1+(evalInActCounter env a)"
| "evalInActCounter env (Snd a) = 1+(evalInActCounter env a)"
| "evalInActCounter env (Hd a) = 1+(evalInActCounter env a)"
| "evalInActCounter env (Last a) = 1+(evalInActCounter env a)"
| "evalInActCounter env (Tl a) = 1+(evalInActCounter env a)"
| "evalInActCounter env Nil = 0"
| "evalInActCounter env (Cons a b) = 1+(evalInActCounter env a)
    +(evalInActCounter env b)"
| "evalInActCounter env (Append a b) = 1+(evalInActCounter env a)
    +(evalInActCounter env b)"
| "evalInActCounter env (Nth a b) = 1+(evalInActCounter env a)
    +(evalInActCounter env b)"
| "evalInActCounter env (Plus a b) = 1+(evalInActCounter env a)
    +(evalInActCounter env b)"
| "evalInActCounter env (Minus a b) = 1+(evalInActCounter env a)
    +(evalInActCounter env b)"
| "evalInActCounter env (Times a b) = 1+(evalInActCounter env a)
    +(evalInActCounter env b)"
| "evalInActCounter env (Neg a) = 1+(evalInActCounter env a)"
| "evalInActCounter env (Prop p) = 1+(evalInBoolCounter env p)"
```

```
| "evalInBoolCounter env (True_p) = 0"
| "evalInBoolCounter env (False_p) = 0"
| "evalInBoolCounter env (And a b) = 1+(evalInBoolCounter env a) +
    (evalInBoolCounter env b)"
| "evalInBoolCounter env (Or a b) = 1+(evalInBoolCounter env a) +
    (evalInBoolCounter env b)"
| "evalInBoolCounter env (Not a) = 1+(evalInBoolCounter env a)"
| "evalInBoolCounter env (Wf a) = 1+(evalInBoolCounter env a)"
| "evalInBoolCounter env (Equal a b) = 1+ (evalInActCounter env a)
    +(evalInActCounter env b)"
| "evalInBoolCounter env (Less a b) =1+ (evalInActCounter env a)
    +(evalInActCounter env b)"
| "evalInBoolCounter env (Elt a b) = 1+ (evalInActCounter env a)
    +(evalInActSetCounter env b)"
| "evalInBoolCounter env (SubSetEq a b) = 1+(evalInActSetCounter env a)
    +(evalInActSetCounter env b)"
| "evalInBoolCounter env (Mem a b) = 1+(evalInActCounter env a)
    +(evalInActCounter env b)"
| "evalInBoolCounter env (SubListEq a b) = 1+(evalInActCounter env a)
    +(evalInActCounter env b)"
| "evalInActSetCounter env (UnionSet a b) = 1+(evalInActSetCounter env
    a) +(evalInActSetCounter env b)"
| "evalInActSetCounter env (InterSet a b) = 1+ (evalInActSetCounter env
    a) + (evalInActSetCounter env b)"
| "evalInActSetCounter env (ActionSet alist) = 1+(evalInActListCounter
    env alist)"
| "evalInActSetCounter env (ActListSet a) = 1+(evalInActCounter env a)"
| "evalInActSetCounter env (ProptSet s p) = 1+(evalInBoolCounter env p)"
| "evalInActListCounter env [] =0"
| "evalInActListCounter env (x#xs) =1+(evalInActCounter env x) +
    (evalInActListCounter env xs)"



(*
Lemmas to prove that the eval functions are actually terminated
*)
lemma termination_aux0: "(evalInActCounter env (replaceVarInAct a s
    (ActVal x))
      < Suc (evalInActCounter env a))
& ( evalInBoolCounter env (replaceVarInBool p s (ActVal x))
      < Suc (evalInBoolCounter env p))
& (evalInActSetCounter env (replaceVarInActSet as s (ActVal x))
      < Suc (evalInActSetCounter env as))
& (evalInActListCounter env (replaceVarInList alist s (ActVal x))
      < Suc (evalInActListCounter env alist))"
apply (rule_tac act="a" and propt="p" and actSet="as" and list="alist"
```

```
in   act_propt_actSet.induct)
apply auto
done


termination evalInAct
apply (relation "measure (x. (case x of (Inl a)  (case a of (Inl
    (env,aa)) evalInActCounter env aa | (Inr (env,ab))
    evalInBoolCounter env ab) | (Inr b)  (case b of (Inl (env,ba))
    evalInActSetCounter env ba | (Inr (env,bb))  evalInActListCounter
    env bb)))")
apply auto
apply (simp add:termination_aux0)
done


(*
The definition for model function
*)
definition model :: "('var par  ('basicData,'chanName) actVal)
    ('var,'basicData, 'chanName) propt  bool" where
"model env p = (case (evalInBool env p) of None  False | Some b b)"


(*
this lemma provides the poof of singleSubActInId and singleSubProInId in
    the locale data for the instance we are creating
*)
lemma singleSubInId_proof : "(s  var
          (replaceVarInAct (replaceVarInAct acts s (IdP y)) var
            (ActVal x) = replaceVarInAct (replaceVarInAct acts var
                (ActVal x)) s (IdP y)))
& (s    var
          (replaceVarInBool (replaceVarInBool p s (IdP y)) var
            (ActVal x) = replaceVarInBool (replaceVarInBool p var
                (ActVal x)) s (IdP y)))
& (s    var
          (replaceVarInActSet (replaceVarInActSet as s (IdP y)) var
            (ActVal x)) = replaceVarInActSet (replaceVarInActSet as var
                (ActVal x)) s (IdP y))
& (s    var
          (replaceVarInList (replaceVarInList alist s (IdP y)) var
            (ActVal x)) = replaceVarInList (replaceVarInList alist var
                (ActVal x)) s (IdP y))"
apply (rule_tac act="acts" and propt="p" and actSet="as" and list="alist"
in   act_propt_actSet.induct)
apply clarsimp+
```

```
done



(*
this lemma provides the poof of singleSubActInValue and
    singleSubActInValue in the locale data for the instance we are
    creating
*)
lemma singleSubInValue_proof: "(s  var
          (replaceVarInAct (replaceVarInAct acts s (ActVal n)) var
            (ActVal x) = replaceVarInAct (replaceVarInAct acts var
                (ActVal x)) s (ActVal n)))
& (s    var
          (replaceVarInBool (replaceVarInBool p s (ActVal n)) var
            (ActVal x) = replaceVarInBool (replaceVarInBool p var
                (ActVal x)) s (ActVal n)))
& (s    var
          (replaceVarInActSet (replaceVarInActSet as s (ActVal n)) var
            (ActVal x)) = replaceVarInActSet (replaceVarInActSet as var
                (ActVal x)) s (ActVal n))
& (s    var
          (replaceVarInList (replaceVarInList alist s (ActVal n)) var
            (ActVal x)) = replaceVarInList (replaceVarInList alist var
                (ActVal x)) s (ActVal n))"
apply (rule_tac act="acts" and propt="p" and actSet="as" and list="alist"
in  act_propt_actSet.induct)
apply clarsimp+
done


(*
these two lemmas provide the poof of actValProInBool and actValProInAct
    in the locale data for the instance we are creating
*)
lemma actValPro_proof_aux:
"    m.
((   a.
((evalInActCounter  a = m))
evalInAct   (replaceVarInAct a s (IdP y)) = evalInAct  (replaceVarInAct
    a s (ActVal ( y))))
(    b.
((evalInBoolCounter  b = m))
evalInBool   (replaceVarInBool b s (IdP y)) = evalInBool
    (replaceVarInBool b s (ActVal ( y))))
(    as.
((evalInActSetCounter  as = m))
```

```
evalInActSet  (replaceVarInActSet as s (IdP y)) = evalInActSet
    (replaceVarInActSet as s (ActVal ( y))))
(   alist.
((evalInActListCounter  alist = m))
evalInActList  (replaceVarInList alist s (IdP y)) = evalInActList
    (replaceVarInList alist s (ActVal ( y)))))"
apply (rule allI)
apply (induct_tac rule: nat_less_induct)
apply (case_tac n)
apply clarsimp
apply (rule conjI)
apply clarify
apply (case_tac "a", clarsimp+)
apply (rule conjI)
apply clarify
apply (case_tac "b", clarsimp+)
apply (rule conjI)
apply clarify
apply (case_tac "as", clarsimp+)
apply (case_tac "alist", clarsimp+)
apply (rule conjI)
apply clarify
apply (case_tac "a",clarsimp+)
apply (frule_tac x = "evalInActCounter  act1" in spec)
apply (erule_tac x = "evalInActCounter  act2" in allE)
apply clarsimp
apply (erule_tac x="act1" in allE)
apply (erule_tac x="act2" in allE)
apply clarsimp
apply (case_tac " (evalInAct  (replaceVarInAct act1 s (ActVal ( y))))")
apply clarsimp
apply clarsimp
apply (case_tac "a")
apply clarsimp+
apply (frule_tac x = "evalInActCounter  act1" in spec)
apply (erule_tac x = "evalInActCounter  act2" in allE)
apply clarsimp
apply (erule_tac x="act1" in allE)
apply (erule_tac x="act2" in allE)
apply clarsimp
apply (case_tac " (evalInAct  (replaceVarInAct act1 s (ActVal ( y))))")
apply clarsimp
apply clarsimp
apply (case_tac "a")
apply clarsimp+
apply (erule_tac x = "evalInActCounter  act" in allE)
apply clarsimp+
```

```
apply (frule_tac x = "evalInActCounter  act1" in spec)
apply (erule_tac x = "evalInActCounter  act2" in allE)
apply clarsimp
apply (erule_tac x="act1" in allE)
apply (erule_tac x="act2" in allE)
apply clarsimp
apply (case_tac " (evalInAct  (replaceVarInAct act1 s (ActVal ( y))))")
apply clarsimp
apply clarsimp
apply (case_tac "a")
apply clarsimp+
apply (erule_tac x = "evalInBoolCounter  propt" in allE)
apply clarsimp+
apply (frule_tac x = "evalInActCounter  act1" in spec)
apply (erule_tac x = "evalInActCounter  act2" in allE)
apply clarsimp
apply (erule_tac x="act1" in allE)
apply (erule_tac x="act2" in allE)
apply clarsimp
apply (case_tac " (evalInAct  (replaceVarInAct act1 s (ActVal ( y))))")
apply clarsimp+
apply (erule_tac x = "evalInActCounter  act" in allE)
apply clarsimp+
apply (erule_tac x = "evalInActCounter  act" in allE)
apply clarsimp+
apply (frule_tac x = "evalInActCounter  act1" in spec)
apply (erule_tac x = "evalInActCounter  act2" in allE)
apply clarsimp
apply (erule_tac x="act1" in allE)
apply (erule_tac x="act2" in allE)
apply clarsimp
apply (case_tac " (evalInAct  (replaceVarInAct act1 s (ActVal ( y))))")
apply clarsimp
apply clarsimp
apply (case_tac "a")
apply clarsimp+
apply (frule_tac x = "evalInActCounter  act1" in spec)
apply (erule_tac x = "evalInActCounter  act2" in allE)
apply clarsimp
apply (erule_tac x="act1" in allE)
apply (erule_tac x="act2" in allE)
apply clarsimp
apply (case_tac " (evalInAct  (replaceVarInAct act1 s (ActVal ( y))))")
apply clarsimp
apply clarsimp
apply (case_tac "a")
apply clarsimp+
```

```
apply (erule_tac x = "evalInActCounter  act" in allE)
apply clarsimp+
apply (frule_tac x = "evalInActCounter  act1" in spec)
apply (erule_tac x = "evalInActCounter  act2" in allE)
apply clarsimp
apply (erule_tac x="act1" in allE)
apply (erule_tac x="act2" in allE)
apply clarsimp
apply (case_tac " (evalInAct  (replaceVarInAct act1 s (ActVal ( y))))")
apply clarsimp+
apply (erule_tac x = "evalInActCounter  act" in allE)
apply clarsimp+
apply (erule_tac x = "evalInActCounter  act" in allE)
apply clarsimp
apply (rule conjI)
apply clarify
apply (case_tac "b",clarsimp+)
apply (frule_tac x = "evalInBoolCounter  propt1" in spec)
apply (erule_tac x = "evalInBoolCounter  propt2" in allE)
apply clarsimp
apply (erule_tac x="act1" in allE)
apply (erule_tac x="act2" in allE)
apply (erule_tac x="propt1" in allE)
apply (erule_tac x="propt2" in allE)
apply clarsimp
apply (case_tac " (evalInBool  (replaceVarInBool propt1 s (ActVal (
    y))))")
apply clarsimp+
apply (frule_tac x = "evalInBoolCounter  propt1" in spec)
apply (erule_tac x = "evalInBoolCounter  propt2" in allE)
apply clarsimp
apply (erule_tac x="act1" in allE)
apply (erule_tac x="act2" in allE)
apply (erule_tac x="propt1" in allE)
apply (erule_tac x="propt2" in allE)
apply clarsimp
apply (case_tac " (evalInBool  (replaceVarInBool propt1 s (ActVal (
    y))))")
apply clarsimp+
apply (erule_tac x = "evalInBoolCounter  propt" in allE)
apply clarsimp+
apply (erule_tac x = "evalInBoolCounter  propt" in allE)
apply clarsimp+
apply (frule_tac x = "evalInActCounter  act1" in spec)
apply (erule_tac x = "evalInActCounter  act2" in allE)
apply clarsimp
apply (erule_tac x="act1" in allE)
```

```
apply (erule_tac x="act2" in allE)
apply clarsimp
apply (case_tac " (evalInAct  (replaceVarInAct act1 s (ActVal ( y))))")
apply clarsimp+
apply (frule_tac x = "evalInActCounter  act1" in spec)
apply (erule_tac x = "evalInActCounter  act2" in allE)
apply clarsimp
apply (erule_tac x="act1" in allE)
apply (erule_tac x="act2" in allE)
apply clarsimp
apply (case_tac " (evalInAct  (replaceVarInAct act1 s (ActVal ( y))))")
apply clarsimp+
apply (case_tac a)
apply clarsimp+
apply (frule_tac x = "evalInActCounter  act" in spec)
apply (erule_tac x = "evalInActSetCounter  actSet" in allE)
apply clarsimp
apply (erule_tac x="act" in allE)
apply (erule_tac x="act" in allE)
apply (erule_tac x="actSet" in allE)
apply (erule_tac x="actSet" in allE)
apply clarsimp
apply (case_tac " (evalInAct  (replaceVarInAct act s (ActVal ( y))))")
apply clarsimp+
apply (frule_tac x = "evalInActSetCounter  actSet1" in spec)
apply (erule_tac x = "evalInActSetCounter  actSet2" in allE)
apply clarsimp
apply (erule_tac x="actSet1" in allE)
apply (erule_tac x="actSet2" in allE)
apply clarsimp
apply (case_tac " (evalInActSet  (replaceVarInActSet actSet1 s (ActVal (
    y))))")
apply clarsimp+
apply (frule_tac x = "evalInActCounter  act1" in spec)
apply (erule_tac x = "evalInActCounter  act2" in allE)
apply clarsimp
apply (erule_tac x="act1" in allE)
apply (erule_tac x="act2" in allE)
apply clarsimp
apply (case_tac " (evalInAct  (replaceVarInAct act1 s (ActVal ( y))))")
apply clarsimp+
apply (frule_tac x = "evalInActCounter  act1" in spec)
apply (erule_tac x = "evalInActCounter  act2" in allE)
apply clarsimp
apply (erule_tac x="act1" in allE)
apply (erule_tac x="act2" in allE)
apply clarsimp
```

```
apply (case_tac " (evalInAct  (replaceVarInAct act1 s (ActVal ( y))))")
apply clarsimp+
apply (case_tac a)
apply clarsimp+
apply (rule conjI)
apply clarify
apply (case_tac "as",clarsimp+)
apply (rule set_eqI)
apply clarsimp
apply (subgoal_tac " (replaceVarInBool (replaceVarInBool propt s (IdP y))
                var (ActVal x))=replaceVarInBool (replaceVarInBool
                    propt var (ActVal x)) s (IdP y)")
apply (subgoal_tac "(replaceVarInBool (replaceVarInBool propt s (ActVal
    ( y))) var
                (ActVal x))=replaceVarInBool (replaceVarInBool propt
                    var (ActVal x)) s (ActVal ( y))")
apply clarsimp
apply (erule_tac x = "evalInBoolCounter  (replaceVarInBool propt var
    (ActVal x))" in allE)
apply (simp add:termination_aux0)
apply (simp add:singleSubInValue_proof)
apply (simp add:singleSubInId_proof)
apply clarsimp
apply (erule_tac x="evalInActListCounter  list" in allE)
apply clarsimp+
apply (erule_tac x = "evalInActCounter  act" in allE)
apply clarsimp+
apply (frule_tac x = "evalInActSetCounter  actSet1" in spec)
apply (erule_tac x = "evalInActSetCounter  actSet2" in allE)
apply clarsimp
apply (erule_tac x="actSet1" in allE)
apply (erule_tac x="actSet2" in allE)
apply clarsimp
apply (case_tac " (evalInActSet  (replaceVarInActSet actSet1 s (ActVal (
    y))))")
apply clarsimp+
apply (frule_tac x = "evalInActSetCounter  actSet1" in spec)
apply (erule_tac x = "evalInActSetCounter  actSet2" in allE)
apply clarsimp
apply (erule_tac x="actSet1" in allE)
apply (erule_tac x="actSet2" in allE)
apply clarsimp
apply (case_tac " (evalInActSet  (replaceVarInActSet actSet1 s (ActVal (
    y))))")
apply clarsimp+
apply (case_tac "alist",clarsimp+)
apply (frule_tac x = "evalInActCounter  a" in spec)
```

```
apply (erule_tac x = "evalInActListCounter  list" in allE)
apply clarsimp
apply (erule_tac x="a" in allE)
apply (erule_tac x="list" in allE)
apply (erule_tac x="list" in allE)
apply clarsimp
apply (case_tac " (evalInAct  (replaceVarInAct a s (ActVal ( y))))")
apply clarsimp+
done

lemma actValPro_proof[rule_format]: "(evalInAct  (replaceVarInAct a s
    (IdP y)) = evalInAct  (replaceVarInAct a s (ActVal ( y))))
&  (evalInBool   (replaceVarInBool b s (IdP y)) = evalInBool
    (replaceVarInBool b s (ActVal ( y))))
& (evalInActSet  (replaceVarInActSet as s (IdP y)) = evalInActSet
    (replaceVarInActSet as s (ActVal ( y))))
& (evalInActList  (replaceVarInList alist s (IdP y)) = evalInActList
    (replaceVarInList alist s (ActVal ( y))))"
apply (cut_tac  = "  " and s = "s" and y="y" in actValPro_proof_aux)
apply (rule conjI)
apply (erule_tac x="evalInActCounter  a" in allE)
apply clarsimp
apply (rule conjI)
apply (erule_tac x="evalInBoolCounter  b" in allE)
apply clarsimp
apply (rule conjI)
apply (erule_tac x="evalInActSetCounter  as" in allE)
apply clarsimp
apply (erule_tac x="evalInActListCounter  alist" in allE)
apply clarsimp
done



(*
The instance of the locale data and the proof of the datatype act and
    propt are the actually instance of the data-locale
*)
interpretation myData: data "IdV :: 'var var
    ('var,'basicData,'chanName) act" "IdP :: 'var par
    ('var,'basicData,'chanName) act" "Equal" "Not" "True_p" "False_p"
    "Wf" "And" "model" "replaceVarInAct" "replaceVarInBool" "evalInAct"
    "evalInBool" "ActVal"
apply unfold_locales
apply clarsimp+
apply (simp add:model_def)
apply (simp add:model_def)
```

```
apply (simp add:model_def)
apply (case_tac "(evalInBool  b)")
apply clarsimp
apply clarsimp
apply (simp add:model_def)
apply (case_tac "evalInAct  a")
apply clarsimp+
apply (case_tac "evalInAct  c")
apply clarsimp+
apply (simp add:model_def)
apply (case_tac "(evalInBool  b)",clarsimp)
apply clarsimp
apply (case_tac "evalInBool  d",clarsimp)
apply clarsimp
apply (simp add:model_def)
apply (case_tac "(evalInBool  b)",clarsimp)
apply clarsimp
apply (simp add:model_def)
apply (case_tac "(evalInBool  b)",clarsimp)
apply clarsimp
apply (simp add:actValPro_proof)
apply (simp add:actValPro_proof)
apply (simp add:singleSubInValue_proof)
apply (simp add:singleSubInId_proof)
apply (simp add:singleSubInValue_proof)
apply (simp add:singleSubInId_proof)
done

end
```

```
theory cspp_function
imports cspp_data cspp_data_real cspp_syntax Main
begin

primrec replaceVarInDoubleAct :: "('var,'basicData,'chanName) act  'var
    ('var,'basicData,'chanName) act  'var  ('var,'basicData,'chanName)
    act  ('var,'basicData,'chanName) act" and replaceVarInDoubleActSet
    :: "('var,'basicData,'chanName) actSet  'var
    ('var,'basicData,'chanName) act  'var  ('var,'basicData,'chanName)
    act  ('var,'basicData,'chanName) actSet" and replaceVarInDoubleList
    :: "('var,'basicData,'chanName) act list  'var
    ('var,'basicData,'chanName) act  'var  ('var,'basicData,'chanName)
    act  ('var,'basicData,'chanName) act list" and
    replaceVarInDoubleBool :: "('var,'basicData,'chanName) propt  'var
    ('var,'basicData,'chanName) act  'var  ('var,'basicData,'chanName)
    act  ('var,'basicData,'chanName) propt" where
```

```
"replaceVarInDoubleAct (BasicAct n) s c s' a = (BasicAct n)"
| "replaceVarInDoubleAct (Chan n) s c s' a = (Chan n)"
|"replaceVarInDoubleAct (Num n) s c s' a= (Num n)"
|"replaceVarInDoubleAct (ActVal n) s c s' a = ActVal n"
| "replaceVarInDoubleAct (Hd a) s c s' a' = Hd (replaceVarInDoubleAct a
    s c s' a')"
| "replaceVarInDoubleAct (Tl a) s c s' a' = Tl (replaceVarInDoubleAct a
    s c s' a')"
| "replaceVarInDoubleAct (Last a) s c s' a' = Last
    (replaceVarInDoubleAct a s c s' a')"
| "replaceVarInDoubleAct (IdV a') s c s' a = (if a'=(Var s) then c else
    if a'=(Var s') then a else (IdV a'))"
| "replaceVarInDoubleAct (IdP a') s c s' a = (IdP a')"
| "replaceVarInDoubleAct (Prop p) s c s' a = Prop
    (replaceVarInDoubleBool p s c s' a)"
| "replaceVarInDoubleAct (Pair x y) s c s' a = Pair
    (replaceVarInDoubleAct x s c s' a) (replaceVarInDoubleAct y s c s'
    a)"
| "replaceVarInDoubleAct (Fst x) s c s' a = Fst (replaceVarInDoubleAct x
    s c s' a)"
| "replaceVarInDoubleAct (Snd x) s c s' a = Snd (replaceVarInDoubleAct x
    s c s' a)"
| "replaceVarInDoubleAct Nil s c s' a = Nil"
| "replaceVarInDoubleAct (Cons x y) s c s' a = Cons
    (replaceVarInDoubleAct x s c s' a) (replaceVarInDoubleAct y s c s'
    a)"
| "replaceVarInDoubleAct (Append x y) s c s' a = Append
    (replaceVarInDoubleAct x s c s' a) (replaceVarInDoubleAct y s c s'
    a)"
| "replaceVarInDoubleAct (Nth x y) s c s' a = Nth (replaceVarInDoubleAct
    x s c s' a) (replaceVarInDoubleAct y s c s' a)"
| "replaceVarInDoubleAct (Plus x y) s c s' a = Plus
    (replaceVarInDoubleAct x s c s' a) (replaceVarInDoubleAct y s c s'
    a)"
| "replaceVarInDoubleAct (Minus x y) s c s' a = Minus
    (replaceVarInDoubleAct x s c s' a) (replaceVarInDoubleAct y s c s'
    a)"
| "replaceVarInDoubleAct (Times x y) s c s' a = Times
    (replaceVarInDoubleAct x s c s' a) (replaceVarInDoubleAct y s c s'
    a)"
| "replaceVarInDoubleAct (Neg x) s c s' a = Neg (replaceVarInDoubleAct x
    s c s' a)"
| "replaceVarInDoubleList [] s c s' a = []"
| "replaceVarInDoubleList (x#xs) s c s' a = ((replaceVarInDoubleAct x s
    c s' a)#(replaceVarInDoubleList xs s c s' a))"
|"replaceVarInDoubleActSet (ProptSet ss p) s c s' a = (if ((ss =
    s)|(ss=s')) then (ProptSet ss p) else (ProptSet ss
```

```
    (replaceVarInDoubleBool p s c s' a)))"
| "replaceVarInDoubleActSet (ActionSet alist) s c s' a = ActionSet
    (replaceVarInDoubleList alist s c s' a)"
| "replaceVarInDoubleActSet (ActListSet a) s c s' a' = ActListSet
    (replaceVarInDoubleAct a s c s' a')"
| "replaceVarInDoubleActSet (InterSet x y) s c s' a = InterSet
    (replaceVarInDoubleActSet x s c s' a) (replaceVarInDoubleActSet y s
    c s' a)"
| "replaceVarInDoubleActSet (UnionSet x y) s c s' a = UnionSet
    (replaceVarInDoubleActSet x s c s' a) (replaceVarInDoubleActSet y s
    c s' a)"
| "replaceVarInDoubleBool True_p s c s' a = True_p"
| "replaceVarInDoubleBool False_p s c s' a = False_p"
| "replaceVarInDoubleBool (And x y) s c s' a = And
    (replaceVarInDoubleBool x s c s' a) (replaceVarInDoubleBool y s c
    s' a)"
| "replaceVarInDoubleBool (Or x y) s c s' a = Or (replaceVarInDoubleBool
    x s c s' a) (replaceVarInDoubleBool y s c s' a)"
| "replaceVarInDoubleBool (Not x) s c s' a = Not (replaceVarInDoubleBool
    x s c s' a)"
| "replaceVarInDoubleBool (Wf x) s c s' a = Wf (replaceVarInDoubleBool x
    s c s' a)"
| "replaceVarInDoubleBool (Equal x y) s c s' a = Equal
    (replaceVarInDoubleAct x s c s' a) (replaceVarInDoubleAct y s c s'
    a)"
| "replaceVarInDoubleBool (Less x y) s c s' a = Less
    (replaceVarInDoubleAct x s c s' a) (replaceVarInDoubleAct y s c s'
    a)"
| "replaceVarInDoubleBool (Elt x y) s c s' a = Elt
    (replaceVarInDoubleAct x s c s' a) (replaceVarInDoubleActSet y s c
    s' a)"
| "replaceVarInDoubleBool (SubSetEq x y) s c s' a = SubSetEq
    (replaceVarInDoubleActSet x s c s' a) (replaceVarInDoubleActSet y s
    c s' a)"
| "replaceVarInDoubleBool (Mem x y) s c s' a =Mem (replaceVarInDoubleAct
    x s c s' a) (replaceVarInDoubleAct y s c s' a)"
| "replaceVarInDoubleBool (SubListEq x y) s c s' a = SubListEq
    (replaceVarInDoubleAct x s c s' a) (replaceVarInDoubleAct y s c s'
    a)"

primrec replaceVarInCspp :: "(('var,'basicData,'chanName) act  'var
    ('var,'basicData,'chanName) act  ('var,'basicData,'chanName) act)
    'var  ('var,'basicData,'chanName) act  'pname  'pname)
    ('var,('var,'basicData,'chanName) act, ('var,'basicData,'chanName)
    propt,'pname) cspp  'var  ('var,'basicData,'chanName) act
    ('var,('var,'basicData,'chanName) act, ('var,'basicData,'chanName)
    propt,'pname) cspp" where
```

```
"replaceVarInCspp pf LSKIP s a = LSKIP"
|"replaceVarInCspp pf omega s a=omega"
| "replaceVarInCspp pf LSTOP s a = LSTOP"
| "replaceVarInCspp pf LDIV s a = LDIV"
| "replaceVarInCspp pf (LProc_name p) s a = LProc_name (pf
    replaceVarInAct s a p)"
| "replaceVarInCspp pf (LAct_prefix c aa P) s a = LAct_prefix
    (replaceVarInAct c s a) (replaceVarInAct aa s a) (replaceVarInCspp
    pf P s a)"
| "replaceVarInCspp pf (LExt_choice P Q) s a = LExt_choice
    (replaceVarInCspp pf P s a) (replaceVarInCspp pf Q s a)"
| "replaceVarInCspp pf (LInt_choice P Q) s a = LInt_choice
    (replaceVarInCspp pf P s a) (replaceVarInCspp pf Q s a)"
| "replaceVarInCspp pf (LSeq_compo P Q) s a = LSeq_compo
    (replaceVarInCspp pf P s a) (replaceVarInCspp pf Q s a)"
| "replaceVarInCspp pf (LIF b P Q) s a = LIF (replaceVarInBool b s a)
    (replaceVarInCspp pf P s a) (replaceVarInCspp pf Q s a)"
| "replaceVarInCspp pf (LParallel P c s us Q) s' a = LParallel
    (replaceVarInCspp pf P s' a) c s (if ((c=s') | (s=s')) then us else
    (replaceVarInBool us s' a)) (replaceVarInCspp pf Q s' a)"
| "replaceVarInCspp pf (LExt_pre_choice c ss aset P) s a =(if (s = ss)
    then LExt_pre_choice (replaceVarInAct c s a) ss aset P else
    LExt_pre_choice (replaceVarInAct c s a) ss (replaceVarInBool aset s
    a) (replaceVarInCspp pf P s a))"
| "replaceVarInCspp pf (LRep_int_choice ss aset P) s a =(if (s = ss)
    then LRep_int_choice ss aset P else LRep_int_choice ss
    (replaceVarInBool aset s a) (replaceVarInCspp pf P s a))"
| "replaceVarInCspp pf (LHid P c s us) s' a= LHid (replaceVarInCspp pf P
    s' a) c s (if ((c=s') | (s=s')) then us else (replaceVarInBool us
    s' a))"



primrec oneStepEvalFun :: "('pname  ('var,('var,'basicData,'chanName)
    act, ('var,'basicData,'chanName) propt,'pname) cspp)  (('var par)
    list  'var par)  ((('var,'basicData,'chanName) act  'var
    ('var,'basicData,'chanName) act  ('var,'basicData,'chanName) act)
    'var  ('var,'basicData,'chanName) act  'pname  'pname)   (('var
    par) list) ('var,'basicData,'chanName) propt  ('var,
    ('var,'basicData,'chanName) act, ('var,'basicData,'chanName) propt
    , 'pname) cspp  (('var par * 'var par) label * ('var par) list *
    ('var,'basicData,'chanName) propt * ('var,
    ('var,'basicData,'chanName) act, ('var,'basicData,'chanName) propt
    , 'pname) cspp) list"
where
"oneStepEvalFun procEnv getNewVar replaceVarInProcName varSet varEnv
    omega = []"
```

```
| "oneStepEvalFun procEnv getNewVar replaceVarInProcName varSet varEnv
    LSKIP = [(Check,varSet,varEnv,omega)]"
| "oneStepEvalFun procEnv getNewVar replaceVarInProcName varSet varEnv
    LDIV = [(Tau,varSet,varEnv,LDIV)]"
| "oneStepEvalFun procEnv getNewVar replaceVarInProcName varSet varEnv
    LSTOP = []"
| "oneStepEvalFun procEnv getNewVar replaceVarInProcName varSet varEnv
    (LAct_prefix c a P) = (case (getNewVar varSet) of y  (case
    (getNewVar ((y)# varSet)) of z  [(Action ((y),(z)),(z# (y#
    varSet)),And (Equal (IdP y) c) (And (Equal (IdP z) a) varEnv),P)]))"
| "oneStepEvalFun procEnv getNewVar replaceVarInProcName varSet varEnv
    (LInt_choice P Q) = [(Tau,varSet,varEnv,P), (Tau,varSet,varEnv,Q)]"
| "oneStepEvalFun procEnv getNewVar replaceVarInProcName varSet varEnv
    (LExt_choice P Q) = (case (oneStepEvalFun procEnv getNewVar
    replaceVarInProcName varSet varEnv P) of someList  (case
    (oneStepEvalFun procEnv getNewVar replaceVarInProcName varSet
    varEnv Q) of aList  (List.map ( (a,varSet',varEnv',P'). (case a of
    Tau  (Tau,varSet',varEnv',LExt_choice P' Q) | _
    (a,varSet',varEnv',P'))) someList)@(List.map (
    (a,varSet',varEnv',Q'). (case a of Tau
    (Tau,varSet',varEnv',LExt_choice P Q') | _
    (a,varSet',varEnv',Q'))) aList)))"
| "oneStepEvalFun procEnv getNewVar replaceVarInProcName varSet varEnv
    (LSeq_compo P Q) = (case (oneStepEvalFun procEnv getNewVar
    replaceVarInProcName varSet varEnv P) of
someList    List.map ( (a,varSet',varEnv',P'). (case a of Check
    (Tau,varSet',varEnv',Q) | _  (a,varSet',varEnv',LSeq_compo P' Q)))
    someList)"
| "oneStepEvalFun procEnv getNewVar replaceVarInProcName varSet varEnv
    (LIF B P Q) = [(Tau,varSet,And B varEnv,P),(Tau,varSet,And (And
    (Not B) (Wf B)) varEnv,Q)]"
| "oneStepEvalFun procEnv getNewVar replaceVarInProcName varSet varEnv
    (LProc_name p) = [(Tau,varSet,varEnv,procEnv p)]"
| "oneStepEvalFun procEnv getNewVar replaceVarInProcName varSet varEnv
    (LExt_pre_choice c s p P) = (case (getNewVar varSet) of y  (case
    (getNewVar (y# varSet)) of z  [((Action ((y),(z))),z# (y#
    varSet),And (Equal (IdP y) c) (And varEnv (replaceVarInBool p s
    (IdP z))),replaceVarInCspp replaceVarInProcName P s (IdP z))]))"
|"oneStepEvalFun procEnv getNewVar replaceVarInProcName varSet varEnv
    (LRep_int_choice s p P) = (case (getNewVar varSet) of y  [(Tau,y#
    varSet,(And varEnv (replaceVarInBool p s (IdP y))),replaceVarInCspp
    replaceVarInProcName P s (IdP y))])"
| "oneStepEvalFun procEnv getNewVar replaceVarInProcName varSet varEnv
    (LParallel P s s' X Q) = (case (P,Q) of (omega,omega)
    [(Check,varSet,varEnv,omega)] | _  (case (oneStepEvalFun procEnv
    getNewVar replaceVarInProcName varSet varEnv P) of someList
    (List.foldl ( col (acts,varSet',varEnv',P').
```

```
  (case acts of Check   ((Tau,varSet',varEnv',LParallel omega s s' X
      Q)#col)
                | Tau    ((Tau,varSet',varEnv',LParallel P' s s' X
                    Q)#col)
                | Action (c,a)
                      (case (oneStepEvalFun procEnv getNewVar
                          replaceVarInProcName varSet' varEnv' Q) of
                          aList
                        (List.foldl (  col'
                            (acts',varSet'',varEnv'',Q'). (case acts'
                            of Action (c',a')  ((Action
                            (c,a),varSet'',And (And (Equal (IdP a)
                            (IdP a')) (And varEnv''
                            (replaceVarInDoubleBool X s (IdP c) s'
                            (IdP a)))) (Equal (IdP c) (IdP
                            c')),LParallel P' s s' X Q')#col') | _
                            col')) [] aList)@((Action
                            (c,a),varSet',(And varEnv' (Not
                            (replaceVarInDoubleBool X s (IdP c) s'
                            (IdP a)))),LParallel P' s s' X Q)#col))))
                            [] someList)
   @ (case (oneStepEvalFun procEnv getNewVar replaceVarInProcName
       varSet varEnv Q) of aList  (List.map (
       (acts,varSet',varEnv',Q'). (case acts of Check
       (Tau,varSet',varEnv',LParallel P s s' X omega) | Tau
       (Tau,varSet',varEnv',LParallel P s s' X Q') | Action (c,a)
       (Action (c,a),varSet',(And varEnv' (Not (replaceVarInDoubleBool
       X s (IdP c) s' (IdP a)))),LParallel P s s' X Q'))) aList))))"
| "oneStepEvalFun procEnv getNewVar replaceVarInProcName varSet varEnv
    (LHid P s s' X)= (case (oneStepEvalFun procEnv getNewVar
    replaceVarInProcName varSet varEnv P) of someList  (List.foldl (
    col (acts,varSet',varEnv',P').
          (case acts of Check   ((Check,varSet',varEnv',omega)#col)
                  | Tau    ((Tau,varSet',varEnv',LHid P' s s' X)#col)
                  | Action (c,a)   [(Tau,varSet',And varEnv'
                      ((replaceVarInDoubleBool X s (IdP c) s' (IdP
                      a))),LHid P' s s' X),(Action (c,a),varSet',And
                      varEnv' (Not (replaceVarInDoubleBool X s (IdP
                      c) s' (IdP a))),LHid P' s s' X)]@col)) []
                      someList))"

export_code oneStepEvalFun
  in OCaml
  module_name oneStepEval file "oneStepEval.ml"

end
```

```
module oneStepEval : sig
  type 'a equal = {equal : 'a -> 'a -> bool}
  val equal : 'a equal -> 'a -> 'a -> bool
  val eq : 'a equal -> 'a -> 'a -> bool
  type num = One | Bit0 of num | Bit1 of num
  type int = Zero_int | Pos of num | Neg of num
  val map : ('a -> 'b) -> 'a list -> 'b list
  val foldl : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
  type 'a par = Par of 'a
  type 'a var = Var of 'a
  type 'a label = Tau | Check | Action of 'a
  type ('a, 'b, 'c, 'd) cspp = Omega | Lskip | Lstop | Ldiv | LProc_name
      of 'd |
    LAct_prefix of 'b * 'b * ('a, 'b, 'c, 'd) cspp |
    LExt_choice of ('a, 'b, 'c, 'd) cspp * ('a, 'b, 'c, 'd) cspp |
    LInt_choice of ('a, 'b, 'c, 'd) cspp * ('a, 'b, 'c, 'd) cspp |
    Lif of 'c * ('a, 'b, 'c, 'd) cspp * ('a, 'b, 'c, 'd) cspp |
    LParallel of ('a, 'b, 'c, 'd) cspp * 'a * 'a * 'c * ('a, 'b, 'c, 'd)
        cspp |
    LSeq_compo of ('a, 'b, 'c, 'd) cspp * ('a, 'b, 'c, 'd) cspp |
    LExt_pre_choice of 'b * 'a * 'c * ('a, 'b, 'c, 'd) cspp |
    LRep_int_choice of 'a * 'c * ('a, 'b, 'c, 'd) cspp |
    LHid of ('a, 'b, 'c, 'd) cspp * 'a * 'a * 'c
  type ('a, 'b) actVal = Null | BasicVal of 'a | ChanVal of 'b | PropVal
      of bool
    | NumVal of int | PairVal of (('a, 'b) actVal * ('a, 'b) actVal) |
    ListVal of ('a, 'b) actVal list
  type ('a, 'b, 'c) act = BasicAct of 'b | ActVal of ('b, 'c) actVal |
    IdP of 'a par | IdV of 'a var | Num of int |
    Plus of ('a, 'b, 'c) act * ('a, 'b, 'c) act |
    Minus of ('a, 'b, 'c) act * ('a, 'b, 'c) act | Nega of ('a, 'b, 'c)
        act |
    Times of ('a, 'b, 'c) act * ('a, 'b, 'c) act | Prop of ('a, 'b, 'c)
        propt |
    Nil | Cons of ('a, 'b, 'c) act * ('a, 'b, 'c) act | Hd of ('a, 'b,
        'c) act |
    Tl of ('a, 'b, 'c) act | Append of ('a, 'b, 'c) act * ('a, 'b, 'c)
        act |
    Nth of ('a, 'b, 'c) act * ('a, 'b, 'c) act | Last of ('a, 'b, 'c)
        act |
    Chan of 'c | Pair of ('a, 'b, 'c) act * ('a, 'b, 'c) act |
    Fst of ('a, 'b, 'c) act | Snd of ('a, 'b, 'c) act
  and ('a, 'b, 'c) propt = True_p | False_p |
    And of ('a, 'b, 'c) propt * ('a, 'b, 'c) propt |
    Or of ('a, 'b, 'c) propt * ('a, 'b, 'c) propt | Not of ('a, 'b, 'c)
        propt |
```

```
  Wf of ('a, 'b, 'c) propt | Equal of ('a, 'b, 'c) act * ('a, 'b, 'c)
      act |
  Less of ('a, 'b, 'c) act * ('a, 'b, 'c) act |
  Elt of ('a, 'b, 'c) act * ('a, 'b, 'c) actSet |
  SubSetEq of ('a, 'b, 'c) actSet * ('a, 'b, 'c) actSet |
  Mem of ('a, 'b, 'c) act * ('a, 'b, 'c) act |
  SubListEq of ('a, 'b, 'c) act * ('a, 'b, 'c) act
and ('a, 'b, 'c) actSet = ProptSet of 'a * ('a, 'b, 'c) propt |
  ActionSet of ('a, 'b, 'c) act list | ActListSet of ('a, 'b, 'c) act |
  InterSet of ('a, 'b, 'c) actSet * ('a, 'b, 'c) actSet |
  UnionSet of ('a, 'b, 'c) actSet * ('a, 'b, 'c) actSet
val equal_var : 'a equal -> 'a var -> 'a var -> bool
val replaceVarInBool :
  'a equal ->
    ('a, 'b, 'c) propt -> 'a -> ('a, 'b, 'c) act -> ('a, 'b, 'c) propt
val replaceVarInAct :
  'a equal -> ('a, 'b, 'c) act -> 'a -> ('a, 'b, 'c) act -> ('a, 'b,
      'c) act
val replaceVarInList :
  'a equal ->
    ('a, 'b, 'c) act list -> 'a -> ('a, 'b, 'c) act -> ('a, 'b, 'c)
        act list
val replaceVarInActSet :
  'a equal ->
    ('a, 'b, 'c) actSet -> 'a -> ('a, 'b, 'c) act -> ('a, 'b, 'c)
        actSet
val oneStepEvalFun :
  'b equal ->
    ('a -> ('b, ('b, 'c, 'd) act, ('b, 'c, 'd) propt, 'a) cspp) ->
      ('b par list -> 'b par) ->
        ((((('b, 'c, 'd) act -> 'b -> ('b, 'c, 'd) act -> ('b, 'c, 'd)
            act) ->
           'b -> ('b, 'c, 'd) act -> 'a -> 'a) ->
          ('b, ('b, 'c, 'd) act, ('b, 'c, 'd) propt, 'a) cspp ->
            'b -> ('b, 'c, 'd) act ->
                  ('b, ('b, 'c, 'd) act, ('b, 'c, 'd) propt, 'a) cspp)
                      ->
          (((('b, 'c, 'd) act -> 'b -> ('b, 'c, 'd) act -> ('b, 'c, 'd)
              act) ->
            'b -> ('b, 'c, 'd) act -> 'a -> 'a) ->
            (('b, 'c, 'd) propt ->
              'b -> ('b, 'c, 'd) act ->
                     'b -> ('b, 'c, 'd) act -> ('b, 'c, 'd) propt) ->
              'b par list ->
                ('b, 'c, 'd) propt ->
                  ('b, ('b, 'c, 'd) act, ('b, 'c, 'd) propt, 'a) cspp ->
                    (('b par * 'b par) label *
```

```
                    ('b par list *
                      (('b, 'c, 'd) propt *
                        ('b, ('b, 'c, 'd) act, ('b, 'c, 'd) propt, 'a)
                          cspp))) list
end = struct

type 'a equal = {equal : 'a -> 'a -> bool};;
let equal _A = _A.equal;;

let rec eq _A a b = equal _A a b;;

type num = One | Bit0 of num | Bit1 of num;;

type int = Zero_int | Pos of num | Neg of num;;

let rec map
  f x1 = match f, x1 with f, [] -> []
    | f, x :: xs -> f x :: map f xs;;

let rec foldl
  f a x2 = match f, a, x2 with f, a, [] -> a
    | f, a, x :: xs -> foldl f (f a x) xs;;

type 'a par = Par of 'a;;

type 'a var = Var of 'a;;

type 'a label = Tau | Check | Action of 'a;;

type ('a, 'b, 'c, 'd) cspp = Omega | Lskip | Lstop | Ldiv | LProc_name
    of 'd |
  LAct_prefix of 'b * 'b * ('a, 'b, 'c, 'd) cspp |
  LExt_choice of ('a, 'b, 'c, 'd) cspp * ('a, 'b, 'c, 'd) cspp |
  LInt_choice of ('a, 'b, 'c, 'd) cspp * ('a, 'b, 'c, 'd) cspp |
  Lif of 'c * ('a, 'b, 'c, 'd) cspp * ('a, 'b, 'c, 'd) cspp |
  LParallel of ('a, 'b, 'c, 'd) cspp * 'a * 'a * 'c * ('a, 'b, 'c, 'd)
      cspp |
  LSeq_compo of ('a, 'b, 'c, 'd) cspp * ('a, 'b, 'c, 'd) cspp |
  LExt_pre_choice of 'b * 'a * 'c * ('a, 'b, 'c, 'd) cspp |
  LRep_int_choice of 'a * 'c * ('a, 'b, 'c, 'd) cspp |
  LHid of ('a, 'b, 'c, 'd) cspp * 'a * 'a * 'c;;

type ('a, 'b) actVal = Null | BasicVal of 'a | ChanVal of 'b | PropVal
    of bool |
  NumVal of int | PairVal of (('a, 'b) actVal * ('a, 'b) actVal) |
  ListVal of ('a, 'b) actVal list;;
```

```
type ('a, 'b, 'c) act = BasicAct of 'b | ActVal of ('b, 'c) actVal |
  IdP of 'a par | IdV of 'a var | Num of int |
  Plus of ('a, 'b, 'c) act * ('a, 'b, 'c) act |
  Minus of ('a, 'b, 'c) act * ('a, 'b, 'c) act | Nega of ('a, 'b, 'c)
      act |
  Times of ('a, 'b, 'c) act * ('a, 'b, 'c) act | Prop of ('a, 'b, 'c)
      propt |
  Nil | Cons of ('a, 'b, 'c) act * ('a, 'b, 'c) act | Hd of ('a, 'b, 'c)
      act |
  Tl of ('a, 'b, 'c) act | Append of ('a, 'b, 'c) act * ('a, 'b, 'c) act
      |
  Nth of ('a, 'b, 'c) act * ('a, 'b, 'c) act | Last of ('a, 'b, 'c) act |
  Chan of 'c | Pair of ('a, 'b, 'c) act * ('a, 'b, 'c) act |
  Fst of ('a, 'b, 'c) act | Snd of ('a, 'b, 'c) act
and ('a, 'b, 'c) propt = True_p | False_p |
  And of ('a, 'b, 'c) propt * ('a, 'b, 'c) propt |
  Or of ('a, 'b, 'c) propt * ('a, 'b, 'c) propt | Not of ('a, 'b, 'c)
      propt |
  Wf of ('a, 'b, 'c) propt | Equal of ('a, 'b, 'c) act * ('a, 'b, 'c)
      act |
  Less of ('a, 'b, 'c) act * ('a, 'b, 'c) act |
  Elt of ('a, 'b, 'c) act * ('a, 'b, 'c) actSet |
  SubSetEq of ('a, 'b, 'c) actSet * ('a, 'b, 'c) actSet |
  Mem of ('a, 'b, 'c) act * ('a, 'b, 'c) act |
  SubListEq of ('a, 'b, 'c) act * ('a, 'b, 'c) act
and ('a, 'b, 'c) actSet = ProptSet of 'a * ('a, 'b, 'c) propt |
  ActionSet of ('a, 'b, 'c) act list | ActListSet of ('a, 'b, 'c) act |
  InterSet of ('a, 'b, 'c) actSet * ('a, 'b, 'c) actSet |
  UnionSet of ('a, 'b, 'c) actSet * ('a, 'b, 'c) actSet;;

let rec equal_var _A (Var aa) (Var a) = eq _A aa a;;

let rec replaceVarInBool _A
  x0 s a = match x0, s, a with True_p, s, a -> True_p
    | False_p, s, a -> False_p
    | And (x, y), s, a ->
        And (replaceVarInBool _A x s a, replaceVarInBool _A y s a)
    | Or (x, y), s, a ->
        Or (replaceVarInBool _A x s a, replaceVarInBool _A y s a)
    | Not x, s, a -> Not (replaceVarInBool _A x s a)
    | Wf x, s, a -> Wf (replaceVarInBool _A x s a)
    | Equal (x, y), s, a ->
        Equal (replaceVarInAct _A x s a, replaceVarInAct _A y s a)
    | Less (x, y), s, a ->
        Less (replaceVarInAct _A x s a, replaceVarInAct _A y s a)
    | Elt (x, y), s, a ->
        Elt (replaceVarInAct _A x s a, replaceVarInActSet _A y s a)
```

```
    | SubSetEq (x, y), s, a ->
       SubSetEq (replaceVarInActSet _A x s a, replaceVarInActSet _A y s
           a)
    | Mem (x, y), s, a ->
       Mem (replaceVarInAct _A x s a, replaceVarInAct _A y s a)
    | SubListEq (x, y), s, a ->
       SubListEq (replaceVarInAct _A x s a, replaceVarInAct _A y s a)
and replaceVarInAct _A
  x0 s a = match x0, s, a with BasicAct n, s, a -> BasicAct n
    | Chan n, s, a -> Chan n
    | Num n, s, a -> Num n
    | ActVal n, s, a -> ActVal n
    | Hd aa, s, a -> Hd (replaceVarInAct _A aa s a)
    | Tl aa, s, a -> Tl (replaceVarInAct _A aa s a)
    | Last aa, s, a -> Last (replaceVarInAct _A aa s a)
    | IdP sa, s, a -> IdP sa
    | IdV sa, s, a -> (if equal_var _A sa (Var s) then a else IdV sa)
    | Prop p, s, a -> Prop (replaceVarInBool _A p s a)
    | Pair (x, y), s, a ->
       Pair (replaceVarInAct _A x s a, replaceVarInAct _A y s a)
    | Fst x, s, a -> Fst (replaceVarInAct _A x s a)
    | Snd x, s, a -> Snd (replaceVarInAct _A x s a)
    | Nil, s, a -> Nil
    | Cons (x, y), s, a ->
       Cons (replaceVarInAct _A x s a, replaceVarInAct _A y s a)
    | Append (x, y), s, a ->
       Append (replaceVarInAct _A x s a, replaceVarInAct _A y s a)
    | Nth (x, y), s, a ->
       Nth (replaceVarInAct _A x s a, replaceVarInAct _A y s a)
    | Plus (x, y), s, a ->
       Plus (replaceVarInAct _A x s a, replaceVarInAct _A y s a)
    | Minus (x, y), s, a ->
       Minus (replaceVarInAct _A x s a, replaceVarInAct _A y s a)
    | Times (x, y), s, a ->
       Times (replaceVarInAct _A x s a, replaceVarInAct _A y s a)
    | Nega x, s, a -> Nega (replaceVarInAct _A x s a)
and replaceVarInList _A
  x0 s a = match x0, s, a with [], s, a -> []
    | x :: xs, s, a -> replaceVarInAct _A x s a :: replaceVarInList _A
        xs s a
and replaceVarInActSet _A
  x0 s a = match x0, s, a with
    ProptSet (sa, p), s, a ->
      (if eq _A s sa then ProptSet (sa, p)
       else ProptSet (sa, replaceVarInBool _A p s a))
    | ActionSet alist, s, a -> ActionSet (replaceVarInList _A alist s a)
    | ActListSet aa, s, a -> ActListSet (replaceVarInAct _A aa s a)
```

```
    | InterSet (x, y), s, a ->
       InterSet (replaceVarInActSet _A x s a, replaceVarInActSet _A y s
           a)
    | UnionSet (x, y), s, a ->
       UnionSet (replaceVarInActSet _A x s a, replaceVarInActSet _A y s
           a);;

let rec oneStepEvalFun _B
  procEnv getNewVar replaceVarInCspp replaceVarInProcName
      replaceVarInDoubleBool
    varSet varEnv x7 = match
    procEnv, getNewVar, replaceVarInCspp, replaceVarInProcName,
      replaceVarInDoubleBool, varSet, varEnv, x7
    with
    procEnv, getNewVar, replaceVarInCspp, replaceVarInProcName,
      replaceVarInDoubleBool, varSet, varEnv, Omega
      -> []
    | procEnv, getNewVar, replaceVarInCspp, replaceVarInProcName,
       replaceVarInDoubleBool, varSet, varEnv, Lskip
       -> [(Check, (varSet, (varEnv, Omega)))]
    | procEnv, getNewVar, replaceVarInCspp, replaceVarInProcName,
       replaceVarInDoubleBool, varSet, varEnv, Ldiv
       -> [(Tau, (varSet, (varEnv, Ldiv)))]
    | procEnv, getNewVar, replaceVarInCspp, replaceVarInProcName,
       replaceVarInDoubleBool, varSet, varEnv, Lstop
       -> []
    | procEnv, getNewVar, replaceVarInCspp, replaceVarInProcName,
       replaceVarInDoubleBool, varSet, varEnv, LAct_prefix (c, a, p)
       -> [(Action (getNewVar varSet, getNewVar (getNewVar varSet ::
           varSet)),
            (getNewVar (getNewVar varSet :: varSet) ::
               getNewVar varSet :: varSet,
              (And (Equal (IdP (getNewVar varSet), c),
                    And (Equal (IdP (getNewVar (getNewVar varSet ::
                        varSet)),
                                a),
                         varEnv)),
               p)))]
    | procEnv, getNewVar, replaceVarInCspp, replaceVarInProcName,
       replaceVarInDoubleBool, varSet, varEnv, LInt_choice (p, q)
       -> [(Tau, (varSet, (varEnv, p))); (Tau, (varSet, (varEnv, q)))]
    | procEnv, getNewVar, replaceVarInCspp, replaceVarInProcName,
       replaceVarInDoubleBool, varSet, varEnv, LExt_choice (p, q)
       -> map (fun (a, (varSeta, (varEnva, pa))) ->
               (match a
                 with Tau -> (Tau, (varSeta, (varEnva, LExt_choice (pa,
                     q))))
```

```
              | Check -> (a, (varSeta, (varEnva, pa)))
              | Action _ -> (a, (varSeta, (varEnva, pa)))))
         (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
          replaceVarInProcName replaceVarInDoubleBool varSet varEnv
              p) @
         map (fun (a, (varSeta, (varEnva, qa))) ->
              (match a
               with Tau ->
                 (Tau, (varSeta, (varEnva, LExt_choice (p, qa))))
               | Check -> (a, (varSeta, (varEnva, qa)))
               | Action _ -> (a, (varSeta, (varEnva, qa)))))
          (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
           replaceVarInProcName replaceVarInDoubleBool varSet
               varEnv q)
| procEnv, getNewVar, replaceVarInCspp, replaceVarInProcName,
   replaceVarInDoubleBool, varSet, varEnv, LSeq_compo (p, q)
   -> map (fun (a, (varSeta, (varEnva, pa))) ->
           (match a
            with Tau -> (a, (varSeta, (varEnva, LSeq_compo (pa,
                q))))
            | Check -> (Tau, (varSeta, (varEnva, q)))
            | Action _ -> (a, (varSeta, (varEnva, LSeq_compo (pa,
                q))))))
        (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
         replaceVarInProcName replaceVarInDoubleBool varSet varEnv
            p)
| procEnv, getNewVar, replaceVarInCspp, replaceVarInProcName,
   replaceVarInDoubleBool, varSet, varEnv, Lif (b, p, q)
   -> [(Tau, (varSet, (And (b, varEnv), p)));
       (Tau, (varSet, (And (And (Not b, Wf b), varEnv), q)))]
| procEnv, getNewVar, replaceVarInCspp, replaceVarInProcName,
   replaceVarInDoubleBool, varSet, varEnv, LProc_name p
   -> [(Tau, (varSet, (varEnv, procEnv p)))]
| procEnv, getNewVar, replaceVarInCspp, replaceVarInProcName,
   replaceVarInDoubleBool, varSet, varEnv, LExt_pre_choice (c, s,
       pa, p)
   -> [(Action (getNewVar varSet, getNewVar (getNewVar varSet ::
       varSet)),
        (getNewVar (getNewVar varSet :: varSet) ::
          getNewVar varSet :: varSet,
         (And (Equal (IdP (getNewVar varSet), c),
               And (varEnv,
                    replaceVarInBool _B pa s
                      (IdP (getNewVar (getNewVar varSet ::
                          varSet))))),
          replaceVarInCspp replaceVarInProcName p s
            (IdP (getNewVar (getNewVar varSet :: varSet))))))]
```

```
  | procEnv, getNewVar, replaceVarInCspp, replaceVarInProcName,
     replaceVarInDoubleBool, varSet, varEnv, LRep_int_choice (s, pa,
         p)
     -> [(Tau, (getNewVar varSet :: varSet,
                (And (varEnv,
                      replaceVarInBool _B pa s (IdP (getNewVar
                          varSet))),
                 replaceVarInCspp replaceVarInProcName p s
                  (IdP (getNewVar varSet)))))]
  | procEnv, getNewVar, replaceVarInCspp, replaceVarInProcName,
     replaceVarInDoubleBool, varSet, varEnv, LParallel (p, sa, s, x,
         q)
     -> (match (p, q)
          with (Omega, Omega) -> [(Check, (varSet, (varEnv, Omega)))]
          | (Omega, Lskip) ->
            foldl (fun col a ->
                   (match a
                    with (Tau, (varSeta, (varEnva, pa))) ->
                      (Tau, (varSeta,
                             (varEnva, LParallel (pa, sa, s, x, q))))
                                 ::
                        col
                    | (Check, (varSeta, (varEnva, pa))) ->
                      (Tau, (varSeta,
                             (varEnva,
                              LParallel (Omega, sa, s, x, q)))) ::
                        col
                    | (Action (c, aa), (varSeta, (varEnva, pa))) ->
                      foldl (fun cola b ->
                             (match b
                              with (Tau, (varSetb, (varEnvb, qa))) ->
                                cola
                              | (Check, (varSetb, (varEnvb, qa))) ->
                                  cola
                              | (Action (ca, aaa),
(varSetb, (varEnvb, qa)))
                                -> (Action (c, aa),
   (varSetb,
     (And (And (Equal (IdP aa, IdP aaa),
                And (varEnvb,
                     replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
           Equal (IdP c, IdP ca)),
      LParallel (pa, sa, s, x, qa)))) ::
   cola))
                        [] (oneStepEvalFun _B procEnv getNewVar
                            replaceVarInCspp replaceVarInProcName
```

```
                                replaceVarInDoubleBool varSeta varEnva q)
                                    @
                               (Action (c, aa),
                                 (varSeta,
                                   (And (varEnva,
Not (replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                                     LParallel (pa, sa, s, x, q)))) ::
                                col))
                    [] (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                        replaceVarInProcName replaceVarInDoubleBool varSet
                            varEnv
                        p) @
                    map (fun a ->
                         (match a
                           with (Tau, (varSeta, (varEnva, qa))) ->
                             (Tau, (varSeta,
                                    (varEnva, LParallel (p, sa, s, x, qa))))
                           | (Check, (varSeta, (varEnva, qa))) ->
                             (Tau, (varSeta,
                                    (varEnva, LParallel (p, sa, s, x,
                                        Omega))))
                           | (Action (c, aa), (varSeta, (varEnva, qa))) ->
                             (Action (c, aa),
                               (varSeta,
                                 (And (varEnva,
                                       Not
(replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                                   LParallel (p, sa, s, x, qa))))))
                      (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                       replaceVarInProcName replaceVarInDoubleBool varSet
                           varEnv
                       q)
                | (Omega, Lstop) ->
                  foldl (fun col a ->
                         (match a
                           with (Tau, (varSeta, (varEnva, pa))) ->
                             (Tau, (varSeta,
                                    (varEnva, LParallel (pa, sa, s, x, q))))
                                        ::
                               col
                           | (Check, (varSeta, (varEnva, pa))) ->
                             (Tau, (varSeta,
                                    (varEnva,
                                      LParallel (Omega, sa, s, x, q)))) ::
                               col
                           | (Action (c, aa), (varSeta, (varEnva, pa))) ->
                             foldl (fun cola b ->
```

```
                                (match b
                                  with (Tau, (varSetb, (varEnvb, qa))) ->
                                    cola
                                  | (Check, (varSetb, (varEnvb, qa))) ->
                                      cola
                                  | (Action (ca, aaa),
 (varSetb, (varEnvb, qa)))
                                    -> (Action (c, aa),
    (varSetb,
      (And (And (Equal (IdP aa, IdP aaa),
                 And (varEnvb,
                      replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
            Equal (IdP c, IdP ca)),
       LParallel (pa, sa, s, x, qa)))) ::
    cola))
                           [] (oneStepEvalFun _B procEnv getNewVar
                               replaceVarInCspp replaceVarInProcName
                               replaceVarInDoubleBool varSeta varEnva q)
                                   @
                           (Action (c, aa),
                             (varSeta,
                               (And (varEnva,
Not (replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                                  LParallel (pa, sa, s, x, q)))) ::
                             col))
                 [] (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                     replaceVarInProcName replaceVarInDoubleBool varSet
                         varEnv
                     p) @
                 map (fun a ->
                       (match a
                         with (Tau, (varSeta, (varEnva, qa))) ->
                           (Tau, (varSeta,
                                  (varEnva, LParallel (p, sa, s, x, qa))))
                         | (Check, (varSeta, (varEnva, qa))) ->
                           (Tau, (varSeta,
                                  (varEnva, LParallel (p, sa, s, x,
                                      Omega))))
                         | (Action (c, aa), (varSeta, (varEnva, qa))) ->
                           (Action (c, aa),
                             (varSeta,
                               (And (varEnva,
                                     Not
(replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                                 LParallel (p, sa, s, x, qa))))))
                  (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
```

```
                    replaceVarInProcName replaceVarInDoubleBool varSet
                        varEnv
                    q)
              | (Omega, Ldiv) ->
                foldl (fun col a ->
                       (match a
                        with (Tau, (varSeta, (varEnva, pa))) ->
                          (Tau, (varSeta,
                                 (varEnva, LParallel (pa, sa, s, x, q))))
                                     ::
                            col
                        | (Check, (varSeta, (varEnva, pa))) ->
                          (Tau, (varSeta,
                                 (varEnva,
                                  LParallel (Omega, sa, s, x, q)))) ::
                            col
                        | (Action (c, aa), (varSeta, (varEnva, pa))) ->
                          foldl (fun cola b ->
                                 (match b
                                  with (Tau, (varSetb, (varEnvb, qa))) ->
                                    cola
                                  | (Check, (varSetb, (varEnvb, qa))) ->
                                      cola
                                  | (Action (ca, aaa),
 (varSetb, (varEnvb, qa)))
                                    -> (Action (c, aa),
    (varSetb,
      (And (And (Equal (IdP aa, IdP aaa),
                 And (varEnvb,
                      replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
            Equal (IdP c, IdP ca)),
       LParallel (pa, sa, s, x, qa)))) ::
    cola))
                            [] (oneStepEvalFun _B procEnv getNewVar
                                replaceVarInCspp replaceVarInProcName
                                replaceVarInDoubleBool varSeta varEnva q)
                                    @
                            (Action (c, aa),
                              (varSeta,
                                (And (varEnva,
 Not (replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                                 LParallel (pa, sa, s, x, q)))) ::
                              col))
                  [] (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                      replaceVarInProcName replaceVarInDoubleBool varSet
                          varEnv
                      p) @
```

```
                map (fun a ->
                      (match a
                       with (Tau, (varSeta, (varEnva, qa))) ->
                         (Tau, (varSeta,
                                (varEnva, LParallel (p, sa, s, x, qa))))
                       | (Check, (varSeta, (varEnva, qa))) ->
                         (Tau, (varSeta,
                                (varEnva, LParallel (p, sa, s, x,
                                    Omega))))
                       | (Action (c, aa), (varSeta, (varEnva, qa))) ->
                         (Action (c, aa),
                           (varSeta,
                             (And (varEnva,
                                   Not
(replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                                LParallel (p, sa, s, x, qa))))))
                  (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                   replaceVarInProcName replaceVarInDoubleBool varSet
                       varEnv
                   q)
              | (Omega, LProc_name _) ->
                foldl (fun col a ->
                       (match a
                        with (Tau, (varSeta, (varEnva, pa))) ->
                          (Tau, (varSeta,
                                 (varEnva, LParallel (pa, sa, s, x, q))))
                                     ::
                            col
                        | (Check, (varSeta, (varEnva, pa))) ->
                          (Tau, (varSeta,
                                 (varEnva,
                                   LParallel (Omega, sa, s, x, q)))) ::
                            col
                        | (Action (c, aa), (varSeta, (varEnva, pa))) ->
                          foldl (fun cola b ->
                                 (match b
                                   with (Tau, (varSetb, (varEnvb, qa))) ->
                                     cola
                                   | (Check, (varSetb, (varEnvb, qa))) ->
                                       cola
                                   | (Action (ca, aaa),
 (varSetb, (varEnvb, qa)))
                                     -> (Action (c, aa),
    (varSetb,
      (And (And (Equal (IdP aa, IdP aaa),
                 And (varEnvb,
                      replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
```

```
            Equal (IdP c, IdP ca)),
        LParallel (pa, sa, s, x, qa)))) ::
    cola))
                          [] (oneStepEvalFun _B procEnv getNewVar
                              replaceVarInCspp replaceVarInProcName
                              replaceVarInDoubleBool varSeta varEnva q)
                                  @
                          (Action (c, aa),
                            (varSeta,
                              (And (varEnva,
Not (replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                                 LParallel (pa, sa, s, x, q)))) ::
                            col))
                 [] (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                     replaceVarInProcName replaceVarInDoubleBool varSet
                         varEnv
                     p) @
                 map (fun a ->
                       (match a
                         with (Tau, (varSeta, (varEnva, qa))) ->
                           (Tau, (varSeta,
                                  (varEnva, LParallel (p, sa, s, x, qa))))
                        | (Check, (varSeta, (varEnva, qa))) ->
                          (Tau, (varSeta,
                                  (varEnva, LParallel (p, sa, s, x,
                                      Omega))))
                        | (Action (c, aa), (varSeta, (varEnva, qa))) ->
                          (Action (c, aa),
                            (varSeta,
                              (And (varEnva,
                                    Not
(replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                                LParallel (p, sa, s, x, qa))))))
                   (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                    replaceVarInProcName replaceVarInDoubleBool varSet
                        varEnv
                    q)
             | (Omega, LAct_prefix (_, _, _)) ->
               foldl (fun col a ->
                       (match a
                         with (Tau, (varSeta, (varEnva, pa))) ->
                           (Tau, (varSeta,
                                  (varEnva, LParallel (pa, sa, s, x, q))))
                                      ::
                            col
                        | (Check, (varSeta, (varEnva, pa))) ->
                          (Tau, (varSeta,
```

```
                                (varEnva,
                                  LParallel (Omega, sa, s, x, q)))) ::
                            col
                        | (Action (c, aa), (varSeta, (varEnva, pa))) ->
                          foldl (fun cola b ->
                                (match b
                                  with (Tau, (varSetb, (varEnvb, qa))) ->
                                    cola
                                  | (Check, (varSetb, (varEnvb, qa))) ->
                                      cola
                                  | (Action (ca, aaa),
 (varSetb, (varEnvb, qa)))
                                    -> (Action (c, aa),
    (varSetb,
      (And (And (Equal (IdP aa, IdP aaa),
                 And (varEnvb,
                      replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
            Equal (IdP c, IdP ca)),
       LParallel (pa, sa, s, x, qa)))) ::
    cola))
                          [] (oneStepEvalFun _B procEnv getNewVar
                              replaceVarInCspp replaceVarInProcName
                              replaceVarInDoubleBool varSeta varEnva q)
                                  @
                          (Action (c, aa),
                            (varSeta,
                              (And (varEnva,
Not (replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                                 LParallel (pa, sa, s, x, q)))) ::
                            col))
                [] (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                    replaceVarInProcName replaceVarInDoubleBool varSet
                        varEnv
                    p) @
                map (fun a ->
                     (match a
                       with (Tau, (varSeta, (varEnva, qa))) ->
                         (Tau, (varSeta,
                                (varEnva, LParallel (p, sa, s, x, qa))))
                       | (Check, (varSeta, (varEnva, qa))) ->
                         (Tau, (varSeta,
                                (varEnva, LParallel (p, sa, s, x,
                                    Omega))))
                       | (Action (c, aa), (varSeta, (varEnva, qa))) ->
                         (Action (c, aa),
                           (varSeta,
                             (And (varEnva,
```

```
                                  Not
(replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                              LParallel (p, sa, s, x, qa))))))
                (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                 replaceVarInProcName replaceVarInDoubleBool varSet
                     varEnv
                 q)
           | (Omega, LExt_choice (_, _)) ->
            foldl (fun col a ->
                   (match a
                    with (Tau, (varSeta, (varEnva, pa))) ->
                      (Tau, (varSeta,
                             (varEnva, LParallel (pa, sa, s, x, q))))
                          ::
                        col
                    | (Check, (varSeta, (varEnva, pa))) ->
                      (Tau, (varSeta,
                             (varEnva,
                              LParallel (Omega, sa, s, x, q)))) ::
                        col
                    | (Action (c, aa), (varSeta, (varEnva, pa))) ->
                      foldl (fun cola b ->
                             (match b
                              with (Tau, (varSetb, (varEnvb, qa))) ->
                                cola
                              | (Check, (varSetb, (varEnvb, qa))) ->
                                  cola
                              | (Action (ca, aaa),
 (varSetb, (varEnvb, qa)))
                                -> (Action (c, aa),
   (varSetb,
     (And (And (Equal (IdP aa, IdP aaa),
                And (varEnvb,
                     replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
           Equal (IdP c, IdP ca)),
       LParallel (pa, sa, s, x, qa)))) ::
   cola))
                        [] (oneStepEvalFun _B procEnv getNewVar
                            replaceVarInCspp replaceVarInProcName
                            replaceVarInDoubleBool varSeta varEnva q)
                               @
                        (Action (c, aa),
                          (varSeta,
                            (And (varEnva,
Not (replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                               LParallel (pa, sa, s, x, q)))) ::
                          col))
```

```
                [] (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                    replaceVarInProcName replaceVarInDoubleBool varSet
                        varEnv
                    p) @
                map (fun a ->
                     (match a
                       with (Tau, (varSeta, (varEnva, qa))) ->
                         (Tau, (varSeta,
                                (varEnva, LParallel (p, sa, s, x, qa))))
                       | (Check, (varSeta, (varEnva, qa))) ->
                         (Tau, (varSeta,
                                (varEnva, LParallel (p, sa, s, x,
                                    Omega))))
                       | (Action (c, aa), (varSeta, (varEnva, qa))) ->
                         (Action (c, aa),
                           (varSeta,
                             (And (varEnva,
                                   Not
(replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                              LParallel (p, sa, s, x, qa))))))
                  (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                   replaceVarInProcName replaceVarInDoubleBool varSet
                       varEnv
                   q)
              | (Omega, LInt_choice (_, _)) ->
                foldl (fun col a ->
                       (match a
                         with (Tau, (varSeta, (varEnva, pa))) ->
                           (Tau, (varSeta,
                                  (varEnva, LParallel (pa, sa, s, x, q))))
                                      ::
                             col
                         | (Check, (varSeta, (varEnva, pa))) ->
                           (Tau, (varSeta,
                                  (varEnva,
                                    LParallel (Omega, sa, s, x, q)))) ::
                             col
                         | (Action (c, aa), (varSeta, (varEnva, pa))) ->
                           foldl (fun cola b ->
                                  (match b
                                    with (Tau, (varSetb, (varEnvb, qa))) ->
                                      cola
                                    | (Check, (varSetb, (varEnvb, qa))) ->
                                        cola
                                    | (Action (ca, aaa),
 (varSetb, (varEnvb, qa)))
                                      -> (Action (c, aa),
```

```
    (varSetb,
      (And (And (Equal (IdP aa, IdP aaa),
                 And (varEnvb,
                      replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
            Equal (IdP c, IdP ca)),
       LParallel (pa, sa, s, x, qa)))) ::
    cola))
                          [] (oneStepEvalFun _B procEnv getNewVar
                              replaceVarInCspp replaceVarInProcName
                              replaceVarInDoubleBool varSeta varEnva q)
                                  @
                          (Action (c, aa),
                            (varSeta,
                              (And (varEnva,
Not (replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                                LParallel (pa, sa, s, x, q)))) ::
                            col))
                [] (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                    replaceVarInProcName replaceVarInDoubleBool varSet
                        varEnv
                    p) @
                map (fun a ->
                     (match a
                      with (Tau, (varSeta, (varEnva, qa))) ->
                        (Tau, (varSeta,
                               (varEnva, LParallel (p, sa, s, x, qa))))
                      | (Check, (varSeta, (varEnva, qa))) ->
                        (Tau, (varSeta,
                               (varEnva, LParallel (p, sa, s, x,
                                   Omega))))
                      | (Action (c, aa), (varSeta, (varEnva, qa))) ->
                        (Action (c, aa),
                          (varSeta,
                            (And (varEnva,
                                  Not
(replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                              LParallel (p, sa, s, x, qa))))))
                  (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                   replaceVarInProcName replaceVarInDoubleBool varSet
                       varEnv
                   q)
            | (Omega, Lif (_, _, _)) ->
              foldl (fun col a ->
                     (match a
                      with (Tau, (varSeta, (varEnva, pa))) ->
                        (Tau, (varSeta,
```

```
                                (varEnva, LParallel (pa, sa, s, x, q))))
                                    ::
                              col
                          | (Check, (varSeta, (varEnva, pa))) ->
                            (Tau, (varSeta,
                                   (varEnva,
                                    LParallel (Omega, sa, s, x, q)))) ::
                              col
                          | (Action (c, aa), (varSeta, (varEnva, pa))) ->
                            foldl (fun cola b ->
                                   (match b
                                    with (Tau, (varSetb, (varEnvb, qa))) ->
                                      cola
                                    | (Check, (varSetb, (varEnvb, qa))) ->
                                        cola
                                    | (Action (ca, aaa),
 (varSetb, (varEnvb, qa)))
                                      -> (Action (c, aa),
    (varSetb,
      (And (And (Equal (IdP aa, IdP aaa),
                 And (varEnvb,
                      replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
            Equal (IdP c, IdP ca)),
       LParallel (pa, sa, s, x, qa)))) ::
    cola))
                            [] (oneStepEvalFun _B procEnv getNewVar
                                replaceVarInCspp replaceVarInProcName
                                replaceVarInDoubleBool varSeta varEnva q)
                                    @
                            (Action (c, aa),
                              (varSeta,
                                (And (varEnva,
 Not (replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                                   LParallel (pa, sa, s, x, q)))) ::
                              col))
                  [] (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                      replaceVarInProcName replaceVarInDoubleBool varSet
                          varEnv
                      p) @
                  map (fun a ->
                       (match a
                        with (Tau, (varSeta, (varEnva, qa))) ->
                          (Tau, (varSeta,
                                 (varEnva, LParallel (p, sa, s, x, qa))))
                        | (Check, (varSeta, (varEnva, qa))) ->
                          (Tau, (varSeta,
```

```
                                (varEnva, LParallel (p, sa, s, x,
                                    Omega))))
                          | (Action (c, aa), (varSeta, (varEnva, qa))) ->
                            (Action (c, aa),
                              (varSeta,
                                (And (varEnva,
                                      Not
(replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                                 LParallel (p, sa, s, x, qa))))))
                  (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                    replaceVarInProcName replaceVarInDoubleBool varSet
                      varEnv
                    q)
              | (Omega, LParallel (_, _, _, _, _)) ->
                foldl (fun col a ->
                       (match a
                        with (Tau, (varSeta, (varEnva, pa))) ->
                          (Tau, (varSeta,
                                 (varEnva, LParallel (pa, sa, s, x, q))))
                                     ::
                            col
                        | (Check, (varSeta, (varEnva, pa))) ->
                          (Tau, (varSeta,
                                 (varEnva,
                                   LParallel (Omega, sa, s, x, q)))) ::
                            col
                        | (Action (c, aa), (varSeta, (varEnva, pa))) ->
                          foldl (fun cola b ->
                                 (match b
                                   with (Tau, (varSetb, (varEnvb, qa))) ->
                                     cola
                                   | (Check, (varSetb, (varEnvb, qa))) ->
                                       cola
                                   | (Action (ca, aaa),
(varSetb, (varEnvb, qa)))
                                     -> (Action (c, aa),
   (varSetb,
     (And (And (Equal (IdP aa, IdP aaa),
                And (varEnvb,
                     replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
           Equal (IdP c, IdP ca)),
      LParallel (pa, sa, s, x, qa)))) ::
   cola))
                            [] (oneStepEvalFun _B procEnv getNewVar
                                replaceVarInCspp replaceVarInProcName
                                replaceVarInDoubleBool varSeta varEnva q)
                                    @
```

```
                          (Action (c, aa),
                            (varSeta,
                              (And (varEnva,
Not (replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                                 LParallel (pa, sa, s, x, q)))) ::
                            col))
                [] (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                    replaceVarInProcName replaceVarInDoubleBool varSet
                        varEnv
                    p) @
                map (fun a ->
                     (match a
                      with (Tau, (varSeta, (varEnva, qa))) ->
                        (Tau, (varSeta,
                               (varEnva, LParallel (p, sa, s, x, qa))))
                      | (Check, (varSeta, (varEnva, qa))) ->
                        (Tau, (varSeta,
                               (varEnva, LParallel (p, sa, s, x,
                                   Omega))))
                      | (Action (c, aa), (varSeta, (varEnva, qa))) ->
                        (Action (c, aa),
                          (varSeta,
                            (And (varEnva,
                                  Not
(replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                               LParallel (p, sa, s, x, qa))))))
                  (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                   replaceVarInProcName replaceVarInDoubleBool varSet
                       varEnv
                   q)
              | (Omega, LSeq_compo (_, _)) ->
                foldl (fun col a ->
                       (match a
                        with (Tau, (varSeta, (varEnva, pa))) ->
                          (Tau, (varSeta,
                                 (varEnva, LParallel (pa, sa, s, x, q))))
                                     ::
                            col
                        | (Check, (varSeta, (varEnva, pa))) ->
                          (Tau, (varSeta,
                                 (varEnva,
                                   LParallel (Omega, sa, s, x, q)))) ::
                            col
                        | (Action (c, aa), (varSeta, (varEnva, pa))) ->
                          foldl (fun cola b ->
                                 (match b
                                   with (Tau, (varSetb, (varEnvb, qa))) ->
```

```
                                  cola
                                | (Check, (varSetb, (varEnvb, qa))) ->
                                    cola
                                | (Action (ca, aaa),
 (varSetb, (varEnvb, qa)))
                                  -> (Action (c, aa),
    (varSetb,
      (And (And (Equal (IdP aa, IdP aaa),
                 And (varEnvb,
                      replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
            Equal (IdP c, IdP ca)),
       LParallel (pa, sa, s, x, qa)))) ::
    cola))
                          [] (oneStepEvalFun _B procEnv getNewVar
                              replaceVarInCspp replaceVarInProcName
                              replaceVarInDoubleBool varSeta varEnva q)
                                  @
                          (Action (c, aa),
                            (varSeta,
                              (And (varEnva,
Not (replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                                 LParallel (pa, sa, s, x, q)))) ::
                            col))
                 [] (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                     replaceVarInProcName replaceVarInDoubleBool varSet
                         varEnv
                     p) @
                 map (fun a ->
                       (match a
                         with (Tau, (varSeta, (varEnva, qa))) ->
                           (Tau, (varSeta,
                                  (varEnva, LParallel (p, sa, s, x, qa))))
                        | (Check, (varSeta, (varEnva, qa))) ->
                          (Tau, (varSeta,
                                  (varEnva, LParallel (p, sa, s, x,
                                      Omega))))
                        | (Action (c, aa), (varSeta, (varEnva, qa))) ->
                          (Action (c, aa),
                            (varSeta,
                              (And (varEnva,
                                    Not
(replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                                LParallel (p, sa, s, x, qa))))))
                   (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                    replaceVarInProcName replaceVarInDoubleBool varSet
                        varEnv
                    q)
```

```
          | (Omega, LExt_pre_choice (_, _, _, _)) ->
            foldl (fun col a ->
                   (match a
                    with (Tau, (varSeta, (varEnva, pa))) ->
                      (Tau, (varSeta,
                             (varEnva, LParallel (pa, sa, s, x, q))))
                          ::
                        col
                    | (Check, (varSeta, (varEnva, pa))) ->
                      (Tau, (varSeta,
                             (varEnva,
                              LParallel (Omega, sa, s, x, q)))) ::
                        col
                    | (Action (c, aa), (varSeta, (varEnva, pa))) ->
                      foldl (fun cola b ->
                             (match b
                              with (Tau, (varSetb, (varEnvb, qa))) ->
                                cola
                              | (Check, (varSetb, (varEnvb, qa))) ->
                                  cola
                              | (Action (ca, aaa),
 (varSetb, (varEnvb, qa)))
                                -> (Action (c, aa),
    (varSetb,
      (And (And (Equal (IdP aa, IdP aaa),
                 And (varEnvb,
                      replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
            Equal (IdP c, IdP ca)),
       LParallel (pa, sa, s, x, qa)))) ::
    cola))
                        [] (oneStepEvalFun _B procEnv getNewVar
                            replaceVarInCspp replaceVarInProcName
                            replaceVarInDoubleBool varSeta varEnva q)
                               @
                        (Action (c, aa),
                          (varSeta,
                            (And (varEnva,
Not (replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                             LParallel (pa, sa, s, x, q)))) ::
                          col))
               [] (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                   replaceVarInProcName replaceVarInDoubleBool varSet
                      varEnv
                   p) @
              map (fun a ->
                   (match a
                    with (Tau, (varSeta, (varEnva, qa))) ->
```

```
                        (Tau, (varSeta,
                               (varEnva, LParallel (p, sa, s, x, qa))))
                      | (Check, (varSeta, (varEnva, qa))) ->
                        (Tau, (varSeta,
                               (varEnva, LParallel (p, sa, s, x,
                                   Omega))))
                      | (Action (c, aa), (varSeta, (varEnva, qa))) ->
                        (Action (c, aa),
                          (varSeta,
                            (And (varEnva,
                                  Not
(replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                             LParallel (p, sa, s, x, qa))))))
                (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                 replaceVarInProcName replaceVarInDoubleBool varSet
                     varEnv
                 q)
            | (Omega, LRep_int_choice (_, _, _)) ->
              foldl (fun col a ->
                     (match a
                      with (Tau, (varSeta, (varEnva, pa))) ->
                        (Tau, (varSeta,
                               (varEnva, LParallel (pa, sa, s, x, q))))
                                   ::
                          col
                      | (Check, (varSeta, (varEnva, pa))) ->
                        (Tau, (varSeta,
                               (varEnva,
                                LParallel (Omega, sa, s, x, q)))) ::
                          col
                      | (Action (c, aa), (varSeta, (varEnva, pa))) ->
                        foldl (fun cola b ->
                               (match b
                                with (Tau, (varSetb, (varEnvb, qa))) ->
                                  cola
                                | (Check, (varSetb, (varEnvb, qa))) ->
                                    cola
                                | (Action (ca, aaa),
 (varSetb, (varEnvb, qa)))
                                  -> (Action (c, aa),
   (varSetb,
     (And (And (Equal (IdP aa, IdP aaa),
                And (varEnvb,
                     replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
           Equal (IdP c, IdP ca)),
      LParallel (pa, sa, s, x, qa)))) ::
   cola))
```

```
                        [] (oneStepEvalFun _B procEnv getNewVar
                            replaceVarInCspp replaceVarInProcName
                            replaceVarInDoubleBool varSeta varEnva q)
                                @
                        (Action (c, aa),
                          (varSeta,
                            (And (varEnva,
Not (replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                             LParallel (pa, sa, s, x, q)))) ::
                          col))
               [] (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                   replaceVarInProcName replaceVarInDoubleBool varSet
                       varEnv
                   p) @
              map (fun a ->
                    (match a
                      with (Tau, (varSeta, (varEnva, qa))) ->
                        (Tau, (varSeta,
                               (varEnva, LParallel (p, sa, s, x, qa))))
                      | (Check, (varSeta, (varEnva, qa))) ->
                        (Tau, (varSeta,
                               (varEnva, LParallel (p, sa, s, x,
                                   Omega))))
                      | (Action (c, aa), (varSeta, (varEnva, qa))) ->
                        (Action (c, aa),
                          (varSeta,
                            (And (varEnva,
                                  Not
(replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                             LParallel (p, sa, s, x, qa))))))
                (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                 replaceVarInProcName replaceVarInDoubleBool varSet
                     varEnv
                 q)
            | (Omega, LHid (_, _, _, _)) ->
              foldl (fun col a ->
                     (match a
                       with (Tau, (varSeta, (varEnva, pa))) ->
                         (Tau, (varSeta,
                                (varEnva, LParallel (pa, sa, s, x, q))))
                                    ::
                           col
                      | (Check, (varSeta, (varEnva, pa))) ->
                        (Tau, (varSeta,
                               (varEnva,
                                 LParallel (Omega, sa, s, x, q)))) ::
                          col
```

```
                    | (Action (c, aa), (varSeta, (varEnva, pa))) ->
                      foldl (fun cola b ->
                             (match b
                               with (Tau, (varSetb, (varEnvb, qa))) ->
                                 cola
                               | (Check, (varSetb, (varEnvb, qa))) ->
                                   cola
                               | (Action (ca, aaa),
 (varSetb, (varEnvb, qa)))
                                 -> (Action (c, aa),
    (varSetb,
      (And (And (Equal (IdP aa, IdP aaa),
                 And (varEnvb,
                      replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
           Equal (IdP c, IdP ca)),
        LParallel (pa, sa, s, x, qa)))) ::
    cola))
                        [] (oneStepEvalFun _B procEnv getNewVar
                            replaceVarInCspp replaceVarInProcName
                            replaceVarInDoubleBool varSeta varEnva q)
                               @
                        (Action (c, aa),
                          (varSeta,
                            (And (varEnva,
Not (replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                             LParallel (pa, sa, s, x, q)))) ::
                          col))
                [] (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                    replaceVarInProcName replaceVarInDoubleBool varSet
                        varEnv
                    p) @
                map (fun a ->
                     (match a
                       with (Tau, (varSeta, (varEnva, qa))) ->
                         (Tau, (varSeta,
                                (varEnva, LParallel (p, sa, s, x, qa))))
                       | (Check, (varSeta, (varEnva, qa))) ->
                         (Tau, (varSeta,
                                (varEnva, LParallel (p, sa, s, x,
                                    Omega))))
                       | (Action (c, aa), (varSeta, (varEnva, qa))) ->
                         (Action (c, aa),
                           (varSeta,
                             (And (varEnva,
                                   Not
(replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                              LParallel (p, sa, s, x, qa))))))
```

```
                (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                 replaceVarInProcName replaceVarInDoubleBool varSet
                     varEnv
                 q)
           | (Lskip, b) ->
             foldl (fun col a ->
                    (match a
                     with (Tau, (varSeta, (varEnva, pa))) ->
                       (Tau, (varSeta,
                              (varEnva, LParallel (pa, sa, s, x, q))))
                                  ::
                         col
                     | (Check, (varSeta, (varEnva, pa))) ->
                       (Tau, (varSeta,
                              (varEnva,
                               LParallel (Omega, sa, s, x, q)))) ::
                         col
                     | (Action (c, aa), (varSeta, (varEnva, pa))) ->
                       foldl (fun cola ba ->
                              (match ba
                               with (Tau, (varSetb, (varEnvb, qa))) ->
                                 cola
                               | (Check, (varSetb, (varEnvb, qa))) ->
                                   cola
                               | (Action (ca, aaa),
 (varSetb, (varEnvb, qa)))
                                 -> (Action (c, aa),
    (varSetb,
      (And (And (Equal (IdP aa, IdP aaa),
                 And (varEnvb,
                      replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
            Equal (IdP c, IdP ca)),
       LParallel (pa, sa, s, x, qa)))) ::
    cola))
                         [] (oneStepEvalFun _B procEnv getNewVar
                             replaceVarInCspp replaceVarInProcName
                             replaceVarInDoubleBool varSeta varEnva q)
                                 @
                         (Action (c, aa),
                           (varSeta,
                             (And (varEnva,
 Not (replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                              LParallel (pa, sa, s, x, q)))) ::
                          col))
               [] (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                   replaceVarInProcName replaceVarInDoubleBool varSet
                       varEnv
```

```
                    p) @
                map (fun a ->
                     (match a
                       with (Tau, (varSeta, (varEnva, qa))) ->
                         (Tau, (varSeta,
                                (varEnva, LParallel (p, sa, s, x, qa))))
                       | (Check, (varSeta, (varEnva, qa))) ->
                         (Tau, (varSeta,
                                (varEnva, LParallel (p, sa, s, x,
                                    Omega))))
                       | (Action (c, aa), (varSeta, (varEnva, qa))) ->
                         (Action (c, aa),
                           (varSeta,
                             (And (varEnva,
                                   Not
(replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                                LParallel (p, sa, s, x, qa))))))
                (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                 replaceVarInProcName replaceVarInDoubleBool varSet
                     varEnv
                q)
            | (Lstop, b) ->
              foldl (fun col a ->
                     (match a
                       with (Tau, (varSeta, (varEnva, pa))) ->
                         (Tau, (varSeta,
                                (varEnva, LParallel (pa, sa, s, x, q))))
                                    ::
                           col
                       | (Check, (varSeta, (varEnva, pa))) ->
                         (Tau, (varSeta,
                                (varEnva,
                                  LParallel (Omega, sa, s, x, q)))) ::
                           col
                       | (Action (c, aa), (varSeta, (varEnva, pa))) ->
                         foldl (fun cola ba ->
                                (match ba
                                  with (Tau, (varSetb, (varEnvb, qa))) ->
                                    cola
                                  | (Check, (varSetb, (varEnvb, qa))) ->
                                      cola
                                  | (Action (ca, aaa),
 (varSetb, (varEnvb, qa)))
                                    -> (Action (c, aa),
   (varSetb,
     (And (And (Equal (IdP aa, IdP aaa),
                And (varEnvb,
```

```
                    replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
          Equal (IdP c, IdP ca)),
       LParallel (pa, sa, s, x, qa)))) ::
   cola))
                       [] (oneStepEvalFun _B procEnv getNewVar
                           replaceVarInCspp replaceVarInProcName
                           replaceVarInDoubleBool varSeta varEnva q)
                               @
                       (Action (c, aa),
                         (varSeta,
                           (And (varEnva,
Not (replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                            LParallel (pa, sa, s, x, q)))) ::
                          col))
              [] (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                  replaceVarInProcName replaceVarInDoubleBool varSet
                      varEnv
                  p) @
              map (fun a ->
                   (match a
                     with (Tau, (varSeta, (varEnva, qa))) ->
                       (Tau, (varSeta,
                              (varEnva, LParallel (p, sa, s, x, qa))))
                     | (Check, (varSeta, (varEnva, qa))) ->
                       (Tau, (varSeta,
                              (varEnva, LParallel (p, sa, s, x,
                                  Omega))))
                     | (Action (c, aa), (varSeta, (varEnva, qa))) ->
                       (Action (c, aa),
                         (varSeta,
                           (And (varEnva,
                                 Not
(replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                            LParallel (p, sa, s, x, qa))))))
                (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                 replaceVarInProcName replaceVarInDoubleBool varSet
                     varEnv
                 q)
           | (Ldiv, b) ->
             foldl (fun col a ->
                    (match a
                      with (Tau, (varSeta, (varEnva, pa))) ->
                        (Tau, (varSeta,
                               (varEnva, LParallel (pa, sa, s, x, q))))
                                   ::
                          col
                      | (Check, (varSeta, (varEnva, pa))) ->
```

```
                        (Tau, (varSeta,
                               (varEnva,
                                LParallel (Omega, sa, s, x, q)))) ::
                          col
                      | (Action (c, aa), (varSeta, (varEnva, pa))) ->
                        foldl (fun cola ba ->
                               (match ba
                                with (Tau, (varSetb, (varEnvb, qa))) ->
                                  cola
                                | (Check, (varSetb, (varEnvb, qa))) ->
                                    cola
                                | (Action (ca, aaa),
 (varSetb, (varEnvb, qa)))
                                  -> (Action (c, aa),
   (varSetb,
     (And (And (Equal (IdP aa, IdP aaa),
                And (varEnvb,
                     replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
          Equal (IdP c, IdP ca)),
       LParallel (pa, sa, s, x, qa)))) ::
   cola))
                        [] (oneStepEvalFun _B procEnv getNewVar
                            replaceVarInCspp replaceVarInProcName
                            replaceVarInDoubleBool varSeta varEnva q)
                               @
                        (Action (c, aa),
                          (varSeta,
                            (And (varEnva,
Not (replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                               LParallel (pa, sa, s, x, q)))) ::
                          col))
                [] (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                    replaceVarInProcName replaceVarInDoubleBool varSet
                        varEnv
                    p) @
                map (fun a ->
                     (match a
                      with (Tau, (varSeta, (varEnva, qa))) ->
                        (Tau, (varSeta,
                               (varEnva, LParallel (p, sa, s, x, qa))))
                      | (Check, (varSeta, (varEnva, qa))) ->
                        (Tau, (varSeta,
                               (varEnva, LParallel (p, sa, s, x,
                                   Omega))))
                      | (Action (c, aa), (varSeta, (varEnva, qa))) ->
                        (Action (c, aa),
                          (varSeta,
```

```
                            (And (varEnva,
                                  Not
(replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                             LParallel (p, sa, s, x, qa))))))
                (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                 replaceVarInProcName replaceVarInDoubleBool varSet
                     varEnv
                 q)
           | (LProc_name _, b) ->
             foldl (fun col a ->
                    (match a
                     with (Tau, (varSeta, (varEnva, pa))) ->
                       (Tau, (varSeta,
                              (varEnva, LParallel (pa, sa, s, x, q))))
                                  ::
                         col
                     | (Check, (varSeta, (varEnva, pa))) ->
                       (Tau, (varSeta,
                              (varEnva,
                               LParallel (Omega, sa, s, x, q)))) ::
                         col
                     | (Action (c, aa), (varSeta, (varEnva, pa))) ->
                       foldl (fun cola ba ->
                              (match ba
                               with (Tau, (varSetb, (varEnvb, qa))) ->
                                 cola
                               | (Check, (varSetb, (varEnvb, qa))) ->
                                    cola
                               | (Action (ca, aaa),
 (varSetb, (varEnvb, qa)))
                                 -> (Action (c, aa),
   (varSetb,
     (And (And (Equal (IdP aa, IdP aaa),
                And (varEnvb,
                     replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
           Equal (IdP c, IdP ca)),
       LParallel (pa, sa, s, x, qa)))) ::
   cola))
                        [] (oneStepEvalFun _B procEnv getNewVar
                            replaceVarInCspp replaceVarInProcName
                            replaceVarInDoubleBool varSeta varEnva q)
                                @
                        (Action (c, aa),
                          (varSeta,
                            (And (varEnva,
Not (replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                             LParallel (pa, sa, s, x, q)))) ::
```

```
                        col))
                [] (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                    replaceVarInProcName replaceVarInDoubleBool varSet
                        varEnv
                    p) @
                map (fun a ->
                     (match a
                      with (Tau, (varSeta, (varEnva, qa))) ->
                        (Tau, (varSeta,
                               (varEnva, LParallel (p, sa, s, x, qa))))
                       | (Check, (varSeta, (varEnva, qa))) ->
                        (Tau, (varSeta,
                               (varEnva, LParallel (p, sa, s, x,
                                   Omega))))
                       | (Action (c, aa), (varSeta, (varEnva, qa))) ->
                        (Action (c, aa),
                          (varSeta,
                            (And (varEnva,
                                  Not
(replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                             LParallel (p, sa, s, x, qa))))))
               (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                replaceVarInProcName replaceVarInDoubleBool varSet
                    varEnv
                q)
           | (LAct_prefix (_, _, _), b) ->
             foldl (fun col a ->
                    (match a
                     with (Tau, (varSeta, (varEnva, pa))) ->
                       (Tau, (varSeta,
                              (varEnva, LParallel (pa, sa, s, x, q))))
                                  ::
                         col
                       | (Check, (varSeta, (varEnva, pa))) ->
                        (Tau, (varSeta,
                               (varEnva,
                                 LParallel (Omega, sa, s, x, q)))) ::
                         col
                       | (Action (c, aa), (varSeta, (varEnva, pa))) ->
                        foldl (fun cola ba ->
                               (match ba
                                with (Tau, (varSetb, (varEnvb, qa))) ->
                                  cola
                                | (Check, (varSetb, (varEnvb, qa))) ->
                                    cola
                                | (Action (ca, aaa),
 (varSetb, (varEnvb, qa)))
```

```
                                  -> (Action (c, aa),
    (varSetb,
      (And (And (Equal (IdP aa, IdP aaa),
                 And (varEnvb,
                      replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
            Equal (IdP c, IdP ca)),
       LParallel (pa, sa, s, x, qa)))) ::
    cola))
                         [] (oneStepEvalFun _B procEnv getNewVar
                              replaceVarInCspp replaceVarInProcName
                              replaceVarInDoubleBool varSeta varEnva q)
                                 @
                         (Action (c, aa),
                           (varSeta,
                             (And (varEnva,
Not (replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                                LParallel (pa, sa, s, x, q)))) ::
                           col))
                [] (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                    replaceVarInProcName replaceVarInDoubleBool varSet
                        varEnv
                    p) @
                map (fun a ->
                      (match a
                        with (Tau, (varSeta, (varEnva, qa))) ->
                          (Tau, (varSeta,
                                 (varEnva, LParallel (p, sa, s, x, qa))))
                       | (Check, (varSeta, (varEnva, qa))) ->
                          (Tau, (varSeta,
                                 (varEnva, LParallel (p, sa, s, x,
                                     Omega))))
                       | (Action (c, aa), (varSeta, (varEnva, qa))) ->
                          (Action (c, aa),
                            (varSeta,
                              (And (varEnva,
                                    Not
(replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                                LParallel (p, sa, s, x, qa))))))
                  (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                   replaceVarInProcName replaceVarInDoubleBool varSet
                       varEnv
                   q)
             | (LExt_choice (_, _), b) ->
               foldl (fun col a ->
                      (match a
                        with (Tau, (varSeta, (varEnva, pa))) ->
                          (Tau, (varSeta,
```

```
                                (varEnva, LParallel (pa, sa, s, x, q))))
                                    ::
                              col
                           | (Check, (varSeta, (varEnva, pa))) ->
                             (Tau, (varSeta,
                                    (varEnva,
                                     LParallel (Omega, sa, s, x, q)))) ::
                              col
                           | (Action (c, aa), (varSeta, (varEnva, pa))) ->
                             foldl (fun cola ba ->
                                    (match ba
                                     with (Tau, (varSetb, (varEnvb, qa))) ->
                                       cola
                                     | (Check, (varSetb, (varEnvb, qa))) ->
                                         cola
                                     | (Action (ca, aaa),
 (varSetb, (varEnvb, qa)))
                                       -> (Action (c, aa),
    (varSetb,
      (And (And (Equal (IdP aa, IdP aaa),
                 And (varEnvb,
                      replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
            Equal (IdP c, IdP ca)),
       LParallel (pa, sa, s, x, qa)))) ::
    cola))
                             [] (oneStepEvalFun _B procEnv getNewVar
                                 replaceVarInCspp replaceVarInProcName
                                 replaceVarInDoubleBool varSeta varEnva q)
                                  @
                             (Action (c, aa),
                               (varSeta,
                                 (And (varEnva,
 Not (replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                                  LParallel (pa, sa, s, x, q)))) ::
                              col))
                  [] (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                      replaceVarInProcName replaceVarInDoubleBool varSet
                          varEnv
                      p) @
                  map (fun a ->
                       (match a
                        with (Tau, (varSeta, (varEnva, qa))) ->
                          (Tau, (varSeta,
                                 (varEnva, LParallel (p, sa, s, x, qa))))
                        | (Check, (varSeta, (varEnva, qa))) ->
                          (Tau, (varSeta,
```

```
                                (varEnva, LParallel (p, sa, s, x,
                                    Omega))))
                          | (Action (c, aa), (varSeta, (varEnva, qa))) ->
                            (Action (c, aa),
                              (varSeta,
                                (And (varEnva,
                                      Not
(replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                                 LParallel (p, sa, s, x, qa))))))
                    (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                     replaceVarInProcName replaceVarInDoubleBool varSet
                         varEnv
                     q)
                | (LInt_choice (_, _), b) ->
                  foldl (fun col a ->
                         (match a
                          with (Tau, (varSeta, (varEnva, pa))) ->
                            (Tau, (varSeta,
                                   (varEnva, LParallel (pa, sa, s, x, q))))
                                       ::
                              col
                          | (Check, (varSeta, (varEnva, pa))) ->
                            (Tau, (varSeta,
                                   (varEnva,
                                    LParallel (Omega, sa, s, x, q)))) ::
                              col
                          | (Action (c, aa), (varSeta, (varEnva, pa))) ->
                            foldl (fun cola ba ->
                                   (match ba
                                    with (Tau, (varSetb, (varEnvb, qa))) ->
                                      cola
                                    | (Check, (varSetb, (varEnvb, qa))) ->
                                        cola
                                    | (Action (ca, aaa),
(varSetb, (varEnvb, qa)))
                                      -> (Action (c, aa),
   (varSetb,
     (And (And (Equal (IdP aa, IdP aaa),
                And (varEnvb,
                     replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
           Equal (IdP c, IdP ca)),
      LParallel (pa, sa, s, x, qa)))) ::
   cola))
                              [] (oneStepEvalFun _B procEnv getNewVar
                                  replaceVarInCspp replaceVarInProcName
                                  replaceVarInDoubleBool varSeta varEnva q)
                                      @
```

```
                          (Action (c, aa),
                            (varSeta,
                              (And (varEnva,
Not (replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                                 LParallel (pa, sa, s, x, q)))) ::
                            col))
                [] (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                    replaceVarInProcName replaceVarInDoubleBool varSet
                        varEnv
                    p) @
                map (fun a ->
                     (match a
                      with (Tau, (varSeta, (varEnva, qa))) ->
                        (Tau, (varSeta,
                               (varEnva, LParallel (p, sa, s, x, qa))))
                      | (Check, (varSeta, (varEnva, qa))) ->
                        (Tau, (varSeta,
                               (varEnva, LParallel (p, sa, s, x,
                                   Omega))))
                      | (Action (c, aa), (varSeta, (varEnva, qa))) ->
                        (Action (c, aa),
                          (varSeta,
                            (And (varEnva,
                                  Not
(replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                               LParallel (p, sa, s, x, qa))))))
                  (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                   replaceVarInProcName replaceVarInDoubleBool varSet
                       varEnv
                   q)
            | (Lif (_, _, _), b) ->
              foldl (fun col a ->
                     (match a
                      with (Tau, (varSeta, (varEnva, pa))) ->
                        (Tau, (varSeta,
                               (varEnva, LParallel (pa, sa, s, x, q))))
                                   ::
                          col
                      | (Check, (varSeta, (varEnva, pa))) ->
                        (Tau, (varSeta,
                               (varEnva,
                                 LParallel (Omega, sa, s, x, q)))) ::
                          col
                      | (Action (c, aa), (varSeta, (varEnva, pa))) ->
                        foldl (fun cola ba ->
                               (match ba
                                with (Tau, (varSetb, (varEnvb, qa))) ->
```

```
                                  cola
                                | (Check, (varSetb, (varEnvb, qa))) ->
                                    cola
                                | (Action (ca, aaa),
 (varSetb, (varEnvb, qa)))
                                  -> (Action (c, aa),
    (varSetb,
      (And (And (Equal (IdP aa, IdP aaa),
                 And (varEnvb,
                      replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
            Equal (IdP c, IdP ca)),
       LParallel (pa, sa, s, x, qa)))) ::
    cola))
                          [] (oneStepEvalFun _B procEnv getNewVar
                              replaceVarInCspp replaceVarInProcName
                              replaceVarInDoubleBool varSeta varEnva q)
                                  @
                          (Action (c, aa),
                            (varSeta,
                              (And (varEnva,
Not (replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                                 LParallel (pa, sa, s, x, q)))) ::
                            col))
                 [] (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                     replaceVarInProcName replaceVarInDoubleBool varSet
                         varEnv
                     p) @
                 map (fun a ->
                       (match a
                         with (Tau, (varSeta, (varEnva, qa))) ->
                           (Tau, (varSeta,
                                  (varEnva, LParallel (p, sa, s, x, qa))))
                        | (Check, (varSeta, (varEnva, qa))) ->
                          (Tau, (varSeta,
                                  (varEnva, LParallel (p, sa, s, x,
                                      Omega))))
                        | (Action (c, aa), (varSeta, (varEnva, qa))) ->
                          (Action (c, aa),
                            (varSeta,
                              (And (varEnva,
                                    Not
(replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                                LParallel (p, sa, s, x, qa))))))
                   (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                    replaceVarInProcName replaceVarInDoubleBool varSet
                        varEnv
                    q)
```

```
          | (LParallel (_, _, _, _, _), b) ->
            foldl (fun col a ->
                   (match a
                    with (Tau, (varSeta, (varEnva, pa))) ->
                      (Tau, (varSeta,
                             (varEnva, LParallel (pa, sa, s, x, q))))
                          ::
                        col
                    | (Check, (varSeta, (varEnva, pa))) ->
                      (Tau, (varSeta,
                             (varEnva,
                              LParallel (Omega, sa, s, x, q)))) ::
                        col
                    | (Action (c, aa), (varSeta, (varEnva, pa))) ->
                      foldl (fun cola ba ->
                             (match ba
                              with (Tau, (varSetb, (varEnvb, qa))) ->
                                cola
                              | (Check, (varSetb, (varEnvb, qa))) ->
                                  cola
                              | (Action (ca, aaa),
 (varSetb, (varEnvb, qa)))
                                -> (Action (c, aa),
   (varSetb,
     (And (And (Equal (IdP aa, IdP aaa),
                And (varEnvb,
                     replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
           Equal (IdP c, IdP ca)),
       LParallel (pa, sa, s, x, qa)))) ::
   cola))
                         [] (oneStepEvalFun _B procEnv getNewVar
                             replaceVarInCspp replaceVarInProcName
                             replaceVarInDoubleBool varSeta varEnva q)
                                @
                         (Action (c, aa),
                           (varSeta,
                             (And (varEnva,
Not (replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                              LParallel (pa, sa, s, x, q)))) ::
                           col))
              [] (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                  replaceVarInProcName replaceVarInDoubleBool varSet
                      varEnv
                  p) @
             map (fun a ->
                   (match a
                    with (Tau, (varSeta, (varEnva, qa))) ->
```

```
                        (Tau, (varSeta,
                               (varEnva, LParallel (p, sa, s, x, qa))))
                      | (Check, (varSeta, (varEnva, qa))) ->
                        (Tau, (varSeta,
                               (varEnva, LParallel (p, sa, s, x,
                                   Omega))))
                      | (Action (c, aa), (varSeta, (varEnva, qa))) ->
                        (Action (c, aa),
                          (varSeta,
                            (And (varEnva,
                                  Not
(replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                             LParallel (p, sa, s, x, qa))))))
                (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                 replaceVarInProcName replaceVarInDoubleBool varSet
                     varEnv
                 q)
            | (LSeq_compo (_, _), b) ->
              foldl (fun col a ->
                     (match a
                      with (Tau, (varSeta, (varEnva, pa))) ->
                        (Tau, (varSeta,
                               (varEnva, LParallel (pa, sa, s, x, q))))
                                   ::
                          col
                      | (Check, (varSeta, (varEnva, pa))) ->
                        (Tau, (varSeta,
                               (varEnva,
                                LParallel (Omega, sa, s, x, q)))) ::
                          col
                      | (Action (c, aa), (varSeta, (varEnva, pa))) ->
                        foldl (fun cola ba ->
                               (match ba
                                with (Tau, (varSetb, (varEnvb, qa))) ->
                                  cola
                                | (Check, (varSetb, (varEnvb, qa))) ->
                                    cola
                                | (Action (ca, aaa),
 (varSetb, (varEnvb, qa)))
                                  -> (Action (c, aa),
   (varSetb,
     (And (And (Equal (IdP aa, IdP aaa),
                And (varEnvb,
                     replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
           Equal (IdP c, IdP ca)),
      LParallel (pa, sa, s, x, qa)))) ::
   cola))
```

```
                        [] (oneStepEvalFun _B procEnv getNewVar
                            replaceVarInCspp replaceVarInProcName
                            replaceVarInDoubleBool varSeta varEnva q)
                                @
                        (Action (c, aa),
                          (varSeta,
                            (And (varEnva,
Not (replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                             LParallel (pa, sa, s, x, q)))) ::
                          col))
               [] (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                   replaceVarInProcName replaceVarInDoubleBool varSet
                       varEnv
                   p) @
               map (fun a ->
                    (match a
                      with (Tau, (varSeta, (varEnva, qa))) ->
                        (Tau, (varSeta,
                               (varEnva, LParallel (p, sa, s, x, qa))))
                      | (Check, (varSeta, (varEnva, qa))) ->
                        (Tau, (varSeta,
                               (varEnva, LParallel (p, sa, s, x,
                                   Omega))))
                      | (Action (c, aa), (varSeta, (varEnva, qa))) ->
                        (Action (c, aa),
                          (varSeta,
                            (And (varEnva,
                                  Not
(replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                             LParallel (p, sa, s, x, qa))))))
                 (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                  replaceVarInProcName replaceVarInDoubleBool varSet
                      varEnv
                  q)
             | (LExt_pre_choice (_, _, _, _), b) ->
              foldl (fun col a ->
                     (match a
                       with (Tau, (varSeta, (varEnva, pa))) ->
                         (Tau, (varSeta,
                                (varEnva, LParallel (pa, sa, s, x, q))))
                                    ::
                           col
                       | (Check, (varSeta, (varEnva, pa))) ->
                         (Tau, (varSeta,
                                (varEnva,
                                  LParallel (Omega, sa, s, x, q)))) ::
                           col
```

```
                    | (Action (c, aa), (varSeta, (varEnva, pa))) ->
                      foldl (fun cola ba ->
                             (match ba
                               with (Tau, (varSetb, (varEnvb, qa))) ->
                                 cola
                               | (Check, (varSetb, (varEnvb, qa))) ->
                                   cola
                               | (Action (ca, aaa),
 (varSetb, (varEnvb, qa)))
                                 -> (Action (c, aa),
   (varSetb,
     (And (And (Equal (IdP aa, IdP aaa),
                And (varEnvb,
                     replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
          Equal (IdP c, IdP ca)),
       LParallel (pa, sa, s, x, qa)))) ::
   cola))
                        [] (oneStepEvalFun _B procEnv getNewVar
                            replaceVarInCspp replaceVarInProcName
                            replaceVarInDoubleBool varSeta varEnva q)
                               @
                        (Action (c, aa),
                          (varSeta,
                            (And (varEnva,
Not (replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                             LParallel (pa, sa, s, x, q)))) ::
                          col))
               [] (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                   replaceVarInProcName replaceVarInDoubleBool varSet
                       varEnv
                   p) @
              map (fun a ->
                    (match a
                      with (Tau, (varSeta, (varEnva, qa))) ->
                        (Tau, (varSeta,
                               (varEnva, LParallel (p, sa, s, x, qa))))
                      | (Check, (varSeta, (varEnva, qa))) ->
                        (Tau, (varSeta,
                               (varEnva, LParallel (p, sa, s, x,
                                   Omega))))
                      | (Action (c, aa), (varSeta, (varEnva, qa))) ->
                        (Action (c, aa),
                          (varSeta,
                            (And (varEnva,
                                  Not
(replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                             LParallel (p, sa, s, x, qa))))))
```

```
                (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                 replaceVarInProcName replaceVarInDoubleBool varSet
                     varEnv
                 q)
           | (LRep_int_choice (_, _, _), b) ->
             foldl (fun col a ->
                    (match a
                     with (Tau, (varSeta, (varEnva, pa))) ->
                       (Tau, (varSeta,
                              (varEnva, LParallel (pa, sa, s, x, q))))
                           ::
                         col
                     | (Check, (varSeta, (varEnva, pa))) ->
                       (Tau, (varSeta,
                              (varEnva,
                               LParallel (Omega, sa, s, x, q)))) ::
                         col
                     | (Action (c, aa), (varSeta, (varEnva, pa))) ->
                       foldl (fun cola ba ->
                              (match ba
                               with (Tau, (varSetb, (varEnvb, qa))) ->
                                 cola
                               | (Check, (varSetb, (varEnvb, qa))) ->
                                   cola
                               | (Action (ca, aaa),
 (varSetb, (varEnvb, qa)))
                                 -> (Action (c, aa),
   (varSetb,
     (And (And (Equal (IdP aa, IdP aaa),
                And (varEnvb,
                     replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
           Equal (IdP c, IdP ca)),
      LParallel (pa, sa, s, x, qa)))) ::
   cola))
                       [] (oneStepEvalFun _B procEnv getNewVar
                           replaceVarInCspp replaceVarInProcName
                           replaceVarInDoubleBool varSeta varEnva q)
                              @
                       (Action (c, aa),
                         (varSeta,
                           (And (varEnva,
Not (replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                            LParallel (pa, sa, s, x, q)))) ::
                        col))
              [] (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                  replaceVarInProcName replaceVarInDoubleBool varSet
                      varEnv
```

```
                    p) @
                map (fun a ->
                     (match a
                       with (Tau, (varSeta, (varEnva, qa))) ->
                         (Tau, (varSeta,
                                (varEnva, LParallel (p, sa, s, x, qa))))
                       | (Check, (varSeta, (varEnva, qa))) ->
                         (Tau, (varSeta,
                                (varEnva, LParallel (p, sa, s, x,
                                    Omega))))
                       | (Action (c, aa), (varSeta, (varEnva, qa))) ->
                         (Action (c, aa),
                           (varSeta,
                             (And (varEnva,
                                   Not
(replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                                LParallel (p, sa, s, x, qa))))))
                (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                 replaceVarInProcName replaceVarInDoubleBool varSet
                     varEnv
                q)
           | (LHid (_, _, _, _), b) ->
             foldl (fun col a ->
                    (match a
                      with (Tau, (varSeta, (varEnva, pa))) ->
                        (Tau, (varSeta,
                               (varEnva, LParallel (pa, sa, s, x, q))))
                                   ::
                          col
                      | (Check, (varSeta, (varEnva, pa))) ->
                        (Tau, (varSeta,
                               (varEnva,
                                 LParallel (Omega, sa, s, x, q)))) ::
                          col
                      | (Action (c, aa), (varSeta, (varEnva, pa))) ->
                        foldl (fun cola ba ->
                               (match ba
                                 with (Tau, (varSetb, (varEnvb, qa))) ->
                                   cola
                                 | (Check, (varSetb, (varEnvb, qa))) ->
                                     cola
                                 | (Action (ca, aaa),
 (varSetb, (varEnvb, qa)))
                                   -> (Action (c, aa),
   (varSetb,
     (And (And (Equal (IdP aa, IdP aaa),
                And (varEnvb,
```

```
                    replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
          Equal (IdP c, IdP ca)),
       LParallel (pa, sa, s, x, qa)))) ::
   cola))
                       [] (oneStepEvalFun _B procEnv getNewVar
                           replaceVarInCspp replaceVarInProcName
                           replaceVarInDoubleBool varSeta varEnva q)
                               @
                       (Action (c, aa),
                         (varSeta,
                           (And (varEnva,
Not (replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                            LParallel (pa, sa, s, x, q)))) ::
                          col))
              [] (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                  replaceVarInProcName replaceVarInDoubleBool varSet
                      varEnv
                  p) @
              map (fun a ->
                   (match a
                     with (Tau, (varSeta, (varEnva, qa))) ->
                       (Tau, (varSeta,
                              (varEnva, LParallel (p, sa, s, x, qa))))
                     | (Check, (varSeta, (varEnva, qa))) ->
                       (Tau, (varSeta,
                              (varEnva, LParallel (p, sa, s, x,
                                  Omega))))
                     | (Action (c, aa), (varSeta, (varEnva, qa))) ->
                       (Action (c, aa),
                         (varSeta,
                           (And (varEnva,
                                 Not
(replaceVarInDoubleBool x sa (IdP c) s (IdP aa))),
                            LParallel (p, sa, s, x, qa))))))
                (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
                  replaceVarInProcName replaceVarInDoubleBool varSet
                      varEnv
                  q))
   | procEnv, getNewVar, replaceVarInCspp, replaceVarInProcName,
      replaceVarInDoubleBool, varSet, varEnv, LHid (p, sa, s, x)
      -> foldl (fun col a ->
                (match a
                  with (Tau, (varSeta, (varEnva, pa))) ->
                    (Tau, (varSeta, (varEnva, LHid (pa, sa, s, x)))) ::
                        col
                  | (Check, (varSeta, (varEnva, pa))) ->
                    (Check, (varSeta, (varEnva, Omega))) :: col
```

```
                  | (Action (c, aa), (varSeta, (varEnva, pa))) ->
                    [(Tau, (varSeta,
                            (And (varEnva,
                                  replaceVarInDoubleBool x sa (IdP c) s
 (IdP aa)),
                             LHid (pa, sa, s, x))));
                     (Action (c, aa),
                      (varSeta,
                       (And (varEnva,
                             Not (replaceVarInDoubleBool x sa (IdP
                                 c) s
  (IdP aa))),
                        LHid (pa, sa, s, x))))] @
                    col))
           [] (oneStepEvalFun _B procEnv getNewVar replaceVarInCspp
               replaceVarInProcName replaceVarInDoubleBool varSet
                   varEnv p);;

end;; (*struct oneStepEval*)
```

```
(* stream.ml *)
type 'a stream = Stream of ('a * (unit -> 'a stream)) | Bottom;;

let max_stream_search = ref 10000;;

let hd theStream =match theStream with (Stream (x, s)) -> Some x |
    Bottom -> None;;
let tl theStream =match theStream with (Stream (x,s)) -> s() | Bottom ->
    Bottom;;

let rec first n stream = if n <= 0 then [] else hd stream :: first (n -1)
       (tl stream);;

let rec stream_map f theStream = match theStream with (Stream(x, s))
       -> Stream((f x), fun () -> stream_map f (s())) | Bottom ->
           Bottom;;

let rec bounded_filter bound property stream =
   let rec bnd_filter_aux bound number_failures property theStream =
       match theStream with (Stream(x,s)) -> (
              if property x then Stream (x, fun () ->
                      bounded_filter bound property (s()))
              else if number_failures = bound then Bottom
           else bnd_filter_aux bound (number_failures + 1) property
               (s()) )
                    |Bottom ->Bottom
```

```
                    in bnd_filter_aux bound 0 property stream;;

let filter property stream = bounded_filter (!max_stream_search)
    property stream;;

let stream_of_itemization f =
   let rec stm f n = Stream (f n, fun () -> stm f (n+1))
   in stm f 0;;

let int_to_nat n = if n >= 0 then 2 * n else ((-2) * n) - 1;;

let nat_to_int n = if n mod 2 = 0 then n / 2 else (-1) * ((n + 1) / 2);;

let stream_of_pred property =
   filter property (stream_of_itemization nat_to_int);;
```

```
(* evaluator.ml *)
open OneStepEval
open Z3
open Streams

let read_int () = print_string "do you want to see the next result
    (1/0):"
              ; print_newline ();
              Scanf.bscanf Scanf.Scanning.stdib " %d" (fun x->x);;


let eval procEnv varEnv cspp =let rec evalAux procEnv varEnv
              cspp lastStream hisList =
               match (oneStepEval procEnv varEnv cspp)
              with Bottom -> Stream (Bottom, fun () -> lastEval hisList)
                     | Stream ((someProcEnv,someVarEnv,
                            Skip_as_cspp,None),someFun)
                            -> (match (decideExists someVarEnv)
                            with None ->Stream (Bottom,
                            fun() -> lastEval ((fun ()
                            -> currentEval (someFun())
                                lastStream)::hisList))
                            | Some newBool
                     -> Stream (Stream
                     ((someProcEnv,newBool,
                     Skip_as_cspp,None),fun()->lastStream),
                     fun()
                     -> lastEval ((fun ()
                     ->currentEval (someFun()) lastStream)::hisList)))
```

```
                        | Stream
                            ((someProcEnv,someVarEnv,Stop_as_cspp,None),
                        someFun)
                        -> (match (decideExists someVarEnv)
                        with None -> Stream (Bottom, fun()
                        ->lastEval ((fun ()->currentEval (someFun())
                            lastStream)::hisList))
                        | Some newBool
                        -> Stream (Stream ((someProcEnv,newBool,
                        Stop_as_cspp,None),fun()->lastStream),
                        fun() -> lastEval ((fun ()
                        ->currentEval (someFun()) lastStream)::hisList)))
                        | Stream ((someProcEnv,someVarEnv,
                        Div_as_cspp,None),someFun)
                        -> (match (decideExists someVarEnv)
                        with None -> Stream (Bottom, fun()->
                        lastEval ((fun ()
                        ->currentEval (someFun()) lastStream)::hisList))
                        | Some newBool
                        -> Stream (Stream
                            ((someProcEnv,newBool,Div_as_cspp,None),
                        fun()->lastStream),fun()
                        -> lastEval ((fun ()->currentEval (someFun())
                            lastStream)::hisList)))
                        | Stream ((someProcEnv,someVarEnv,
                        someCspp,someAct),someFun)
                        -> (match (decideExists someVarEnv)
                        with None -> Stream (Bottom,
                        fun()->lastEval ((fun ()->currentEval (someFun())
                            lastStream)::hisList))
                        | Some newBool
                        -> Stream (Stream ((someProcEnv,
                        newBool,someCspp,someAct),fun()
                        ->lastStream),fun() -> evalAux someProcEnv
                        newBool someCspp (Stream ((someProcEnv,newBool
                        ,someCspp,someAct),fun()->lastStream))
                        ((fun ()->currentEval (someFun())
                            lastStream)::hisList)))

and currentEval furStream lastStream = match furStream with Bottom ->
    Bottom
              | Stream ((someProcEnv,someVarEnv,
                     Skip_as_cspp,None),someFun)
              -> (match (decideExists someVarEnv)
              with None -> Stream (Bottom,
              fun()->currentEval (someFun()) lastStream)
       | Some newBool
```

```
        -> Stream (Stream ((someProcEnv,newBool
               ,Skip_as_cspp,None),fun()
               ->lastStream),fun() -> currentEval (someFun())
                   lastStream))
        | Stream ((someProcEnv,someVarEnv
               ,Stop_as_cspp,None),someFun)
        -> (match (decideExists someVarEnv)
        with None -> Stream (Bottom, fun()->currentEval (someFun())
            lastStream)
        | Some newBool -> Stream (Stream
         ((someProcEnv,newBool,Stop_as_cspp,None)
        ,fun()->lastStream),fun()
        -> currentEval (someFun()) lastStream))
        | Stream ((someProcEnv,someVarEnv
        ,Div_as_cspp,None),someFun)
        -> (match (decideExists someVarEnv)
               with None -> Stream (Bottom,
               fun()->currentEval (someFun()) lastStream)
        | Some newBool
               -> Stream (Stream ((someProcEnv
               ,newBool,Div_as_cspp,None),
               fun()->lastStream),fun()
               -> currentEval (someFun()) lastStream))
        | Stream ((someProcEnv,someVarEnv
               ,someCspp,someAct),someFun)
        -> (match (decideExists someVarEnv)
        with None -> Stream (Bottom, fun()->currentEval (someFun())
            lastStream)
        | Some newBool -> Stream (Stream ((someProcEnv
        ,newBool,someCspp,someAct),fun()->lastStream)
        ,fun() -> evalAux someProcEnv newBool someCspp
        (Stream
            ((someProcEnv,newBool,someCspp,someAct),fun()->lastStream))
         ((fun ()->currentEval (someFun()) lastStream)::[])))

and lastEval hisList = match hisList with [] -> Bottom
       |x::xs-> (match (x()) with Bottom -> lastEval xs
       | Stream (someStream,otherFun)-> Stream (someStream,
              fun () -> lastEval ((otherFun)::xs)))

in evalAux procEnv varEnv cspp Bottom [];;
```

References

[1] Brookes, S.D., Hoare, C.A.R., Roscoe, A.W.: A theory of communicating sequential processes. J. ACM **31**(3) (1984) 560–599

[2] Paulson, L.C.: Isabelle: The next 700 theorem provers. In Odifreddi, P., ed.: Logic and Computer Science. Academic Press (1990) 361–386

[3] Gunter, E.L., Yasmeen, A., Gunter, C.A., Nguyen, A.: Specifying and analyzing workflows for automated identification and data capture. In: HICSS, IEEE Computer Society (2009) 1–11

[4] Wong, P.Y.H., Gibbons, J.: A process-algebraic approach to workflow specification and refinement. In: Proceedings of the 6th international conference on Software composition. SC'07, Berlin, Heidelberg, Springer-Verlag (2007) 51–65

[5] Wright, P., Fields, B., Harrison, M.: Deriving human-error tolerance requirements from tasks. In: In Proc. of ICRE94 IEEE Intl. Conf. on Requirements Engineering, IEEE (1994) 135–142

[6] Ltd, F.S.E.: Process behaviour explorer. ProBE user manual. In: ProBE User Manual. (2003)

[7] Scattergood, J.: The Semantics and Implementation of Machine-Readable CSP. D.phil., Oxford University Computing Laboratory (1998)

[8] Hoare, C.A.R.: Communicating sequential processes. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1985)

[9] Brookes, S.D., Roscoe, A.W., Walker, D.J.: An operational semantics for CSP. Technical report, Oxford University Computing Laboratory (1986)

[10] Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10) (October 1969) 576–580

[11] Urban, C.: Nominal techniques in Isabelle/HOL. J. Autom. Reason. **40**(4) (May 2008) 327–356

[12] Isobe, Y., Roggenbach, M.: A complete axiomatic semantics for the CSP stable-failures model. In Baier, C., Hermanns, H., eds.: CONCUR 2006 Concurrency Theory. Volume 4137 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2006) 158–172 10.1007/11817949_11.

[13] Kuncak, V., Kuncak, V.: Quantifier-free boolean algebra with presburger arithmetic is np-complete (2007)

[14] Ltd., F.S.E.: Failures-divergence refinement. FDR2 user manual. In: FDR2 User Manual. (2010)

[15] Sun, J., Liu, Y., Dong, J.S.: Model checking csp revisited: Introducing a process analysis toolkit. In: In ISoLA 2008, Springer (2008) 307–322

[16] Brooke, P.J., Paige, R.F.: Lazy exploration and checking of CSP models with CSPsim. In McEwan, A.A., Schneider, S.A., Ifill, W., Welch, P.H., eds.: CPA. Volume 65 of Concurrent Systems Engineering Series., IOS Press (2007) 33–49

[17] Gibson-Robinson, T., Philip Armstrong, A.B., Roscoe, A.: Fdr3 - a modern refinement checker for csp. In: TACAS. (2014) In Press.

[18] Hennessy, M., Lin, H.: Symbolic bisimulations. Theor. Comput. Sci. **138**(2) (February 1995) 353–389

[19] Sangiorgi, D.: A theory of bisimulation for the pi-calculus. In Best, E., ed.: CONCUR. Volume 715 of Lecture Notes in Computer Science., Springer (1993) 127–142

[20] Bonchi, F., Montanari, U.: Symbolic semantics revisited. In: Proceedings of the Theory and practice of software, 11th international conference on Foundations of software science and computational structures. FOSSACS'08/ETAPS'08, Berlin, Heidelberg, Springer-Verlag (2008) 395–412

[21] Calder, M., Shankland, C.: A symbolic semantics and bisimulation for full lotos. In: PROC. FORMAL TECHNIQUES FOR NETWORKED AND DISTRIBUTED SYSTEMS (FORTE XIV, Kluwer Academic Publishers (2001) 184–200

[22] Pugliese, R., Tiezzi, F., Yoshida, N.: A symbolic semantics for a calculus for service-oriented computing. Electron. Notes Theor. Comput. Sci. **241** (July 2009) 135–164

[23] Groote, J.F., Mathijssen, A., Reniers, M., Usenko, Y., Weerdenburg, M.V.: The formal specification language mcrl2. In: In Proceedings of the Dagstuhl Seminar, MIT Press (2007)