

Formalizing Statecharts using Hierarchical Automata

Steffen Helke and Florian Kammüller

October 13, 2025

Abstract

We formalize in Isabelle/HOL the abstract syntax and a synchronous step semantics for the specification language Statecharts [HN96]. The formalization is based on Hierarchical Automata [MLS97] which allow a structural decomposition of Statecharts into Sequential Automata. To support the composition of Statecharts, we introduce calculating operators to construct a Hierarchical Automaton in a stepwise manner [HK01]. Furthermore, we present a complete semantics of Statecharts including a theory of data spaces, which enables the modelling of racing effects [HK05]. We also adapt CTL for Statecharts to build a bridge for future combinations with model checking. However the main motivation of this work is to provide a sound and complete basis for reasoning on Statecharts. As a central meta theorem we prove that the well-formedness of a Statechart is preserved by the semantics [Hel07].

Contents

1	Contributions to the Standard Library of HOL	1
1.1	Basic definitions and lemmas	1
1.1.1	Maps	1
1.1.2	<i>rtranc1</i>	2
1.1.3	<i>finite</i>	4
1.1.4	<i>override</i>	4
1.1.5	<i>Part</i>	5
1.1.6	Set operators	6
1.1.7	One point rule	7
2	Partitoned Data Spaces for Statecharts	8
2.1	Definitions	8
2.2	Lemmas	8
2.2.1	<i>DataSpace</i>	8
2.2.2	<i>PartNum</i>	9

3	Data Space Assignments	10
3.1	Total data space assignments	10
3.2	Partial data space assignments	12
3.2.1	<i>DefaultPData</i>	14
3.2.2	<i>Data2PData</i>	14
3.2.3	<i>DataOverride</i>	16
3.2.4	<i>OptionOverride</i>	17
4	Update-Functions on Data Spaces	17
4.1	Total update-functions	17
4.1.1	Basic lemmas	18
4.1.2	<i>DefaultUpdate</i>	18
4.2	Partial update-functions	19
4.2.1	Basic lemmas	19
4.2.2	<i>Data2PData</i>	19
4.2.3	<i>PUpdate</i>	20
4.2.4	<i>SequentialRacing</i>	20
5	Label Expressions	21
6	Sequential Automata	26
7	Syntax of Hierarchical Automata	29
7.1	Definitions	30
7.1.1	Well-formedness for the syntax of HA	31
7.1.2	State successor function	32
7.1.3	Configurations	33
7.2	Lemmas	33
7.2.1	<i>HASates</i>	34
7.2.2	<i>HAEvents</i>	34
7.2.3	<i>NoCycles</i>	35
7.2.4	<i>OneAncestor</i>	35
7.2.5	<i>MutuallyDistinct</i>	35
7.2.6	<i>RootEx</i>	36
7.2.7	<i>HARoot</i>	37
7.2.8	<i>CompFun</i>	38
7.2.9	<i>SAs</i>	40
7.2.10	<i>HAInitState</i>	40
7.2.11	<i>Chi</i>	42
7.2.12	<i>ChiRel</i>	45
7.2.13	<i>ChiPlus</i>	50
7.2.14	<i>ChiStar</i>	58
7.2.15	<i>InitConf</i>	59
7.2.16	<i>StepConf</i>	62

8	Semantics of Hierarchical Automata	63
8.1	Definitions	63
8.1.1	<i>Status</i>	63
8.2	Lemmas	67
8.2.1	<i>IsConfSet</i>	68
8.2.2	<i>InitStatus</i>	68
8.2.3	<i>Events</i>	69
8.2.4	<i>StepStatus</i>	70
8.2.5	Enabled Transitions <i>ET</i>	70
8.2.6	Finite Transition Set	70
8.2.7	<i>PUpdate</i>	71
8.2.8	Higher Priority Transitions <i>HPT</i>	71
8.2.9	Delta Transition Set	72
8.2.10	Target Transition Set	74
8.2.11	Source Transition Set	74
8.2.12	<i>StepActEvents</i>	77
8.2.13	<i>UniqueSucStates</i>	77
8.2.14	<i>RootState</i>	77
8.2.15	Configuration <i>Conf</i>	80
8.2.16	<i>RootExSem</i>	85
8.2.17	<i>StepConf</i>	86
8.2.18	<i>StepStatus</i>	90
8.3	Meta Theorem: Preservation for Statecharts	96
9	Kripke Structures and CTL	96
10	Kripke Structures as Hierarchical Automata	98
11	Constructing Hierarchical Automata	101
11.1	Constructing a Composition Function for a PseudoHA	102
11.2	Extending a Composition Function by a SA	103
11.3	Constructing a PseudoHA	110
11.4	Extending a HA by a SA (<i>AddSA</i>)	112
11.5	Theorems for Calculating Wellformedness of HA	121
12	Example Specification for a Car Audio System	124
12.1	Definitions	124
12.1.1	Data space for two Integer-Variables	124
12.1.2	Sequential Automaton <i>Root-CTRL</i>	125
12.1.3	Sequential Automaton <i>CDPlayer-CTRL</i>	126
12.1.4	Sequential Automaton <i>AudioPlayer-CTRL</i>	127
12.1.5	Sequential Automaton <i>On-CTRL</i>	127
12.1.6	Sequential Automaton <i>TunerMode-CTRL</i>	128
12.1.7	Sequential Automaton <i>CDMode-CTRL</i>	129

12.1.8	Hierarchical Automaton <i>CarAudioSystem</i>	130
12.2	Lemmas	130
12.2.1	Sequential Automaton <i>CDMode-CTRL</i>	130
12.2.2	Sequential Automaton <i>CDPlayer-CTRL</i>	131
12.2.3	Sequential Automaton <i>AudioPlayer-CTRL</i>	132
12.2.4	Sequential Automaton <i>On-CTRL</i>	133
12.2.5	Sequential Automaton <i>TunerMode-CTRL</i>	134
12.2.6	Sequential Automaton <i>CDMode-CTRL</i>	135
12.2.7	Hierarchical Automaton <i>CarAudioSystem</i>	136

1 Contributions to the Standard Library of HOL

```
theory Contrib
imports Main HOL-Library.FuncSet
begin
```

1.1 Basic definitions and lemmas

1.1.1 Maps

```
definition chg-map :: ('b => 'b) => 'a => ('a -> 'b) => ('a -> 'b) where
  chg-map f a m = (case m a of None => m | Some b => m(a|->f b))
```

```
lemma map-some-list [simp]:
  map the (map Some L) = L
apply (induct-tac L)
apply auto
done
```

```
lemma dom-ran-the:
  [| ran G = {y}; x ∈ (dom G) |] ==> (the (G x)) = y
apply (unfold ran-def dom-def)
apply auto
done
```

```
lemma dom-None:
  (S ∉ dom F) ==> (F S = None)
by (unfold dom-def, auto)
```

```
lemma ran-dom-the:
  [| y ∉ Union (ran G); x ∈ dom G |] ==> y ∉ the (G x)
by (unfold ran-def dom-def, auto)
```

```
lemma dom-map-upd: dom(m(a|->b)) = insert a (dom m)
apply auto
done
```

1.1.2 rtranc1

lemma *rtranc1-Int*:

$\llbracket (a,b) \in A; (a,b) \in B \rrbracket \implies (a,b) \in (A \cap B)^{\wedge*}$
by *auto*

lemma *rtranc1-mem-Sigma*:

$\llbracket a \neq b; (a, b) \in (A \times A)^{\wedge*} \rrbracket \implies b \in A$
apply (*frule* *rtranc1D*)
apply (*cut-tac* *r=A × A and A=A in tranc1-subset-Sigma*)
apply *auto*
done

lemma *help-rtranc1-Range*:

$\llbracket a \neq b; (a,b) \in R^{\wedge*} \rrbracket \implies b \in \text{Range } R$
apply (*erule* *rtranc1E*)
apply *auto*
done

lemma *rtranc1-Int-help*:

$(a,b) \in (A \cap B)^{\wedge*} \implies (a,b) \in A^{\wedge*} \wedge (a,b) \in B^{\wedge*}$
apply (*unfold* *Int-def*)
apply *auto*
apply (*rule-tac* *b=b in rtranc1-induct*)
apply *auto*
apply (*rule-tac* *b=b in rtranc1-induct*)
apply *auto*
done

lemmas *rtranc1-Int1* = *rtranc1-Int-help* [*THEN* *conjunct1*]

lemmas *rtranc1-Int2* = *rtranc1-Int-help* [*THEN* *conjunct2*]

lemma *tranc1D3* [*rule-format*]:

$(S,T) \in R^{\wedge+} \implies (S,T) \notin R \longrightarrow (\exists U. (S,U) \in R \wedge (U,T) \in R^{\wedge+})$
apply (*rule-tac* *a=S and b=T and r=R in tranc1-induct*)
apply *auto*
done

lemma *tranc1D4* [*rule-format*]:

$(S,T) \in R^{\wedge+} \implies (S,T) \notin R \longrightarrow (\exists U. (S,U) \in R^{\wedge+} \wedge (U,T) \in R)$
apply (*rule-tac* *a=S and b=T and r=R in tranc1-induct*)
apply *auto*
done

lemma *tranc1-collect* [*rule-format*]:

$\llbracket (x,y) \in R^{\wedge*}; S \notin \text{Domain } R \rrbracket \implies y \neq S \longrightarrow (x,y) \in \{p. \text{fst } p \neq S \wedge \text{snd } p \neq S \wedge p \in R\}^{\wedge*}$
apply (*rule-tac* *b=y in rtranc1-induct*)
apply *auto*
apply (*rule* *rtranc1-into-rtranc1*)

apply *fast*
apply *auto*
done

lemma *trancl-subseteq*:
 $\llbracket R \subseteq Q; S \in R^{\hat{*}} \rrbracket \implies S \in Q^{\hat{*}}$
apply (*frule rtrancl-mono*)
apply *fast*
done

lemma *trancl-Int-subset*:
 $(R \cap (A \times A))^+ \subseteq R^+ \cap (A \times A)$
apply (*rule subsetI*)
apply (*rename-tac S*)
apply (*case-tac S*)
apply (*rename-tac T V*)
apply *auto*
apply (*rule-tac a=T and b=V and r=(R $\cap$ A $\times$ A) in converse-trancl-induct,*
auto)+
done

lemma *trancl-Int-mem*:
 $(S, T) \in (R \cap (A \times A))^+ \implies (S, T) \in R^+ \cap A \times A$
by (*rule trancl-Int-subset [THEN subsetD], assumption*)

lemma *Int-expand*:
 $\{(S, S'). P \ S \ S' \wedge Q \ S \ S'\} = (\{(S, S'). P \ S \ S'\} \cap \{(S, S'). Q \ S \ S'\})$
by *auto*

1.1.3 *finite*

lemma *finite-conj*:
 $finite \ (\{(S, S'). P \ S \ S'\} :: (('a * 'b) set)) \longrightarrow$
 $finite \ \{(S, S'). P \ (S :: 'a) \ (S' :: 'b) \wedge Q \ (S :: 'a) \ (S' :: 'b)\}$
apply (*rule impI*)
apply (*subst Int-expand*)
apply (*rule finite-Int*)
apply *auto*
done

lemma *finite-conj2*:
 $\llbracket finite \ A; finite \ B \rrbracket \implies finite \ (\{(S, S'). S : A \wedge S' : B\})$
by *auto*

1.1.4 *override*

lemma *dom-override-the*:
 $(x \in (dom \ G2)) \longrightarrow ((G1 \ ++ \ G2) \ x) = (G2 \ x)$
by (*auto*)

```

lemma dom-override-the2 [simp]:
   $\llbracket \text{dom } G1 \cap \text{dom } G2 = \{\}; x \in (\text{dom } G1) \rrbracket \implies ((G1 ++ G2) x) = (G1 x)$ 
apply (unfold dom-def map-add-def)
apply auto
apply (drule sym)
apply (erule equalityE)
apply (unfold Int-def)
apply auto
apply (erule-tac x=x in allE)
apply auto
done

```

```

lemma dom-override-the3 [simp]:
   $\llbracket x \notin \text{dom } G2; x \in \text{dom } G1 \rrbracket \implies ((G1 ++ G2) x) = (G1 x)$ 
apply (unfold dom-def map-add-def)
apply auto
done

```

```

lemma Union-ran-override [simp]:
   $S \in \text{dom } G \implies \bigcup (\text{ran } (G ++ \text{Map.empty}(S \mapsto \text{insert } SA (\text{the}(G S))))) =$ 
 $(\text{insert } SA (\text{Union } (\text{ran } G)))$ 
apply (unfold dom-def ran-def)
apply auto
apply (rename-tac T)
apply (case-tac T = S)
apply auto
done

```

```

lemma Union-ran-override2 [simp]:
   $S \in \text{dom } G \implies \bigcup (\text{ran } (G(S \mapsto \text{insert } SA (\text{the}(G S))))) = (\text{insert } SA (\text{Union}$ 
 $(\text{ran } G)))$ 
apply (unfold dom-def ran-def)
apply auto
apply (rename-tac T)
apply (case-tac T = S)
apply auto
done

```

```

lemma ran-override [simp]:
   $(\text{dom } A \cap \text{dom } B) = \{\} \implies \text{ran } (A ++ B) = (\text{ran } A) \cup (\text{ran } B)$ 
apply (unfold Int-def ran-def)
apply (simp add: map-add-Some-iff)
apply auto
done

```

```

lemma chg-map-new [simp]:
   $m a = \text{None} \implies \text{chg-map } f a m = m$ 
by (unfold chg-map-def, auto)

```

lemma *chg-map-upd* [*simp*]:
 $m\ a = \text{Some } b \implies \text{chg-map } f\ a\ m = m(a \mapsto f\ b)$
by (*unfold chg-map-def, auto*)

lemma *ran-override-chg-map*:
 $A \in \text{dom } G \implies \text{ran } (G \text{ ++ Map.empty}(A \mapsto B)) = (\text{ran } (\text{chg-map } (\lambda x. B) A\ G))$
apply (*unfold dom-def ran-def*)
apply (*subst map-add-Some-iff [THEN ext]*)
apply *auto*
apply (*rename-tac T*)
apply (*case-tac T = A*)
apply *auto*
done

1.1.5 Part

definition *Part* :: [*'a set, 'b => 'a*] => *'a set* **where**
 $\text{Part } A\ h = A \cap \{x. \exists z. x = h(z)\}$

lemma *Part-UNIV-Inl-comp*:
 $((\text{Part UNIV } (\text{Inl } o\ f)) = (\text{Part UNIV } (\text{Inl } o\ g))) = ((\text{Part UNIV } f) = (\text{Part UNIV } g))$
apply (*unfold Part-def*)
apply *auto*
apply (*erule equalityE*)
apply (*erule subsetCE*)
apply *auto*
apply (*erule equalityE*)
apply (*erule subsetCE*)
apply *auto*
done

lemma *Part-eqI* [*intro*]: $\llbracket a \in A; a=h(b) \rrbracket \implies a \in \text{Part } A\ h$
by (*auto simp add: Part-def*)

lemmas *PartI* = *Part-eqI* [*OF - refl*]

lemma *PartE* [*elim!*]: $\llbracket a \in \text{Part } A\ h; \forall z. \llbracket a \in A; a=h(z) \rrbracket \implies P \rrbracket \implies P$
by (*auto simp add: Part-def*)

lemma *Part-subset*: $\text{Part } A\ h \subseteq A$
by (*auto simp add: Part-def*)

lemma *Part-mono*: $A \subseteq B \implies \text{Part } A\ h \subseteq \text{Part } B\ h$
by *blast*

lemmas *basic-monos* = *basic-monos Part-mono*

lemma *PartD1*: $a \in \text{Part } A \ h \implies a \in A$
by (*simp add: Part-def*)

lemma *Part-id*: $\text{Part } A \ (\lambda x. x) = A$
by *blast*

lemma *Part-Int*: $\text{Part } (A \cap B) \ h = (\text{Part } A \ h) \cap (\text{Part } B \ h)$
by *blast*

lemma *Part-Collect*: $\text{Part } (A \cap \{x. P \ x\}) \ h = (\text{Part } A \ h) \cap \{x. P \ x\}$
by *blast*

1.1.6 Set operators

lemma *subset-lemma*:
 $\llbracket A \cap B = \{\}; A \subseteq B \rrbracket \implies A = \{\}$
by *auto*

lemma *subset-lemma2*:
 $\llbracket B \cap A = \{\}; C \subseteq A \rrbracket \implies C \cap B = \{\}$
by *auto*

lemma *insert-inter*:
 $\llbracket a \notin A; (A \cap B) = \{\} \rrbracket \implies (A \cap (\text{insert } a \ B)) = \{\}$
by *auto*

lemma *insert-notmem*:
 $\llbracket a \neq b; a \notin B \rrbracket \implies a \notin (\text{insert } b \ B)$
by *auto*

lemma *insert-union*:
 $A \cup (\text{insert } a \ B) = \text{insert } a \ A \cup B$
by *auto*

lemma *insert-or*:
 $\{s. s = t1 \ \vee \ (P \ s)\} = \text{insert } t1 \ \{s. P \ s\}$
by *auto*

lemma *Collect-subset*:
 $\{x. x \subseteq A \wedge P \ x\} = \{x \in \text{Pow } A. P \ x\}$
by *auto*

lemma *OneElement-Card* [*simp*]:
 $\llbracket \text{finite } M; \text{card } M \leq \text{Suc } 0; t \in M \rrbracket \implies M = \{t\}$
apply *auto*
apply (*rename-tac s*)
apply (*rule-tac P=finite M in mp*)
apply (*rule-tac P=card M <= Suc 0 in mp*)

```

apply (rule-tac  $P=t \in M$  in mp)
apply (rule-tac  $F=M$  in finite-induct)
apply auto
apply (rule-tac  $P=finite\ M$  in mp)
apply (rule-tac  $P=card\ M \leq Suc\ 0$  in mp)
apply (rule-tac  $P=s \in M$  in mp)
apply (rule-tac  $P=t \in M$  in mp)
apply (rule-tac  $F=M$  in finite-induct)
apply auto
done

```

1.1.7 One point rule

lemma *Ex1-one-point* [simp]:

$(\exists! x. P\ x \wedge x = a) = P\ a$

by auto

lemma *Ex1-one-point2* [simp]:

$(\exists! x. P\ x \wedge Q\ x \wedge x = a) = (P\ a \wedge Q\ a)$

by auto

lemma *Some-one-point* [simp]:

$P\ a \implies (SOME\ x. P\ x \wedge x = a) = a$

by auto

lemma *Some-one-point2* [simp]:

$\llbracket Q\ a; P\ a \rrbracket \implies (SOME\ x. P\ x \wedge Q\ x \wedge x = a) = a$

by auto

end

2 Partitoned Data Spaces for Statecharts

theory *DataSpace*

imports *Contrib*

begin

2.1 Definitions

definition

$DataSpace :: ('d\ set)\ list$

$=> bool$ **where**

$DataSpace\ L = ((distinct\ L) \wedge$

$(\forall\ D1 \in (set\ L). \forall\ D2 \in (set\ L).$

$D1 \neq D2 \implies ((D1 \cap D2) = \{\})) \wedge$

$((\bigcup (set\ L)) = UNIV))$

lemma *DataSpace-EmptySet*:

$[UNIV] \in \{ L \mid L.\ DataSpace\ L \}$

by (*unfold DataSpace-def*, *auto*)

definition *dataspace* = { *L* | (*L*::('d set) list). *DataSpace L*}

typedef 'd *dataspace* = *dataspace* :: 'd set list set
unfolding *dataspace-def*
apply (*rule exI*)
apply (*rule DataSpace-EmptySet*)
done

definition
PartNum :: ('d) *dataspace* => nat **where**
PartNum = *length o Rep-dataspace*

definition
PartDom :: ['d *dataspace*, nat] => ('d set) (**infixl** <!*D*!> 101) **where**
PartDom d n = (*Rep-dataspace d*) ! *n*

2.2 Lemmas

2.2.1 *DataSpace*

lemma *DataSpace-UNIV* [*simp*]:
DataSpace [*UNIV*]
by (*unfold DataSpace-def*, *auto*)

lemma *DataSpace-select*:
DataSpace (*Rep-dataspace L*)
apply (*cut-tac x=L in Rep-dataspace*)
apply (*unfold dataspace-def*)
apply *auto*
done

lemma *UNIV-dataspace* [*simp*]:
[*UNIV*] ∈ *dataspace*
by (*unfold dataspace-def*, *auto*)

lemma *Inl-Inr-DataSpace* [*simp*]:
DataSpace [*Part UNIV Inl*, *Part UNIV Inr*]
apply (*unfold DataSpace-def*)
apply *auto*
apply (*rename-tac d*)
apply (*rule-tac b=(inv Inl) d in Part-eqI*)
apply *auto*
apply (*rule sym*)
apply (*case-tac d*)
apply *auto*
done

lemma *Inl-Inr-dataspace* [*simp*]:

$[Part\ UNIV\ Inl, Part\ UNIV\ Inr] \in dataspace$
by (*unfold dataspace-def*, *auto*)

lemma *InlInr-InlInl-Inr-DataSpace* [*simp*]:
 $DataSpace\ [Part\ UNIV\ (Inl\ o\ Inr), Part\ UNIV\ (Inl\ o\ Inl), Part\ UNIV\ Inr]$
apply (*unfold DataSpace-def*)
apply *auto*
apply (*unfold Part-def*)
apply *auto*
apply (*rename-tac x*)
apply (*case-tac x*)
apply *auto*
apply (*rename-tac a*)
apply (*case-tac a*)
apply *auto*
done

lemma *InlInr-InlInl-Inr-dataspace* [*simp*]:
 $[Part\ UNIV\ (Inl\ o\ Inr), Part\ UNIV\ (Inl\ o\ Inl), Part\ UNIV\ Inr] : dataspace$
by (*unfold dataspace-def*, *auto*)

2.2.2 PartNum

lemma *PartDom-PartNum-distinct*:

$$\llbracket i < PartNum\ d; j < PartNum\ d; \\ i \neq j; p \in (d\ !D!\ i) \rrbracket \implies \\ p \notin (d\ !D!\ j)$$

apply *auto*
apply (*cut-tac L=d in DataSpace-select*)
apply (*unfold DataSpace-def*)
apply *auto*
apply (*erule-tac x=Rep-dataspace d ! i in ballE*)
apply (*erule-tac x=Rep-dataspace d ! j in ballE*)
apply *auto*
apply (*simp add:distinct-conv-nth PartNum-def*)
apply (*unfold PartDom-def PartNum-def*)
apply *auto*
done

lemma *PartDom-PartNum-distinct2*:

$$\llbracket i < PartNum\ d; j < PartNum\ d; \\ i \neq j; p \in (d\ !D!\ j) \rrbracket \implies \\ p \notin (d\ !D!\ i)$$

apply *auto*
apply (*cut-tac L=d in DataSpace-select*)
apply (*unfold DataSpace-def*)
apply *auto*
apply (*erule-tac x=Rep-dataspace d ! i in ballE*)
apply (*erule-tac x=Rep-dataspace d ! j in ballE*)

```

apply auto
apply (simp add:distinct-conv-nth PartNum-def)
apply (unfold PartDom-def PartNum-def)
apply auto
done

lemma PartNum-length [simp]:
  (DataSpace L)  $\implies$  (PartNum (Abs-dataspace L) = (length L))
apply (unfold PartNum-def)
apply auto
apply (subst Abs-dataspace-inverse)
apply (unfold dataspace-def)
apply auto
done

end

```

3 Data Space Assignments

```

theory Data
imports DataSpace
begin

```

3.1 Total data space assignments

```

definition
  Data :: ['d list, 'd dataspace]
     $\Rightarrow$  bool where
    Data L D = (((length L) = (PartNum D))  $\wedge$ 
      ( $\forall i \in \{n. n < (\text{PartNum } D)\}. (L!i) \in (\text{PartDom } D \ i)$ )))

```

```

lemma Data-EmptySet:
  ([@ t. True], Abs-dataspace [UNIV])  $\in$  { (L,D) | L D. Data L D }
apply (unfold Data-def PartDom-def)
apply auto
apply (subst Abs-dataspace-inverse)
apply auto
done

```

```

definition
  data =
    { (L,D) |
      (L::('d list))
      (D::('d dataspace)).
      Data L D }

```

```

typedef 'd data = data :: ('d list * 'd dataspace) set
unfolding data-def
apply (rule exI)

```

apply (*rule Data-EmptySet*)
done

definition

$DataValue :: ('d\ data) => ('d\ list)$ **where**
 $DataValue = fst\ o\ Rep-data$

definition

$DataSpace :: ('d\ data) => ('d\ dataspace)$ **where**
 $DataSpace = snd\ o\ Rep-data$

definition

$DataPart :: ['d\ data, nat] => 'd\ (\l(-\ !P!\ -)\r [10,11]10)$ **where**
 $DataPart\ d\ n = (DataValue\ d)\ !\ n$

lemma *Rep-data-tuple*:

$Rep-data\ D = (DataValue\ D, DataSpace\ D)$
by (*unfold DataValue-def DataSpace-def, simp*)

lemma *Rep-data-select*:

$(DataValue\ D, DataSpace\ D) \in data$
apply (*subst Rep-data-tuple [THEN sym]*)
apply (*rule Rep-data*)
done

lemma *Data-select*:

$Data\ (DataValue\ D)\ (DataSpace\ D)$
apply (*cut-tac D=D in Rep-data-select*)
apply (*unfold data-def*)
apply *auto*
done

lemma *length-DataValue-PartNum [simp]*:

$length\ (DataValue\ D) = PartNum\ (Data.DataSpace\ D)$
apply (*cut-tac D=D in Data-select*)
apply (*unfold Data-def*)
apply *auto*
done

lemma *DataValue-PartDom [simp]*:

$i < PartNum\ (Data.DataSpace\ D) \implies$
 $DataValue\ D\ !\ i \in PartDom\ (Data.DataSpace\ D)\ i$
apply (*cut-tac D=D in Data-select*)
apply (*unfold Data-def*)
apply *auto*
done

lemma *DataPart-PartDom [simp]*:

$i < PartNum\ (Data.DataSpace\ d) \longrightarrow (d\ !P!\ i) \in ((Data.DataSpace\ d)\ !D!\ i)$

```

apply (unfold DataPart-def)
apply auto
done

```

3.2 Partial data space assignments

definition

$$\begin{aligned}
 PData &:: ['d \text{ option list}, 'd \text{ dataspace}] \Rightarrow \text{bool} \text{ where} \\
 PData \ L \ D &== ((\text{length } L) = (\text{PartNum } D)) \wedge \\
 &\quad (\forall \ i \in \{n. \ n < (\text{PartNum } D)\}. \\
 &\quad (L!i) \neq \text{None} \longrightarrow \text{the } (L!i) \in (\text{PartDom } D \ i))
 \end{aligned}$$

lemma *PData-EmptySet*:

$$([\text{Some } (@ \ t. \ \text{True})], \text{Abs-dataspace } [UNIV]) \in \{ (L,D) \mid L \ D. \ PData \ L \ D \}$$

```

apply (unfold PData-def PartDom-def)
apply auto
apply (subst Abs-dataspace-inverse)
apply auto
done

```

definition

$$\begin{aligned}
 pdata &= \\
 &\{ (L,D) \mid \\
 &\quad (L::('d \text{ option list})) \\
 &\quad (D::('d \text{ dataspace})). \\
 &\quad PData \ L \ D \}
 \end{aligned}$$

```

typedef 'd pdata = pdata :: ('d option list * 'd dataspace) set
unfolding pdata-def
apply (rule exI)
apply (rule PData-EmptySet)
done

```

definition

$$\begin{aligned}
 PDataValue &:: ('d \text{ pdata}) \Rightarrow ('d \text{ option list}) \text{ where} \\
 PDataValue &= \text{fst } o \ \text{Rep-pdata}
 \end{aligned}$$

definition

$$\begin{aligned}
 PDataSpace &:: ('d \text{ pdata}) \Rightarrow ('d \text{ dataspace}) \text{ where} \\
 PDataSpace &= \text{snd } o \ \text{Rep-pdata}
 \end{aligned}$$

definition

$$\begin{aligned}
 Data2PData &:: ('d \text{ data}) \Rightarrow ('d \text{ pdata}) \text{ where} \\
 Data2PData \ D &= (\text{let} \\
 &\quad (L,DP) = \text{Rep-data } D; \\
 &\quad OL = \text{map } \text{Some } L \\
 &\text{in} \\
 &\quad \text{Abs-pdata } (OL,DP))
 \end{aligned}$$

definition

$PData2Data :: ('d \text{ pdata}) \Rightarrow ('d \text{ data})$ **where**
 $PData2Data \ D =$ (let
 $(OL, DP) = Rep\text{-}pdata \ D;$
 $L = \text{map the } OL$
 in
 $Abs\text{-}data \ (L, DP)$)

definition

$DefaultPData :: ('d \text{ dataspace}) \Rightarrow ('d \text{ pdata})$ **where**
 $DefaultPData \ D = Abs\text{-}pdata \ (\text{replicate} \ (PartNum \ D) \ None, \ D)$

definition

$OptionOverride :: ('d \text{ option} * 'd) \Rightarrow 'd$ **where**
 $OptionOverride \ P = (\text{if } (fst \ P) = None \text{ then } (snd \ P) \text{ else } (the \ (fst \ P)))$

definition

$DataOverride :: ['d \text{ pdata}, 'd \text{ data}] \Rightarrow 'd \text{ data}$ ($\langle (- [D+] / -) \rangle [10, 11] 10$) **where**
 $DataOverride \ D1 \ D2 =$
 (let
 $(L1, DP1) = Rep\text{-}pdata \ D1;$
 $(L2, DP2) = Rep\text{-}data \ D2;$
 $L = \text{map } OptionOverride \ (\text{zip } L1 \ L2)$
 in
 $Abs\text{-}data \ (L, DP2)$)

lemma Rep-pdata-tuple:

$Rep\text{-}pdata \ D = (PDataValue \ D, PDataSpace \ D)$
apply ($\text{unfold } PDataValue\text{-}def \ PDataSpace\text{-}def$)
apply (simp)
done

lemma Rep-pdata-select:

$(PDataValue \ D, PDataSpace \ D) \in \text{pdata}$
apply ($\text{subst } Rep\text{-}pdata\text{-}tuple \ [THEN \ \text{sym}]$)
apply ($\text{rule } Rep\text{-}pdata$)
done

lemma PData-select:

$PData \ (PDataValue \ D) \ (PDataSpace \ D)$
apply ($\text{cut-tac } D=D \text{ in } Rep\text{-}pdata\text{-}select$)
apply ($\text{unfold } pdata\text{-}def$)
apply auto
done

3.2.1 DefaultPData**lemma PData-DefaultPData [simp]:**

$PData \ (\text{replicate} \ (PartNum \ D) \ None) \ D$


```

apply (unfold PData-def)
apply auto
done

```

```

lemma pdata-DefaultPData [simp]:
  (replicate (PartNum D) None, D) ∈ pdata
apply (unfold pdata-def)
apply auto
done

```

```

lemma PDataSpace-DefaultPData [simp]:
  PDataSpace (DefaultPData D) = D
apply (unfold DataSpace-def PDataSpace-def DefaultPData-def)
apply auto
apply (subst Abs-pdata-inverse)
apply auto
done

```

```

lemma length-PartNum-PData [simp]:
  length (PDataValue P) = PartNum (PDataSpace P)
apply (cut-tac D=P in Rep-pdata-select)
apply (unfold pdata-def PData-def)
apply auto
done

```

3.2.2 Data2PData

```

lemma PData-Data2PData [simp]:
  PData (map Some (DataValue D)) (Data.DataSpace D)
apply (unfold PData-def)
apply auto
done

```

```

lemma pdata-Data2PData [simp]:
  (map Some (DataValue D), Data.DataSpace D) ∈ pdata
apply (unfold pdata-def)
apply auto
done

```

```

lemma DataSpace-Data2PData [simp]:
  (PDataSpace (Data2PData D)) = (Data.DataSpace D)
apply (unfold DataSpace-def PDataSpace-def Data2PData-def Let-def)
apply auto
apply (cut-tac D=D in Rep-data-tuple)
apply auto
apply (subst Abs-pdata-inverse)
apply auto
done

```

```

lemma PDataValue-Data2PData-DataValue [simp]:
  (map the (PDataValue (Data2PData D))) = DataValue D
apply (unfold DataValue-def PDataValue-def Data2PData-def Let-def)
apply auto
apply (cut-tac D=D in Rep-data-tuple)
apply auto
apply (subst Abs-pdata-inverse)
apply simp
apply (simp del: map-map)
done

```

```

lemma DataSpace-PData2Data:
  Data (map the (PDataValue D)) (PDataSpace D)  $\implies$ 
  (Data.DataSpace (PData2Data D) = (PDataSpace D))
apply (unfold DataSpace-def PDataSpace-def PData2Data-def Let-def)
apply auto
apply (cut-tac D=D in Rep-pdata-tuple)
apply auto
apply (subst Abs-data-inverse)
apply (unfold data-def)
apply auto
done

```

```

lemma PartNum-PDataValue-PartDom [simp]:
   $\llbracket i < \text{PartNum } (PDataSpace Q);$ 
     $PDataValue Q ! i = \text{Some } y \rrbracket \implies$ 
     $y \in \text{PartDom } (PDataSpace Q) i$ 
apply (cut-tac D=Q in Rep-pdata-select)
apply (unfold pdata-def PData-def)
apply auto
done

```

3.2.3 DataOverride

```

lemma Data-DataOverride:
  ((PDataSpace P) = (Data.DataSpace Q))  $\implies$ 
  Data (map OptionOverride (zip (PDataValue P) (Data.DataValue Q))) (Data.DataSpace
  Q)
apply (unfold Data-def)
apply auto
apply (unfold OptionOverride-def)
apply auto
apply (rename-tac i D)
apply (case-tac PDataValue P ! i = None)
apply auto
apply (drule sym)
apply auto
done

```

lemma *data-DataOverride*:
 $((PDataSpace\ P) = (Data.DataSpace\ Q)) \implies$
 $(map\ OptionOverride\ (zip\ (PDataValue\ P)\ (Data.DataValue\ Q)),\ Data.DataSpace\ Q) \in data$
apply (*unfold data-def*)
apply *auto*
apply (*rule Data-DataOverride*)
apply *fast*
done

lemma *DataSpace-DataOverride [simp]*:
 $((Data.DataSpace\ D) = (PDataSpace\ E)) \implies$
 $Data.DataSpace\ (E\ [D+]\ D) = (Data.DataSpace\ D)$
apply (*unfold DataSpace-def DataOverride-def Let-def*)
apply *auto*
apply (*cut-tac D=D in Rep-data-tuple*)
apply (*cut-tac D=E in Rep-pdata-tuple*)
apply *auto*
apply (*subst Abs-data-inverse*)
apply *auto*
apply (*drule sym*)
apply *simp*
apply (*rule data-DataOverride*)
apply *auto*
done

lemma *DataValue-DataOverride [simp]*:
 $((PDataSpace\ P) = (Data.DataSpace\ Q)) \implies$
 $(DataValue\ (P\ [D+]\ Q)) = (map\ OptionOverride\ (zip\ (PDataValue\ P)\ (Data.DataValue\ Q)))$
apply (*unfold DataValue-def DataOverride-def Let-def*)
apply *auto*
apply (*cut-tac D=P in Rep-pdata-tuple*)
apply (*cut-tac D=Q in Rep-data-tuple*)
apply *auto*
apply (*subst Abs-data-inverse*)
apply *auto*
apply (*rule data-DataOverride*)
apply *auto*
done

3.2.4 OptionOverride

lemma *DataValue-OptionOverride-nth*:
 $\llbracket ((PDataSpace\ P) = (DataSpace\ Q));$
 $i < PartNum\ (DataSpace\ Q) \rrbracket \implies$
 $(DataValue\ (P\ [D+]\ Q)\ !\ i) =$
 $OptionOverride\ (PDataValue\ P\ !\ i,\ DataValue\ Q\ !\ i)$
apply *auto*

done

lemma *None-OptionOverride* [simp]:

$(fst\ P) = None \implies OptionOverride\ P = (snd\ P)$

apply (unfold *OptionOverride-def*)

apply *auto*

done

lemma *Some-OptionOverride* [simp]:

$(fst\ P) \neq None \implies OptionOverride\ P = the\ (fst\ P)$

apply (unfold *OptionOverride-def*)

apply *auto*

done

end

4 Update-Functions on Data Spaces

theory *Update*

imports *Data*

begin

4.1 Total update-functions

definition

$Update :: ('d\ data) \Rightarrow ('d\ data)) \Rightarrow bool$ **where**

$Update\ U = (\forall\ d. Data.DataSpace\ d = DataSpace\ (U\ d))$

lemma *Update-EmptySet*:

$(\% d. d) \in \{ L \mid L. Update\ L \}$

by (unfold *Update-def*, *auto*)

definition

$update = \{ L \mid (L :: ('d\ data) \Rightarrow ('d\ data))). Update\ L \}$

typedef $'d\ update = update :: ('d\ data \Rightarrow 'd\ data)\ set$

unfolding *update-def*

apply (rule *exI*)

apply (rule *Update-EmptySet*)

done

definition

$UpdateApply :: ['d\ update, 'd\ data] \Rightarrow 'd\ data\ (\langle (-\ !!!\ / -) \rangle [10,11]10)$ **where**

$UpdateApply\ U\ D == Rep-update\ U\ D$

definition

$DefaultUpdate :: ('d\ update)$ **where**

$DefaultUpdate == Abs-update\ (\lambda\ D. D)$

4.1.1 Basic lemmas

```

lemma Update-select:
  Update (Rep-update U)
apply (cut-tac x=U in Rep-update)
apply (unfold update-def)
apply auto
done

```

```

lemma DataSpace-DataSpace-Update [simp]:
  Data.DataSpace (Rep-update U DP) = Data.DataSpace DP
apply (cut-tac U=U in Update-select)
apply (unfold Update-def)
apply auto
done

```

4.1.2 DefaultUpdate

```

lemma Update-DefaultUpdate [simp]:
  Update (λ D. D)
by (unfold Update-def, auto)

```

```

lemma update-DefaultUpdate [simp]:
   $(\lambda D. D) \in \text{update}$ 
by (unfold update-def, auto)

```

```

lemma DataSpace-UpdateApply [simp]:
  Data.DataSpace (U !!! D) = Data.DataSpace D
by (unfold UpdateApply-def, auto)

```

4.2 Partial update-functions

definition

```

PUpdate :: (('d data) => ('d pdata)) => bool where
PUpdate U = ( $\forall d. \text{Data.DataSpace } d = \text{PDataSpace } (U d)$ )

```

```

lemma PUpdate-EmptySet:
   $(\% d. \text{Data2PData } d) \in \{ L \mid L. \text{PUpdate } L \}$ 
by (unfold PUpdate-def, auto)

```

```

definition pupdate = { L | (L::('d data) => ('d pdata)). PUpdate L }

```

```

typedef 'd pupdate = pupdate :: ('d data => 'd pdata) set
  unfolding pupdate-def
  apply (rule exI)
  apply (rule PUpdate-EmptySet)
done

```

definition

```

PUpdateApply :: ['d pupdate, 'd data] => 'd pdata ( $\langle (- !!! -) \rangle [10,11] 10$ ) where

```

$PUpdateApply\ U\ D = Rep-pupdate\ U\ D$

definition

$DefaultPUpdate :: ('d\ pupdate)\ \mathbf{where}$
 $DefaultPUpdate = Abs-pupdate\ (\lambda\ D.\ DefaultPData\ (Data.DataSpace\ D))$

4.2.1 Basic lemmas

lemma $PUpdate-select$:

$PUpdate\ (Rep-pupdate\ U)$
apply $(cut-tac\ x=U\ \mathbf{in}\ Rep-pupdate)$
apply $(unfold\ pupdate-def)$
apply $auto$
done

lemma $DataSpace-PDataSpace-PUpdate\ [simp]$:

$PDataSpace\ (Rep-pupdate\ U\ DP) = Data.DataSpace\ DP$
apply $(cut-tac\ U=U\ \mathbf{in}\ PUpdate-select)$
apply $(unfold\ PUpdate-def)$
apply $auto$
done

4.2.2 $Data2PData$

lemma $PUpdate-Data2PData\ [simp]$:

$PUpdate\ Data2PData$
by $(unfold\ PUpdate-def,\ auto)$

lemma $pupdate-Data2PData\ [simp]$:

$Data2PData \in pupdate$
by $(unfold\ pupdate-def,\ auto)$

4.2.3 $PUpdate$

lemma $PUpdate-DefaultPUpdate\ [simp]$:

$PUpdate\ (\lambda\ D.\ DefaultPData\ (Data.DataSpace\ D))$
apply $(unfold\ PUpdate-def)$
apply $auto$
done

lemma $pupdate-DefaultPUpdate\ [simp]$:

$(\lambda\ D.\ DefaultPData\ (Data.DataSpace\ D)) \in pupdate$
apply $(unfold\ pupdate-def)$
apply $auto$
done

lemma $DefaultPUpdate-None\ [simp]$:

$(DefaultPUpdate\ !!\ D) = DefaultPData\ (DataSpace\ D)$
apply $(unfold\ DefaultPUpdate-def\ PUpdateApply-def)$
apply $(subst\ Abs-pupdate-inverse)$

apply *auto*
done

4.2.4 SequentialRacing

definition

$UpdateOverride :: ['d\ pupdate, 'd\ update] \Rightarrow$
 $\quad 'd\ update \ (\langle (-\ [U+]/\ -) \rangle\ [10,11]10) \textbf{ where}$
 $UpdateOverride\ U\ P = Abs\text{-}update\ (\lambda\ DA.\ (U\ !!\ DA)\ [D+]\ (P\ !!!\ DA))$

inductive

$FoldSet :: ('b \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'b\ set \Rightarrow 'a \Rightarrow bool$
for $h :: 'b \Rightarrow 'a \Rightarrow 'a$
and $z :: 'a$

where

$emptyI\ [intro]: FoldSet\ h\ z\ \{\}\ z$
 $| insertI\ [intro]:$
 $\quad \llbracket x \notin A; FoldSet\ h\ z\ A\ y \rrbracket$
 $\quad \implies FoldSet\ h\ z\ (insert\ x\ A)\ (h\ x\ y)$

definition

$SequentialRacing :: ('d\ pupdate\ set) \Rightarrow ('d\ update\ set) \textbf{ where}$
 $SequentialRacing\ U =$
 $\quad \{u.\ FoldSet\ UpdateOverride\ DefaultUpdate\ U\ u\}$

lemma *FoldSet-imp-finite*:

$FoldSet\ h\ z\ A\ x \implies finite\ A$
by (*induct set: FoldSet*) *auto*

lemma *finite-imp-FoldSet*:

$finite\ A \implies \exists\ x.\ FoldSet\ h\ z\ A\ x$
by (*induct set: finite*) *auto*

lemma *finite-SequentialRacing*:

$finite\ US \implies (SOME\ u.\ u \in SequentialRacing\ US) \in SequentialRacing\ US$

apply (*unfold SequentialRacing-def*)

apply *auto*

apply (*drule-tac h=UpdateOverride and z=DefaultUpdate in finite-imp-FoldSet*)

apply *auto*

apply (*rule someI*)

apply *auto*

done

end

5 Label Expressions

```
theory Expr
imports Update
begin
```

unbundle *no bit-operations-syntax*

```
datatype ('s,'e)expr = true
| In 's
| En 'e
| NOT ('s,'e)expr
| And ('s,'e)expr ('s,'e)expr
| Or ('s,'e)expr ('s,'e)expr
```

```
type-synonym 'd guard = ('d data) => bool
type-synonym ('e,'d)action = ('e set * 'd pupdate)
type-synonym ('s,'e,'d)label = (('s,'e)expr * 'd guard * ('e,'d)action)
type-synonym ('s,'e,'d)trans = ('s * ('s,'e,'d)label * 's)
```

```
primrec
  eval-expr :: [('s set * 'e set), ('s,'e)expr] => bool where
    eval-expr sc true      = True
  | eval-expr sc (En ev)    = (ev ∈ snd sc)
  | eval-expr sc (In st)    = (st ∈ fst sc)
  | eval-expr sc (NOT e1)   = (¬ (eval-expr sc e1))
  | eval-expr sc (And e1 e2) = ((eval-expr sc e1) ∧ (eval-expr sc e2))
  | eval-expr sc (Or e1 e2) = ((eval-expr sc e1) ∨ (eval-expr sc e2))
```

```
primrec
  ExprEvents :: ('s,'e)expr => 'e set where
    ExprEvents true      = {}
  | ExprEvents (En ev)    = {ev}
  | ExprEvents (In st)    = {}
  | ExprEvents (NOT e)    = (ExprEvents e)
  | ExprEvents (And e1 e2) = ((ExprEvents e1) ∪ (ExprEvents e2))
  | ExprEvents (Or e1 e2) = ((ExprEvents e1) ∪ (ExprEvents e2))
```

```
datatype ('s, 'e, dead 'd)atomar =
  TRUE
| FALSE
| IN 's
| EN 'e
```


| VAL 'd data => bool

definition

source :: ('s,'e,'d)trans => 's **where**
source t = fst t

definition

Source :: ('s,'e,'d)trans set => 's set **where**
Source T == *source* ' T

definition

target :: ('s,'e,'d)trans => 's **where**
target t = snd(snd t)

definition

Target :: ('s,'e,'d)trans set => 's set **where**
Target T = *target* ' T

definition

label :: ('s,'e,'d)trans => ('s,'e,'d)label **where**
label t = fst (snd t)

definition

Label :: ('s,'e,'d)trans set => ('s,'e,'d)label set **where**
Label T = *label* ' T

definition

expr :: ('s,'e,'d)label => ('s,'e)expr **where**
expr = fst

definition

action :: ('s,'e,'d)label => ('e,'d)action **where**
action = snd o snd

definition

Action :: ('s,'e,'d)label set => ('e,'d)action set **where**
Action L = *action* ' L

definition

pupdate :: ('s,'e,'d)label => 'd pupdate **where**
pupdate = snd o action

definition

PUpdate :: ('s,'e,'d)label set => ('d pupdate) set **where**
PUpdate L = *pupdate* ' L

definition

actevent :: ('s,'e,'d)label => 'e set **where**
actevent = fst o action

definition

Actevent :: ('s,'e,'d)label set => ('e set) set **where**
Actevent L = *actevent* ' L

definition

guard :: ('s,'e,'d)label => 'd guard **where**
guard = *fst* o *snd*

definition

Guard :: ('s,'e,'d)label set => ('d guard) set **where**
Guard L = *guard* ' L

definition

defaultexpr :: ('s,'e)expr **where**
defaultexpr = *true*

definition

defaultaction :: ('e,'d)action **where**
defaultaction = ({},DefaultPUpdate)

definition

defaultguard :: ('d guard) **where**
defaultguard = (λ d. *True*)

definition

defaultlabel :: ('s,'e,'d)label **where**
defaultlabel = (*defaultexpr*, *defaultguard*, *defaultaction*)

definition

eval :: [(('s set * 'e set * 'd data), ('s,'e,'d)label)] => bool
($\hookleftarrow$ - $\models$ - $\hookrightarrow$ [91,90]90) **where**
eval *scd* l = (*let* (s,e,d) = *scd*
in
((*eval-expr* (s,e) (*expr* l)) $\wedge$ ((*guard* l) d)))

lemma *Source-EmptySet* [*simp*]:

Source {} = {}
by (*unfold Source-def*, *auto*)

lemma *Target-EmptySet* [*simp*]:

Target {} = {}
by (*unfold Target-def*, *auto*)

lemma *Label-EmptySet* [*simp*]:

Label {} = {}
by (*unfold Label-def*, *auto*)

lemma *Action-EmptySet* [*simp*]:

Action $\{\} = \{\}$
by (*unfold Action-def*, *auto*)

lemma *PUpdate-EmptySet* [*simp*]:
PUpdate $\{\} = \{\}$
by (*unfold PUpdate-def*, *auto*)

lemma *Actevent-EmptySet* [*simp*]:
Actevent $\{\} = \{\}$
by (*unfold Actevent-def*, *auto*)

lemma *Union-Actevent-subset*:
 $\llbracket m \in M; ((\bigcup (Actevent (Label (Union M)))) \subseteq (N::'a set)) \rrbracket \implies$
 $((\bigcup (Actevent (Label m))) \subseteq N)$
by (*unfold Actevent-def Label-def*, *auto*)

lemma *action-select* [*simp*]:
action $(a,b,c) = c$
by (*unfold action-def*, *auto*)

lemma *label-select* [*simp*]:
label $(a,b,c) = b$
by (*unfold label-def*, *auto*)

lemma *target-select* [*simp*]:
target $(a,b,c) = c$
by (*unfold target-def*, *auto*)

lemma *actevent-select* [*simp*]:
actevent $(a,b,(c,d)) = c$
by (*unfold actevent-def*, *auto*)

lemma *pupdate-select* [*simp*]:
pupdate $(a,b,c,d) = d$
by (*unfold pupdate-def*, *auto*)

lemma *source-select* [*simp*]:
source $(a,b) = a$
by (*unfold source-def*, *auto*)

lemma *finite-PUpdate* [*simp*]:
 $finite\ S \implies finite(PUpdate\ S)$
by (*unfold PUpdate-def*, *auto*)

lemma *finite-Label* [*simp*]:
 $finite\ S \implies finite(Label\ S)$
by (*unfold Label-def*, *auto*)

lemma *fst-defaultaction* [*simp*]:

```

    fst defaultaction = {}
  by (unfold defaultaction-def, auto)

lemma action-defaultlabel [simp]:
  (action defaultlabel) = defaultaction
  by (unfold defaultlabel-def action-def, auto)

lemma fst-defaultlabel [simp]:
  (fst defaultlabel) = defaultexpr
  by (unfold defaultlabel-def, auto)

lemma ExprEvents-defaultexpr [simp]:
  (ExprEvents defaultexpr) = {}
  by (unfold defaultexpr-def, auto)

lemma defaultlabel-defaultexpr [simp]:
  expr defaultlabel = defaultexpr
  by (unfold defaultlabel-def expr-def, auto)

lemma target-Target [simp]:
  t ∈ T ⇒ target t ∈ Target T
  by (unfold Target-def, auto)

lemma Source-union : Source s ∪ Source t = Source (s ∪ t)
  apply (unfold Source-def)
  apply auto
  done

lemma Target-union : Target s ∪ Target t = Target (s ∪ t)
  apply (unfold Target-def)
  apply auto
  done

end

```

6 Sequential Automata

```

theory SA
imports Expr
begin

```

definition

```

  SeqAuto :: ['s set,
              's,
              (('s,'e,'d)label) set,
              (('s,'e,'d)trans) set]
    => bool where
    SeqAuto S I L D = (I ∈ S ∧ S ≠ {} ∧ finite S ∧ finite D ∧
      (∀ (s,l,t) ∈ D. s ∈ S ∧ t ∈ S ∧ l ∈ L))

```

lemma *SeqAuto-EmptySet*:
 $(\{\text{@}x . \text{True}\}, (\text{@}x . \text{True}), \{\}, \{\}) \in \{(S, I, L, D) \mid S \text{ I } L \text{ D. SeqAuto } S \text{ I } L \text{ D}\}$
by (*unfold SeqAuto-def*, *auto*)

definition

seqauto =
 $\{ (S, I, L, D) \mid$
 $\quad (S :: 's \text{ set})$
 $\quad (I :: 's)$
 $\quad (L :: (('s, 'e, 'd) \text{label}) \text{ set})$
 $\quad (D :: (('s, 'e, 'd) \text{trans}) \text{ set}).$
 $\text{SeqAuto } S \text{ I } L \text{ D} \}$

typedef $('s, 'e, 'd) \text{ seqauto} =$
 $\text{seqauto} :: ('s \text{ set} * 's * (('s, 'e, 'd) \text{label}) \text{ set} * (('s, 'e, 'd) \text{trans}) \text{ set}) \text{ set}$
unfolding *seqauto-def*
apply (*rule exI*)
apply (*rule SeqAuto-EmptySet*)
done

definition

States :: $((s, 'e, 'd) \text{seqauto}) \Rightarrow 's \text{ set}$ **where**
States = *fst o Rep-seqauto*

definition

InitState :: $((s, 'e, 'd) \text{seqauto}) \Rightarrow 's$ **where**
InitState = *fst o snd o Rep-seqauto*

definition

Labels :: $((s, 'e, 'd) \text{seqauto}) \Rightarrow (('s, 'e, 'd) \text{label}) \text{ set}$ **where**
Labels = *fst o snd o snd o Rep-seqauto*

definition

Delta :: $((s, 'e, 'd) \text{seqauto}) \Rightarrow (('s, 'e, 'd) \text{trans}) \text{ set}$ **where**
Delta = *snd o snd o snd o Rep-seqauto*

definition

SAEvents :: $((s, 'e, 'd) \text{seqauto}) \Rightarrow 'e \text{ set}$ **where**
SAEvents *SA* = $(\bigcup l \in \text{Label } (\text{Delta } \text{SA}). (\text{fst } (\text{action } l)) \cup (\text{ExprEvents } (\text{expr } l)))$

lemma *Rep-seqauto-tuple*:

Rep-seqauto *SA* = (*States* *SA*, *InitState* *SA*, *Labels* *SA*, *Delta* *SA*)
by (*unfold States-def InitState-def Labels-def Delta-def*, *auto*)

lemma *Rep-seqauto-select*:

$(\text{States } \text{SA}, \text{InitState } \text{SA}, \text{Labels } \text{SA}, \text{Delta } \text{SA}) \in \text{seqauto}$
by (*rule Rep-seqauto-tuple [THEN subst]*, *rule Rep-seqauto*)

```

lemma SeqAuto-select:
  SeqAuto (States SA) (InitState SA) (Labels SA) (Delta SA)
by (cut-tac SA=SA in Rep-seqauto-select, unfold seqauto-def, auto)

lemma neq-States [simp]:
  States SA  $\neq$  {}
apply (cut-tac Rep-seqauto-select)
apply auto
apply (unfold seqauto-def SeqAuto-def)
apply auto
done

lemma SA-States-disjunct :
  (States A)  $\cap$  (States A') = {}  $\implies$  A'  $\neq$  A
by auto

lemma SA-States-disjunct2 :
   $\llbracket$  (States A)  $\cap$  C = {} ; States B  $\subseteq$  C  $\rrbracket \implies$  B  $\neq$  A
apply (rule SA-States-disjunct)
apply auto
done

lemma SA-States-disjunct3 :
   $\llbracket$  C  $\cap$  States A = {} ; States B  $\subseteq$  C  $\rrbracket \implies$  States A  $\cap$  States B = {}
apply (cut-tac SA=B in neq-States)
apply fast
done

lemma EX-State-SA [simp]:
   $\exists$  S. S  $\in$  States SA
apply (cut-tac Rep-seqauto-select)
apply (unfold seqauto-def SeqAuto-def)
apply auto
done

lemma finite-States [simp]:
  finite (States A)
apply (cut-tac Rep-seqauto-select)
apply (unfold seqauto-def SeqAuto-def)
apply auto
done

lemma finite-Delta [simp]:
  finite (Delta A)
apply (cut-tac Rep-seqauto-select)
apply (unfold seqauto-def SeqAuto-def)
apply auto
done

```

lemma *InitState-States* [simp]:

InitState A $\in$ *States A*
apply (*cut-tac Rep-seqauto-select*)
apply (*unfold seqauto-def SeqAuto-def*)
apply *auto*
done

lemma *SeqAuto-EmptySet-States* [simp]:

(*States* (*Abs-seqauto* ($\{\text{@}x. \text{True}\}$, ($\text{@}x. \text{True}$), $\{\}$, $\{\}$))) = $\{\text{@}x. \text{True}\}$
apply (*unfold States-def*)
apply (*simp*)
apply (*subst Abs-seqauto-inverse*)
apply (*unfold seqauto-def*)
apply (*rule SeqAuto-EmptySet*)
apply *auto*
done

lemma *SeqAuto-EmptySet-SAEvents* [simp]:

(*SAEvents* (*Abs-seqauto* ($\{\text{@}x. \text{True}\}$, ($\text{@}x. \text{True}$), $\{\}$, $\{\}$))) = $\{\}$
apply (*unfold SAEvents-def Delta-def*)
apply *simp*
apply (*subst Abs-seqauto-inverse*)
apply (*unfold seqauto-def*)
apply (*rule SeqAuto-EmptySet*)
apply *auto*
done

lemma *Label-Delta-subset* [simp]:

(*Label* (*Delta SA*)) $\subseteq$ *Labels SA*
apply (*unfold Label-def label-def*)
apply *auto*
apply (*cut-tac SA=SA in SeqAuto-select*)
apply (*unfold SeqAuto-def*)
apply *auto*
done

lemma *Target-SAs-Delta-States*:

Target ($\bigcup (\text{Delta ' (SAs HA)})$) $\subseteq$ $\bigcup (\text{States ' (SAs HA)})$
apply (*unfold image-def Target-def target-def*)
apply *auto*
apply (*rename-tac SA Source Trigger Guard Action Update Target*)
apply (*cut-tac SA=SA in SeqAuto-select*)
apply (*unfold SeqAuto-def*)
apply *auto*
done

lemma *States-Int-not-mem*:

$(\bigcup (\text{States ' F}) \text{ Int States SA}) = \{\} \implies \text{SA} \notin F$

```

apply (unfold Int-def)
apply auto
apply (subgoal-tac  $\exists S. S \in \text{States } SA$ )
prefer 2
apply (rule EX-State-SA)
apply (erule exE)
apply (rename-tac T)
apply (erule-tac  $x=T$  in allE)
apply auto
done

lemma Delta-target-States [simp]:
   $\llbracket T \in \text{Delta } A \rrbracket \implies \text{target } T \in \text{States } A$ 
apply (cut-tac  $SA=A$  in SeqAuto-select)
apply (unfold SeqAuto-def source-def target-def)
apply auto
done

lemma Delta-source-States [simp]:
   $\llbracket T \in \text{Delta } A \rrbracket \implies \text{source } T \in \text{States } A$ 
apply (cut-tac  $SA=A$  in SeqAuto-select)
apply (unfold SeqAuto-def source-def target-def)
apply auto
done

end

```

7 Syntax of Hierarchical Automata

```

theory HA
imports SA
begin

```

7.1 Definitions

```

definition
  RootEx :: [ $((s', e, d) \text{seqauto}) \text{ set},$ 
     $s' \rightarrow (s', e, d) \text{seqauto set}] \Rightarrow \text{bool}$  where
     $\text{RootEx } F \ G = (\exists! A. A \in F \wedge A \notin \bigcup (\text{ran } G))$ 

```

```

definition
  Root :: [ $((s', e, d) \text{seqauto}) \text{ set},$ 
     $s' \rightarrow (s', e, d) \text{seqauto set}]$ 
     $\Rightarrow (s', e, d) \text{seqauto}$  where
     $\text{Root } F \ G = (@ A. A \in F \wedge A \notin \bigcup (\text{ran } G))$ 

```

```

definition

```


$MutuallyDistinct :: ((s, e, d)seqauto) \text{ set} \Rightarrow \text{bool}$ **where**
 $MutuallyDistinct F =$
 $(\forall a \in F. \forall b \in F. a \neq b \longrightarrow (States a) \cap (States b) = \{\})$

definition

$OneAncestor :: ((s, e, d)seqauto) \text{ set},$
 $s \rightarrow (s, e, d) \text{ seqauto set}] \Rightarrow \text{bool}$ **where**
 $OneAncestor F G =$
 $(\forall A \in F - \{Root F G\} .$
 $\exists! s. s \in (\bigcup A' \in F - \{A\} . States A') \wedge$
 $A \in the (G s))$

definition

$NoCycles :: ((s, e, d)seqauto) \text{ set},$
 $s \rightarrow (s, e, d) \text{ seqauto set}] \Rightarrow \text{bool}$ **where**
 $NoCycles F G =$
 $(\forall S \in Pow (\bigcup A \in F. States A).$
 $S \neq \{\} \longrightarrow (\exists s \in S. S \cap (\bigcup A \in the (G s). States A) = \{\}))$

definition

$IsCompFun :: ((s, e, d)seqauto) \text{ set},$
 $s \rightarrow (s, e, d) \text{ seqauto set}] \Rightarrow \text{bool}$ **where**
 $IsCompFun F G = ((dom G = (\bigcup A \in F. States A)) \wedge$
 $(\bigcup (ran G) = (F - \{Root F G\})) \wedge$
 $(RootEx F G) \wedge$
 $(OneAncestor F G) \wedge$
 $(NoCycles F G))$

7.1.1 Well-formedness for the syntax of HA

definition

$HierAuto :: [d \text{ data},$
 $((s, e, d)seqauto) \text{ set},$
 $e \text{ set},$
 $s \rightarrow ((s, e, d)seqauto) \text{ set}]$
 $\Rightarrow \text{bool}$ **where**
 $HierAuto D F E G = ((\bigcup A \in F. SAEvents A) \subseteq E \wedge$
 $MutuallyDistinct F \wedge$
 $finite F \wedge$
 $IsCompFun F G)$

lemma *HierAuto-EmptySet:*

$((@x. True), \{Abs-seqauto (\{@x. True\}, (@x. True), \{\}, \{\}), \{\},$

```

    Map.empty ( @x. True  $\mapsto$  {}))  $\in$  {(D,F,E,G) | D F E G. HierAuto D F E G}
  apply (unfold HierAuto-def IsCompFun-def Root-def RootEx-def MutuallyDistinct-def
        OneAncestor-def NoCycles-def)
  apply auto
done

```

definition

```

  hierauto =
    {(D,F,E,G) |
      (D::'d data)
      (F::('s,'e,'d) seqauto) set)
      (E::('e set))
      (G::('s  $\rightarrow$  (('s,'e,'d) seqauto) set)).
      HierAuto D F E G}

```

typedef ('s,'e,'d) hierauto =

```

  hierauto :: ('d data * ('s,'e,'d) seqauto set * 'e set * ('s  $\rightarrow$  ('s,'e,'d) seqauto
set)) set
  unfolding hierauto-def
  apply (rule exI)
  apply (rule HierAuto-EmptySet)
done

```

definition

```

  SAs :: (('s,'e,'d) hierauto)  $\Rightarrow$  (('s,'e,'d) seqauto) set where
  SAs = fst o snd o Rep-hierauto

```

definition

```

  HAEvents :: (('s,'e,'d) hierauto)  $\Rightarrow$  ('e set) where
  HAEvents = fst o snd o snd o Rep-hierauto

```

definition

```

  CompFun :: (('s,'e,'d) hierauto)  $\Rightarrow$  ('s  $\rightarrow$  ('s,'e,'d) seqauto set) where
  CompFun = (snd o snd o snd o Rep-hierauto)

```

definition

```

  HAStates :: (('s,'e,'d) hierauto)  $\Rightarrow$  ('s set) where
  HAStates HA = ( $\bigcup$  A  $\in$  (SAs HA). States A)

```

definition

```

  HADelta :: (('s,'e,'d) hierauto)  $\Rightarrow$  (('s,'e,'d)trans)set where
  HADelta HA = ( $\bigcup$  F  $\in$  (SAs HA). Delta F)

```

definition

```

  HAINitValue :: (('s,'e,'d) hierauto)  $\Rightarrow$  'd data where
  HAINitValue == fst o Rep-hierauto

```

definition

```

  HAINitStates :: (('s,'e,'d) hierauto)  $\Rightarrow$  's set where

```

$H\text{InitStates } HA == \bigcup A \in (SAs \ HA). \{ \text{InitState } A \}$

definition

$HARoot :: ((s,e,d) \text{ hierauto}) \Rightarrow (s,e,d) \text{ seqauto}$ **where**
 $HARoot \ HA == \text{Root } (SAs \ HA) \ (\text{CompFun } HA)$

definition

$H\text{InitState} :: ((s,e,d) \text{ hierauto}) \Rightarrow s$ **where**
 $H\text{InitState } HA == \text{InitState } (HARoot \ HA)$

7.1.2 State successor function

definition

$Chi :: (s,e,d) \text{ hierauto} \Rightarrow s \Rightarrow s$ **set where**
 $Chi \ A == (\lambda \ S \in (H\text{States } A) .$
 $\{ S' . \exists \ SA \in (SAs \ A) . SA \in \text{the } ((\text{CompFun } A) \ S) \wedge S' \in \text{States } SA$
 $\})$

definition

$ChiRel :: (s,e,d) \text{ hierauto} \Rightarrow (s * s)$ **set where**
 $ChiRel \ A == \{ (S,S') . S \in H\text{States } A \wedge S' \in H\text{States } A \wedge S' \in (Chi \ A) \ S \}$

definition

$ChiPlus :: (s,e,d) \text{ hierauto} \Rightarrow (s * s)$ **set where**
 $ChiPlus \ A == (ChiRel \ A) \hat{+}$

definition

$ChiStar :: (s,e,d) \text{ hierauto} \Rightarrow (s * s)$ **set where**
 $ChiStar \ A == (ChiRel \ A) \hat{*}$

definition

$HigherPriority :: [(s,e,d) \text{ hierauto},$
 $(s,e,d) \text{ trans} * (s,e,d) \text{ trans}] \Rightarrow \text{bool}$ **where**
 $HigherPriority \ A ==$
 $\lambda (t,t') \in (H\Delta A) \times (H\Delta A).$
 $(\text{source } t', \text{source } t) \in ChiPlus \ A$

7.1.3 Configurations

definition

$\text{InitConf} :: (s,e,d) \text{ hierauto} \Rightarrow s$ **set where**
 $\text{InitConf } A == (((H\text{InitStates } A) \times (H\text{InitStates } A)) \cap (ChiRel \ A)) \hat{*}$
 $\text{“ } \{ H\text{InitState } A \}$

definition

StepConf :: [(*'s, 'e, 'd*)hierauto, *'s set*,
 (*'s, 'e, 'd*)trans set] => *'s set* **where**

StepConf *A C TS* ==
 (*C* - ((*ChiStar* *A*) “ (*Source TS*))) ∪
 (*Target TS*) ∪
 ((*ChiRel* *A*) “ (*Target TS*)) ∩ (*HInitStates* *A*) ∪
 ((((*ChiRel* *A*) ∩ ((*HInitStates* *A*) × (*HInitStates* *A*))))⁺)
 “ (((*ChiRel* *A*) “ (*Target TS*)) ∩ (*HInitStates* *A*)))

7.2 Lemmas**lemma** *Rep-hierauto-tuple*:

Rep-hierauto *HA* = (*HInitValue* *HA*, *SAs* *HA*, *HAEvents* *HA*, *CompFun* *HA*)
by (*unfold* *SAs-def* *HAEvents-def* *CompFun-def* *HInitValue-def*, *simp*)

lemma *Rep-hierauto-select*:

(*HInitValue* *HA*, *SAs* *HA*, *HAEvents* *HA*, *CompFun* *HA*): *hierauto*
by (*rule* *Rep-hierauto-tuple* [*THEN* *subst*], *rule* *Rep-hierauto*)

lemma *HierAuto-select* [*simp*]:

HierAuto (*HInitValue* *HA*) (*SAs* *HA*) (*HAEvents* *HA*) (*CompFun* *HA*)
by (*cut-tac* *Rep-hierauto-select*, *unfold* *hierauto-def*, *simp*)

7.2.1 HStates**lemma** *finite-HStates* [*simp*]:

finite (*HStates* *HA*)
apply (*cut-tac* *Rep-hierauto-select*)
apply (*unfold* *hierauto-def* *HierAuto-def*)
apply *auto*
apply (*simp* *add*: *HStates-def*)
apply (*rule* *finite-UN-I*)
apply *fast*

apply (*rule finite-States*)
done

lemma *HASates-SA-mem*:
 $\llbracket SA \in SAs\ A; S \in States\ SA \rrbracket \implies S \in HASates\ A$
by (*unfold HASates-def, auto*)

lemma *ChiRel-HASates* [*simp*]:
 $(a,b) \in ChiRel\ A \implies a \in HASates\ A$
apply (*unfold ChiRel-def*)
apply *auto*
done

lemma *ChiRel-HASates2* [*simp*]:
 $(a,b) \in ChiRel\ A \implies b \in HASates\ A$
apply (*unfold ChiRel-def*)
apply *auto*
done

7.2.2 *HAEvents*

lemma *HAEvents-SAEvents-SAs*:
 $\bigcup (SAEvents\ ' (SAs\ HA)) \subseteq HAEvents\ HA$
apply (*cut-tac Rep-hierauto-select*)
apply (*unfold hierauto-def HierAuto-def*)
apply *fast*
done

7.2.3 *NoCycles*

lemma *NoCycles-EmptySet* [*simp*]:
 $NoCycles\ \{\} S$
by (*unfold NoCycles-def, auto*)

lemma *NoCycles-HA* [*simp*]:
 $NoCycles\ (SAs\ HA)\ (CompFun\ HA)$
apply (*cut-tac Rep-hierauto-select*)
apply (*unfold hierauto-def HierAuto-def IsCompFun-def*)
apply *auto*
done

7.2.4 *OneAncestor*

lemma *OneAncestor-HA* [*simp*]:
 $OneAncestor\ (SAs\ HA)\ (CompFun\ HA)$
apply (*cut-tac Rep-hierauto-select*)
apply (*unfold hierauto-def HierAuto-def IsCompFun-def*)
apply *auto*
done

7.2.5 MutuallyDistinct

lemma *MutuallyDistinct-Single* [simp]:
 $\text{MutuallyDistinct } \{SA\}$
by (unfold *MutuallyDistinct-def*, auto)

lemma *MutuallyDistinct-EmptySet* [simp]:
 $\text{MutuallyDistinct } \{\}$
by (unfold *MutuallyDistinct-def*, auto)

lemma *MutuallyDistinct-Insert*:
 $\llbracket \text{MutuallyDistinct } S; (\text{States } A) \cap (\bigcup B \in S. \text{States } B) = \{\} \rrbracket$
 $\implies \text{MutuallyDistinct } (\text{insert } A \ S)$
by (unfold *MutuallyDistinct-def*, safe, fast+)

lemma *MutuallyDistinct-Union*:
 $\llbracket \text{MutuallyDistinct } A; \text{MutuallyDistinct } B;$
 $(\bigcup C \in A. \text{States } C) \cap (\bigcup C \in B. \text{States } C) = \{\} \rrbracket$
 $\implies \text{MutuallyDistinct } (A \cup B)$
by (unfold *MutuallyDistinct-def*, safe, blast+)

lemma *MutuallyDistinct-HA* [simp]:
 $\text{MutuallyDistinct } (SAs \ HA)$
apply (cut-tac *Rep-hierauto-select*)
apply (unfold *hierauto-def HierAuto-def IsCompFun-def*)
apply auto
done

7.2.6 RootEx

lemma *RootEx-Root* [simp]:
 $\text{RootEx } F \ G \implies \text{Root } F \ G \in F$
apply (unfold *RootEx-def Root-def*)
apply (erule *ex1E*)
apply (erule *conjE*)
apply (rule *someI2*)
apply blast+
done

lemma *RootEx-Root-ran* [simp]:
 $\text{RootEx } F \ G \implies \text{Root } F \ G \notin \bigcup (\text{ran } G)$
apply (unfold *RootEx-def Root-def*)
apply (erule *ex1E*)
apply (erule *conjE*)
apply (rule *someI2*)
apply blast+
done

lemma *RootEx-States-Subset* [simp]:
 $(\text{RootEx } F \ G) \implies \text{States } (\text{Root } F \ G) \subseteq (\bigcup x \in F. \text{States } x)$

```

apply (unfold RootEx-def Root-def)
apply (erule ex1E)
apply (erule conjE)
apply (rule someI2)
apply fast
apply (unfold UNION-eq)
apply (simp add: subset-eq)
apply auto
done

```

```

lemma RootEx-States-notdisjunct [simp]:
  RootEx F G  $\implies$  States (Root F G)  $\cap$  ( $\bigcup x \in F . \text{States } x$ )  $\neq \{\}$ 
apply (frule RootEx-States-Subset)
apply (case-tac States (Root F G)= $\{\}$ )
prefer 2
apply fast
apply simp
done

```

```

lemma Root-neq-SA [simp]:
   $\llbracket \text{RootEx } F \ G; (\bigcup x \in F . \text{States } x) \cap \text{States } SA = \{\} \rrbracket \implies \text{Root } F \ G \neq SA$ 
apply (rule SA-States-disjunct)
apply (frule RootEx-States-Subset)
apply fast
done

```

```

lemma RootEx-HA [simp]:
  RootEx (SAs HA) (CompFun HA)
apply (cut-tac Rep-hierauto-select)
apply (unfold hierauto-def HierAuto-def IsCompFun-def)
apply fast
done

```

7.2.7 HARoot

```

lemma HARoot-SAs [simp]:
  (HARoot HA)  $\in$  SAs HA
apply (unfold HARoot-def)
apply (cut-tac Rep-hierauto-select)
apply (unfold hierauto-def HierAuto-def)
apply auto
done

```

```

lemma States-HARoot-HAStates:
  States (HARoot HA)  $\subseteq$  HAStates HA
apply (unfold HAStates-def)
apply auto
apply (rule-tac x=HARoot HA in bexI)
apply auto

```

done

lemma *SAEvents-HARoot-HAEvents*:

$SAEvents (HARoot HA) \subseteq HAEvents HA$

apply (*cut-tac Rep-hierauto-select*)

apply (*unfold hierauto-def HierAuto-def*)

apply *auto*

apply (*rename-tac S*)

apply (*unfold UNION-eq*)

apply (*simp add: subset-eq*)

apply (*erule-tac x=S in allE*)

apply *auto*

done

lemma *HARoot-ran-CompFun*:

$HARoot HA \notin Union (ran (CompFun HA))$

apply (*unfold HARoot-def*)

apply (*cut-tac Rep-hierauto-select*)

apply (*unfold IsCompFun-def hierauto-def HierAuto-def*)

apply *fast*

done

lemma *HARoot-ran-CompFun2*:

$S \in ran (CompFun HA) \longrightarrow HARoot HA \notin S$

apply (*unfold HARoot-def*)

apply (*cut-tac Rep-hierauto-select*)

apply (*unfold IsCompFun-def hierauto-def HierAuto-def*)

apply *fast*

done

7.2.8 *CompFun*

lemma *IsCompFun-HA [simp]*:

$IsCompFun (SAs HA) (CompFun HA)$

apply (*cut-tac Rep-hierauto-select*)

apply (*unfold hierauto-def HierAuto-def*)

apply *auto*

done

lemma *dom-CompFun [simp]*:

$dom (CompFun HA) = HAsStates HA$

apply (*cut-tac HA=HA in IsCompFun-HA*)

apply (*unfold IsCompFun-def HAsStates-def*)

apply *auto*

done

lemma *ran-CompFun [simp]*:

$Union (ran (CompFun HA)) = ((SAs HA) - \{Root (SAs HA)(CompFun HA)\})$

apply (*cut-tac HA=HA in IsCompFun-HA*)

apply (*unfold IsCompFun-def*)
apply *fast*
done

lemma *ran-CompFun-subseteq*:
 $Union\ (ran\ (CompFun\ HA)) \subseteq (SAs\ HA)$
apply (*cut-tac HA=HA in IsCompFun-HA*)
apply (*unfold IsCompFun-def*)
apply *fast*
done

lemma *ran-CompFun-is-not-SA*:
 $\neg Sas \subseteq (SAs\ HA) \implies Sas \notin (ran\ (CompFun\ HA))$
apply (*cut-tac HA=HA in IsCompFun-HA*)
apply (*unfold IsCompFun-def*)
apply *fast*
done

lemma *HASates-HARoot-CompFun [simp]*:
 $S \in HASates\ HA \implies HARoot\ HA \notin the\ (CompFun\ HA\ S)$
apply (*rule ran-dom-the*)
back
apply (*simp add: HARoot-ran-CompFun2 HARoot-def HASates-def*) +
done

lemma *HASates-CompFun-SAs*:
 $S \in HASates\ A \implies the\ (CompFun\ A\ S) \subseteq SAs\ A$
apply *auto*
apply (*rename-tac T*)
apply (*cut-tac HA=A in ran-CompFun*)
apply (*erule equalityE*)
apply (*erule-tac c=T in subsetCE*)
apply (*erule ran-dom-the*)
apply *auto*
done

lemma *HASates-CompFun-notmem [simp]*:
 $\llbracket S \in HASates\ A; SA \in the\ (CompFun\ A\ S) \rrbracket \implies S \notin States\ SA$
apply (*unfold HASates-def*)
apply *auto*
apply (*rename-tac T*)
apply (*cut-tac HA=A in MutuallyDistinct-HA*)
apply (*unfold MutuallyDistinct-def*)
apply (*erule-tac x=SA in ballE*)
apply (*erule-tac x=T in ballE*)
apply *auto*
prefer 2
apply (*cut-tac A=A and S=S in HASates-CompFun-SAs*)
apply (*unfold HASates-def*)

```

apply simp
apply fast
apply fast
apply (cut-tac  $HA=A$  in NoCycles-HA)
apply (unfold NoCycles-def)
apply (erule-tac  $x=\{S\}$  in ballE)
apply auto
done

```

lemma *CompFun-Int-disjoint*:

$\llbracket S \neq T; S \in \text{HASStates } A; T \in \text{HASStates } A \rrbracket \implies \text{the } (\text{CompFun } A \ T) \cap \text{the } (\text{CompFun } A \ S) = \{\}$

```

apply auto
apply (rename-tac U)
apply (cut-tac  $HA=A$  in OneAncestor-HA)
apply (unfold OneAncestor-def)
apply (erule-tac  $x=U$  in ballE)
prefer 2
apply simp
apply (fold HARoot-def)
apply (frule HASStates-HARoot-CompFun)
apply simp
apply (frule HASStates-CompFun-SAs)
apply auto
apply (erule-tac  $x=S$  in allE)
apply (erule-tac  $x=T$  in allE)
apply auto
apply (cut-tac  $HA=A$  in NoCycles-HA)
apply (unfold NoCycles-def)
apply (simp only: HASStates-def)
apply safe
apply (erule-tac  $x=\{S\}$  in ballE)
apply simp
apply fast
apply simp
apply (cut-tac  $HA=A$  in NoCycles-HA)
apply (unfold NoCycles-def)
apply (simp only: HASStates-def)
apply safe
apply (erule-tac  $x=\{T\}$  in ballE)
apply simp
apply fast
apply simp
done

```

7.2.9 SAs

lemma *finite-SAs* [*simp*]:

finite (*SAs* *HA*)

```

apply (cut-tac Rep-hierauto-select)
apply (unfold hierauto-def HierAuto-def)
apply fast
done

```

```

lemma HASates-SAs-disjunct:
   $HASates\ HA1 \cap HASates\ HA2 = \{\} \implies SAs\ HA1 \cap SAs\ HA2 = \{\}$ 
apply (unfold UNION-eq HASates-def Int-def)
apply auto
apply (rename-tac SA)
apply (cut-tac SA=SA in EX-State-SA)
apply (erule exE)
apply auto
done

```

```

lemma HASates-CompFun-SAs-mem [simp]:
   $\llbracket S \in HASates\ A; T \in the\ (CompFun\ A\ S) \rrbracket \implies T \in SAs\ A$ 
apply (cut-tac A=A and S=S in HASates-CompFun-SAs)
apply auto
done

```

```

lemma SAs-States-HASates:
   $SA \in SAs\ A \implies States\ SA \subseteq HASates\ A$ 
by (unfold HASates-def, auto)

```

7.2.10 *HASatesInitState*

```

lemma HASatesInitState-HARoot [simp]:
   $HASatesInitState\ A \in States\ (HARoot\ A)$ 
by (unfold HASatesInitState-def, auto)

```

```

lemma HASatesInitState-HARoot2 [simp]:
   $HASatesInitState\ A \in States\ (Root\ (SAs\ A)\ (CompFun\ A))$ 
by (fold HARoot-def, simp)

```

```

lemma HASatesInitStates-HASates [simp]:
   $HASatesInitStates\ A \subseteq HASates\ A$ 
apply (unfold HASatesInitStates-def HASates-def)
apply auto
done

```

```

lemma HASatesInitStates-HASates2 [simp]:
   $S \in HASatesInitStates\ A \implies S \in HASates\ A$ 
apply (cut-tac A=A in HASatesInitStates-HASates)
apply fast
done

```

```

lemma HASatesInitState-HASates [simp]:
   $HASatesInitState\ A \in HASates\ A$ 

```

```

apply (unfold HASStates-def)
apply auto
apply (rule-tac  $x=HARoot\ A$  in bexI)
apply auto
done

```

```

lemma HASInitState-HASInitStates [simp]:
  HASInitState  $A \in HASInitStates  $A$ 
by (unfold HASInitStates-def HASInitState-def, auto)$ 
```

```

lemma CompFun-HASInitStates-HASStates [simp]:
   $\llbracket S \in HAS$ States  $A; SA \in the\ (CompFun\ A\ S) \rrbracket \implies (InitState\ SA) \in HASInitStates
   $A$ 
apply (unfold HASInitStates-def)
apply auto
done$ 
```

```

lemma CompFun-HASInitState-HASInitStates [simp]:
   $\llbracket SA \in the\ (CompFun\ A\ (HASInitState  $A)) \rrbracket \implies (InitState\ SA) \in HASInitStates
   $A$ 
apply (unfold HASInitStates-def)
apply auto
apply (rule-tac  $x=SA$  in bexI)
apply auto
apply (cut-tac  $A=A$  and  $S=HASInitState  $A$  in HASStates-CompFun-SAs)
apply auto
done$$$ 
```

```

lemma HASInitState-notmem-States [simp]:
   $\llbracket S \in HAS$ States  $A; SA \in the\ (CompFun\ A\ S) \rrbracket \implies HASInitState  $A \notin States\ SA$ 
apply (cut-tac  $HA=A$  in MutuallyDistinct-HA)
apply (unfold MutuallyDistinct-def)
apply (erule-tac  $x=SA$  in ballE)
apply (erule-tac  $x=HARoot\ A$  in ballE)
apply auto
done$ 
```

```

lemma InitState-notmem-States [simp]:
   $\llbracket S \in HAS$ States  $A; SA \in the\ (CompFun\ A\ S);$ 
     $T \in HASInitStates  $A; T \neq InitState\ SA \rrbracket$ 
     $\implies T \notin States\ SA$ 
apply (unfold HASInitStates-def)
apply auto
apply (rename-tac SAA)
apply (cut-tac  $HA=A$  in MutuallyDistinct-HA)
apply (unfold MutuallyDistinct-def)
apply (erule-tac  $x=SA$  in ballE)
apply (erule-tac  $x=SAA$  in ballE)$ 
```

apply *auto*
done

lemma *InitState-States-notmem* [*simp*]:
 $\llbracket B \in \text{SAs } A; C \in \text{SAs } A; B \neq C \rrbracket \implies \text{InitState } B \notin \text{States } C$
apply *auto*
apply (*cut-tac* $HA=A$ **in** *MutuallyDistinct-HA*)
apply (*unfold* *MutuallyDistinct-def*)
apply *force*
done

lemma *OneHAINitState-SASates*:
 $\llbracket S \in \text{HAINitStates } A; T \in \text{HAINitStates } A; \\ S \in \text{States } SA; T \in \text{States } SA; SA \in \text{SAs } A \rrbracket \implies \\ S = T$
apply (*unfold* *HAINitStates-def*)
apply *auto*
apply (*rename-tac* AA AAA)
apply (*case-tac* $AA = SA$)
apply *auto*
apply (*case-tac* $AAA = SA$)
apply *auto*
done

7.2.11 *Chi*

lemma *HARootStates-notmem-Chi* [*simp*]:
 $\llbracket S \in \text{HASates } A; T \in \text{States } (\text{HARoot } A) \rrbracket \implies T \notin \text{Chi } A$
apply (*unfold* *Chi-def* *restrict-def*, *auto*)
apply (*rename-tac* SA)
apply (*cut-tac* $HA=A$ **in** *MutuallyDistinct-HA*)
apply (*unfold* *MutuallyDistinct-def*)
apply (*erule-tac* $x=\text{HARoot } A$ **in** *ballE*)
apply (*erule-tac* $x=SA$ **in** *ballE*)
apply *auto*
done

lemma *SASates-notmem-Chi* [*simp*]:
 $\llbracket S \in \text{States } SA; T \in \text{States } SA; \\ SA \in \text{SAs } A \rrbracket \implies T \notin \text{Chi } A$
apply (*unfold* *Chi-def* *restrict-def*, *auto*)
apply (*rename-tac* SAA)
apply (*cut-tac* $HA=A$ **in** *MutuallyDistinct-HA*)
apply (*unfold* *MutuallyDistinct-def*)
apply (*erule-tac* $x=SAA$ **in** *ballE*)
apply (*erule-tac* $x=SA$ **in** *ballE*)
apply *auto*
apply (*unfold* *HASates-def*)
apply *auto*

done

lemma *HASInitState-notmem-Chi* [simp]:
 $S \in HASStates\ A \implies HASInitState\ A \notin Chi\ A\ S$
by (unfold *Chi-def restrict-def*, auto)

lemma *Chi-HASStates* [simp]:
 $T \in HASStates\ A \implies (Chi\ A\ T) \subseteq HASStates\ A$
apply (unfold *Chi-def restrict-def*)
apply (auto)
apply (cut-tac $A=A$ and $S=T$ in *HASStates-CompFun-SAs*)
apply (unfold *HASStates-def*)
apply auto
done

lemma *Chi-HASStates-Self* [simp]:
 $s \in HASStates\ a \implies s \notin (Chi\ a\ s)$
by (unfold *Chi-def restrict-def*, auto)

lemma *ChiRel-HASStates-Self* [simp]:
 $(s,s) \notin (ChiRel\ a)$
by (unfold *ChiRel-def*, auto)

lemma *HASStates-Chi-NoCycles*:
 $\llbracket s \in HASStates\ a; t \in HASStates\ a; s \in Chi\ a\ t \rrbracket \implies t \notin Chi\ a\ s$
apply (unfold *Chi-def restrict-def*)
apply auto
apply (cut-tac $HA=a$ in *NoCycles-HA*)
apply (unfold *NoCycles-def*)
apply (erule-tac $x=\{s,t\}$ in *ballE*)
apply auto
done

lemma *HASStates-Chi-NoCycles-trans*:
 $\llbracket s \in HASStates\ a; t \in HASStates\ a; u \in HASStates\ a;$
 $t \in Chi\ a\ s; u \in Chi\ a\ t \rrbracket \implies s \notin Chi\ a\ u$
apply (unfold *Chi-def restrict-def*)
apply auto
apply (cut-tac $HA=a$ in *NoCycles-HA*)
apply (unfold *NoCycles-def*)
apply (erule-tac $x=\{s,t,u\}$ in *ballE*)
prefer 2
apply simp
apply (unfold *HASStates-def*)
apply auto
done

lemma *Chi-range-disjoint*:
 $\llbracket S \neq T; T \in HASStates\ A; S \in HASStates\ A; U \in Chi\ A\ S \rrbracket \implies U \notin Chi\ A\ T$

```

apply (frule CompFun-Int-disjoint)
apply auto
apply (unfold Chi-def restrict-def)
apply auto
apply (rename-tac SA SAA)
apply (cut-tac HA=A in MutuallyDistinct-HA)
apply (unfold MutuallyDistinct-def)
apply (erule-tac x=SA in ballE)
apply (erule-tac x=SAA in ballE)
apply auto
done

lemma SASates-Chi-trans [rule-format]:
  
$$\llbracket U \in \text{Chi } A \ T; S \in \text{Chi } A \ U; T \in \text{States } SA; \\ SA \in \text{SAs } A; U \in \text{HASates } A \rrbracket \implies S \notin \text{States } SA$$

apply (frule HASates-SA-mem)
apply auto
apply (unfold Chi-def restrict-def)
apply auto
apply (rename-tac SAA SAAA)
apply (cut-tac HA=A in NoCycles-HA)
apply (unfold NoCycles-def)
apply (erule-tac x={U,T} in ballE)
prefer 2
apply (simp only: HASates-def)
apply auto
apply (cut-tac HA=A in MutuallyDistinct-HA)
apply (unfold MutuallyDistinct-def)
apply (rotate-tac -1)
apply (erule-tac x=SA in ballE)
apply (rotate-tac -1)
apply (erule-tac x=SAAA in ballE)
apply auto
done

```

7.2.12 *ChiRel*

```

lemma finite-ChiRel [simp]:
  finite (ChiRel A)
apply (rule-tac B=HASates A  $\times$  HASates A in finite-subset)
apply auto
done

```

```

lemma ChiRel-HASates-subseteq [simp]:
   $(\text{ChiRel } A) \subseteq (\text{HASates } A \times \text{HASates } A)$ 
apply (unfold ChiRel-def Chi-def restrict-def)
apply auto
done

```

lemma *ChiRel-CompFun*:
 $s \in \text{HStates } a \implies \text{ChiRel } a \text{ “ } \{s\} = (\bigcup x \in \text{the } (\text{CompFun } a \ s)). \text{ States } x)$
apply (*unfold ChiRel-def Chi-def restrict-def Image-def*)
apply *simp*
apply *auto*
apply (*frule HStates-CompFun-SAs-mem*)
apply *fast*
apply (*unfold HStates-def*)
apply *fast*
done

lemma *ChiRel-HARoot*:
 $\llbracket (x,y) \in \text{ChiRel } A \rrbracket \implies y \notin \text{States } (\text{HARoot } A)$
apply (*unfold ChiRel-def Chi-def*)
apply *auto*
apply (*rename-tac SA*)
apply (*frule HStates-HARoot-CompFun*)
apply (*cut-tac HA=A in MutuallyDistinct-HA*)
apply (*unfold MutuallyDistinct-def*)
apply *auto*
apply (*erule-tac x=SA in ballE*)
apply (*erule-tac x=HARoot A in ballE*)
apply *auto*
done

lemma *HStates-CompFun-States-ChiRel*:
 $S \in \text{HStates } A \implies \bigcup (\text{States ‘ the } (\text{CompFun } A \ S)) = \text{ChiRel } A \text{ “ } \{S\}$
apply (*unfold ChiRel-def Chi-def restrict-def*)
apply *auto*
apply (*drule HStates-CompFun-SAs*)
apply (*subst HStates-def*)
apply *fast*
done

lemma *HInitState-notmem-Range-ChiRel [simp]*:
 $\text{HInitState } A \notin \text{Range } (\text{ChiRel } A)$
by (*unfold ChiRel-def, auto*)

lemma *HInitState-notmem-Range-ChiRel2 [simp]*:
 $(S, \text{HInitState } A) \notin (\text{ChiRel } A)$
by (*unfold ChiRel-def, auto*)

lemma *ChiRel-OneAncestor-notmem*:
 $\llbracket S \neq T; (S,U) \in \text{ChiRel } A \rrbracket \implies (T,U) \notin \text{ChiRel } A$
apply (*unfold ChiRel-def*)
apply *auto*
apply (*simp only: Chi-range-disjoint*)
done

lemma *ChiRel-OneAncestor*:
 $\llbracket (S1, U) \in \text{ChiRel } A; (S2, U) \in \text{ChiRel } A \rrbracket \implies S1 = S2$
apply (*rule notnotD*, *rule notI*)
apply (*simp add: ChiRel-OneAncestor-notmem*)
done

lemma *CompFun-ChiRel*:
 $\llbracket S1 \in \text{HASstates } A; SA \in \text{the } (\text{CompFun } A \ S1);$
 $S2 \in \text{States } SA \rrbracket \implies (S1, S2) \in \text{ChiRel } A$
apply (*unfold ChiRel-def Chi-def restrict-def*)
apply *auto*
apply (*cut-tac A=A and S=S1 in HASstates-CompFun-SAs*)
apply (*unfold HASstates-def*)
apply *auto*
done

lemma *CompFun-ChiRel2*:
 $\llbracket (S, T) \in \text{ChiRel } A; T \in \text{States } SA; SA \in \text{SAs } A \rrbracket \implies SA \in \text{the } (\text{CompFun } A \ S)$
apply (*unfold ChiRel-def Chi-def restrict-def*)
apply *auto*
apply (*rename-tac SAA*)
apply (*cut-tac HA=A in MutuallyDistinct-HA*)
apply (*unfold MutuallyDistinct-def*)
apply (*erule-tac x=SA in ballE*)
apply (*rotate-tac -1*)
apply (*erule-tac x=SAA in ballE*)
apply *auto*
done

lemma *ChiRel-HASstates-NoCycles*:
 $(s, t) \in (\text{ChiRel } a) \implies (t, s) \notin (\text{ChiRel } a)$
apply (*unfold ChiRel-def*)
apply *auto*
apply (*frule HASstates-Chi-NoCycles*)
apply *auto*
done

lemma *HASstates-ChiRel-NoCycles-trans*:
 $\llbracket (s, t) \in (\text{ChiRel } a); (t, u) \in (\text{ChiRel } a) \rrbracket \implies (u, s) \notin (\text{ChiRel } a)$
apply (*unfold ChiRel-def*)
apply *auto*
apply (*frule HASstates-Chi-NoCycles-trans*)
apply *fast*
back
back
prefer 3
apply *fast*
apply *auto*

done

lemma *SASates-ChiRel*:

[[$S \in \text{States } SA; T \in \text{States } SA;$
 $SA \in \text{SAs } A$]] $\implies (S, T) \notin (\text{ChiRel } A)$
by (*unfold ChiRel-def*, *auto*)

lemma *ChiRel-SA-OneAncestor*:

[[$(S, T) \in \text{ChiRel } A; T \in \text{States } SA;$
 $U \in \text{States } SA; SA \in \text{SAs } A$]] $\implies$
 $(S, U) \in \text{ChiRel } A$
apply (*frule CompFun-ChiRel2*)
apply *auto*
apply (*rule CompFun-ChiRel*)
apply *auto*
done

lemma *ChiRel-OneAncestor2*:

[[$S \in \text{HASates } A; S \notin \text{States } (\text{HARoot } A)$]] $\implies$
 $\exists! T. (T, S) \in \text{ChiRel } A$
apply (*unfold ChiRel-def*)
apply *auto*
prefer 2
apply (*rename-tac T U*)
prefer 2
apply (*unfold Chi-def restrict-def*)
apply *auto*
prefer 2
apply (*rename-tac SA SAA*)
prefer 2
apply (*cut-tac HA=A in OneAncestor-HA*)
apply (*unfold OneAncestor-def*)
apply (*fold HARoot-def*)
apply *auto*
apply (*simp cong: rev-conj-cong*)
apply (*unfold HASates-def*)
apply *auto*
apply (*rename-tac SA*)
apply (*erule-tac x=SA in ballE*)
apply *auto*
apply (*case-tac T = U*)
apply *auto*
apply (*frule CompFun-Int-disjoint*)
apply (*unfold HASates-def*)
apply *auto*
apply (*case-tac SA=SAA*)
apply *auto*
apply (*cut-tac HA=A in MutuallyDistinct-HA*)
apply (*unfold MutuallyDistinct-def*)

```

apply (erule-tac  $x=SAA$  in ballE)
apply (erule-tac  $x=SA$  in ballE)
apply auto
apply (cut-tac  $S=T$  and  $A=A$  in HASates-CompFun-SAs)
apply (unfold HASates-def)
apply fast
apply fast
apply (cut-tac  $S=U$  and  $A=A$  in HASates-CompFun-SAs)
apply (unfold HASates-def)
apply fast
apply fast
done

```

```

lemma HARootStates-notmem-Range-ChiRel [simp]:
   $S \in \text{States } (HARoot\ A) \implies S \notin \text{Range } (ChiRel\ A)$ 
by (unfold ChiRel-def, auto)

```

```

lemma ChiRel-int-disjoint:
   $S \neq T \implies (ChiRel\ A\ \text{“}\ \{S\}) \cap (ChiRel\ A\ \text{“}\ \{T\}) = \{\}$ 
apply (unfold ChiRel-def)
apply auto
apply (simp only: Chi-range-disjoint)
done

```

```

lemma SASates-ChiRel-trans [rule-format]:
   $\llbracket (S,U) \in (ChiRel\ A); (U,T) \in ChiRel\ A; S \in \text{States } SA; SA \in SAs\ A \rrbracket \implies T \notin \text{States } SA$ 
apply auto
apply (unfold ChiRel-def)
apply auto
apply (frule SASates-Chi-trans)
back
apply fast+
done

```

```

lemma HASates-InitState-trancl:
   $\llbracket S \in HASates\ (HA\ ST); A \in the\ (CompFun\ (HA\ ST)\ S) \rrbracket \implies$ 
   $(S, InitState\ A) \in (ChiRel\ (HA\ ST) \cap HASates\ (HA\ ST) \times HASates\ (HA\ ST))^+$ 
apply (case-tac  $S \in HASates\ (HA\ ST)$ )
apply (frule CompFun-ChiRel)
apply fast+
apply (rule InitState-States)
apply auto
apply (rule r-into-trancl')
apply auto
apply (rule CompFun-HASates-HASates)
apply auto
done

```

lemma *HASInitStates-InitState-trancl2*:
 $\llbracket S \in HASStates (HA ST); A \in the (CompFun (HA ST) S);$
 $(x, S) \in (ChiRel (HA ST) \cap HASInitStates (HA ST) \times HASInitStates (HA ST))^+$
 $\rrbracket$
 $\implies (x, InitState A) \in (ChiRel (HA ST) \cap HASInitStates (HA ST) \times HASInitStates (HA ST))^+$
apply (*rule-tac a=x and b=S and r=ChiRel (HA ST) $\cap$ HASInitStates (HA ST)*
 $\times HASInitStates (HA ST)$ **in** *converse-trancl-induct*)
apply *auto*
prefer 2
apply (*rename-tac T U*)
prefer 2
apply (*case-tac S $\in$ HASStates (HA ST)*)
apply (*frule CompFun-ChiRel*)
apply *fast*
apply (*rule InitState-States*)
apply *simp*
apply (*rule trancl-trans [of - S]*)
apply (*rule r-into-trancl'*)
apply *auto*
apply (*rule r-into-trancl'*)
apply *auto*
apply (*rule CompFun-HASInitStates-HASStates*)
prefer 2
apply *fast*
apply (*cut-tac A=HA ST in HASInitStates-HASStates, fast*)
apply (*rule-tac y = U in trancl-trans*)
apply (*rule r-into-trancl'*)
apply *auto*
done

7.2.13 ChiPlus

lemma *ChiPlus-ChiRel [simp]*:
 $(S, T) \in ChiRel A \implies (S, T) \in ChiPlus A$
apply (*unfold ChiPlus-def*)
apply (*frule r-into-trancl*)
apply *auto*
done

lemma *ChiPlus-HASStates [simp]*:
 $(ChiPlus A) \subseteq (HASStates A \times HASStates A)$
apply (*unfold ChiPlus-def*)
apply (*rule trancl-subset-Sigma*)
apply *auto*
done

lemma *ChiPlus-subset-States*:

```

  ChiPlus a “ {t}  $\subseteq$   $\bigcup$  (States ‘ (SAs a))
apply (cut-tac A=a in ChiPlus-HAStates)
apply (unfold HAStates-def)
apply auto
done

```

```

lemma finite-ChiPlus [simp]:
  finite (ChiPlus A)
apply (rule-tac B=HAStates A  $\times$  HAStates A in finite-subset)
apply auto
done

```

```

lemma ChiPlus-OneAncestor:
   $\llbracket S \in \text{HAStates } A; S \notin \text{States } (\text{HARoot } A) \rrbracket \implies$ 
   $\exists T. (T, S) \in \text{ChiPlus } A$ 
apply (unfold ChiPlus-def)
apply (frule ChiRel-OneAncestor2)
apply auto
done

```

```

lemma ChiPlus-HAStates-Left:
   $(S, T) \in \text{ChiPlus } A \implies S \in \text{HAStates } A$ 
apply (cut-tac A=A in ChiPlus-HAStates)
apply (unfold HAStates-def)
apply auto
done

```

```

lemma ChiPlus-HAStates-Right:
   $(S, T) \in \text{ChiPlus } A \implies T \in \text{HAStates } A$ 
apply (cut-tac A=A in ChiPlus-HAStates)
apply (unfold HAStates-def)
apply auto
done

```

```

lemma ChiPlus-ChiRel-int [rule-format]:
   $\llbracket (T, S) \in (\text{ChiPlus } A) \rrbracket \implies (\text{ChiPlus } A \text{ “ } \{T\}) \cap (\text{ChiRel } A \text{ “ } \{S\}) = (\text{ChiRel } A \text{ “ } \{S\})$ 
apply (unfold ChiPlus-def)
apply (rule-tac a=T and b=S and r=(ChiRel A) in converse-trancl-induct)
apply auto
done

```

```

lemma ChiPlus-ChiPlus-int [rule-format]:
   $\llbracket (T, S) \in (\text{ChiPlus } A) \rrbracket \implies (\text{ChiPlus } A \text{ “ } \{T\}) \cap (\text{ChiPlus } A \text{ “ } \{S\}) = (\text{ChiPlus } A \text{ “ } \{S\})$ 
apply (unfold ChiPlus-def)
apply (rule-tac a=T and b=S and r=(ChiRel A) in converse-trancl-induct)
apply auto
done

```

lemma *ChiPlus-ChiRel-NoCycle-1* [rule-format]:

$\llbracket (T, S) \in \text{ChiPlus } A \rrbracket \implies$
 $(\text{insert } S (\text{insert } T (\{U. (T, U) \in \text{ChiPlus } A \wedge (U, S) \in \text{ChiPlus } A\}))) \cap (\text{ChiRel } A \text{ “ } \{T\}) \neq \{\}$
apply (unfold *ChiPlus-def*)
apply (rule-tac *a=T and b=S and r=(ChiRel A) in converse-trancl-induct*)
apply (unfold *Image-def Int-def*)
apply *auto*
done

lemma *ChiPlus-ChiRel-NoCycle-2* [rule-format]:

$\llbracket (T, S) \in \text{ChiPlus } A \rrbracket \implies (S, T) \in (\text{ChiRel } A) \longrightarrow$
 $(\text{insert } S (\text{insert } T (\{U. (T, U) \in \text{ChiPlus } A \wedge (U, S) \in \text{ChiPlus } A\}))) \cap (\text{ChiRel } A \text{ “ } \{S\}) \neq \{\}$
apply (unfold *ChiPlus-def*)
apply (rule-tac *a=T and b=S and r=(ChiRel A) in converse-trancl-induct*)
apply (unfold *Image-def Int-def*)
apply *auto*
done

lemma *ChiPlus-ChiRel-NoCycle-3* [rule-format]:

$\llbracket (T, S) \in \text{ChiPlus } A \rrbracket \implies (S, T) \in (\text{ChiRel } A) \longrightarrow (T, U) \in \text{ChiPlus } A \longrightarrow (U, S) \in \text{ChiPlus } A \longrightarrow$
 $(\text{insert } S (\text{insert } T (\{U. (T, U) \in \text{ChiPlus } A \wedge (U, S) \in \text{ChiPlus } A\}))) \cap (\text{ChiRel } A \text{ “ } \{U\}) \neq \{\}$
apply (unfold *ChiPlus-def*)
apply (rule-tac *a=T and b=S and r=(ChiRel A) in trancl-induct*)
apply (unfold *Image-def Int-def, simp*)
apply (rename-tac *V*)
prefer 2
apply (rename-tac *V W*)
prefer 2
apply (*simp, safe*)
apply (*simp only: ChiRel-HAStates-NoCycles*)
apply *simp*
apply (*case-tac (U, W) ∈ (ChiRel A), fast, rotate-tac 5, frule tranclD3, fast, blast intro: trancl-into-trancl*)
done

lemma *ChiPlus-ChiRel-NoCycle-4* [rule-format]:

$\llbracket (T, S) \in \text{ChiPlus } A \rrbracket \implies (S, T) \in (\text{ChiRel } A) \longrightarrow ((\text{ChiPlus } A \text{ “ } \{T\}) \cap (\text{ChiRel } A \text{ “ } \{S\})) \neq \{\}$
apply (unfold *ChiPlus-def*)
apply (rule-tac *a=T and b=S and r=(ChiRel A) in trancl-induct*)
apply (unfold *Image-def Int-def*)
apply *auto*
apply (*simp only: ChiRel-HAStates-NoCycles*)
apply (rule-tac *x=T in exI*)

```

apply simp
apply (rule-tac  $x=T$  in exI)
apply simp
done

```

```

lemma ChiRel-ChiPlus-NoCycles:
   $(S,T) \in (ChiRel\ A) \implies (T,S) \notin (ChiPlus\ A)$ 
apply (cut-tac  $HA=A$  in NoCycles-HA)
apply (unfold NoCycles-def)
apply (erule-tac  $x=insert\ S\ (insert\ T\ (\{U. (T,U) \in ChiPlus\ A \wedge (U,S) \in ChiPlus\ A\}))$  in ballE)
prefer 2
apply (simp add: ChiPlus-subset-States)
apply (cut-tac  $A=A$  in ChiPlus-HAStates)
apply (unfold HAStates-def)
apply auto
apply (frule ChiPlus-ChiRel-NoCycle-2)
apply fast
apply (simp add: ChiRel-CompFun)
apply (frule ChiPlus-ChiRel-NoCycle-1)
apply (simp add: ChiRel-CompFun)
apply (frule ChiPlus-ChiRel-NoCycle-3)
apply fast
apply fast
back
apply fast
apply (rename-tac V)
apply (case-tac  $V \in HAStates\ A$ )
apply (simp add: ChiRel-CompFun)
apply (simp only: ChiPlus-HAStates-Right)
apply fast
done

```

```

lemma ChiPlus-ChiPlus-NoCycles:
   $(S,T) \in (ChiPlus\ A) \implies (T,S) \notin (ChiPlus\ A)$ 
apply (unfold ChiPlus-def)
apply (rule-tac  $a=S$  and  $b=T$  and  $r=(ChiRel\ A)$  in trancl-induct)
apply fast
apply (frule ChiRel-ChiPlus-NoCycles)
apply (auto intro: trancl-into-trancl2 simp add: ChiPlus-def)
done

```

```

lemma ChiPlus-NoCycles [rule-format]:
   $(S,T) \in (ChiPlus\ A) \implies S \neq T$ 
apply (frule ChiPlus-ChiPlus-NoCycles)
apply auto
done

```

```

lemma ChiPlus-NoCycles-2 [simp]:

```

$(S, S) \notin (ChiPlus\ A)$
apply (*rule notI*)
apply (*frule ChiPlus-NoCycles*)
apply *fast*
done

lemma *ChiPlus-ChiPlus-NoCycles-2*:
 $\llbracket (S, U) \in ChiPlus\ A; (U, T) \in ChiPlus\ A \rrbracket \implies (T, S) \notin ChiPlus\ A$
apply (*rule ChiPlus-ChiPlus-NoCycles*)
apply (*auto intro: trancl-trans simp add: ChiPlus-def*)
done

lemma *ChiRel-ChiPlus-trans*:
 $\llbracket (U, S) \in ChiPlus\ A; (S, T) \in ChiRel\ A \rrbracket \implies (U, T) \in ChiPlus\ A$
apply (*unfold ChiPlus-def*)
apply *auto*
done

lemma *ChiRel-ChiPlus-trans2*:
 $\llbracket (U, S) \in ChiRel\ A; (S, T) \in ChiPlus\ A \rrbracket \implies (U, T) \in ChiPlus\ A$
apply (*unfold ChiPlus-def*)
apply *auto*
done

lemma *ChiPlus-ChiRel-Ex* [*rule-format*]:
 $\llbracket (S, T) \in ChiPlus\ A \rrbracket \implies (S, T) \notin ChiRel\ A \longrightarrow$
 $(\exists U. (S, U) \in ChiPlus\ A \wedge (U, T) \in ChiRel\ A)$
apply (*unfold ChiPlus-def*)
apply (*rule-tac a=S and b=T and r=(ChiRel A) in converse-trancl-induct*)
apply *auto*
apply (*rename-tac U*)
apply (*rule-tac x=U in exI*)
apply *auto*
done

lemma *ChiPlus-ChiRel-Ex2* [*rule-format*]:
 $\llbracket (S, T) \in ChiPlus\ A \rrbracket \implies (S, T) \notin ChiRel\ A \longrightarrow$
 $(\exists U. (S, U) \in ChiRel\ A \wedge (U, T) \in ChiPlus\ A)$
apply (*unfold ChiPlus-def*)
apply (*rule-tac a=S and b=T and r=(ChiRel A) in converse-trancl-induct*)
apply *auto*
done

lemma *HARootStates-Range-ChiPlus* [*simp*]:
 $\llbracket S \in States\ (HARoot\ A) \rrbracket \implies S \notin Range\ (ChiPlus\ A)$
by (*unfold ChiPlus-def, auto*)

lemma *HARootStates-Range-ChiPlus2* [*simp*]:
 $\llbracket S \in States\ (HARoot\ A) \rrbracket \implies (x, S) \notin (ChiPlus\ A)$

by (frule HARootStates-Range-ChiPlus, unfold Domain-converse [symmetric], fast)

lemma *SASates-ChiPlus-ChiRel-NoCycle-1* [rule-format]:

$\llbracket (S, U) \in \text{ChiPlus } A; SA \in \text{SAs } A \rrbracket \implies (U, T) \in (\text{ChiRel } A) \longrightarrow S \in \text{States } SA \longrightarrow T \in \text{States } SA \longrightarrow$
 $(\text{insert } S (\text{insert } U (\{V. (S, V) \in \text{ChiPlus } A \wedge (V, U) \in \text{ChiPlus } A\}))) \cap (\text{ChiRel } A \text{ “ } \{U\}) \neq \{\}$
apply (unfold ChiPlus-def)
apply (rule-tac a=S and b=U and r=(ChiRel A) in converse-trancl-induct)
apply (simp, safe)
apply (simp only: SASates-ChiRel-trans)
apply (simp add: ChiRel-CompFun)
apply safe
apply (erule-tac x=SA in ballE)
apply (simp add: CompFun-ChiRel2)+
apply (simp add: Int-def, fast)
apply auto
apply (fold ChiPlus-def)
apply (rename-tac W)
apply (frule-tac U=U and T=U and S=W in ChiRel-ChiPlus-trans2)
apply auto
done

lemma *SASates-ChiPlus-ChiRel-NoCycle-2* [rule-format]:

$\llbracket (S, U) \in \text{ChiPlus } A \rrbracket \implies (U, T) \in (\text{ChiRel } A) \longrightarrow$
 $(\text{insert } S (\text{insert } U (\{V. (S, V) \in \text{ChiPlus } A \wedge (V, U) \in \text{ChiPlus } A\}))) \cap$
 $(\text{ChiRel } A \text{ “ } \{S\}) \neq \{\}$
apply (unfold ChiPlus-def)
apply (rule-tac a=S and b=U and r=(ChiRel A) in converse-trancl-induct)
apply (unfold Image-def Int-def)
apply auto
done

lemma *SASates-ChiPlus-ChiRel-NoCycle-3* [rule-format]:

$\llbracket (S, U) \in \text{ChiPlus } A \rrbracket \implies (U, T) \in (\text{ChiRel } A) \longrightarrow (S, s) \in \text{ChiPlus } A \longrightarrow$
 $(s, U) \in \text{ChiPlus } A \longrightarrow$
 $(\text{insert } S (\text{insert } U (\{V. (S, V) \in \text{ChiPlus } A \wedge (V, U) \in \text{ChiPlus } A\}))) \cap$
 $(\text{ChiRel } A \text{ “ } \{s\}) \neq \{\}$
apply (unfold ChiPlus-def)
apply (rule-tac a=S and b=U and r=(ChiRel A) in trancl-induct)
apply fast
apply (rename-tac W)
prefer 2
apply (rename-tac W X)
prefer 2
apply (unfold Image-def Int-def)
apply (simp, safe)

```

apply (fold ChiPlus-def)
apply (case-tac ( $s, W \in \text{ChiRel } A$ )
apply fast
apply (frule-tac  $S=s$  and  $T=W$  in ChiPlus-ChiRel-Ex2)
apply simp
apply safe
apply (rename-tac  $X$ )
apply (rule-tac  $x=X$  in exI)
apply (fast intro: ChiRel-ChiPlus-trans)
apply simp
apply (case-tac ( $s, X \in \text{ChiRel } A$ )
apply force
apply (frule-tac  $S=s$  and  $T=X$  in ChiPlus-ChiRel-Ex2)
apply simp
apply safe
apply (rename-tac  $Y$ )
apply (erule-tac  $x=Y$  in allE)
apply simp
apply (fast intro: ChiRel-ChiPlus-trans)
apply simp
apply (case-tac ( $s, X \in \text{ChiRel } A$ )
apply force
apply (frule-tac  $S=s$  and  $T=X$  in ChiPlus-ChiRel-Ex2)
apply simp
apply safe
apply (rename-tac  $Y$ )
apply (erule-tac  $x=Y$  in allE)
apply simp
apply (fast intro: ChiRel-ChiPlus-trans)
apply fastforce
apply simp
apply (erule-tac  $x=W$  in allE)
apply simp
apply simp
apply (rename-tac  $Y$ )
apply (erule-tac  $x=Y$  in allE)
apply simp
apply (fast intro: ChiRel-ChiPlus-trans)
done

```

lemma *SASates-ChiPlus-ChiRel-trans* [*rule-format*]:

```

 $\llbracket (S, U) \in (\text{ChiPlus } A); (U, T) \in (\text{ChiRel } A); S \in \text{States } SA; SA \in \text{SAs } A \rrbracket \implies$ 
 $T \notin \text{States } SA$ 
apply (cut-tac  $HA=A$  in NoCycles-HA)
apply (unfold NoCycles-def)
apply (erule-tac  $x=\text{insert } S$  (insert  $U$  ( $\{V. (S, V) \in \text{ChiPlus } A \wedge (V, U) \in \text{ChiPlus } A\}$ )) in ballE)
prefer 2
apply (simp add: ChiPlus-subset-States)

```

```

apply (cut-tac  $A=A$  in ChiPlus-HAStates)
apply (unfold HAStates-def)
apply auto[1]
apply safe
apply fast
apply (frule SAStates-ChiPlus-ChiRel-NoCycle-2)
apply fast
apply (frule HAStates-SA-mem)
apply fast
apply (simp only: ChiRel-CompFun)
apply (frule SAStates-ChiPlus-ChiRel-NoCycle-1)
apply auto[3]
apply fast
apply (simp add: ChiRel-CompFun)
apply (frule SAStates-ChiPlus-ChiRel-NoCycle-3)
apply fast
apply fast
back
apply fast
apply (simp only: ChiPlus-HAStates-Left ChiRel-CompFun)
done

```

```

lemma SAStates-ChiPlus2 [rule-format]:
   $\llbracket (S,T) \in \text{ChiPlus } A; SA \in \text{SAs } A \rrbracket \implies S \in \text{States } SA \longrightarrow T \notin \text{States } SA$ 
apply (unfold ChiPlus-def)
apply (rule-tac  $a=S$  and  $b=T$  and  $r=(\text{ChiRel } A)$  in trancl-induct)
apply auto
apply (rename-tac  $U$ )
apply (frule-tac  $S=S$  and  $T=U$  in SAStates-ChiRel)
apply auto
apply (fold ChiPlus-def)
apply (simp only: SAStates-ChiPlus-ChiRel-trans)
done

```

```

lemma SAStates-ChiPlus [rule-format]:
   $\llbracket S \in \text{States } SA; T \in \text{States } SA; SA \in \text{SAs } A \rrbracket \implies (S,T) \notin \text{ChiPlus } A$ 
apply auto
apply (simp only: SAStates-ChiPlus2)
done

```

```

lemma SAStates-ChiPlus-ChiRel-OneAncestor [rule-format]:
   $\llbracket T \in \text{States } SA; SA \in \text{SAs } A; (S,U) \in \text{ChiPlus } A \rrbracket \implies S \neq T \longrightarrow S \in \text{States } SA \longrightarrow (T,U) \notin \text{ChiRel } A$ 
apply (unfold ChiPlus-def)
apply (rule-tac  $a=S$  and  $b=U$  and  $r=(\text{ChiRel } A)$  in trancl-induct)
apply auto
apply (simp add: ChiRel-OneAncestor-notmem)
apply (rename-tac  $V W$ )
apply (fold ChiPlus-def)

```

```

apply (case-tac  $V=T$ )
apply (simp add: ChiRel-OneAncestor-notmem SASates-ChiPlus) +
done

lemma SASates-ChiPlus-OneAncestor [rule-format]:
   $\llbracket T \in \text{States } SA; SA \in \text{SAs } A; (S,U) \in \text{ChiPlus } A \rrbracket \implies S \neq T \longrightarrow$ 
   $S \in \text{States } SA \longrightarrow (T,U) \notin \text{ChiPlus } A$ 
apply (unfold ChiPlus-def)
apply (rule-tac a=S and b=U and r=(ChiRel A) in trancl-induct)
apply auto
apply (fold ChiPlus-def)
apply (rename-tac V)
apply (frule-tac T=S and S=T and U=V in SASates-ChiPlus-ChiRel-OneAncestor)
apply auto
apply (rename-tac V W)
apply (frule-tac S=T and T=W in ChiPlus-ChiRel-Ex)
apply auto
apply (frule-tac T=T and S=S and U=W in SASates-ChiPlus-ChiRel-OneAncestor)
apply auto
apply (rule ChiRel-ChiPlus-trans)
apply auto
apply (rename-tac X)
apply (case-tac V=X)
apply simp
apply (simp add: ChiRel-OneAncestor-notmem)
done

lemma ChiRel-ChiPlus-OneAncestor [rule-format]:
   $\llbracket (T,U) \in \text{ChiPlus } A \rrbracket \implies T \neq S \longrightarrow (S,U) \in \text{ChiRel } A \longrightarrow (T,S) \in \text{ChiPlus}$ 
   $A$ 
apply (unfold ChiPlus-def)
apply (rule-tac a=T and b=U and r=(ChiRel A) in trancl-induct)
apply auto
apply (fast intro: ChiRel-OneAncestor)
apply (rename-tac V W)
apply (case-tac S=V)
apply auto
apply (fast intro: ChiRel-OneAncestor)
done

lemma ChiPlus-SA-OneAncestor [rule-format]:
   $\llbracket (S,T) \in \text{ChiPlus } A;$ 
   $U \in \text{States } SA; SA \in \text{SAs } A \rrbracket \implies T \in \text{States } SA \longrightarrow$ 
   $(S,U) \in \text{ChiPlus } A$ 
apply (unfold ChiPlus-def)
apply (rule-tac a=S and b=T and r=(ChiRel A) in converse-trancl-induct)
apply auto
apply (frule ChiRel-SA-OneAncestor)
apply fast+

```

done

7.2.14 *ChiStar*

lemma *ChiPlus-ChiStar* [simp]:
 $\llbracket (S, T) \in \text{ChiPlus } A \rrbracket \implies (S, T) \in \text{ChiStar } A$
by (unfold *ChiPlus-def* *ChiStar-def*, auto)

lemma *HARootState-Range-ChiStar* [simp]:
 $\llbracket x \neq S; S \in \text{States } (\text{HARoot } A) \rrbracket \implies (x, S) \notin (\text{ChiStar } A)$
apply (unfold *ChiStar-def*)
apply (subst *rtrancl-eq-or-trancl*)
apply (fold *ChiPlus-def*)
apply auto
done

lemma *ChiStar-Self* [simp]:
 $(S, S) \in \text{ChiStar } A$
apply (unfold *ChiStar-def*)
apply simp
done

lemma *ChiStar-Image* [simp]:
 $S \in M \implies S \in (\text{ChiStar } A \text{ `` } M)$
apply (unfold *Image-def*)
apply (auto intro: *ChiStar-Self*)
done

lemma *ChiStar-ChiPlus-noteq*:
 $\llbracket S \neq T; (S, T) \in \text{ChiStar } A \rrbracket \implies (S, T) \in \text{ChiPlus } A$
apply (unfold *ChiPlus-def* *ChiStar-def*)
apply (simp add: *rtrancl-eq-or-trancl*)
done

lemma *ChiRel-ChiStar-trans*:
 $\llbracket (S, U) \in \text{ChiStar } A; (U, T) \in \text{ChiRel } A \rrbracket \implies (S, T) \in \text{ChiStar } A$
apply (unfold *ChiStar-def*)
apply auto
done

7.2.15 *InitConf*

lemma *InitConf-HAStates* [simp]:
 $\text{InitConf } A \subseteq \text{HAStates } A$
apply (unfold *InitConf-def* *HAStates-def*)
apply auto
apply (rule *rtrancl-induct*)
back
apply auto
apply (rule-tac $x = \text{HARoot } A$ in *bexI*)

```

apply auto
apply (unfold HASStates-def ChiRel-def)
apply auto
done

```

```

lemma InitConf-HASStates2 [simp]:
   $S \in \text{InitConf } A \implies S \in \text{HASStates } A$ 
apply (cut-tac  $A=A$  in InitConf-HASStates)
apply fast
done

```

```

lemma HAINitState-InitConf [simp]:
   $\text{HAINitState } A \in \text{InitConf } A$ 
by (unfold HAINitState-def InitConf-def, auto)

```

```

lemma InitConf-HAINitState-HARoot:
   $\llbracket S \in \text{InitConf } A; S \neq \text{HAINitState } A \rrbracket \implies S \notin \text{States } (\text{HARoot } A)$ 
apply (unfold InitConf-def)
apply auto
apply (rule mp)
prefer 2
apply fast
back
apply (rule mp)
prefer 2
apply fast
back
back
back
apply (rule-tac  $b=S$  in rtrancl-induct)
apply auto
apply (simp add: ChiRel-HARoot)+
done

```

```

lemma InitConf-HARoot-HAINitState [simp]:
   $\llbracket S \in \text{InitConf } A; S \in \text{States } (\text{HARoot } A) \rrbracket \implies S = \text{HAINitState } A$ 
apply (subst not-not [THEN sym])
apply (rule notI)
apply (simp add: InitConf-HAINitState-HARoot)
done

```

```

lemma HAINitState-CompFun-InitConf [simp]:
   $\llbracket SA \in \text{the } (\text{CompFun } A \ (\text{HAINitState } A)) \rrbracket \implies (\text{InitState } SA) \in \text{InitConf } A$ 
apply (unfold InitConf-def HASStates-def)
apply auto
apply (rule rtrancl-Int)
apply auto
apply (cut-tac  $A=A$  and  $S=\text{HAINitState } A$  in HASStates-CompFun-States-ChiRel)
apply auto
apply (rule Image-singleton-iff [THEN subst])

```

```

apply (rotate-tac -1)
apply (drule sym)
apply simp
apply (rule-tac x=SA in bexI)
apply auto
done

```

```

lemma InitState-CompFun-InitConf:
  [| S ∈ HStates A; SA ∈ the (CompFun A S); S ∈ InitConf A |] ==> (InitState
  SA) ∈ InitConf A
apply (unfold InitConf-def)
apply auto
apply (rule-tac b=S in rtrancl-into-rtrancl)
apply fast
apply (frule rtrancl-Int1)
apply auto
apply (case-tac S = HInitState A)
apply simp
apply (rule rtrancl-mem-Sigma)
apply auto
apply (cut-tac A=A and S=S in HStates-CompFun-States-ChiRel)
apply auto
apply (rule Image-singleton-iff [THEN subst])
apply (rotate-tac -1)
apply (drule sym)
apply simp
apply (rule-tac x=SA in bexI)
apply auto
done

```

```

lemma InitConf-HInitStates:
  InitConf A ⊆ HInitStates A
apply (unfold InitConf-def)
apply (rule subsetI)
apply auto
apply (frule rtrancl-Int1)
apply (case-tac x = HInitState A)
apply simp
apply (rule rtrancl-mem-Sigma)
apply auto
done

```

```

lemma InitState-notmem-InitConf:
  [| SA ∈ the (CompFun A S); S ∈ InitConf A; T ∈ States SA;
  T ≠ InitState SA |] ==> T ∉ InitConf A
apply (frule InitConf-HStates2)
apply (unfold InitConf-def)
apply auto
apply (rule mp)

```



```

apply (erule rtrancLE)
apply auto
apply (rename-tac U)
apply (cut-tac A=A in HAINitState-notmem-Range-ChiRel)
apply auto
apply (case-tac U = S)
apply (auto simp add: ChiRel-OneAncestor)
done

lemma InitConf-CompFun-Ancestor:
   $\llbracket S \in HStates\ A; SA \in the\ (CompFun\ A\ S); T \in InitConf\ A; T \in States\ SA \rrbracket$ 
   $\implies S \in InitConf\ A$ 
apply (rule InitConf-ChiRel-Ancestor)
apply auto
apply (rule CompFun-ChiRel)
apply auto
done

7.2.16 StepConf

lemma StepConf-EmptySet [simp]:
  StepConf A C {} = C
by (unfold StepConf-def, auto)

end

```

8 Semantics of Hierarchical Automata

```

theory HASem
imports HA
begin

```

8.1 Definitions

```

definition
  RootExSem ::  $[(('s, 'e, 'd)seqauto)\ set, 's \rightarrow ('s, 'e, 'd)seqauto\ set,$ 
     $'s\ set] \Rightarrow bool$  where
    RootExSem F G C ==  $(\exists! S. S \in States\ (Root\ F\ G) \wedge S \in C)$ 

```

```

definition
  UniqueSucStates ::  $[(('s, 'e, 'd)seqauto)\ set, 's \rightarrow ('s, 'e, 'd)seqauto\ set,$ 
     $'s\ set] \Rightarrow bool$  where
    UniqueSucStates F G C ==  $\forall S \in (\bigcup (States\ 'F)).$ 
       $\forall A \in the\ (G\ S).$ 
      if  $(S \in C)$  then
         $\exists! S' . S' \in States\ A \wedge S' \in C$ 
      else
         $\forall S \in States\ A. S \notin C$ 

```

definition

$IsConfSet :: [((s,e,d)seqauto) \text{ set}, s \rightarrow (s,e,d)seqauto \text{ set},$
 $s \text{ set}] \Rightarrow \text{bool}$ **where**
 $IsConfSet \ F \ G \ C ==$
 $C \subseteq (\bigcup (States \ 'F)) \ \&$
 $RootExSem \ F \ G \ C \ \&$
 $UniqueSucStates \ F \ G \ C$

definition

$Status :: [(s,e,d)hierauto,$
 $s \text{ set},$
 $e \text{ set},$
 $d \text{ data}] \Rightarrow \text{bool}$ **where**
 $Status \ HA \ C \ E \ D == E \subseteq HAEvents \ HA \wedge$
 $IsConfSet \ (SAs \ HA) \ (CompFun \ HA) \ C \wedge$
 $Data.DataSpace \ (HAINitValue \ HA) = Data.DataSpace \ D$

8.1.1 Status**lemma Status-EmptySet:**

$(Abs-hierauto ((@ \ x . True),$
 $\{Abs-seqauto (\{ @ \ x . True\}, (@ \ x . True), \{\}, \{\}), \{\}, Map.empty(@ \ x . True$
 $\mapsto \{\})),$
 $\{ @ \ x . True\}, \{\}, @ \ x . True) \in$
 $\{(HA, C, E, D) \mid HA \ C \ E \ D. Status \ HA \ C \ E \ D\}$
apply (unfold Status-def CompFun-def SAs-def)
apply auto
apply (subst Abs-hierauto-inverse)
apply (subst hierauto-def)
apply (rule HierAuto-EmptySet)
apply (subst Abs-hierauto-inverse)
apply (subst hierauto-def)
apply (rule HierAuto-EmptySet)
apply auto
apply (unfold IsConfSet-def UniqueSucStates-def RootExSem-def)
apply auto
apply (unfold States-def)
apply auto
apply (unfold Root-def)
apply (rule someI2)
apply (rule conjI)
apply fast
apply (simp add: ran-def)
apply simp
apply (subst Abs-seqauto-inverse)
apply (subst seqauto-def)
apply (rule SeqAuto-EmptySet)
apply simp
apply (unfold HAINitValue-def)

```

apply auto
apply (subst Abs-hierauto-inverse)
apply (subst hierauto-def)
apply (rule HierAuto-EmptySet)
apply simp
done

```

definition

```

status =
  {(HA,C,E,D) |
    (HA::('s,'e,'d)hierauto)
    (C::('s set))
    (E::('e set))
    (D::('d data). Status HA C E D)}

```

```

typedef ('s,'e,'d) status =
  status :: (('s,'e,'d)hierauto * 's set * 'e set * 'd data) set
unfolding status-def
apply (rule exI)
apply (rule Status-EmptySet)
done

```

definition

```

HA :: ('s,'e,'d) status => ('s,'e,'d) hierauto where
HA == fst o Rep-status

```

definition

```

Conf :: ('s,'e,'d) status => 's set where
Conf == fst o snd o Rep-status

```

definition

```

Events :: ('s,'e,'d) status => 'e set where
Events == fst o snd o snd o Rep-status

```

definition

```

Value :: ('s,'e,'d) status => 'd data where
Value == snd o snd o snd o Rep-status

```

definition

```

RootState :: ('s,'e,'d) status => 's where
RootState ST == @ S. S ∈ Conf ST ∧ S ∈ States (HARoot (HA ST))

```

definition

```

EnabledTrans :: (('s,'e,'d)status * ('s,'e,'d)seqauto *

```

$(\text{'s, 'e, 'd})\text{trans}) \text{ set } \mathbf{where}$
 $EnabledTrans == \{(ST, SA, T) .$
 $SA \in SAs (HA ST) \wedge$
 $T \in Delta SA \wedge$
 $source T \in Conf ST \wedge$
 $(Conf ST, Events ST, Value ST) \models (label T) \}$

definition

$ET :: (\text{'s, 'e, 'd}) \text{ status} \Rightarrow ((\text{'s, 'e, 'd}) \text{ trans}) \text{ set } \mathbf{where}$
 $ET ST == \bigcup SA \in SAs (HA ST). (EnabledTrans \text{ `` } \{ST\}) \text{ `` } \{SA\}$

definition

$MaxNonConflict :: [(\text{'s, 'e, 'd})\text{status},$
 $(\text{'s, 'e, 'd})\text{trans set}] \Rightarrow \text{bool } \mathbf{where}$
 $MaxNonConflict ST T ==$
 $(T \subseteq ET ST) \wedge$
 $(\forall A \in SAs (HA ST). \text{card } (T \text{ Int } Delta A) \leq 1) \wedge$
 $(\forall t \in (ET ST). (t \in T) = (\neg (\exists t' \in ET ST. \text{HigherPriority } (HA ST)$
 $(t', t))))$

definition

$ResolveRacing :: (\text{'s, 'e, 'd})\text{trans set}$
 $\Rightarrow (\text{'d update set}) \mathbf{where}$
 $ResolveRacing TS ==$
 let
 $U = PUpdate (Label TS)$
 in
 $SequentialRacing U$

definition

$HPT :: (\text{'s, 'e, 'd})\text{status} \Rightarrow ((\text{'s, 'e, 'd})\text{trans set}) \text{ set } \mathbf{where}$
 $HPT ST == \{ T. MaxNonConflict ST T \}$

definition

InitStatus :: ('s,'e,'d)hierauto => ('s,'e,'d)status **where**
InitStatus A ==
Abs-status (A,InitConf A,{}, HAINitValue A)

definition

StepActEvent :: ('s,'e,'d)trans set => 'e set **where**
StepActEvent TS == Union (Actevent (Label TS))

definition

StepStatus :: [('s,'e,'d)status, ('s,'e,'d)trans set, 'd update]
 => ('s,'e,'d)status **where**
StepStatus ST TS U =
 (let
 (A,C,E,D) = Rep-status ST;
 C' = StepConf A C TS;
 E' = StepActEvent TS;
 D' = U !!! D
 in
Abs-status (A,C',E',D'))

definition

StepRelSem :: ('s,'e,'d)hierauto
 => (('s,'e,'d)status * ('s,'e,'d)status) set **where**
StepRelSem A == {(ST,ST'). (HA ST) = A ∧
 ((HPT ST ≠ {}) →
 (∃ TS ∈ HPT ST.
 ∃ U ∈ ResolveRacing TS.
 ST' = StepStatus ST TS U)) &
 ((HPT ST = {}) →
 (ST' = StepStatus ST {} DefaultUpdate))}

inductive-set

$ReachStati :: ('s, 'e, 'd)hierauto \Rightarrow ('s, 'e, 'd) status set$
for $A :: ('s, 'e, 'd)hierauto$
where
 $Status0 : InitStatus A \in ReachStati A$
| $StatusStep :$
 $\llbracket ST \in ReachStati A; TS \in HPT ST; U \in ResolveRacing TS \rrbracket$
 $\implies StepStatus ST TS U \in ReachStati A$

8.2 Lemmas**lemma** *Rep-status-tuple*:

$Rep\text{-}status ST = (HA ST, Conf ST, Events ST, Value ST)$
by (*unfold HA-def Conf-def Events-def Value-def, simp*)

lemma *Rep-status-select*:

$(HA ST, Conf ST, Events ST, Value ST) \in status$
by (*rule Rep-status-tuple [THEN subst], rule Rep-status*)

lemma *Status-select [simp]*:

$Status (HA ST) (Conf ST) (Events ST) (Value ST)$
apply (*cut-tac Rep-status-select*)
apply (*unfold status-def*)
apply *simp*
done

8.2.1 IsConfSet**lemma** *IsConfSet-Status [simp]*:

$IsConfSet (SAs (HA ST)) (CompFun (HA ST)) (Conf ST)$
apply (*cut-tac Rep-status-select*)
apply (*unfold status-def Status-def*)
apply *auto*
done

8.2.2 InitStatus**lemma** *IsConfSet-InitConf [simp]*:

$IsConfSet (SAs A) (CompFun A) (InitConf A)$
apply (*unfold IsConfSet-def RootExSem-def UniqueSucStates-def, fold HARoot-def*)
apply (*rule conjI*)
apply (*fold HAsStates-def, simp*)
apply (*rule conjI*)
apply (*rule-tac a=HAINitState A in ex1I*)
apply *auto*
apply (*rename-tac S SA*)

```

apply (case-tac  $S \in \text{InitConf } A$ )
apply auto
apply (rule-tac  $x = \text{InitState } SA \text{ in } exI$ )
apply auto
apply (rule InitState-CompFun-InitConf)
apply auto
apply (rename-tac  $S \ SA \ T \ U$ )
apply (case-tac  $U = \text{InitState } SA$ )
apply auto
apply (simp only:InitConf-CompFun-Ancestor HAStates-SA-mem, simp)+
done

```

```

lemma InitConf-status [simp]:
   $(A, \text{InitConf } A, \{\}, \text{HAINitValue } A) \in \text{status}$ 
apply (cut-tac Rep-status-select)
apply (unfold status-def Status-def)
apply auto
done

```

```

lemma Conf-InitStatus-InitConf [simp]:
   $\text{Conf } (\text{InitStatus } A) = \text{InitConf } A$ 
apply (unfold Conf-def InitStatus-def)
apply simp
apply (subst Abs-status-inverse)
apply auto
done

```

```

lemma HAINitValue-Value-DataSpace-Status [simp]:
   $\text{Data.DataSpace } (\text{HAINitValue } (HA \ ST)) = \text{Data.DataSpace } (\text{Value } ST)$ 
apply (cut-tac Rep-status-select)
apply (unfold status-def Status-def)
apply fast
done

```

```

lemma Value-InitStatus-HAINitValue [simp]:
   $\text{Value } (\text{InitStatus } A) = \text{HAINitValue } A$ 
apply (unfold Value-def InitStatus-def)
apply simp
apply (subst Abs-status-inverse)
apply auto
done

```

```

lemma HA-InitStatus [simp]:
   $HA \ (\text{InitStatus } A) = A$ 
apply (unfold InitStatus-def HA-def)
apply auto
apply (subst Abs-status-inverse)
apply auto
done

```

8.2.3 Events

lemma *Events-HAEvents-Status*:
 $(Events\ ST) \subseteq HAEvents\ (HA\ ST)$
apply (*cut-tac Rep-status-select*)
apply (*unfold status-def Status-def*)
apply *fast*
done

lemma *TS-EventSet*:
 $TS \subseteq ET\ ST \implies \bigcup (Actevent\ (Label\ TS)) \subseteq HAEvents\ (HA\ ST)$
apply (*unfold Actevent-def actevent-def ET-def EnabledTrans-def Action-def Label-def*)
apply (*cut-tac HA=HA ST in HAEvents-SAEvents-SAs*)
apply *auto*
apply (*rename-tac Event Source Trigger Guard Action Update Target*)
apply (*unfold SAEvents-def*)
apply (*erule subsetCE*)
apply *auto*
apply (*rename-tac SA*)
apply (*erule subsetCE*)
apply *auto*
apply (*erule-tac x=SA in ballE*)
apply *auto*
apply (*erule-tac x=(Trigger, Guard, Action, Update) in ballE*)
apply *auto*
apply (*cut-tac SA=SA in Label-Delta-subset*)
apply (*erule subsetCE*)
apply (*unfold Label-def image-def*)
apply *auto*
done

8.2.4 StepStatus

lemma *StepStatus-empty*:
 $Abs-status\ (HA\ ST, Conf\ ST, \{\}, U\ !!!\ (Value\ ST)) = StepStatus\ ST\ \{\}\ U$
apply (*unfold StepStatus-def Let-def*)
apply *auto*
apply (*subst Rep-status-tuple*)
apply *auto*
apply (*unfold StepActEvent-def*)
apply *auto*
done

lemma *status-empty-eventset [simp]*:
 $(HA\ ST, Conf\ ST, \{\}, U\ !!!\ (Value\ ST)) \in status$
apply (*unfold status-def Status-def*)
apply *auto*
done


```

lemma HA-StepStatus-emptyTS [simp]:
   $HA (StepStatus\ ST\ \{\}\ U) = HA\ ST$ 
apply (subst StepStatus-empty [THEN sym])
apply (unfold HA-def)
apply auto
apply (subst Abs-status-inverse)
apply auto
apply (subst Rep-status-tuple)
apply auto
done

```

8.2.5 Enabled Transitions *ET*

```

lemma HPT-ETI:
   $TS \in HPT\ ST \implies TS \subseteq ET\ ST$ 
by (unfold HPT-def MaxNonConflict-def, auto)

```

```

lemma finite-ET [simp]:
  finite (ET ST)
by (unfold ET-def Image-def EnabledTrans-def, auto)

```

8.2.6 Finite Transition Set

```

lemma finite-MaxNonConflict [simp]:
   $MaxNonConflict\ ST\ TS \implies finite\ TS$ 
apply (unfold MaxNonConflict-def)
apply auto
apply (subst finite-subset)
apply auto
done

```

```

lemma finite-HPT [simp]:
   $TS \in HPT\ ST \implies finite\ TS$ 
by (unfold HPT-def, auto)

```

8.2.7 *PUpdate*

```

lemma finite-Update:
   $finite\ TS \implies finite\ ((\lambda\ F.\ (Rep-pupdate\ F)\ (Value\ ST))\ '\ (PUpdate\ (Label\ TS)))$ 
by (rule finite-imageI, auto)

```

```

lemma finite-PUpdate:
   $TS \in HPT\ S \implies finite\ (Expr.PUpdate\ (Label\ TS))$ 
apply auto
done

```

```

lemma HPT-ResolveRacing-Some [simp]:
   $TS \in HPT\ S \implies (SOME\ u.\ u \in ResolveRacing\ TS) \in ResolveRacing\ TS$ 
apply (unfold ResolveRacing-def Let-def)
apply (rule finite-SequentialRacing)

```

apply *auto*
done

8.2.8 Higher Priority Transitions *HPT*

lemma *finite-HPT2* [*simp*]:
 finite (*HPT ST*)
apply (*cut-tac ST=ST in finite-ET*)
apply (*unfold HPT-def MaxNonConflict-def*)
apply (*subst Collect-subset*)
apply (*frule finite-Collect-subsets*)
apply *auto*
done

lemma *HPT-target-StepConf* [*simp*]:
 $\llbracket TS \in HPT\ ST; T \in TS \rrbracket \implies \text{target } T \in StepConf\ (HA\ ST)\ (Conf\ ST)\ TS$
apply (*unfold StepConf-def*)
apply *auto*
done

lemma *HPT2-target-StepConf2* [*simp*]:
 $\llbracket TS \in HPT\ ST; (S,L,T) \in TS \rrbracket \implies T \in StepConf\ (HA\ ST)\ (Conf\ ST)\ TS$
apply (*unfold StepConf-def Target-def Source-def source-def target-def image-def*)
apply *auto*
apply *auto*
done

8.2.9 Delta Transition Set

lemma *ET-Delta*:
 $\llbracket TS \subseteq ET\ ST; t \in TS; \text{source } t \in States\ A; A \in SAs\ (HA\ ST) \rrbracket \implies t \in Delta\ A$
apply (*unfold ET-def EnabledTrans-def*)
apply *simp*
apply (*erule subsetCE*)
apply *auto*
apply (*rename-tac SA*)
apply (*case-tac A = SA*)
apply *auto*
apply (*cut-tac HA=HA ST in MutuallyDistinct-HA*)
apply (*unfold MutuallyDistinct-def*)
apply *force*
done

lemma *ET-Delta-target*:
 $\llbracket TS \subseteq ET\ ST; t \in TS; \text{target } t \in States\ A; A \in SAs\ (HA\ ST) \rrbracket \implies t \in Delta\ A$
apply (*unfold ET-def EnabledTrans-def*)
apply *simp*
apply (*erule subsetCE*)

```

apply auto
apply (rename-tac SA)
apply (case-tac A = SA)
apply auto
apply (cut-tac HA=HA ST in MutuallyDistinct-HA)
apply (unfold MutuallyDistinct-def)
apply force
done

```

```

lemma ET-HADelta:
   $\llbracket TS \subseteq ET\ ST; t \in TS \rrbracket \implies t \in HADelta\ (HA\ ST)$ 
apply (unfold HADelta-def)
apply auto
apply (unfold ET-def EnabledTrans-def Image-def)
apply auto
done

```

```

lemma HPT-HADelta:
   $\llbracket TS \in HPT\ ST; t \in TS \rrbracket \implies t \in HADelta\ (HA\ ST)$ 
apply (rule ET-HADelta)
apply (unfold HPT-def MaxNonConflict-def)
apply auto
done

```

```

lemma HPT-Delta:
   $\llbracket TS \in HPT\ ST; t \in TS; \text{source } t \in \text{States } A; A \in \text{SAs } (HA\ ST) \rrbracket \implies t \in \text{Delta}_A$ 
apply (rule ET-Delta)
apply auto
apply (unfold HPT-def MaxNonConflict-def)
apply fast
done

```

```

lemma HPT-Delta-target:
   $\llbracket TS \in HPT\ ST; t \in TS; \text{target } t \in \text{States } A; A \in \text{SAs } (HA\ ST) \rrbracket \implies t \in \text{Delta}_A$ 
apply (rule ET-Delta-target)
apply auto
apply (unfold HPT-def MaxNonConflict-def)
apply fast
done

```

```

lemma OneTrans-HPT-SA:
   $\llbracket TS \in HPT\ ST; T \in TS; \text{source } T \in \text{States } SA; \\ U \in TS; \text{source } U \in \text{States } SA; SA \in \text{SAs } (HA\ ST) \rrbracket \implies T = U$ 
apply (unfold HPT-def MaxNonConflict-def Source-def)
apply auto
apply (erule-tac x=SA in ballE)
apply (case-tac finite (TS  $\cap$  Delta SA))

```

```

apply (frule-tac  $t=T$  in OneElement-Card)
apply fast
apply (frule-tac  $t=T$  and  $A=SA$  in ET-Delta)
apply assumption+
apply fast
apply (frule-tac  $t=U$  in OneElement-Card)
apply fast
apply (frule-tac  $t=U$  and  $A=SA$  in ET-Delta)
apply auto
done

```

lemma *OneTrans-HPT-SA2*:

```

   $\llbracket TS \in HPT\ ST; T \in TS; \text{target } T \in \text{States } SA;$ 
   $U \in TS; \text{target } U \in \text{States } SA; SA \in SAs\ (HA\ ST) \rrbracket \implies T = U$ 
apply (unfold HPT-def MaxNonConflict-def Target-def)
apply auto
apply (erule-tac  $x=SA$  in ballE)
apply (case-tac finite ( $TS \cap Delta\ SA$ ))
apply (frule-tac  $t=T$  in OneElement-Card)
apply fast
apply (frule-tac  $t=T$  and  $A=SA$  in ET-Delta-target)
apply assumption+
apply fast
apply (frule-tac  $t=U$  in OneElement-Card)
apply fast
apply (frule-tac  $t=U$  and  $A=SA$  in ET-Delta-target)
apply auto
done

```

8.2.10 Target Transition Set

lemma *ET-Target-HAStates*:

```

   $TS \subseteq ET\ ST \implies \text{Target } TS \subseteq HAStates\ (HA\ ST)$ 
apply (unfold HAStates-def Target-def target-def ET-def EnabledTrans-def Action-def
Label-def)
apply (cut-tac  $HA=HA\ ST$  in Target-SAs-Delta-States)
apply auto
apply (rename-tac Source Trigger Guard Action Update Target)
apply (unfold Target-def)
apply (erule subsetCE)
apply auto
apply (rename-tac SA)
apply (erule subsetCE)
apply auto
apply (unfold image-def)
apply auto
apply (metis target-select)
done

```

lemma *HPT-Target-HAStates*:
 $TS \in \text{HPT } ST \implies \text{Target } TS \subseteq \text{HAStates } (HA \text{ } ST)$
apply (*rule HPT-ETI [THEN ET-Target-HAStates]*)
apply *assumption*
done

lemma *HPT-Target-HAStates2 [simp]*:
 $\llbracket TS \in \text{HPT } ST; S \in \text{Target } TS \rrbracket \implies S \in \text{HAStates } (HA \text{ } ST)$
apply (*cut-tac HPT-Target-HAStates*)
apply *fast+*
done

lemma *OneState-HPT-Target*:
 $\llbracket TS \in \text{HPT } ST; S \in \text{Target } TS;$
 $T \in \text{Target } TS; S \in \text{States } SA;$
 $T \in \text{States } SA; SA \in \text{SAs } (HA \text{ } ST) \rrbracket$
 $\implies S = T$
apply (*unfold Target-def*)
apply (*auto dest: OneTrans-HPT-SA2[rotated -1]*)
done

8.2.11 Source Transition Set

lemma *ET-Source-Conf*:
 $TS \subseteq \text{ET } ST \implies (\text{Source } TS) \subseteq \text{Conf } ST$
apply (*unfold Source-def ET-def EnabledTrans-def*)
apply *auto*
done

lemma *HPT-Source-Conf [simp]*:
 $TS \in \text{HPT } ST \implies (\text{Source } TS) \subseteq \text{Conf } ST$
apply (*unfold HPT-def MaxNonConflict-def*)
apply (*rule ET-Source-Conf*)
apply *auto*
done

lemma *ET-Source-Target [simp]*:
 $\llbracket SA \in \text{SAs } (HA \text{ } ST); TS \subseteq \text{ET } ST; \text{States } SA \cap \text{Source } TS = \{\} \rrbracket \implies \text{States } SA \cap \text{Target } TS = \{\}$
apply (*unfold ET-def EnabledTrans-def Source-def Target-def*)
apply *auto*
apply (*rename-tac Source Trigger Guard Action Update Target*)
apply (*erule subsetCE*)
apply *auto*
apply (*rename-tac SAA*)
apply (*unfold image-def source-def Int-def*)
apply *auto*
apply (*erule-tac x=Source in allE*)
apply *auto*

```

apply (frule Delta-source-States)
apply (unfold source-def)
apply auto
apply (case-tac SA=SAA)
apply auto
apply (cut-tac HA=HA ST in MutuallyDistinct-HA)
apply (unfold MutuallyDistinct-def)
apply (erule-tac x=SA in ballE)
apply (erule-tac x=SAA in ballE)
apply auto
apply (frule Delta-target-States)
apply (unfold target-def)
apply force
done

```

```

lemma HPT-Source-Target [simp]:
   $\llbracket TS \in HPT\ ST; States\ SA \cap Source\ TS = \{\}; SA \in SAs\ (HA\ ST) \rrbracket \implies States\ SA \cap Target\ TS = \{\}$ 
apply (unfold HPT-def MaxNonConflict-def)
apply auto
done

```

```

lemma ET-target-source:
   $\llbracket TS \subseteq ET\ ST; t \in TS; target\ t \in States\ A; A \in SAs\ (HA\ ST) \rrbracket \implies source\ t \in States\ A$ 
apply (frule ET-Delta-target)
apply auto
done

```

```

lemma ET-source-target:
   $\llbracket TS \subseteq ET\ ST; t \in TS; source\ t \in States\ A; A \in SAs\ (HA\ ST) \rrbracket \implies target\ t \in States\ A$ 
apply (frule ET-Delta)
apply auto
done

```

```

lemma HPT-target-source:
   $\llbracket TS \in HPT\ ST; t \in TS; target\ t \in States\ A; A \in SAs\ (HA\ ST) \rrbracket \implies source\ t \in States\ A$ 
apply (rule ET-target-source)
apply auto
apply (unfold HPT-def MaxNonConflict-def)
apply fast
done

```

```

lemma HPT-source-target:
   $\llbracket TS \in HPT\ ST; t \in TS; source\ t \in States\ A; A \in SAs\ (HA\ ST) \rrbracket \implies target\ t \in States\ A$ 
apply (rule ET-source-target)

```

```

apply auto
apply (unfold HPT-def MaxNonConflict-def)
apply fast
done

lemma HPT-source-target2 [simp]:
   $\llbracket TS \in \text{HPT } ST; (s, l, t) \in TS; s \in \text{States } A; A \in \text{SAs } (HA \text{ } ST) \rrbracket \implies t \in \text{States } A$ 
apply (cut-tac ST=ST and TS=TS and t=(s,l,t) in HPT-source-target)
apply auto
done

lemma ChiRel-ChiStar-Source-notmem:
   $\llbracket TS \in \text{HPT } ST; (S, T) \in \text{ChiRel } (HA \text{ } ST); S \in \text{Conf } ST; \\ T \notin \text{ChiStar } (HA \text{ } ST) \text{ ``Source } TS \rrbracket \implies \\ S \notin \text{ChiStar } (HA \text{ } ST) \text{ ``Source } TS$ 
apply auto
apply (rename-tac U)
apply (simp only: Image-def)
apply auto
apply (erule-tac x=U in ballE)
apply (fast intro: ChiRel-ChiStar-trans)+
done

lemma ChiRel-ChiStar-notmem:
   $\llbracket TS \in \text{HPT } ST; (S, T) \in \text{ChiRel } (HA \text{ } ST); \\ S \in \text{ChiStar } (HA \text{ } ST) \text{ ``Source } TS \rrbracket \implies T \notin \text{Source } TS$ 
using [hypsubst-thin = true]
apply (unfold HPT-def MaxNonConflict-def HigherPriority-def restrict-def)
apply auto
apply (rename-tac U)
apply (unfold Source-def image-def)
apply auto
apply (rename-tac SSource STrigger SGuard SAction SUpdate STarget
  TSource TTrigger TGuard TAction TUpdate TTarget)
apply (erule-tac x=(SSource, (STrigger, SGuard, SAction, SUpdate), STarget) in ballE)
apply auto
apply (erule-tac x=(TSource, (TTrigger, TGuard, TAction, TUpdate), TTarget) in ballE)
apply auto
apply (simp add: ET-HADelta)
apply (case-tac SSource=S)
apply auto
apply (frule ChiStar-ChiPlus-noteq)
apply fast
apply (fast intro: ChiRel-ChiPlus-trans)
done

```

8.2.12 *StepActEvents*

lemma *StepActEvent-empty* [simp]:
 $\text{StepActEvent } \{\} = \{\}$
by (*unfold StepActEvent-def*, *auto*)

lemma *StepActEvent-HAEvents*:
 $TS \in \text{HPT } ST \implies \text{StepActEvent } TS \subseteq \text{HAEvents } (HA \ ST)$
apply (*unfold StepActEvent-def image-def*)
apply (*rule HPT-ETI [THEN TS-EventSet]*)
apply *assumption*
done

8.2.13 *UniqueSucStates*

lemma *UniqueSucStates-Status* [simp]:
 $\text{UniqueSucStates } (SAs \ (HA \ ST)) \ (\text{CompFun } (HA \ ST)) \ (\text{Conf } ST)$
apply (*cut-tac Rep-status-select*)
apply (*unfold status-def Status-def IsConfSet-def*)
apply *auto*
done

8.2.14 *RootState*

lemma *RootExSem-Status* [simp]:
 $\text{RootExSem } (SAs \ (HA \ ST)) \ (\text{CompFun } (HA \ ST)) \ (\text{Conf } ST)$
apply (*cut-tac Rep-status-select*)
apply (*unfold status-def Status-def IsConfSet-def*)
apply *auto*
done

lemma *RootState-HARootState* [simp]:
 $(\text{RootState } ST) \in \text{States } (\text{HARoot } (HA \ ST))$
apply (*unfold RootState-def*)
apply (*cut-tac ST=ST in RootExSem-Status*)
apply (*unfold RootExSem-def HARoot-def HAStates-def*)
apply *auto*
apply (*subst some1-equality*)
apply *auto*
done

lemma *RootState-Conf* [simp]:
 $(\text{RootState } ST) \in (\text{Conf } ST)$
apply (*unfold RootState-def*)
apply (*cut-tac ST=ST in RootExSem-Status*)
apply (*unfold RootExSem-def HARoot-def HAStates-def*)
apply *auto*
apply (*subst some1-equality*)
apply *auto*
done

lemma *RootState-notmem-Chi* [simp]:
 $S \in HAStates (HA ST) \implies (RootState ST) \notin Chi (HA ST) S$
by *auto*

lemma *RootState-notmem-Range-ChiRel* [simp]:
 $RootState ST \notin Range (ChiRel (HA ST))$
by *auto*

lemma *RootState-Range-ChiPlus* [simp]:
 $RootState ST \notin Range (ChiPlus (HA ST))$
by *auto*

lemma *RootState-Range-ChiStar* [simp]:
 $\llbracket x \neq RootState ST \rrbracket \implies (x, RootState ST) \notin (ChiStar (HA ST))$
by *auto*

lemma *RootState-notmem-ChiRel* [simp]:
 $(x, RootState ST) \notin (ChiRel (HA ST))$
by (*unfold ChiRel-def, auto*)

lemma *RootState-notmem-ChiRel2* [simp]:
 $\llbracket S \in States (HARoot (HA ST)) \rrbracket \implies (x, S) \notin (ChiRel (HA ST))$
by (*unfold ChiRel-def, auto*)

lemma *RootState-Conf-StepConf* [simp]:
 $\llbracket RootState ST \notin Source TS \rrbracket \implies RootState ST \in StepConf (HA ST) (Conf ST) TS$
apply (*unfold StepConf-def*)
apply *auto*
apply (*rename-tac S*)
apply (*case-tac S=RootState ST*)
apply *fast*
apply *auto*
apply (*rename-tac S*)
apply (*case-tac S=RootState ST*)
apply *fast*
apply *auto*
done

lemma *OneRootState-Conf* [simp]:
 $\llbracket S \in States (HARoot (HA ST)); S \in Conf ST \rrbracket \implies S = RootState ST$
apply (*cut-tac ST=ST in IsConfSet-Status*)
apply (*unfold IsConfSet-def RootExSem-def*)
apply (*fold HARoot-def*)
apply *auto*
done

lemma *OneRootState-Source*:

$\llbracket TS \in HPT\ ST; S \in Source\ TS; S \in States\ (HARoot\ (HA\ ST)) \rrbracket \implies S = RootState\ ST$
apply (*cut-tac* $ST=ST$ **and** $TS=TS$ **in** *HPT-Source-Conf*, *assumption*)
apply (*cut-tac* $ST=ST$ **in** *OneRootState-Conf*)
apply *fast+*
done

lemma *OneState-HPT-Target-Source*:
 $\llbracket TS \in HPT\ ST; S \in States\ SA; SA \in SAs\ (HA\ ST);$
 $States\ SA \cap Source\ TS = \{\} \rrbracket$
 $\implies S \notin Target\ TS$
apply (*unfold* *Target-def*)
apply *auto*
apply (*unfold* *Source-def* *Image-def* *Int-def*)
apply *auto*
apply (*frule* *HPT-target-source*)
apply *auto*
done

lemma *RootState-notmem-Target* [*simp*]:
 $\llbracket TS \in HPT\ ST; S \in States\ (HARoot\ (HA\ ST)); RootState\ ST \notin Source\ TS \rrbracket$
 $\implies S \notin Target\ TS$
apply *auto*
apply (*frule* *OneState-HPT-Target-Source*)
prefer 4
apply *fast+*
apply *simp*
apply (*unfold* *Int-def*)
apply *auto*
apply (*frule* *OneRootState-Source*)
apply *fast+*
done

8.2.15 Configuration Conf

lemma *Conf-HAStates*:
 $Conf\ ST \subseteq HAStates\ (HA\ ST)$
apply (*cut-tac* *Rep-status-select*)
apply (*unfold* *IsConfSet-def* *status-def* *Status-def* *HAStates-def*)
apply *fast*
done

lemma *Conf-HAStates2* [*simp*]:
 $S \in Conf\ ST \implies S \in HAStates\ (HA\ ST)$
apply (*cut-tac* $ST=ST$ **in** *Conf-HAStates*)
apply *fast*
done

lemma *OneState-Conf* [*intro*]:

```

   $\llbracket S \in \text{Conf } ST; T \in \text{Conf } ST; S \in \text{States } SA; T \in \text{States } SA; \\ SA \in \text{SAs } (HA \text{ } ST) \rrbracket \implies T = S$ 
apply (cut-tac  $ST=ST$  in IsConfSet-Status)
apply (unfold IsConfSet-def UniqueSucStates-def)
apply (case-tac  $SA = HARoot \text{ } (HA \text{ } ST)$ )
apply (cut-tac  $ST=ST$  and  $S=S$  in OneRootState-Conf)
apply fast+
apply (simp only:OneRootState-Conf)
apply (erule conjE)+
apply (cut-tac  $HA=HA \text{ } ST$  in OneAncestor-HA)
apply (unfold OneAncestor-def)
apply (fold HARoot-def)
apply (erule-tac  $x=SA$  in ballE)
apply (erule ex1-implies-ex)
apply (erule exE)
apply (rename-tac  $U$ )
apply (erule-tac  $x=U$  in ballE)
apply (erule-tac  $x=SA$  in ballE)
apply (case-tac  $U \in \text{Conf } ST$ )
apply simp
apply safe
apply fast+
apply simp
apply fast
done

```

lemma *OneState-HPT-SA*:

```

   $\llbracket TS \in \text{HPT } ST; S \in \text{Source } TS; T \in \text{Source } TS; \\ S \in \text{States } SA; T \in \text{States } SA; \\ SA \in \text{SAs } (HA \text{ } ST) \rrbracket \implies S = T$ 
apply (rule OneState-Conf)
apply auto
apply (frule HPT-Source-Conf, fast)+
done

```

lemma *HPT-SASates-Target-Source*:

```

   $\llbracket TS \in \text{HPT } ST; A \in \text{SAs } (HA \text{ } ST); S \in \text{States } A; T \in \text{States } A; S \in \text{Conf } ST; \\ T \in \text{Target } TS \rrbracket \implies S \in \text{Source } TS$ 
apply (case-tac  $\text{States } A \cap \text{Source } TS = \{\}$ )
apply (frule OneState-HPT-Target-Source)
apply fast
back
apply simp+
apply auto
apply (rename-tac  $U$ )
apply (cut-tac  $ST=ST$  in HPT-Source-Conf)
apply fast
apply (frule-tac  $S=S$  and  $T=U$  in OneState-Conf)
apply fast+

```

done

lemma *HPT-Conf-Target-Source*:

$\llbracket TS \in \text{HPT } ST; S \in \text{Conf } ST; \\ S \in \text{Target } TS \rrbracket \implies S \in \text{Source } TS$

apply (*frule* *Conf-HAStates2*)

apply (*unfold* *HAStates-def*)

apply *auto*

apply (*simp only: HPT-SAStates-Target-Source*)

done

lemma *Conf-SA*:

$S \in \text{Conf } ST \implies \exists A \in \text{SAs } (HA \text{ } ST). S \in \text{States } A$

apply (*cut-tac* *ST=ST in IsConfSet-Status*)

apply (*unfold* *IsConfSet-def*)

apply *fast*

done

lemma *HPT-Source-HAStates* [*simp*]:

$\llbracket TS \in \text{HPT } ST; S \in \text{Source } TS \rrbracket \implies S \in \text{HAStates } (HA \text{ } ST)$

apply (*frule* *HPT-Source-Conf*)

apply (*rule* *Conf-HAStates2*)

apply *fast*

done

lemma *Conf-Ancestor*:

$\llbracket S \in \text{Conf } ST; A \in \text{the } (\text{CompFun } (HA \text{ } ST) \text{ } S) \rrbracket \implies \exists! T \in \text{States } A. T \in \text{Conf } ST$

apply (*cut-tac* *ST=ST in IsConfSet-Status*)

apply (*unfold* *IsConfSet-def UniqueSucStates-def*)

apply *safe*

apply (*erule-tac* *x=S in ballE*)

prefer 2

apply *blast*

apply (*erule-tac* *x=A in ballE*)

prefer 2

apply *fast*

apply *simp*

apply (*fast intro: HAStates-CompFun-SAs-mem Conf-HAStates2*)

done

lemma *Conf-ChiRel*:

$\llbracket (S, T) \in \text{ChiRel } (HA \text{ } ST); T \in \text{Conf } ST \rrbracket \implies S \in \text{Conf } ST$

apply (*unfold* *ChiRel-def Chi-def restrict-def*)

apply *simp*

apply *safe*

apply *simp*

apply *safe*

```

apply (rename-tac SA)
apply (unfold HASStates-def)
apply simp
apply safe
apply (rename-tac U)
apply (cut-tac ST=ST in UniqueSucStates-Status)
apply (unfold UniqueSucStates-def)
apply (erule-tac x=S in ballE)
apply (erule-tac x=SA in ballE)
apply auto
apply (case-tac S ∈ Conf ST)
apply simp+
done

```

lemma *Conf-ChiPlus*:

```

   $\llbracket (T, S) \in \text{ChiPlus } (HA \ ST) \rrbracket \implies S \in \text{Conf } ST \longrightarrow T \in \text{Conf } ST$ 
apply (unfold ChiPlus-def)
apply (rule-tac a=T and b=S and r=(ChiRel (HA ST)) in trancl-induct)
apply (fast intro: Conf-ChiRel)+
done

```

lemma *HPT-Conf-Target-Source-ChiPlus*:

```

   $\llbracket TS \in \text{HPT } ST; S \in \text{Conf } ST; S \in \text{ChiPlus } (HA \ ST) \text{ “ Target } TS \rrbracket$ 
     $\implies S \in \text{ChiStar } (HA \ ST) \text{ “ Source } TS$ 
apply auto
apply (rename-tac T)
apply (simp add: Image-def)
apply (frule HPT-Target-HASStates2)
apply fast
apply (unfold HASStates-def)
apply auto
apply (rename-tac SA)
apply (case-tac States SA ∩ Source TS = {})
apply (simp only: OneState-HPT-Target-Source)
apply auto
apply (rename-tac U)
apply (erule-tac x=U in ballE)
apply auto
apply (case-tac U=T)
apply auto
apply (frule Conf-ChiPlus)
apply simp
apply (frule HPT-Conf-Target-Source)
apply fast
back
apply fast
apply (simp add: OneState-HPT-SA)
done

```

```

lemma OneState-HPT-Target-ChiRel:
  [[  $TS \in \text{HPT } ST$ ;  $(U, T) \in \text{ChiRel } (HA \text{ } ST)$ ;
     $U \in \text{Target } TS$ ;  $A \in \text{SAs } (HA \text{ } ST)$ ;  $T \in \text{States } A$ ;
     $S \in \text{States } A$  ]]  $\implies S \notin \text{Target } TS$ 
using [[hypsubst-thin = true]]
apply auto
apply (unfold HigherPriority-def restrict-def HPT-def MaxNonConflict-def Target-def)
apply auto
apply (rename-tac SSource STTrigger SGuard SAction SUpdate STarget
        TSource TTrigger TGuard TAction TUpdate TTarget)
apply (cut-tac  $t=(TSource, (TTrigger, TGuard, TAction, TUpdate), TTarget)$ 
  and  $TS=TS$  and  $ST=ST$  and  $A=A$  in ET-target-source)
apply assumption+
apply simp
apply assumption
apply (frule ChiRel-HAStates)
apply (unfold HAStates-def)
apply safe
apply (cut-tac  $t=(SSource, (STTrigger, SGuard, SAction, SUpdate), STarget)$  and
   $A=x$  and  $ST=ST$  and  $TS=TS$  in ET-target-source)
apply assumption+
apply simp
apply assumption
apply simp
apply (erule-tac  $x=(SSource, (STTrigger, SGuard, SAction, SUpdate), STarget)$  in
ballE)
apply simp
apply (erule-tac  $x=(TSource, (TTrigger, TGuard, TAction, TUpdate), TTarget)$ 
in ballE)
apply (simp add: ET-HADelta)
apply (cut-tac  $A=HA \text{ } ST$  and  $S=STarget$  and  $T=T$  and  $U=TSource$  in ChiRel-SA-OneAncestor)
apply fast+
apply (frule ET-Source-Conf)
apply (unfold Source-def image-def)
apply (case-tac  $SSource \in \text{Conf } ST$ )
prefer 2
apply (erule subsetCE)
back
apply fast
back
apply simp
apply (case-tac  $TSource \in \text{Conf } ST$ )
prefer 2
apply (erule subsetCE)
back
apply fast
apply simp
apply (case-tac  $STarget=SSource$ )

```

```

apply simp
apply (fast intro: Conf-ChiRel)+
done

lemma OneState-HPT-Target-ChiPlus [rule-format]:
  
$$\llbracket TS \in \text{HPT } ST; (U, T) \in \text{ChiPlus } (HA \text{ } ST);$$


$$S \in \text{Target } TS; A \in \text{SAs } (HA \text{ } ST);$$


$$S \in \text{States } A \rrbracket \implies T \in \text{States } A \longrightarrow U \notin \text{Target } TS$$

using [hypsubst-thin = true]
apply (unfold ChiPlus-def)
apply (rule-tac a=U and b=T and r=(ChiRel (HA ST)) in converse-trancl-induct)
apply auto
apply (simp only: OneState-HPT-Target-ChiRel)
apply (rename-tac V W)
apply (fold ChiPlus-def)
apply (unfold HPT-def MaxNonConflict-def Target-def HigherPriority-def restrict-def)
apply auto
apply (rename-tac SSource STrigger SGuard SAction SUpdate STarget

$$TSource TTrigger TGuard TAction TUpdate TTarget$$
)
apply (cut-tac t=(SSource, (STrigger, SGuard, SAction, SUpdate), STarget) and

$$ST=ST \text{ and } TS=TS \text{ and } A=A \text{ in } ET\text{-target-source}$$
)
apply assumption+
apply simp
apply assumption
apply simp
apply (frule ChiRel-HAStates)
apply (unfold HAStates-def)
apply safe
apply (cut-tac t=(TSource, (TTrigger, TGuard, TAction, TUpdate), TTarget)

$$\text{and } A=x \text{ and } TS=TS \text{ and } ST=ST \text{ in } ET\text{-target-source}$$
)
apply assumption+
apply simp
apply assumption
apply simp
apply (erule-tac x=(TSource, (TTrigger, TGuard, TAction, TUpdate), TTarget)

$$\text{in } ballE$$
)
apply simp
apply (erule-tac x=(SSource, (STrigger, SGuard, SAction, SUpdate), STarget) in

$$ballE$$
)
apply (simp add: ET-HADelta)
apply (cut-tac A=HA ST and S=TTarget and T=T and U=SSource in ChiPlus-SA-OneAncestor)
apply (fast intro: ChiRel-ChiPlus-trans2)
apply fast+
apply (frule ET-Source-Conf)
apply (unfold Source-def image-def)
apply (case-tac SSource  $\in$  Conf ST)
prefer 2
apply (erule subsetCE)
back

```

```

apply fast
apply simp
apply (case-tac  $TSource \in Conf\ ST$ )
prefer 2
apply (erule subsetCE)
back
apply fast
back
apply simp
apply (case-tac  $TTarget = SSource$ )
apply simp
apply (frule-tac  $T = TTarget$  and  $S = SSource$  in Conf-ChiPlus)
apply simp
apply (frule-tac  $T = TSource$  and  $S = TTarget$  in OneState-Conf)
apply fast
done

```

8.2.16 *RootExSem*

```

lemma RootExSem-StepConf:
   $\llbracket TS \in HPT\ ST \rrbracket \implies$ 
     $RootExSem\ (SAs\ (HA\ ST))\ (CompFun\ (HA\ ST))\ (StepConf\ (HA\ ST)\ (Conf\ ST)\ TS)$ 
apply (unfold RootExSem-def)
apply (fold HARoot-def)
apply auto
apply (case-tac  $RootState\ ST \notin Source\ TS$ )
apply (rule-tac  $x = RootState\ ST$  in exI)
apply simp
apply simp
apply (unfold Source-def image-def)
apply simp
apply (erule bexE)
apply (rename-tac  $T$ )
apply (rule-tac  $x = target\ T$  in exI)
apply simp
apply (rule HPT-source-target)
apply auto
apply (rename-tac  $S\ T$ )
apply (case-tac  $S \in Conf\ ST$ )
apply (case-tac  $T \in Conf\ ST$ )
apply (frule OneRootState-Conf)
apply auto
apply (frule OneRootState-Conf)
apply auto
apply (frule OneRootState-Conf)
apply auto
apply (case-tac  $RootState\ ST \in Source\ TS$ )
apply (case-tac  $T \in Source\ TS$ )

```



```

apply (frule HPT-Source-Conf)
apply fast
apply (unfold StepConf-def)
apply auto
apply (frule OneState-HPT-Target)
apply (frule-tac  $SA=HARoot\ (HA\ ST)\ \text{and}\ TS=TS\ \text{and}\ S=T\ \text{and}\ T=RootState\ ST$  in OneState-HPT-Target)
apply fast+
apply simp+
apply (frule tranc1-Int-mem, fold ChiPlus-def, force) +
prefer 2
apply (frule OneState-HPT-Target)
apply fast+
back
apply simp+
apply (case-tac  $RootState\ ST \in Source\ TS$ )
apply (case-tac  $T = RootState\ ST$ )
apply auto
apply (frule tranc1-Int-mem, fold ChiPlus-def, force) +
done

```

8.2.17 *StepConf*

lemma *Target-StepConf*:

```

 $S \in Target\ TS \implies S \in StepConf\ (HA\ ST)\ (Conf\ ST)\ TS$ 
apply (unfold StepConf-def)
apply auto
done

```

lemma *Target-ChiRel-HAInit-StepConf*:

```

 $\llbracket S \in Target\ TS; (S, T) \in ChiRel\ A; \\ T \in HInitStates\ A \rrbracket \implies T \in StepConf\ A\ C\ TS$ 
apply (unfold StepConf-def)
apply auto
done

```

lemma *StepConf-HAStates*:

```

 $TS \in HPT\ ST \implies StepConf\ (HA\ ST)\ (Conf\ ST)\ TS \subseteq HStates\ (HA\ ST)$ 
apply (unfold StepConf-def)
apply auto
apply (frule tranc1D2)
apply auto
done

```

lemma *RootState-Conf-StepConf2* [*simp*]:

```

 $\llbracket source\ T = RootState\ ST; T \in TS \rrbracket \implies target\ T \in StepConf\ (HA\ ST)\ (Conf\ ST)\ TS$ 
apply (unfold StepConf-def)
apply auto

```

done

lemma *HPT-StepConf-HAStates* [simp]:
 $\llbracket TS \in \text{HPT } ST; S \in \text{StepConf } (HA \ ST) \ (Conf \ ST) \ TS \rrbracket \implies S \in \text{HAStates } (HA \ ST)$
apply (unfold StepConf-def)
apply auto
apply (frule tranclD2)
apply auto
done

lemma *StepConf-Target-HAInitStates*:
 $\llbracket S \in \text{StepConf } (HA \ ST) \ (Conf \ ST) \ TS; S \notin \text{Target } TS; S \notin \text{Conf } ST \rrbracket \implies S \in \text{HAInitStates } (HA \ ST)$
apply (unfold StepConf-def)
apply auto
apply (frule tranclD2)
apply auto
done

lemma *InitSucState-StepConf*:
 $\llbracket TS \in \text{HPT } ST; S \notin \text{Target } TS; A \in \text{the } (CompFun \ (HA \ ST) \ S); S \notin \text{Conf } ST; S \in \text{StepConf } (HA \ ST) \ (Conf \ ST) \ TS \rrbracket \implies$
 $\text{InitState } A \in \text{StepConf } (HA \ ST) \ (Conf \ ST) \ TS$
apply (frule StepConf-HAStates [THEN subsetD, THEN CompFun-HAInitStates-HAStates])
apply fast+
apply (subst (asm) StepConf-def)
apply safe
apply (unfold StepConf-def)
apply (fast intro: HAAInitStates-InitState-trancl)
apply (frule trancl-Int-mem, fold ChiPlus-def)
apply (fast intro: ChiPlus-HAStates-Right [THEN HAAInitStates-InitState-trancl2])
done

lemma *InitSucState-Target-StepConf*:
 $\llbracket TS \in \text{HPT } ST; S \in \text{Target } TS; A \in \text{the } (CompFun \ (HA \ ST) \ S) \rrbracket \implies$
 $\text{InitState } A \in \text{StepConf } (HA \ ST) \ (Conf \ ST) \ TS$
apply (frule HPT-Target-HAStates2 [THEN CompFun-HAInitStates-HAStates])
apply fast+
apply (frule HPT-Target-HAStates2 [THEN CompFun-ChiRel])
apply (fast intro: InitState-States)+
apply (unfold StepConf-def)
apply auto
done

lemma *InitSucState-Conf-StepConf*:
 $\llbracket TS \in \text{HPT } ST; S \in \text{StepConf } (HA \ ST) \ (Conf \ ST) \ TS; S \notin \text{Target } TS; A \in \text{the } (CompFun \ (HA \ ST) \ S);$

$S \in \text{Conf } ST; S \in \text{ChiStar } (HA \ ST) \text{ “ (Source } TS) \parallel \implies$
 $\text{InitState } A \in \text{StepConf } (HA \ ST) (\text{Conf } ST) \ TS$
apply (frule Conf-HAStates2 [THEN CompFun-HAInitStates-HAStates])
apply fast
apply (subst (asm) StepConf-def)
apply safe
apply fast
apply (unfold StepConf-def)
apply (fast intro:HAInitStates-InitState-trancl)
apply (rename-tac T U V)
apply (frule trancl-Int-mem, fold ChiPlus-def, safe)
apply (subst (asm) Image-def, safe)
apply (rule-tac $x=U$ in bexI)
apply (simp only: ChiPlus-HAStates-Right [THEN HAAInitStates-InitState-trancl2])
apply fast
apply (subst (asm) Image-def, safe)
apply (rule-tac $x=U$ in bexI)
apply (simp only: ChiPlus-HAStates-Right [THEN HAAInitStates-InitState-trancl2])
apply fast
done

lemma SucState-Conf-StepConf:

$\parallel TS \in \text{HPT } ST; S \in \text{StepConf } (HA \ ST) (\text{Conf } ST) \ TS;$
 $S \notin \text{Target } TS; A \in \text{the } (\text{CompFun } (HA \ ST) \ S);$
 $S \in \text{Conf } ST; \text{States } A \cap \text{ChiStar } (HA \ ST) \text{ “ (Source } TS) = \{\} \parallel \implies$
 $\exists x. x \in \text{States } A \wedge x \in \text{StepConf } (HA \ ST) (\text{Conf } ST) \ TS$
apply (unfold StepConf-def)
apply (cut-tac $ST=ST$ in UniqueSucStates-Status)
apply (unfold UniqueSucStates-def)
apply (cut-tac $ST=ST$ in Conf-HAStates2)
apply fast
apply (fold HAStates-def)
apply (erule-tac $x=S$ in ballE)
apply (erule-tac $x=A$ in ballE)
apply simp
apply fast+
done

lemma SucState-Conf-Source-StepConf:

$\parallel TS \in \text{HPT } ST; S \in \text{StepConf } (HA \ ST) (\text{Conf } ST) \ TS;$
 $S \notin \text{Target } TS; A \in \text{the } (\text{CompFun } (HA \ ST) \ S);$
 $S \in \text{Conf } ST; \text{States } A \cap \text{ChiStar } (HA \ ST) \text{ “ (Source } TS) \neq \{\};$
 $S \notin \text{ChiStar } (HA \ ST) \text{ “ (Source } TS) \parallel \implies$
 $\exists x. x \in \text{States } A \wedge x \in \text{StepConf } (HA \ ST) (\text{Conf } ST) \ TS$
apply safe
apply (rename-tac T U)
apply (frule Conf-HAStates2 [THEN CompFun-ChiRel])
apply fast+
apply simp

```

apply (case-tac  $U=T$ )
apply simp
apply (rotate-tac  $-5$ )
apply (simp only:Source-def Target-def image-def)
apply safe
apply (rename-tac Source Trigger Guard Action Update Target)
apply (erule-tac  $x=Target$  in allE)
apply simp
apply (frule HPT-source-target2)
apply fast+
apply (rule HASStates-CompFun-SAs-mem)
apply (rule Conf-HASStates2)
apply fast+
apply (frule ChiStar-ChiPlus-noteq)
apply fast
apply (case-tac  $U=S$ )
apply (fast intro:ChiStar-Self ChiRel-ChiPlus-OneAncestor ChiPlus-ChiStar)+
done

```

```

lemma SucState-StepConf:
   $\llbracket TS \in HPT\ ST; S \in StepConf\ (HA\ ST)\ (Conf\ ST)\ TS;$ 
   $A \in the\ (CompFun\ (HA\ ST)\ S) \rrbracket \implies$ 
   $\exists x. x \in States\ A \wedge x \in StepConf\ (HA\ ST)\ (Conf\ ST)\ TS$ 
apply (case-tac  $S \in Target\ TS$ )
apply (fast intro: InitSucState-Target-StepConf InitState-States)
apply (case-tac  $S \in Conf\ ST$ )
prefer 2
apply (fast intro: InitSucState-StepConf InitState-States)
apply (case-tac  $S \in ChiStar\ (HA\ ST)$  “(Source TS)”)
apply (fast intro: InitSucState-Conf-StepConf InitState-States)
apply (case-tac  $States\ A \cap ChiStar\ (HA\ ST)$  “(Source TS) = {}”)
apply (fast intro: SucState-Conf-StepConf SucState-Conf-Source-StepConf)+
done

```

8.2.18 StepStatus

```

lemma StepStatus-expand:
   $Abs-status\ (HA\ ST, StepConf\ (HA\ ST)\ (Conf\ ST)\ TS,$ 
   $StepActEvent\ TS, U\ !!!\ (Value\ ST))$ 
   $= (StepStatus\ ST\ TS\ U)$ 
apply (unfold StepStatus-def Let-def)
apply (subst Rep-status-tuple)
apply auto
done

```

```

lemma UniqueSucState-Conf-Source-StepConf:
   $\llbracket TS \in HPT\ ST; S \in StepConf\ (HA\ ST)\ (Conf\ ST)\ TS; A \in SAs\ (HA\ ST);$ 
   $A \in the\ (CompFun\ (HA\ ST)\ S); T \in States\ A; U \in States\ A;$ 
   $T \in StepConf\ (HA\ ST)\ (Conf\ ST)\ TS; T \neq U; U \in Conf\ ST \rrbracket \implies$ 

```

$U \in \text{ChiStar } (HA \ ST) \text{ “ Source } TS$
apply (*frule-tac* $?S2.0=T$ **in** *StepConf-HAStates* [*THEN subsetD*, *THEN Comp-Fun-ChiRel*])
apply *fast+*
apply (*frule-tac* $?S2.0=U$ **in** *StepConf-HAStates* [*THEN subsetD*, *THEN Comp-Fun-ChiRel*])
apply *fast+*
apply (*frule-tac* $S=S$ **and** $T=U$ **in** *Conf-ChiRel*, *fast*)
apply (*case-tac* $S \in \text{ChiStar } (HA \ ST) \text{ “ Source } TS$)
apply (*fast intro*: *ChiRel-ChiStar-trans*)
apply (*case-tac* $U \in \text{Source } TS$)
apply *force*
apply (*unfold StepConf-def*)
apply *simp*
apply *safe*
apply (*fast intro*: *HPT-SAStates-Target-Source*) +
apply (*rename-tac* V)
apply (*case-tac* $V=S$)
apply (*frule-tac* $S=S$ **in** *HPT-Conf-Target-Source*, *fast+*)
apply (*fast intro*: *ChiStar-Image ChiRel-OneAncestor*) +
apply (*rename-tac* $V \ W$)
apply (*frule* *trancl-Int-mem*, *fold ChiPlus-def*, *safe*)
apply (*cut-tac* $ST=ST$ **and** $S=S$ **in** *HPT-Conf-Target-Source-ChiPlus*)
apply *fast+*
apply (*simp only*: *Image-def*, *safe*)
apply (*case-tac* $(V, T) \notin \text{ChiRel } (HA \ ST)$)
apply (*frule-tac* $S=V$ **and** $T=T$ **in** *ChiPlus-ChiRel-Ex*)
apply (*fast*, *safe*)
apply (*rename-tac* X)
apply (*case-tac* $X=S$)
apply (*rule-tac* $x=W$ **in** *bexI*)
prefer 4
apply (*case-tac* $V=S$)
prefer 2
apply *simp*
apply (*fast intro*: *ChiPlus-ChiRel ChiRel-ChiPlus-trans2 ChiRel-OneAncestor*) +
done

lemma *UniqueSucState-Target-StepConf*:

$\llbracket TS \in \text{HPT } ST; S \in \text{StepConf } (HA \ ST) \ (Conf \ ST) \ TS; A \in \text{SAs } (HA \ ST);$
 $A \in \text{the } (CompFun \ (HA \ ST) \ S); T \in \text{States } A; U \in \text{States } A;$
 $T \in \text{StepConf } (HA \ ST) \ (Conf \ ST) \ TS; T \neq U \rrbracket \implies$
 $U \notin \text{Target } TS$

apply *auto*
apply (*frule-tac* $ST=ST$ **in** *Target-StepConf*)
apply (*subst (asm)* (2) *StepConf-def*)
apply *simp*
apply *safe*
apply (*cut-tac* $TS=TS$ **and** $ST=ST$ **and** $S=S$ **and** $T=U$ **in** *UniqueSucState-Conf-Source-StepConf*)

apply *fast+*
apply (*simp add: OneState-HPT-Target*)
apply (*simp only: OneState-HPT-Target-ChiRel*)
apply (*rename-tac V W*)
apply (*frule-tac U=W and S=V and T=T in ChiRel-ChiPlus-trans2*)
apply (*frule trancl-Int-mem, fold ChiPlus-def, force*)
apply (*simp only: OneState-HPT-Target-ChiPlus*)
done

lemma *UniqueSucState-Target-ChiRel-StepConf*:

$\llbracket TS \in \text{HPT } ST; S \in \text{StepConf } (HA \ ST) \ (Conf \ ST) \ TS; A \in \text{SAs } (HA \ ST);$
 $A \in \text{the } (CompFun \ (HA \ ST) \ S); T \in \text{States } A; U \in \text{States } A;$
 $T \in \text{StepConf } (HA \ ST) \ (Conf \ ST) \ TS; T \neq U; (V, U) \in \text{ChiRel } (HA \ ST);$
 $U \in \text{HAINitStates } (HA \ ST) \rrbracket$
 $\implies V \notin \text{Target } TS$

apply *auto*
apply (*frule-tac A=HA ST and C=Conf ST in Target-ChiRel-HAINit-StepConf*)
apply *fast+*
apply (*subst (asm) (2) StepConf-def, safe*)
apply (*fast intro: UniqueSucState-Conf-Source-StepConf*)
apply (*simp only: OneState-HPT-Target-ChiRel*)
apply (*fast intro: OneHAINitState-SASates*)
apply (*frule trancl-Int-mem, fold ChiPlus-def*)
apply (*fast intro: OneHAINitState-SASates*)
done

lemma *UniqueSucState-Target-ChiPlus-StepConf [rule-format]*:

$\llbracket TS \in \text{HPT } ST; (S, T) \in \text{ChiRel } (HA \ ST); (S, U) \in \text{ChiRel } (HA \ ST);$
 $V \in \text{Target } TS; (V, W) \in \text{ChiRel } (HA \ ST); T \notin \text{ChiStar } (HA \ ST) \text{ “ Source}$
 $TS;$
 $(W, U) \in (\text{ChiRel } (HA \ ST) \cap \text{HAINitStates } (HA \ ST) \times \text{HAINitStates } (HA \ ST))^+;$
 $T \in \text{Conf } ST \rrbracket \implies (S, U) \in \text{ChiRel } (HA \ ST) \longrightarrow T=U$

apply (*frule-tac S=S and T=T in Conf-ChiRel*)
apply *fast*
apply (*rule-tac a=W and b=U and r=ChiRel (HA ST) $\cap$ HAINitStates (HA ST) $\times$ HAINitStates (HA ST) in trancl-induct*)
apply *safe*
apply (*rename-tac X*)
apply (*case-tac W=S*)
apply *simp*
prefer 2
apply (*simp add: ChiRel-OneAncestor*)
prefer 2
apply (*rename-tac X Y*)
apply (*case-tac X=S*)
apply *simp*
prefer 2
apply (*simp add: ChiRel-OneAncestor*)

```

prefer 2
apply (frule-tac  $a=V$  in ChiRel-HAStates)
apply (unfold HAStates-def)
apply (simp,safe)
apply (rename-tac  $Y$ )
apply (case-tac  $States\ Y \cap Source\ TS = \{\}$ )
apply (simp add:OneState-HPT-Target-Source)
apply (subst (asm) Int-def, safe)
apply (rename-tac  $Z$ )
apply (frule-tac  $S=V$  and  $T=S$  in Conf-ChiRel)
apply fast
apply (frule HPT-Conf-Target-Source)
prefer 2
apply fast
apply fast
apply (frule-tac  $S=Z$  and  $T=V$  in OneState-HPT-SA)
apply fast+
apply simp
apply (fast intro: ChiPlus-ChiRel ChiRel-ChiPlus-trans ChiPlus-ChiStar)
apply (simp add: Image-def)
apply (frule trancl-Int-mem, fold ChiPlus-def, simp, safe)
back
apply (frule-tac  $T=W$  and  $S=S$  in Conf-ChiPlus)
apply simp
apply (frule-tac  $S=V$  and  $T=W$  in Conf-ChiRel)
apply fast
apply (frule-tac  $a=V$  in ChiRel-HAStates)
apply (unfold HAStates-def)
apply (simp, safe)
apply (rename-tac  $Z$ )
apply (case-tac  $States\ Z \cap Source\ TS = \{\}$ )
apply (simp add:OneState-HPT-Target-Source)
apply (subst (asm) Int-def, safe)
apply (frule-tac  $S=V$  in HPT-Conf-Target-Source)
apply fast+
apply (rename-tac  $P$ )
apply (frule-tac  $S=P$  and  $T=V$  in OneState-HPT-SA)
apply fast+
apply (frule-tac  $U=V$  and  $S=W$  and  $T=S$  in ChiRel-ChiPlus-trans2)
apply fast+
apply (fast intro: ChiPlus-ChiRel ChiRel-ChiPlus-trans ChiPlus-ChiStar)
apply (case-tac  $T=S$ )
apply (simp add: ChiRel-OneAncestor)+
done

```

lemma *UniqueSucStates-SAStates-StepConf*:

$$\llbracket TS \in HPT\ ST; S \in StepConf\ (HA\ ST)\ (Conf\ ST)\ TS; A \in SAs\ (HA\ ST); \\ A \in the\ (CompFun\ (HA\ ST)\ S); T \in States\ A; U \in States\ A; \\ T \in StepConf\ (HA\ ST)\ (Conf\ ST)\ TS; T \neq U \rrbracket \implies$$

```

       $U \notin \text{StepConf } (HA \ ST) \ (Conf \ ST) \ TS$ 
apply (subst StepConf-def)
apply (simp, safe)
apply (rule UniqueSucState-Conf-Source-StepConf)
apply fast+
apply (frule-tac U=U in UniqueSucState-Target-StepConf)
apply fast+
apply (frule-tac U=U in UniqueSucState-Target-ChiRel-StepConf)
apply fast+
apply (rename-tac V W)
apply (frule trancl-Int-mem, fold ChiPlus-def)
apply (simp, safe)
apply (frule-tac ?S2.0=T in StepConf-HAStates [THEN subsetD, THEN Comp-
Fun-ChiRel])
apply fast+
apply (frule-tac ?S2.0=U in StepConf-HAStates [THEN subsetD, THEN Comp-
Fun-ChiRel])
apply fast+
apply (subst (asm) (2) StepConf-def)
apply (simp, safe)
apply (fast intro: UniqueSucState-Target-ChiPlus-StepConf)
apply (frule-tac U=W and T=U and S=T in OneState-HPT-Target-ChiPlus)
apply (fast intro: ChiPlus-ChiRel ChiRel-ChiPlus-trans2 OneHAINitState-SASates) +
apply (frule trancl-Int-mem, fold ChiPlus-def, simp, safe)
apply (fast intro: OneHAINitState-SASates)
done

```

lemma *UniqueSucStates-Ancestor-StepConf*:

```

   $\llbracket TS \in \text{HPT } ST; S \in \text{HAStates } (HA \ ST); SA \in \text{the } (CompFun \ (HA \ ST) \ S);$ 
   $T \in \text{States } SA; T \in \text{StepConf } (HA \ ST) \ (Conf \ ST) \ TS \rrbracket$ 
   $\implies S \in \text{StepConf } (HA \ ST) \ (Conf \ ST) \ TS$ 
apply (rule notnotD, rule notI)
apply (subst (asm) StepConf-def)
apply (simp, safe)
apply (frule CompFun-ChiRel, fast+)
apply (frule Conf-ChiRel, fast)
apply (frule ChiRel-ChiStar-Source-notmem, fast+)
apply (subst (asm) StepConf-def)
apply force
apply (case-tac States SA  $\cap$  Source TS = {})
apply (simp add: OneState-HPT-Target-Source)
apply (subst (asm) Int-def)
apply (simp, safe)
apply (rename-tac U)
apply (frule-tac ?S2.0=U in CompFun-ChiRel, fast+)
apply (frule Conf-ChiRel)
apply (frule HPT-Source-Conf, fast)
apply (case-tac S  $\in$  ChiStar (HA ST) “ Source TS)
apply (subst (asm) StepConf-def)

```



```

apply simp
apply (frule ChiRel-ChiStar-notmem)
apply fast+
apply (case-tac  $U=S$ )
apply (subst (asm) StepConf-def)
apply force
apply (subst (asm) StepConf-def)
apply force
apply (rename-tac  $U$ )
apply (case-tac  $U=S$ )
apply (subst (asm) StepConf-def)
apply force
apply (frule CompFun-ChiRel, fast+)
apply (simp add: ChiRel-OneAncestor)
apply (rename-tac  $U\ V$ )
apply (frule trancl-Int-mem, fold ChiPlus-def, simp, safe)
apply (frule tranclD2)
apply safe
apply (rename-tac  $W$ )
apply (case-tac  $W=S$ )
apply simp
prefer 2
apply (frule CompFun-ChiRel, fast+)
apply (simp only: ChiRel-OneAncestor)
apply (subst (asm) StepConf-def)
apply safe
apply (simp add: Image-def)
apply (erule-tac  $x=U$  in ballE)
apply (case-tac  $U=S$ )
apply fast
apply (simp add: rtrancl-eq-or-trancl)
apply fast
apply (simp add: Image-def)
apply (rename-tac  $W$ )
apply (erule-tac  $x=U$  in ballE)
apply (simp add: rtrancl-eq-or-trancl)
apply fast+
done

```

lemma *UniqueSucStates-StepConf*:

```

   $\llbracket TS \in HPT\ ST \rrbracket \implies$ 
     $UniqueSucStates\ (SAs\ (HA\ ST))\ (CompFun\ (HA\ ST))\ (StepConf\ (HA\ ST))$ 
  (Conf  $ST$ )  $TS$ )
apply (unfold UniqueSucStates-def)
apply auto
apply (simp only: SucState-StepConf)
apply (rule notnotD, rule notI)
apply (frule UniqueSucStates-SASates-StepConf)
apply fast

```

```

prefer 2
apply fast
apply (rule HASStates-CompFun-SAs-mem)
prefer 2
apply fast
apply (simp only: HASStates-def, fast)
apply fast+
back
apply (frule UniqueSucStates-Ancestor-StepConf)
prefer 2
apply fast
apply (simp only: HASStates-def, fast)
apply fast+
done

lemma Status-Step:
   $\llbracket TS \in \text{HPT } ST; U \in \text{ResolveRacing } TS \rrbracket \implies$ 
   $(HA \text{ } ST, \text{StepConf } (HA \text{ } ST) (\text{Conf } ST) \text{ } TS, \text{StepActEvent } TS, U \text{ } !!! \text{ } (Value$ 
   $ST)) \in \text{status}$ 
apply (unfold status-def Status-def)
apply auto
apply (frule StepActEvent-HAEvents)
apply blast
apply (unfold IsConfSet-def)
apply (rule conjI, frule StepConf-HASStates, unfold HASStates-def, assumption)
apply (rule conjI, rule RootExSem-StepConf, assumption)
apply (rule UniqueSucStates-StepConf, assumption)
done

```

8.3 Meta Theorem: Preservation for Statecharts

```

lemma IsConfSet-StepConf:
   $TS \in \text{HPT } ST \implies \text{IsConfSet } (SAs \text{ } (HA \text{ } ST)) (\text{CompFun } (HA \text{ } ST))$ 
   $(\text{StepConf } (HA \text{ } ST) (\text{Conf } ST) \text{ } TS)$ 
apply (unfold IsConfSet-def)
apply auto
apply (frule StepConf-HASStates)
apply (unfold HASStates-def, fast)
apply (rule RootExSem-StepConf, assumption)
apply (rule UniqueSucStates-StepConf, assumption)
done

```

```

lemma HA-StepStatus-HPT-ResolveRacing [simp]:
   $\llbracket TS \in \text{HPT } ST; U \in \text{ResolveRacing } TS \rrbracket \implies$ 
   $HA (\text{StepStatus } ST \text{ } TS \text{ } U) = HA \text{ } ST$ 
apply (subst StepStatus-expand [THEN sym])
apply (subst HA-def)
apply auto
apply (subst Abs-status-inverse)

```

```

apply auto
apply (rule Status-Step)
apply auto
done

end

```

9 Kripke Structures and CTL

```

theory Kripke
imports Main
begin

```

definition

```

Kripke :: ['s set,
            's set,
            ('s * 's) set,
            ('s  $\rightarrow$  'a set)]
        => bool where

```

```

Kripke S S0 R L =
    (S0  $\subseteq$  S  $\wedge$ 
     R  $\leq$  S  $\times$  S  $\wedge$ 
     (Domain R) = S  $\wedge$ 
     (dom L) = S)

```

lemma *Kripke-EmptySet*:

```

({@x. True}, {@x. True}, {@x. True, @x. True}, Map.empty(@x. True  $\mapsto$  {@x.
True}))  $\in$ 
  {(S, S0, R, L) | S S0 R L. Kripke S S0 R L}
by (unfold Kripke-def Domain-unfold, auto)

```

definition

```

kripke =
  {(S, S0, T, L) |
   (S::('s set))
   (S0::('s set))
   (T::(('s * 's) set))
   (L::('s  $\rightarrow$  ('a set))).
   Kripke S S0 T L}

```

typedef ('s, 'a) *kripke* =

```

  kripke :: ('s set * 's set * ('s * 's) set * ('s  $\rightarrow$  'a set)) set
unfolding kripke-def
apply (rule exI)
apply (rule Kripke-EmptySet)
done

```

definition

$Statutes :: ('s, 'a) \text{ kripke} \Rightarrow 's \text{ set}$ **where**
 $Statutes = fst \ o \ Rep\text{-kripke}$

definition

$InitStatutes :: ('s, 'a) \text{ kripke} \Rightarrow 's \text{ set}$ **where**
 $InitStatutes == fst \ o \ snd \ o \ Rep\text{-kripke}$

definition

$StepRel :: ('s, 'a) \text{ kripke} \Rightarrow ('s * 's) \text{ set}$ **where**
 $StepRel == fst \ o \ snd \ o \ snd \ o \ Rep\text{-kripke}$

definition

$LabelFun :: ('s, 'a) \text{ kripke} \Rightarrow ('s \rightarrow 'a \text{ set})$ **where**
 $LabelFun == snd \ o \ snd \ o \ snd \ o \ Rep\text{-kripke}$

definition

$Paths :: [('s, 'a) \text{ kripke}, 's] \Rightarrow$
 $(nat \Rightarrow 's) \text{ set}$ **where**
 $Paths \ M \ S == \{ p \cdot S = p \ (0::nat) \wedge (\forall i. (p \ i, p \ (i+1)) \in (StepRel \ M)) \}$

datatype $('s, 'a) \text{ ctl} =$ $Atom \ 'a$

$| \ AND \ ('s, 'a) \text{ ctl} \ ('s, 'a) \text{ ctl}$
 $| \ OR \ ('s, 'a) \text{ ctl} \ ('s, 'a) \text{ ctl}$
 $| \ IMPLIES \ ('s, 'a) \text{ ctl} \ ('s, 'a) \text{ ctl}$
 $| \ CAX \ ('s, 'a) \text{ ctl}$
 $| \ AF \ ('s, 'a) \text{ ctl}$
 $| \ AG \ ('s, 'a) \text{ ctl}$
 $| \ AU \ ('s, 'a) \text{ ctl} \ ('s, 'a) \text{ ctl}$
 $| \ AR \ ('s, 'a) \text{ ctl} \ ('s, 'a) \text{ ctl}$

primrec

$eval\text{-ctl} :: [('s, 'a) \text{ kripke}, 's, ('s, 'a) \text{ ctl}] \Rightarrow bool \ (\prec, - \models c = \rightarrow [92, 91, 90] 90)$
where
 $(M, S \models c = (Atom \ P)) = (P \in the \ ((LabelFun \ M) \ S))$
 $| \ (M, S \models c = (AND \ F1 \ F2)) = ((M, S \models c = F1) \wedge (M, S \models c = F2))$
 $| \ (M, S \models c = (OR \ F1 \ F2)) = ((M, S \models c = F1) \vee (M, S \models c = F2))$
 $| \ (M, S \models c = (IMPLIES \ F1 \ F2)) = ((M, S \models c = F1) \longrightarrow (M, S \models c = F2))$
 $| \ (M, S \models c = (CAX \ F)) = (\forall \ T. (S, T) \in (StepRel \ M) \longrightarrow (M, T \models c = F))$
 $| \ (M, S \models c = (AF \ F)) = (\forall \ P \in Paths \ M \ S. \exists \ i. (M, (P \ i) \models c = F))$
 $| \ (M, S \models c = (AG \ F)) = (\forall \ P \in Paths \ M \ S. \forall \ i. (M, (P \ i) \models c = F))$
 $| \ (M, S \models c = (AU \ F \ G)) = (\forall \ P \in Paths \ M \ S.$
 $\quad \exists \ i. (M, (P \ i) \models c = G) \wedge$
 $\quad (\forall \ j. j < i \longrightarrow (M, (P \ j) \models c = F)))$
 $| \ (M, S \models c = (AR \ F \ G)) = (\forall \ P \in Paths \ M \ S.$
 $\quad \forall \ i. (M, (P \ i) \models c = G) \vee$
 $\quad (\exists \ j. j < i \wedge (M, (P \ j) \models c = F)))$

end

10 Kripke Structures as Hierarchical Automata

theory *HAKripke*

imports *HASem Kripke*

begin

type-synonym $(s,e,d)hakripke = ((s,e,d)status, (s,e,d)atomar)kripke$

type-synonym $(s,e,d)hactl = ((s,e,d)status, (s,e,d)atomar)ctl$

definition

LabelFunSem :: $(s,e,d)hierauto$
 $=> ((s,e,d)status \rightarrow (((s,e,d) atomar) set))$ **where**
LabelFunSem *a* = $(\lambda ST.$
 (if (*HA* *ST* = *a*) then
 (let
In-preds = $(\lambda s. (IN\ s)) \text{ ' } (Conf\ ST);$
En-preds = $(\lambda e. (EN\ e)) \text{ ' } (Events\ ST);$
Val-preds = $\{ x . (\exists P. (x = (VAL\ P)) \wedge P\ (Value\ ST)) \}$
 in
 Some (*In-preds* $\cup$ *En-preds* $\cup$ *Val-preds* $\cup$ {*atomar.TRUE*})
 else
 None))

definition

HA2Kripke :: $(s,e,d)hierauto \Rightarrow (s,e,d)hakripke$ **where**
HA2Kripke *a* =
 Abs-kripke ($\{ST. HA\ ST = a\},$
 {*InitStatus* *a*},
 StepRelSem *a*,
 LabelFunSem *a*)

definition

eval-ctl-HA :: $[(s,e,d)hierauto, (s,e,d)hactl]$
 $\Rightarrow bool \ (\prec \cdot \models_H \rightarrow [92,91]90)$ **where**

eval-ctl-HA *a f* = $((HA2Kripke\ a), (InitStatus\ a) \models_c f)$

lemma *Kripke-HA* [*simp*]:

Kripke $\{ST. HA\ ST = a\} \{InitStatus\ a\} (StepRelSem\ a) (LabelFunSem\ a)$

apply (*unfold Kripke-def*)

apply *auto*

apply (*unfold StepRelSem-def*)

apply *auto*

apply (*unfold LabelFunSem-def Let-def If-def dom-def*)

apply *auto*

prefer 2

apply (*rename-tac ST S*)

```

apply (case-tac HA ST = a)
apply auto
apply (rename-tac ST)
apply (case-tac HPT ST = {})
apply auto
apply (rename-tac TSS)
apply (erule-tac x=StepStatus ST TSS (@ u. u : ResolveRacing TSS) in allE)
apply (erule-tac x=TSS in ballE)
apply auto
done

```

```

lemma LabelFun-LabelFunSem [simp]:
  (LabelFun (HA2Kripke a)) = (LabelFunSem a)
apply (unfold HA2Kripke-def LabelFun-def)
apply auto
apply (subst Abs-kripke-inverse)
apply auto
apply (unfold kripke-def)
apply auto
done

```

```

lemma InitStatuses-InitStatus [simp]:
  (InitStatuses (HA2Kripke a)) = {(InitStatus a)}
apply (unfold HA2Kripke-def InitStatuses-def)
apply simp
apply (subst Abs-kripke-inverse)
apply (unfold kripke-def)
apply auto
done

```

```

lemma Statuses-StatusesOfHA [simp]:
  (Statuses (HA2Kripke a)) = {ST. HA ST = a}
apply (unfold HA2Kripke-def Statuses-def)
apply simp
apply (subst Abs-kripke-inverse)
apply (unfold kripke-def)
apply auto
done

```

```

lemma StepRel-StepRelSem [simp]:
  (StepRel (HA2Kripke a)) = (StepRelSem a)
apply (unfold HA2Kripke-def StepRel-def)
apply simp
apply (subst Abs-kripke-inverse)
apply (unfold kripke-def)
apply auto
done

```

```

lemma TRUE-LabelFunSem [simp]:

```

$atomar.TRUE \in the (LabelFunSem (HA ST) ST)$
apply (unfold LabelFunSem-def Let-def)
apply auto
done

lemma FALSE-LabelFunSem [simp]:
 $atomar.FALSE \notin the (LabelFunSem (HA ST) ST)$
apply (unfold LabelFunSem-def Let-def)
apply auto
done

lemma Conf-LabelFunSem [simp]:
 $((IN S) \in the (LabelFunSem (HA ST) ST)) = (S \in (Conf ST))$
apply (unfold LabelFunSem-def Let-def)
apply auto
done

lemma Events-LabelFunSem [simp]:
 $((EN S) \in the (LabelFunSem (HA ST) ST)) = (S \in (Events ST))$
apply (unfold LabelFunSem-def Let-def)
apply auto
done

lemma Value-LabelFunSem [simp]:
 $((VAL P) \in the (LabelFunSem (HA ST) ST)) = (P (Value ST))$
apply (unfold LabelFunSem-def Let-def)
apply auto
done

lemma AtomTRUE-EvalCTLHA [simp]:
 $a \models_H (Atom (atomar.TRUE))$
apply (unfold eval-ctl-HA-def)
apply auto
apply (subst HA-InitStatus [THEN sym])
apply (rule TRUE-LabelFunSem)
done

lemma AtomFalse-EvalCTLHA [simp]:
 $\neg a \models_H (Atom (atomar.FALSE))$
apply (unfold eval-ctl-HA-def)
apply auto
apply (subst (asm) HA-InitStatus [THEN sym])
apply (simp only: FALSE-LabelFunSem)
done

lemma Events-InitStatus-EvalCTLHA [simp]:
 $(a \models_H (Atom (EN S))) = (S \in (Events (InitStatus a)))$
apply (unfold eval-ctl-HA-def)
apply simp

```

apply (subst HA-InitStatus [THEN sym])
apply (rule Events-LabelFunSem)
done

lemma Conf-InitStatus-EvalCTLHA [simp]:
  ( $a \models H = (\text{Atom } (IN\ S))) = (S \in (\text{Conf } (\text{InitStatus } a)))$ )
apply (unfold eval-ctl-HA-def)
apply simp
apply (subst HA-InitStatus [THEN sym])
apply (subst Conf-LabelFunSem)
apply simp
done

lemma HAInitValue-EvalCTLHA [simp]:
  ( $a \models H = (\text{Atom } (VAL\ P))) = (P\ (\text{HAInitValue } a))$ )
apply (unfold eval-ctl-HA-def)
apply simp
apply (subst HA-InitStatus [THEN sym])
apply (subst Value-LabelFunSem)
apply auto
done

end

```

11 Constructing Hierarchical Automata

```

theory HAOps
imports HA
begin

```

11.1 Constructing a Composition Function for a PseudoHA

definition

EmptyMap :: 's set => ('s $\rightarrow$ (('s, 'e, 'd) seqauto) set) **where**
EmptyMap *S* = ($\lambda\ a\ .\ \text{if } a \in S \text{ then Some } \{\} \text{ else None}$)

lemma *EmptyMap-dom* [simp]:

$\text{dom } (\text{EmptyMap } S) = S$

by (unfold *dom-def EmptyMap-def*, *auto*)

lemma *EmptyMap-ran* [simp]:

$S \neq \{\} \implies \text{ran } (\text{EmptyMap } S) = \{\{\}\}$

by (unfold *ran-def EmptyMap-def*, *auto*)

lemma *EmptyMap-the* [simp]:

$x \in S \implies \text{the } ((\text{EmptyMap } S) x) = \{\}$

by (unfold *ran-def EmptyMap-def*, *auto*)

lemma *EmptyMap-ran-override*:


```

  [  $S \neq \{\}$ ;  $(S \cap (\text{dom } G)) = \{\}$  ]  $\implies$ 
   $\text{ran } (G ++ \text{EmptyMap } S) = \text{insert } \{\} (\text{ran } G)$ 
apply (subst ran-override)
apply (simp add: Int-commute)
apply simp
done

```

```

lemma EmptyMap-Union-ran-override:
  [  $S \neq \{\}$ ;
     $S \cap \text{dom } G = \{\}$  ]  $\implies$ 
   $(\text{Union } (\text{ran } (G ++ (\text{EmptyMap } S)))) = (\text{Union } (\text{ran } G))$ 
apply (subst EmptyMap-ran-override)
apply auto
done

```

```

lemma EmptyMap-Union-ran-override2:
  [  $S \neq \{\}$ ;  $S \cap \text{dom } G1 = \{\}$ ;
     $\text{dom } G1 \cap \text{dom } G2 = \{\}$  ]  $\implies$ 
   $\bigcup (\text{ran } (G1 ++ \text{EmptyMap } S ++ G2)) = (\bigcup (\text{ran } G1 \cup \text{ran } G2))$ 
apply (unfold Union-eq UNION-eq EmptyMap-def Int-def ran-def)
apply (simp add: map-add-Some-iff)
apply (unfold dom-def)
apply simp
apply (rule equalityI)
apply (rule subsetI)
apply simp
apply fast
apply (rule subsetI)
apply (rename-tac t)
apply simp
apply (erule bexE)
apply (rename-tac U)
apply simp
apply (erule disjE)
apply (erule exE)
apply (rename-tac v)
apply (rule-tac  $x=U$  in exI)
apply simp
apply (rule-tac  $x=v$  in exI)
apply auto
done

```

```

lemma EmptyMap-Root [simp]:
   $\text{Root } \{SA\} (\text{EmptyMap } (\text{States } SA)) = SA$ 
by (unfold Root-def, auto)

```

```

lemma EmptyMap-RootEx [simp]:
   $\text{RootEx } \{SA\} (\text{EmptyMap } (\text{States } SA))$ 
by (unfold RootEx-def, auto)

```

lemma *EmptyMap-OneAncestor* [simp]:
OneAncestor {SA} (*EmptyMap* (*States* SA))
by (*unfold OneAncestor-def*, *auto*)

lemma *EmptyMap-NoCycles* [simp]:
NoCycles {SA} (*EmptyMap* (*States* SA))
by (*unfold NoCycles-def EmptyMap-def*, *auto*)

lemma *EmptyMap-IsCompFun* [simp]:
IsCompFun {SA} (*EmptyMap* (*States* SA))
by (*unfold IsCompFun-def*, *auto*)

lemma *EmptyMap-hierauto* [simp]:
(*D*, {SA}, *SAEvents* SA, *EmptyMap* (*States* SA)) $\in$ *hierauto*
by (*unfold hierauto-def HierAuto-def*, *auto*)

11.2 Extending a Composition Function by a SA

definition

FAddSA :: [(*'s* $\rightarrow$ ((*'s*, *'e*, *'d*)seqauto) set), *'s* * (*'s*, *'e*, *'d*)seqauto]
 $\Rightarrow$ (*'s* $\rightarrow$ ((*'s*, *'e*, *'d*)seqauto) set)
($\langle$ (- [*f*+] / -) $\rangle$ [10,11]10) **where**
FAddSA *G* *SSA* = (let (*S*, *SA*) = *SSA*
in
if ((*S* $\in$ dom *G*) $\wedge$ (*S* $\notin$ *States* SA)) then
(*G* ++ (*Map.empty*(*S* $\mapsto$ (*insert* SA (*the* (*G* *S*))))))
++ *EmptyMap* (*States* SA))
else *G*))

lemma *FAddSA-dom* [simp]:
(*S* $\notin$ (dom (*A*::(*'a* $\Rightarrow$ (*'a*, *'c*, *'d*)seqauto set option)))) $\Rightarrow$
((*A* [*f*+] (*S*, (*SA*::(*'a*, *'c*, *'d*)seqauto))) = *A*)
by (*unfold FAddSA-def Let-def*, *auto*)

lemma *FAddSA-States* [simp]:
(*S* $\in$ (*States* (*SA*::(*'a*, *'c*, *'d*)seqauto))) $\Rightarrow$
(((*A*::(*'a* $\Rightarrow$ (*'a*, *'c*, *'d*)seqauto set option)) [*f*+] (*S*, *SA*)) = *A*)
by (*unfold FAddSA-def Let-def*, *auto*)

lemma *FAddSA-dom-insert* [simp]:
 $\llbracket S \in$ (dom *A*); *S* $\notin$ *States* SA $\rrbracket \Rightarrow$
(((*A* [*f*+] (*S*, *SA*)) *S*) = *Some* (*insert* SA (*the* (*A* *S*))))
by (*unfold FAddSA-def Let-def restrict-def*, *auto*)

lemma *FAddSA-States-neq* [simp]:
 $\llbracket S' \notin$ *States* (*SA*::(*'a*, *'c*, *'d*)seqauto); *S* $\neq$ *S'* $\rrbracket \Rightarrow$
((((*A*::(*'a* $\Rightarrow$ (*'a*, *'c*, *'d*)seqauto set option)) [*f*+] (*S*, *SA*)) *S'*) = (*A* *S'*))
apply (*case-tac* *S* $\in$ dom *A*)

```

apply (case-tac  $S \in \text{States } SA$ )
apply auto
apply (case-tac  $S' \in \text{dom } A$ )
apply (unfold FAddSA-def Let-def)
apply auto
apply (simp add: dom-None)
done

```

```

lemma FAddSA-dom-emptyset [simp]:
   $\llbracket S \in (\text{dom } A); S \notin \text{States } SA; S' \in \text{States } (SA::('a,'c,'d)\text{seqauto}) \rrbracket \implies$ 
   $(((((A::('a \Rightarrow ('a,'c,'d)\text{seqauto set option}))) [f+] (S,SA)) S') = (\text{Some } \{\}))$ 
apply (unfold FAddSA-def Let-def)
apply auto
apply (unfold EmptyMap-def)
apply auto
done

```

```

lemma FAddSA-dom-dom-States [simp]:
   $\llbracket S \in (\text{dom } F); S \notin \text{States } SA \rrbracket \implies$ 
   $(\text{dom } ((F::('a \rightarrow (('a,'b,'d)\text{seqauto set})) [f+] (S, SA))) =$ 
   $((\text{dom } F) \cup (\text{States } (SA::('a,'b,'d)\text{seqauto}))))$ 
by (unfold FAddSA-def Let-def, auto)

```

```

lemma FAddSA-dom-dom [simp]:
   $S \notin (\text{dom } F) \implies$ 
   $(\text{dom } ((F::('a \rightarrow (('a,'b,'d)\text{seqauto set})) [f+] (S, (SA::('a,'b,'d)\text{seqauto})))) = (\text{dom } F)$ 
by (unfold FAddSA-def Let-def, auto)

```

```

lemma FAddSA-States-dom [simp]:
   $S \in (\text{States } SA) \implies$ 
   $(\text{dom } ((F::('a \rightarrow (('a,'b,'d)\text{seqauto set})) [f+] (S, (SA::('a,'b,'d)\text{seqauto})))) = (\text{dom } F)$ 
by (unfold FAddSA-def Let-def, auto)

```

```

lemma FAddSA-dom-insert-dom-disjunct [simp]:
   $\llbracket S \in \text{dom } G; \text{States } SA \cap \text{dom } G = \{\} \rrbracket \implies ((G [f+] (S,SA)) S) = \text{Some}$ 
   $(\text{insert } SA (\text{the } (G S)))$ 
apply (rule FAddSA-dom-insert)
apply auto
done

```

```

lemma FAddSA-Union-ran:
   $\llbracket S \in \text{dom } G; (\text{States } SA) \cap (\text{dom } G) = \{\} \rrbracket \implies$ 
   $(\bigcup (\text{ran } (G [f+] (S,SA)))) = (\text{insert } SA (\bigcup (\text{ran } G)))$ 
apply (unfold FAddSA-def Let-def)
apply simp
apply (rule conjI)
prefer 2

```

```

apply (rule impI)
apply (unfold Int-def)
apply simp
apply (fold Int-def)
apply (rule impI)
apply (subst EmptyMap-Union-ran-override)
apply auto
done

```

```

lemma FAddSA-Union-ran2:
   $\llbracket S \in \text{dom } G1; (\text{States } SA) \cap (\text{dom } G1) = \{\}; (\text{dom } G1 \cap \text{dom } G2) = \{\} \rrbracket \implies$ 
   $(\bigcup (\text{ran } ((G1 [f+] (S,SA)) ++ G2))) = (\text{insert } SA (\bigcup ((\text{ran } G1) \cup (\text{ran } G2))))$ 
apply (unfold FAddSA-def Let-def)
apply (simp (no-asm-simp))
apply (rule conjI)
apply (rule impI)
apply (subst EmptyMap-Union-ran-override2)
apply simp
apply simp
apply simp
apply fast
apply (subst Union-Un-distrib)
apply (subst Union-ran-override2)
apply auto
done

```

```

lemma FAddSA-ran:
   $\llbracket \forall T \in \text{dom } G . T \neq S \longrightarrow (\text{the } (G T) \cap \text{the } (G S)) = \{\};$ 
   $S \in \text{dom } G; (\text{States } SA) \cap (\text{dom } G) = \{\} \rrbracket \implies$ 
   $\text{ran } (G [f+] (S,SA)) = \text{insert } \{\} (\text{insert } (\text{insert } SA (\text{the } (G S))) (\text{ran } G - \{\text{the } (G S)\}))$ 
apply (unfold FAddSA-def Let-def)
apply simp
apply (rule conjI)
apply (rule impI)+
prefer 2
apply fast
apply (simp add: EmptyMap-ran-override)
apply (unfold ran-def)
apply auto
apply (rename-tac Y X a xa xb)
apply (erule-tac x=a in allE)
apply simp
apply (erule-tac x=a in allE)
apply simp
done

```

```

lemma FAddSA-RootEx-def:
   $\llbracket S \in \text{dom } G; (\text{States } SA) \cap (\text{dom } G) = \{\} \rrbracket \implies$ 

```

$RootEx\ F\ (G\ [f+]\ (S,SA)) = (\exists! A . A \in F \wedge A \notin insert\ SA\ (\bigcup\ (ran\ G)))$
apply (*unfold RootEx-def*)
apply (*simp only: FAddSA-Union-ran Int-commute*)
done

lemma *FAddSA-RootEx*:
 $\llbracket \bigcup\ (ran\ G) = F - \{Root\ F\ G\};$
 $dom\ G = \bigcup\ (States\ 'F);$
 $(dom\ G \cap States\ SA) = \{\}; S \in dom\ G;$
 $RootEx\ F\ G \rrbracket \implies RootEx\ (insert\ SA\ F)\ (G\ [f+]\ (S,SA))$
apply (*simp add: FAddSA-RootEx-def Int-commute cong: rev-conj-cong*)
apply (*auto cong: conj-cong*)
done

lemma *FAddSA-Root-def*:
 $\llbracket S \in dom\ G; (States\ SA) \cap (dom\ G) = \{\} \rrbracket \implies$
 $(Root\ F\ (G\ [f+]\ (S,SA)) = (@\ A . A \in F \wedge A \notin insert\ SA\ (\bigcup\ (ran\ G))))$
apply (*unfold Root-def*)
apply (*simp only: FAddSA-Union-ran Int-commute*)
done

lemma *FAddSA-RootEx-Root*:
 $\llbracket Union\ (ran\ G) = F - \{Root\ F\ G\};$
 $\bigcup\ (States\ 'F) = dom\ G;$
 $(dom\ G \cap States\ SA) = \{\}; S \in dom\ G;$
 $RootEx\ F\ G \rrbracket \implies (Root\ (insert\ SA\ F)\ (G\ [f+]\ (S,SA))) = (Root\ F\ G)$
apply (*simp add: FAddSA-Root-def Int-commute cong: rev-conj-cong*)
apply (*simp cong: conj-cong*)
done

lemma *FAddSA-OneAncestor*:
 $\llbracket \bigcup\ (ran\ G) = F - \{Root\ F\ G\};$
 $(dom\ G \cap States\ SA) = \{\}; S \in dom\ G;$
 $\bigcup\ (States\ 'F) = dom\ G; RootEx\ F\ G;$
 $OneAncestor\ F\ G \rrbracket \implies OneAncestor\ (insert\ SA\ F)\ (G\ [f+]\ (S,SA))$
apply (*subst OneAncestor-def*)
apply *simp*
apply (*rule ballI*)
apply (*rename-tac SAA*)
apply (*case-tac SA = SAA*)
apply (*rule-tac a=S in ex1I*)
apply (*rule conjI*)
apply *simp*
apply *fast*
apply (*subst FAddSA-dom-insert*)
apply *simp*
apply (*simp add: Int-def*)
apply *simp*
apply (*rename-tac T*)

```

apply (erule conjE bexE exE disjE)+
apply (rename-tac SAAA)
apply simp
apply (erule conjE)
apply (subst not-not [THEN sym])
apply (rule notI)
apply (case-tac T ∈ States SAA)
apply blast
apply (drule-tac A=G and S=S and SA=SAA in FAddSA-States-neg)
apply fast
apply simp
apply (case-tac SAA ∉ Union (ran G))
apply (frule ran-dom-the)
prefer 2
apply fast
apply blast
apply simp
apply (erule conjE)
apply (simp add: States-Int-not-mem)
apply (unfold OneAncestor-def)
apply (drule-tac G=G and S=S and SA=SA in FAddSA-RootEx-Root)
apply simp
apply simp
apply simp
apply simp
apply (erule-tac x=SAA in ballE)
prefer 2
apply simp
apply simp
apply (erule conjE bexE ex1E exE disjE)+
apply (rename-tac T SAAA)
apply (rule-tac a=T in ex1I)
apply (rule conjI)
apply fast
apply (case-tac T = S)
apply simp
apply (case-tac S ∉ States SA)
apply simp
apply simp
apply (subst FAddSA-States-neg)
apply blast
apply (rule not-sym)
apply simp
apply simp
apply (rename-tac U)
apply simp
apply (erule conjE bexE)+
apply (rename-tac SAAAA)
apply simp

```

```

apply (erule conjE disjE)+
apply (frule FAddSA-dom-emptyset)
prefer 2
apply fast
back
back
apply simp
apply blast
apply simp
apply (erule-tac x=U in allE)
apply (erule impE)
prefer 2
apply simp
apply (rule conjI)
apply fast
apply (case-tac S ≠ U)
apply (subgoal-tac U ∉ States SA)
apply (drule-tac A=G in FAddSA-States-neg)
apply fast
apply simp
apply blast
apply (drule-tac A=G and SA=SA in FAddSA-dom-insert)
apply simp
apply blast
apply auto
done

```

```

lemma FAddSA-NoCycles:
  
$$\llbracket (States\ SA \cap dom\ G) = \{\};\ S \in dom\ G;\$$


$$dom\ G = \bigcup (States\ 'F);\ NoCycles\ F\ G \rrbracket \implies$$


$$NoCycles\ (insert\ SA\ F)\ (G\ [f+]\ (S,SA))$$

apply (unfold NoCycles-def)
apply (rule ballI impI)+
apply (rename-tac SAA)
apply (case-tac ∃ s ∈ SAA. s ∈ States SA)
apply simp
apply (erule bexE)+
apply (rename-tac SAAA T)
apply (rule-tac x=T in bexI)
apply simp
apply (subst FAddSA-dom-emptyset)
apply simp
apply fast
apply blast
apply simp
apply simp
apply simp
apply simp
apply (erule-tac x=SAA in ballE)

```

```

prefer 2
apply simp
apply auto[1]
apply (unfold UNION-eq Pow-def)
apply simp
apply (case-tac SAA = {})
apply fast
apply simp
apply (erule bexE)+
apply (rename-tac SAAA T)
apply (rule-tac x=T in bexI)
prefer 2
apply simp
apply (case-tac T=S)
apply simp
apply (subst FAddSA-dom-insert)
apply auto
done

lemma FAddSA-IsCompFun:
   $\llbracket (States\ SA \cap (\bigcup (States\ 'F))) = \{\};$ 
   $S \in (\bigcup (States\ 'F));$ 
   $IsCompFun\ F\ G \rrbracket \implies IsCompFun\ (insert\ SA\ F)\ (G\ [f+]\ (S,SA))$ 
apply (unfold IsCompFun-def)
apply (erule conjE)+
apply (simp add: Int-commute FAddSA-RootEx-Root FAddSA-RootEx FAddSA-OneAncestor
FAddSA-NoCycles)
apply (rule conjI)
apply (subst FAddSA-dom-dom-States)
apply simp
apply blast
apply (simp add: Un-commute)
apply (simp add: FAddSA-Union-ran)
apply (case-tac SA = Root F G)
prefer 2
apply blast
apply (subgoal-tac States (Root F G)  $\subseteq \bigcup (States\ 'F)$ )
apply simp
apply (frule subset-lemma)
apply auto
done

lemma FAddSA-HierAuto:
   $\llbracket (States\ SA \cap (\bigcup (States\ 'F))) = \{\};$ 
   $S \in (\bigcup (States\ 'F));$ 
   $HierAuto\ D\ F\ E\ G \rrbracket \implies HierAuto\ D\ (insert\ SA\ F)\ (E \cup SAEvents\ SA)\ (G$ 
   $[f+]\ (S,SA))$ 
apply (unfold HierAuto-def)
apply auto

```


apply (*simp add: MutuallyDistinct-Insert*)
apply (*rule FAddSA-IsCompFun*)
apply *auto*
done

lemma *FAddSA-HierAuto-insert* [*simp*]:

$$\llbracket (States\ SA \cap HStates\ HA) = \{\};$$

$$S \in HStates\ HA \rrbracket \implies$$

$$HierAuto\ (HInitValue\ HA)$$

$$(insert\ SA\ (SAs\ HA))$$

$$(HAEvents\ HA \cup SAEvents\ SA)$$

$$(CompFun\ HA\ [f+]\ (S, SA))$$

apply (*unfold HStates-def*)
apply (*rule FAddSA-HierAuto*)
apply *auto*
done

11.3 Constructing a PseudoHA

definition

$PseudoHA :: [(s, e, d)seqauto, d\ data] \Rightarrow (s, e, d)hierauto$ **where**
 $PseudoHA\ SA\ D = Abs-hierauto(D, \{SA\}, SAEvents\ SA, EmptyMap\ (States\ SA))$

lemma *PseudoHA-SAs* [*simp*]:
 $SAs\ (PseudoHA\ SA\ D) = \{SA\}$
by (*unfold PseudoHA-def SAs-def, simp add: Abs-hierauto-inverse*)

lemma *PseudoHA-Events* [*simp*]:
 $HAEvents\ (PseudoHA\ SA\ D) = SAEvents\ SA$
by (*unfold PseudoHA-def HAEvents-def, simp add: Abs-hierauto-inverse*)

lemma *PseudoHA-CompFun* [*simp*]:
 $CompFun\ (PseudoHA\ SA\ D) = EmptyMap\ (States\ SA)$
by (*unfold PseudoHA-def CompFun-def, simp add: Abs-hierauto-inverse*)

lemma *PseudoHA-HStates* [*simp*]:
 $HStates\ (PseudoHA\ SA\ D) = (States\ SA)$
by (*unfold HStates-def, auto*)

lemma *PseudoHA-HInitValue* [*simp*]:
 $(HInitValue\ (PseudoHA\ SA\ D)) = D$
by (*unfold PseudoHA-def Let-def HInitValue-def, simp add: Abs-hierauto-inverse*)

lemma *PseudoHA-CompFun-the* [*simp*]:
 $S \in States\ A \implies (the\ (CompFun\ (PseudoHA\ A\ D)\ S)) = \{\}$
by *simp*

lemma *PseudoHA-CompFun-ran* [*simp*]:
 $(ran\ (CompFun\ (PseudoHA\ SA\ D))) = \{\{\}\}$

by *auto*

lemma *PseudoHA-HARoot* [*simp*]:
 $(HARoot (PseudoHA SA D)) = SA$
by (*unfold HARoot-def*, *auto*)

lemma *PseudoHA-HAInitState* [*simp*]:
 $HAInitState (PseudoHA A D) = InitState A$
apply (*unfold HAHInitState-def*)
apply *simp*
done

lemma *PseudoHA-HAInitStates* [*simp*]:
 $HAInitStates (PseudoHA A D) = \{InitState A\}$
apply (*unfold HAHInitStates-def*)
apply *simp*
done

lemma *PseudoHA-Chi* [*simp*]:
 $S \in States A \implies Chi (PseudoHA A D) S = \{\}$
apply (*unfold Chi-def restrict-def*)
apply *auto*
done

lemma *PseudoHA-ChiRel* [*simp*]:
 $ChiRel (PseudoHA A D) = \{\}$
apply (*unfold ChiRel-def*)
apply *simp*
done

lemma *PseudoHA-InitConf* [*simp*]:
 $InitConf (PseudoHA A D) = \{InitState A\}$
apply (*unfold InitConf-def*)
apply *simp*
done

11.4 Extending a HA by a SA (*AddSA*)

definition

$AddSA :: [(s,e,d)hierauto, s * (s,e,d)seqauto]$
 $\Rightarrow (s,e,d)hierauto$
 $(\langle (- [++]/ -) \rangle [10,11]10) \textbf{ where}$
 $AddSA HA SSA = (let (S,SA) = SSA;$
 $DNew = HAHInitValue HA;$
 $FNew = insert SA (SAs HA);$
 $ENew = HAHEvents HA \cup SAEvents SA;$
 $GNew = CompFun HA [f+] (S,SA)$
 $\textbf{ in}$
 $Abs-hierauto(DNew,FNew,ENew,GNew))$

definition

```

AddHA :: [(('s','e','d')hierauto, 's * ('s','e','d')hierauto)]
      => ('s','e','d')hierauto
      (⟨(- [**]/ -)⟩ [10,11]10) where
AddHA HA1 SHA =
  (let (S,HA2)      = SHA;
      (D1,F1,E1,G1) = Rep-hierauto (HA1 [++] (S,HARoot HA2));
      (D2,F2,E2,G2) = Rep-hierauto HA2;
      FNew          = F1 ∪ F2;
      ENew          = E1 ∪ E2;
      GNew          = G1 ++ G2
  in
    Abs-hierauto(D1,FNew,ENew,GNew))

```

lemma AddSA-SAs:

```

[[ (States SA ∩ HAsStates HA) = {}];
  S ∈ HAsStates HA ]] ⇒ (SAs (HA [++] (S,SA))) = insert SA (SAs HA)
apply (unfold Let-def AddSA-def)
apply (subst SAs-def)
apply (simp add: hierauto-def Abs-hierauto-inverse)
done

```

lemma AddSA-Events:

```

[[ (States SA ∩ HAsStates HA) = {}];
  S ∈ HAsStates HA ]] ⇒
  HAsEvents (HA [++] (S,SA)) = (HAsEvents HA) ∪ (SAsEvents SA)
apply (unfold Let-def AddSA-def)
apply (subst HAsEvents-def)
apply (simp add: hierauto-def Abs-hierauto-inverse)
done

```

lemma AddSA-CompFun:

```

[[ (States SA ∩ HAsStates HA) = {}];
  S ∈ HAsStates HA ]] ⇒
  CompFun (HA [++] (S,SA)) = (CompFun HA [f+]) (S,SA)
apply (unfold Let-def AddSA-def)
apply (subst CompFun-def)
apply (simp add: hierauto-def Abs-hierauto-inverse)
done

```

lemma AddSA-HAsStates:

```

[[ (States SA ∩ HAsStates HA) = {}];
  S ∈ HAsStates HA ]] ⇒
  HAsStates (HA [++] (S,SA)) = (HAsStates HA) ∪ (States SA)
apply (unfold HAsStates-def)
apply (subst AddSA-SAs)
apply (unfold HAsStates-def)
apply auto

```

done

lemma *AddSA-HAInitValue*:

$\llbracket (States\ SA \cap HAStates\ HA) = \{\};$
 $S \in HAStates\ HA \rrbracket \implies$
 $(HAInitValue\ (HA\ [++]\ (S,SA))) = (HAInitValue\ HA)$
apply (*unfold Let-def AddSA-def*)
apply (*subst HAHInitValue-def*)
apply (*simp add: hierauto-def Abs-hierauto-inverse*)
done

lemma *AddSA-HARoot*:

$\llbracket (States\ SA \cap HAStates\ HA) = \{\};$
 $S \in HAStates\ HA \rrbracket \implies$
 $(HARoot\ (HA\ [++]\ (S,SA))) = (HARoot\ HA)$
apply (*unfold HARoot-def*)
apply (*simp add: AddSA-CompFun AddSA-SAs*)
apply (*subst FAddSA-RootEx-Root*)
apply *auto*
apply (*simp only: HAStates-SA-mem*)
apply (*unfold HAStates-def*)
apply *fast*
done

lemma *AddSA-CompFun-the*:

$\llbracket (States\ SA \cap HAStates\ A) = \{\};$
 $S \in HAStates\ A \rrbracket \implies$
 $(the\ ((CompFun\ (A\ [++]\ (S,SA)))\ S)) = insert\ SA\ (the\ ((CompFun\ A)\ S))$
by (*simp add: AddSA-CompFun*)

lemma *AddSA-CompFun-the2*:

$\llbracket S' \in States\ (SA::('a,'c,'d)seqauto);$
 $(States\ SA \cap HAStates\ A) = \{\};$
 $S \in HAStates\ A \rrbracket \implies$
 $the\ ((CompFun\ (A\ [++]\ (S,SA)))\ S') = \{\}$
apply (*simp add: AddSA-CompFun*)
apply (*subst FAddSA-dom-emptyset*)
apply *auto*
done

lemma *AddSA-CompFun-the3*:

$\llbracket S' \notin States\ (SA::('a,'c,'d)seqauto);$
 $S \neq S';$
 $(States\ SA \cap HAStates\ A) = \{\};$
 $S \in HAStates\ A \rrbracket \implies$
 $(the\ ((CompFun\ (A\ [++]\ (S,SA)))\ S')) = (the\ ((CompFun\ A)\ S'))$
by (*simp add: AddSA-CompFun*)

lemma *AddSA-CompFun-ran*:

```

[[ (States SA  $\cap$  HASates A) = {}];
  S  $\in$  HASates A ]]  $\implies$ 
  ran (CompFun (A [++] (S,SA))) =
    insert {} (insert (insert SA (the ((CompFun A) S))) (ran (CompFun A) -
{the ((CompFun A) S)}))
apply (simp add: AddSA-CompFun)
apply (subst FAddSA-ran)
apply auto
apply (fast dest: CompFun-Int-disjoint)
done

```

```

lemma AddSA-CompFun-ran2:
[[ (States SA1  $\cap$  HASates A) = {}];
  (States SA2  $\cap$  (HASates A  $\cup$  States SA1)) = {}];
  S  $\in$  HASates A;
  T  $\in$  States SA1 ]]  $\implies$ 
  ran (CompFun ((A [++] (S,SA1)) [++] (T,SA2))) =
    insert {} (insert {SA2} (ran (CompFun (A [++] (S,SA1)))))
apply (simp add: AddSA-HASates AddSA-CompFun)
apply (subst FAddSA-ran)
apply (rule ballI)
apply (rule impI)
apply (subst AddSA-CompFun [THEN sym])
apply simp
apply simp
apply (subst AddSA-CompFun [THEN sym])
apply simp
apply simp
apply (rule CompFun-Int-disjoint)
apply simp
apply (simp add: AddSA-HASates)
apply (simp add: AddSA-HASates)
apply (case-tac S  $\in$  States SA1)
apply simp
apply (simp only: dom-CompFun [THEN sym])
apply (frule FAddSA-dom-dom-States)
apply fast
apply simp
apply (case-tac S  $\in$  States SA1)
apply simp
apply fast
apply (subst FAddSA-dom-dom-States)
apply simp
apply simp
apply simp
apply (case-tac S  $\in$  States SA1)
apply simp
apply fast
apply (subst FAddSA-dom-dom-States)

```

```

apply simp
apply simp
apply simp
apply (case-tac  $S \in \text{States } SA1$ )
apply simp
apply fast
apply simp
apply fast
done

lemma AddSA-CompFun-ran-not-mem:
   $\llbracket \text{States } SA2 \cap (\text{HStates } A \cup \text{States } SA1) = \{\};$ 
   $\text{States } SA1 \cap \text{HStates } A = \{\};$ 
   $S \in \text{HStates } A \rrbracket \implies$ 
   $\{SA2\} \notin \text{ran } (\text{CompFun } A [f+](S, SA1))$ 
apply (cut-tac  $HA=A [++](S, SA1)$  and  $Sas=\{SA2\}$  in ran-CompFun-is-not-SA)
apply (metis AddSA-HStates SA-States-disjunct2 SAs-States-HStates insert-subset)
apply (simp add: AddSA-HStates AddSA-CompFun)
done

lemma AddSA-CompFun-ran3:
   $\llbracket (\text{States } SA1 \cap \text{HStates } A) = \{\};$ 
   $(\text{States } SA2 \cap (\text{HStates } A \cup \text{States } SA1)) = \{\};$ 
   $(\text{States } SA3 \cap (\text{HStates } A \cup \text{States } SA1 \cup \text{States } SA2)) = \{\};$ 
   $S \in \text{HStates } A;$ 
   $T \in \text{States } SA1 \rrbracket \implies$ 
   $\text{ran } (\text{CompFun } ((A [++](S, SA1)) [++](T, SA2) [++](T, SA3))) =$ 
   $\text{insert } \{\} (\text{insert } \{SA3, SA2\} (\text{ran } (\text{CompFun } (A [++](S, SA1)))))$ 
apply (simp add: AddSA-HStates AddSA-CompFun)
apply (subst FAddSA-ran)
apply (metis AddSA-CompFun AddSA-HStates CompFun-Int-disjoint UnCI
dom-CompFun)
apply (metis AddSA-CompFun AddSA-HStates UnCI dom-CompFun)
apply (metis AddSA-CompFun AddSA-HStates UnCI dom-CompFun)

apply (subst AddSA-CompFun [THEN sym])
back
apply simp
apply simp

apply (subst AddSA-CompFun [THEN sym])
back
apply (simp add: AddSA-HStates)
apply (simp add: AddSA-HStates)
apply (subst AddSA-CompFun-ran2)
apply fast
apply fast
apply fast
apply fast

```

```

apply (simp add: AddSA-CompFun)
apply (subst FAddSA-dom-insert)
  apply (subst FAddSA-dom-dom-States)
    apply simp
    apply fast
    apply simp
    apply fast
apply (subst FAddSA-dom-emptyset)
  apply simp
  apply fast
  apply simp
apply simp
apply (subst FAddSA-dom-insert)
  apply (subst FAddSA-dom-dom-States)
    apply simp
    apply fast
    apply simp
    apply fast
apply (subst FAddSA-dom-emptyset)
  apply simp
  apply fast
  apply simp
apply simp
by (simp add: AddSA-CompFun-ran-not-mem insert-Diff-if insert-commute)

lemma AddSA-CompFun-PseudoHA-ran:
   $\llbracket S \in \text{States RootSA};$ 
   $\text{States RootSA} \cap \text{States SA} = \{\} \rrbracket \implies$ 
   $(\text{ran } (\text{CompFun } ((\text{PseudoHA RootSA } D) \text{ } [++] (S, \text{SA})))) = (\text{insert } \{\} \{\{\text{SA}\}\})$ 
apply (subst AddSA-CompFun-ran)
apply auto
done

lemma AddSA-CompFun-PseudoHA-ran2:
   $\llbracket \text{States SA1} \cap \text{States RootSA} = \{\};$ 
   $\text{States SA2} \cap (\text{States RootSA} \cup \text{States SA1}) = \{\};$ 
   $S \in \text{States RootSA} \rrbracket \implies$ 
   $(\text{ran } (\text{CompFun } ((\text{PseudoHA RootSA } D) \text{ } [++] (S, \text{SA1}) \text{ } [++] (S, \text{SA2})))) =$ 
   $(\text{insert } \{\} \{\{\text{SA2}, \text{SA1}\}\})$ 
apply (subst AddSA-CompFun-ran)
prefer 3
apply (subst AddSA-CompFun-the)
apply simp
apply simp
apply (subst AddSA-CompFun-PseudoHA-ran)
apply fast
apply fast
apply (subst AddSA-CompFun-the)
apply simp

```

```

apply simp
apply simp
apply fast
apply (simp add: AddSA-HAStates)
apply (simp add: AddSA-HAStates)
done

lemma AddSA-HAInitStates [simp]:
   $\llbracket \text{States } SA \cap \text{HAStates } A = \{\};$ 
   $S \in \text{HAStates } A \rrbracket \implies$ 
   $\text{HAInitStates } (A \text{ [++]} (S, SA)) = \text{insert } (\text{InitState } SA) (\text{HAInitStates } A)$ 
apply (unfold HAINitStates-def)
apply (simp add: AddSA-SAs)
done

lemma AddSA-HAInitState [simp]:
   $\llbracket \text{States } SA \cap \text{HAStates } A = \{\};$ 
   $S \in \text{HAStates } A \rrbracket \implies$ 
   $\text{HAInitState } (A \text{ [++]} (S, SA)) = (\text{HAInitState } A)$ 
apply (unfold HAINitState-def)
apply (simp add: AddSA-HARoot)
done

lemma AddSA-Chi [simp]:
   $\llbracket \text{States } SA \cap \text{HAStates } A = \{\};$ 
   $S \in \text{HAStates } A \rrbracket \implies$ 
   $\text{Chi } (A \text{ [++]} (S, SA)) \ S = (\text{States } SA) \cup (\text{Chi } A \ S)$ 
apply (unfold Chi-def restrict-def)
apply (simp add: AddSA-SAs AddSA-HAStates AddSA-CompFun-the)
apply auto
done

lemma AddSA-Chi2 [simp]:
   $\llbracket \text{States } SA \cap \text{HAStates } A = \{\};$ 
   $S \in \text{HAStates } A;$ 
   $T \in \text{States } SA \rrbracket \implies$ 
   $\text{Chi } (A \text{ [++]} (S, SA)) \ T = \{\}$ 
apply (unfold Chi-def restrict-def)
apply (simp add: AddSA-SAs AddSA-HAStates AddSA-CompFun-the2)
done

lemma AddSA-Chi3 [simp]:
   $\llbracket \text{States } SA \cap \text{HAStates } A = \{\};$ 
   $S \in \text{HAStates } A;$ 
   $T \notin \text{States } SA; T \neq S \rrbracket \implies$ 
   $\text{Chi } (A \text{ [++]} (S, SA)) \ T = \text{Chi } A \ T$ 
apply (unfold Chi-def restrict-def)
apply (simp add: AddSA-SAs AddSA-HAStates AddSA-CompFun-the3)
apply auto

```


done

lemma *AddSA-ChiRel* [simp]:

$\llbracket \text{States } SA \cap \text{HAStates } A = \{\} \rrbracket$

$S \in \text{HAStates } A \rrbracket \implies$

$\text{ChiRel } (A \text{ } [++]) (S, SA) = \{ (T, T') . T = S \wedge T' \in \text{States } SA \} \cup (\text{ChiRel } A)$

apply (*unfold ChiRel-def*)

apply (*simp add: AddSA-HAStates*)

apply *safe*

apply (*rename-tac T U*)

apply (*case-tac T ∈ States SA*)

apply *simp*

apply *simp*

apply (*rename-tac T U*)

apply (*case-tac T ≠ S*)

apply (*case-tac T ∈ States SA*)

apply *simp*

apply *simp*

apply *simp*

apply (*rename-tac T U*)

apply (*case-tac T ∈ States SA*)

apply *simp*

apply *simp*

apply (*cut-tac A=A and T=T in Chi-HAStates*)

apply *fast*

apply (*case-tac T ∈ States SA*)

apply *simp*

apply *simp*

apply (*cut-tac A=A and T=T in Chi-HAStates*)

apply *fast*

apply *fast*

apply (*rename-tac T U*)

apply (*case-tac T ≠ S*)

apply (*case-tac T ∈ States SA*)

apply *simp*

apply *simp*

apply *simp*

apply (*rename-tac T U*)

apply (*case-tac T ∈ States SA*)

apply *auto*

apply (*metis AddSA-Chi AddSA-Chi3 Int-iff Un-iff empty-iff*)

done

lemma *help-InitConf*:

$\llbracket \text{States } SA \cap \text{HAStates } A = \{\} \rrbracket \implies \{p. \text{fst } p \neq \text{InitState } SA \wedge \text{snd } p \neq \text{InitState}$

$SA \wedge$

$p \in \text{insert } (\text{InitState } SA) (\text{HAIInitStates } A) \times \text{insert } (\text{InitState } SA) (\text{HAIInitStates}$

$A) \wedge$

$(p \in \{S\} \times \text{States } SA \vee p \in \text{ChiRel } A)\} =$

```

    (HInitStates A × HInitStates A ∩ ChiRel A)
  apply auto
  apply (cut-tac A=SA in InitState-States)
  apply (cut-tac A=A in HInitStates-HAStates, fast)
  apply (cut-tac A=SA in InitState-States)
  apply (cut-tac A=A in HInitStates-HAStates, fast)
done

lemma AddSA-InitConf [simp]:
  [| States SA ∩ HAStates A = {};
   S ∈ InitConf A |] ⇒
    InitConf (A [++] (S,SA)) = insert (InitState SA) (InitConf A)
  apply (frule InitConf-HAStates2)
  apply (unfold InitConf-def)
  apply (simp del: insert-Times-insert)
    apply auto
    apply (rename-tac T)
    apply (case-tac T=S)
    apply auto
  prefer 3
  apply (rule-tac R=(HInitStates A) × (HInitStates A) ∩ ChiRel A in trancl-subseteq)
  apply auto
  apply (rotate-tac 3)
  apply (frule trancl-collect)
    prefer 2
    apply fast
    apply auto
    apply (cut-tac A=SA in InitState-States)
    apply (frule ChiRel-HAStates)
    apply fast
    apply (frule ChiRel-HAStates)
    apply (cut-tac A=SA in InitState-States)
    apply fast
  apply (subst help-InitConf [THEN sym])
  apply fast
  apply auto
  apply (rule-tac b=S in rtrancl-into-rtrancl)
  apply auto
  prefer 2
  apply (erule rtranclE)
  apply auto
  prefer 2
  apply (erule rtranclE)
  apply auto
  apply (rule-tac R=(HInitStates A) × (HInitStates A) ∩ ChiRel A in trancl-subseteq)
  apply auto
done

lemma AddSA-InitConf2 [simp]:

```

```

[[ States SA  $\cap$  HStates A = {};
   S  $\notin$  InitConf A;
   S  $\in$  HStates A ]]  $\implies$ 
   InitConf (A [++] (S,SA)) = InitConf A
apply (unfold InitConf-def)
apply simp
apply auto
apply (rename-tac T)
prefer 2
apply (rule-tac R=(HInitStates A)  $\times$  (HInitStates A)  $\cap$  ChiRel A in trancl-subseteq)
  apply auto
apply (case-tac T=InitState SA)
  apply auto
  prefer 2
  apply (rotate-tac 3)
  apply (frule trancl-collect)
    prefer 2
    apply fast
  apply auto
  apply (cut-tac A=SA in InitState-States)
  apply (frule ChiRel-HStates)
  apply fast
  apply (cut-tac A=SA in InitState-States)
  apply (frule ChiRel-HStates)
  apply fast
  apply (cut-tac A=SA in InitState-States)
  apply (cut-tac A=A in HInitStates-HStates)
  apply (subst help-InitConf [THEN sym])
  apply fast
  apply auto
apply (rule-tac b=InitState SA in rtrancl-induct)
  apply auto
  apply (frule ChiRel-HStates2)
  apply (cut-tac A=SA in InitState-States)
  apply fast
  prefer 2
  apply (frule ChiRel-HStates)
  apply (cut-tac A=SA in InitState-States)
  apply fast
apply (rule rtrancl-into-rtrancl)
  apply auto
done

```

11.5 Theorems for Calculating Wellformedness of HA

lemma *PseudoHA-HStates-IFF*:

(States SA) = X $\implies$ (HStates (PseudoHA SA D)) = X

```

apply simp
done

```

lemma *AddSA-SAs-IFF*:
 $\llbracket \text{States } SA \cap \text{HASstates } HA = \{\};$
 $S \in \text{HASstates } HA;$
 $(\text{SAs } HA) = X \rrbracket \implies (\text{SAs } (HA \text{ } [++] (S, SA))) = (\text{insert } SA \ X)$
apply (*subst AddSA-SAs*)
apply *auto*
done

lemma *AddSA-Events-IFF*:
 $\llbracket \text{States } SA \cap \text{HASstates } HA = \{\};$
 $S \in \text{HASstates } HA;$
 $(\text{HAEvents } HA) = \text{HAE};$
 $(\text{SAEvents } SA) = \text{SAE};$
 $(\text{HAE} \cup \text{SAE}) = X \rrbracket \implies (\text{HAEvents } (HA \text{ } [++] (S, SA))) = X$
apply (*subst AddSA-Events*)
apply *auto*
done

lemma *AddSA-CompFun-IFF*:
 $\llbracket \text{States } SA \cap \text{HASstates } HA = \{\};$
 $S \in \text{HASstates } HA;$
 $(\text{CompFun } HA) = \text{HAG};$
 $(\text{HAG } [f+] (S, SA)) = X \rrbracket \implies (\text{CompFun } (HA \text{ } [++] (S, SA))) = X$
apply (*subst AddSA-CompFun*)
apply *auto*
done

lemma *AddSA-HASstates-IFF*:
 $\llbracket \text{States } SA \cap \text{HASstates } HA = \{\};$
 $S \in \text{HASstates } HA;$
 $(\text{HASstates } HA) = \text{HAS};$
 $(\text{States } SA) = \text{SAS};$
 $(\text{HAS} \cup \text{SAS}) = X \rrbracket \implies (\text{HASstates } (HA \text{ } [++] (S, SA))) = X$
apply (*subst AddSA-HASstates*)
apply *auto*
done

lemma *AddSA-HAInitValue-IFF*:
 $\llbracket \text{States } SA \cap \text{HASstates } HA = \{\};$
 $S \in \text{HASstates } HA;$
 $(\text{HAInitValue } HA) = X \rrbracket \implies (\text{HAInitValue } (HA \text{ } [++] (S, SA))) = X$
apply (*subst AddSA-HAInitValue*)
apply *auto*
done

lemma *AddSA-HARoot-IFF*:
 $\llbracket \text{States } SA \cap \text{HASstates } HA = \{\};$
 $S \in \text{HASstates } HA;$

$(HARoot\ HA) = X \implies (HARoot\ (HA\ [++]\ (S,\ SA))) = X$
apply (subst AddSA-HARoot)
apply auto
done

lemma AddSA-InitConf-IFF:

$\llbracket\ InitConf\ A = Y;$
 $\quad States\ SA \cap HStates\ A = \{\};$
 $\quad S \in HStates\ A;$
 $\quad (if\ S \in Y\ then\ insert\ (InitState\ SA)\ Y\ else\ Y) = X \rrbracket \implies$
 $\quad InitConf\ (A\ [++]\ (S,SA)) = X$
apply (case-tac $S \in Y$)
apply auto
done

lemma AddSA-CompFun-ran-IFF:

$\llbracket\ (States\ SA \cap HStates\ A) = \{\};$
 $\quad S \in HStates\ A;$
 $\quad (insert\ \{\}\ (insert\ (insert\ SA\ (the\ ((CompFun\ A)\ S)))\ (ran\ (CompFun\ A) -$
 $\{the\ ((CompFun\ A)\ S)\}))) = X \rrbracket \implies$
 $\quad ran\ (CompFun\ (A\ [++]\ (S,SA))) = X$
apply (subst AddSA-CompFun-ran)
apply auto
done

lemma AddSA-CompFun-ran2-IFF:

$\llbracket\ (States\ SA1 \cap HStates\ A) = \{\};$
 $\quad (States\ SA2 \cap (HStates\ A \cup States\ SA1)) = \{\};$
 $\quad S \in HStates\ A;$
 $\quad T \in States\ SA1;$
 $\quad insert\ \{\}\ (insert\ \{SA2\}\ (ran\ (CompFun\ (A\ [++]\ (S,SA1)))))) = X \rrbracket \implies$
 $\quad ran\ (CompFun\ ((A\ [++]\ (S,SA1))\ [++]\ (T,SA2))) = X$
apply (subst AddSA-CompFun-ran2)
apply auto
done

lemma AddSA-CompFun-ran3-IFF:

$\llbracket\ (States\ SA1 \cap HStates\ A) = \{\};$
 $\quad (States\ SA2 \cap (HStates\ A \cup States\ SA1)) = \{\};$
 $\quad (States\ SA3 \cap (HStates\ A \cup States\ SA1 \cup States\ SA2)) = \{\};$
 $\quad S \in HStates\ A;$
 $\quad T \in States\ SA1;$
 $\quad insert\ \{\}\ (insert\ \{SA3,SA2\}\ (ran\ (CompFun\ (A\ [++]\ (S,SA1)))))) = X \rrbracket \implies$
 $\quad ran\ (CompFun\ ((A\ [++]\ (S,SA1))\ [++]\ (T,SA2)\ [++]\ (T,SA3))) = X$
apply (subst AddSA-CompFun-ran3)
apply auto
done

lemma AddSA-CompFun-PseudoHA-ran-IFF:

```

  [  $S \in \text{States RootSA};$ 
     $\text{States RootSA} \cap \text{States SA} = \{\};$ 
     $(\text{insert } \{\} \{\{SA\}\}) = X ] \implies$ 
     $(\text{ran } (\text{CompFun } ((\text{PseudoHA RootSA } D) [++] (S, SA)))) = X$ 
apply (subst AddSA-CompFun-PseudoHA-ran)
apply auto
done

```

lemma AddSA-CompFun-PseudoHA-ran2-IFF:

```

  [  $\text{States SA1} \cap \text{States RootSA} = \{\};$ 
     $\text{States SA2} \cap (\text{States RootSA} \cup \text{States SA1}) = \{\};$ 
     $S \in \text{States RootSA};$ 
     $(\text{insert } \{\} \{\{SA2, SA1\}\}) = X ] \implies$ 
     $(\text{ran } (\text{CompFun } ((\text{PseudoHA RootSA } D) [++] (S, SA1) [++] (S, SA2)))) = X$ 
apply (subst AddSA-CompFun-PseudoHA-ran2)
apply auto
done

```

ML <

```

val AddSA-SAs-IFF = @{thm AddSA-SAs-IFF};
val AddSA-Events-IFF = @{thm AddSA-Events-IFF};
val AddSA-CompFun-IFF = @{thm AddSA-CompFun-IFF};
val AddSA-HAStates-IFF = @{thm AddSA-HAStates-IFF};
val PseudoHA-HAStates-IFF = @{thm PseudoHA-HAStates-IFF};
val AddSA-HAInitValue-IFF = @{thm AddSA-HAInitValue-IFF};
val AddSA-CompFun-ran-IFF = @{thm AddSA-CompFun-ran-IFF};
val AddSA-HARoot-IFF = @{thm AddSA-HARoot-IFF};
val insert-inter = @{thm insert-inter};
val insert-notmem = @{thm insert-notmem};
val PseudoHA-CompFun = @{thm PseudoHA-CompFun};
val PseudoHA-Events = @{thm PseudoHA-Events};
val PseudoHA-SAs = @{thm PseudoHA-SAs};
val PseudoHA-HARoot = @{thm PseudoHA-HARoot};
val PseudoHA-HAInitValue = @{thm PseudoHA-HAInitValue};
val PseudoHA-CompFun-ran = @{thm PseudoHA-CompFun-ran};
val Un-empty-right = @{thm Un-empty-right};
val insert-union = @{thm insert-union};

```

```

fun wellformed-tac ctxt L i =
  FIRST[resolve-tac ctxt [AddSA-SAs-IFF] i,
        resolve-tac ctxt [AddSA-Events-IFF] i,
        resolve-tac ctxt [AddSA-CompFun-IFF] i,
        resolve-tac ctxt [AddSA-HAStates-IFF] i,
        resolve-tac ctxt [PseudoHA-HAStates-IFF] i,
        resolve-tac ctxt [AddSA-HAInitValue-IFF] i,
        resolve-tac ctxt [AddSA-HARoot-IFF] i,

```

```

    resolve-tac ctxt [AddSA-CompFun-ran-IFF] i,
    resolve-tac ctxt [insert-inter] i,
    resolve-tac ctxt [insert-notmem] i,
    CHANGED (simp-tac (put-simpset HOL-basic-ss ctxt addsimps
[PseudoHA-HARoot, PseudoHA-CompFun, PseudoHA-CompFun-ran, PseudoHA-Events, PseudoHA-SAs
    PseudoHA-HAInitValue, Un-empty-right]@ L) i),
    fast-tac ctxt i,
    CHANGED (simp-tac ctxt i)];
  ›

method-setup wellformed = ‹Attrib.thms ‹› (fn thms => fn ctxt => (METHOD
(fn facts =>
                                                                    (HEADGOAL (wellformed-tac ctxt (facts @
thms))))))›

end

```

12 Example Specification for a Car Audio System

```

theory CarAudioSystem
imports HAKripke HAOps
begin

```

12.1 Definitions

12.1.1 Data space for two Integer-Variables

```

datatype d = V0 int
           | V1 int

```

```

primrec
  Sel0 :: d => int where
    Sel0 (V0 i) = i

```

```

primrec
  Sel1 :: d => int where
    Sel1 (V1 i) = i

```

```

definition
  Select0 :: [d data] => int where
    Select0 d = Sel0 (DataPart d 0)

```

```

definition
  Select1 :: [d data] => int where
    Select1 d = Sel1 (DataPart d 1)

```

definition

DSpace :: *d dataspace* **where**
DSpace = *Abs-dataspace* ([*range V0*, *range V1*])

definition

LiftInitData :: (*d list*) => *d data* **where**
LiftInitData L = *Abs-data* (*L*, *DSpace*)

definition

LiftPUpdate :: (*d data* => ((*d option*) *list*)) => *d pupdate* **where**
LiftPUpdate L = *Abs-pupdate*
 (λ *d*. if ((*DataSpace d*) = *DSpace*) then
 Abs-pdata (*L d*, *DSpace*)
 else (*Data2PData d*))

12.1.2 Sequential Automaton *Root-CTRL***definition**

Root-CTRL-States :: *string set* **where**
Root-CTRL-States = {"CarAudioSystem"}

definition

Root-CTRL-Init :: *string* **where**
Root-CTRL-Init = "CarAudioSystem"

definition

Root-CTRL-Labels :: (*string, string, d*)*label set* **where**
Root-CTRL-Labels = {}

definition

Root-CTRL-Delta :: (*string, string, d*)*trans set* **where**
Root-CTRL-Delta = {}

definition

Root-CTRL :: (*string, string, d*)*seqauto* **where**
Root-CTRL = *Abs-seqauto* (*Root-CTRL-States*, *Root-CTRL-Init*,
 Root-CTRL-Labels, *Root-CTRL-Delta*)

12.1.3 Sequential Automaton *CDPlayer-CTRL***definition**

CDPlayer-CTRL-States :: *string set* **where**
CDPlayer-CTRL-States = {"ReadTracks", "CDFull", "CDEmpty"}

definition

CDPlayer-CTRL-Init :: *string* **where**
CDPlayer-CTRL-Init = "CDEmpty"

definition

CDPlayer-CTRL-Update1 :: *d* *pupdate* **where**
CDPlayer-CTRL-Update1 = *LiftPUpdate* (% *d*. [*None*, *Some* (*V1* ((*Select0* *d*) + 1))])

definition

CDPlayer-CTRL-Action1 :: (*string*,*d*)*action* **where**
CDPlayer-CTRL-Action1 = ({}, *CDPlayer-CTRL-Update1*)

definition

CDPlayer-CTRL-Update2 :: *d* *pupdate* **where**
CDPlayer-CTRL-Update2 = *LiftPUpdate* (% *d*. [*Some* (*V0* ((*Select0* *d*) + 1)), *None*])

definition

CDPlayer-CTRL-Action2 :: (*string*,*d*)*action* **where**
CDPlayer-CTRL-Action2 = ({}, *CDPlayer-CTRL-Update2*)

definition

CDPlayer-CTRL-Update3 :: *d* *pupdate* **where**
CDPlayer-CTRL-Update3 = *LiftPUpdate* (% *d*. [*Some* (*V0* 0), *None*])

definition

CDPlayer-CTRL-Action3 :: (*string*,*d*)*action* **where**
CDPlayer-CTRL-Action3 = ({}, *CDPlayer-CTRL-Update3*)

definition

CDPlayer-CTRL-Labels :: (*string*,*string*,*d*)*label set* **where**
CDPlayer-CTRL-Labels = {(En "*LastTrack*", *defaultguard*, *CDPlayer-CTRL-Action1*),
(En "*NewTrack*", *defaultguard*, *CDPlayer-CTRL-Action2*),
(And (En "*CDEject*") (In "*On*"), *defaultguard*, *CD-Player-CTRL-Action3*),
(En "*CDIn*", *defaultguard*, *defaultaction*)}

definition

CDPlayer-CTRL-Delta :: (*string*,*string*,*d*)*trans set* **where**
CDPlayer-CTRL-Delta = {("*ReadTracks*", (En "*LastTrack*", *defaultguard*, *CD-Player-CTRL-Action1*),
"*CDFull*",
("*CDFull*", (And (En "*CDEject*") (In "*On*"), *defaultguard*,
CDPlayer-CTRL-Action3),
"*CDEmpty*",
("*ReadTracks*", (En "*NewTrack*", *defaultguard*, *CD-Player-CTRL-Action2*),
"*ReadTracks*",
("*CDEmpty*", (En "*CDIn*", *defaultguard*, *defaultaction*),
"*ReadTracks*")}

definition

CDPlayer-CTRL :: (string,string,d)seqauto **where**
CDPlayer-CTRL = Abs-seqauto (*CDPlayer-CTRL-States*, *CDPlayer-CTRL-Init*,
CDPlayer-CTRL-Labels, *CDPlayer-CTRL-Delta*)

12.1.4 Sequential Automaton *AudioPlayer-CTRL*

definition

AudioPlayer-CTRL-States :: string set **where**
AudioPlayer-CTRL-States = {"Off","On"}

definition

AudioPlayer-CTRL-Init :: string **where**
AudioPlayer-CTRL-Init = "Off"

definition

AudioPlayer-CTRL-Labels :: (string,string,d)label set **where**
AudioPlayer-CTRL-Labels = {(En "O", defaultguard, defaultaction)}

definition

AudioPlayer-CTRL-Delta :: (string,string,d)trans set **where**
AudioPlayer-CTRL-Delta = {"Off", (En "O", defaultguard, defaultaction), "On",
("On", (En "O", defaultguard, defaultaction), "Off")}

definition

AudioPlayer-CTRL :: (string,string,d)seqauto **where**
AudioPlayer-CTRL = Abs-seqauto (*AudioPlayer-CTRL-States*, *AudioPlayer-CTRL-Init*,
AudioPlayer-CTRL-Labels, *AudioPlayer-CTRL-Delta*)

12.1.5 Sequential Automaton *On-CTRL*

definition

On-CTRL-States :: string set **where**
On-CTRL-States = {"TunerMode","CDMode"}

definition

On-CTRL-Init :: string **where**
On-CTRL-Init = "TunerMode"

definition

On-CTRL-Labels :: (string,string,d)label set **where**
On-CTRL-Labels = {(And (En "Src") (In "CDFull"), defaultguard, defaultac-
tion),
(En "Src", defaultguard, defaultaction),
(En "CDEject", defaultguard, defaultaction),
(En "EndOfTitle", (λ d. (DataPart d 0) = (DataPart d 1)),
defaultaction)}

definition

On-CTRL-Delta :: (string,string,d)trans set **where**

$On-CTRL-Delta = \{("TunerMode", (And (En "Src") (In "CDFull"), defaultguard, defaultaction), "CDMode"),$
 $(("CDMode", (En "Src", defaultguard, defaultaction), "TunerMode"),$
 $(("CDMode", (En "CDEject", defaultguard, defaultaction),$
 $"TunerMode"),$
 $(("CDMode", (En "EndOfTitle", (\lambda d. (DataPart d 0) = (DataPart d 1)), defaultaction),$
 $"TunerMode"))\}$

definition

$On-CTRL :: (string, string, d)seqauto$ **where**
 $On-CTRL = Abs-seqauto (On-CTRL-States, On-CTRL-Init,$
 $On-CTRL-Labels, On-CTRL-Delta)$

12.1.6 Sequential Automaton *TunerMode-CTRL*

definition

$TunerMode-CTRL-States :: string set$ **where**
 $TunerMode-CTRL-States = \{"1", "2", "3", "4"\}$

definition

$TunerMode-CTRL-Init :: string$ **where**
 $TunerMode-CTRL-Init = "1"$

definition

$TunerMode-CTRL-Labels :: (string, string, d)label set$ **where**
 $TunerMode-CTRL-Labels = \{(En "Next", defaultguard, defaultaction),$
 $(En "Back", defaultguard, defaultaction)\}$

definition

$TunerMode-CTRL-Delta :: (string, string, d)trans set$ **where**
 $TunerMode-CTRL-Delta = \{("1", (En "Next", defaultguard, defaultaction), "2"),$
 $(("2", (En "Next", defaultguard, defaultaction), "3"),$
 $(("3", (En "Next", defaultguard, defaultaction), "4"),$
 $(("4", (En "Next", defaultguard, defaultaction), "1"),$
 $(("1", (En "Back", defaultguard, defaultaction), "4"),$
 $(("4", (En "Back", defaultguard, defaultaction), "3"),$
 $(("3", (En "Back", defaultguard, defaultaction), "2"),$
 $(("2", (En "Back", defaultguard, defaultaction), "1"))\}$

definition

$TunerMode-CTRL :: (string, string, d)seqauto$ **where**
 $TunerMode-CTRL = Abs-seqauto (TunerMode-CTRL-States, TunerMode-CTRL-Init,$
 $TunerMode-CTRL-Labels, TunerMode-CTRL-Delta)$

12.1.7 Sequential Automaton *CDMode-CTRL*

definition

$CDMode-CTRL-States :: string set$ **where**
 $CDMode-CTRL-States = \{"Playing", "SelectingNextTrack",$

"SelectingPreviousTrack"}

definition

CDMode-CTRL-Init :: *string* **where**
CDMode-CTRL-Init = *"Playing"*

definition

CDMode-CTRL-Update1 :: *d* *pupdate* **where**
CDMode-CTRL-Update1 = *LiftPUpdate* (% *d*. [*Some* (*V0* ((*Select0* *d*) + 1)),
None])

definition

CDMode-CTRL-Action1 :: (*string*,*d*)*action* **where**
CDMode-CTRL-Action1 = ({},*CDMode-CTRL-Update1*)

definition

CDMode-CTRL-Update2 :: *d* *pupdate* **where**
CDMode-CTRL-Update2 = *LiftPUpdate* (% *d*. [*Some* (*V0* ((*Select0* *d*) - 1)),
None])

definition

CDMode-CTRL-Action2 :: (*string*,*d*)*action* **where**
CDMode-CTRL-Action2 = ({},*CDMode-CTRL-Update2*)

definition

CDMode-CTRL-Labels :: (*string*,*string*,*d*)*label set* **where**
CDMode-CTRL-Labels = {(En *"Next"*, *defaultguard*, *defaultaction*),
(En *"Back"*, *defaultguard*, *defaultaction*),
(En *"Ready"*, *defaultguard*, *CDMode-CTRL-Action1*),
(En *"Ready"*, *defaultguard*, *CDMode-CTRL-Action2*),
(En *"EndOfTitle"*, (λ (*d*:: *d data*). (*Select0* *d*) < (*Select1* *d*)),
defaultaction)}

definition

CDMode-CTRL-Delta :: (*string*,*string*,*d*)*trans set* **where**
CDMode-CTRL-Delta = {"Playing", (En *"Next"*, *defaultguard*, *defaultaction*) ,
"SelectingNextTrack"),
(En *"SelectingNextTrack"*, (En *"Ready"*, *defaultguard*, *CD-*
Mode-CTRL-Action1) ,*"Playing"*),
(En *"Playing"*, (En *"Back"*, *defaultguard*, *defaultaction*)
, *"SelectingPreviousTrack"*),
(En *"SelectingPreviousTrack"*, (En *"Ready"*, *defaultguard*,
CDMode-CTRL-Action2),
"Playing"),
(En *"Playing"*, (En *"EndOfTitle"*, (λ (*d*:: *d data*). (*Select0* *d*) <
(*Select1* *d*)), *defaultaction*),
"SelectingNextTrack")}

definition

CDMode-CTRL :: (string,string,d)seqauto **where**
CDMode-CTRL = Abs-seqauto (*CDMode-CTRL-States*, *CDMode-CTRL-Init*,
CDMode-CTRL-Labels, *CDMode-CTRL-Delta*)

12.1.8 Hierarchical Automaton *CarAudioSystem*

definition

CarAudioSystem :: (string,string,d)hierauto **where**
CarAudioSystem = ((*PseudoHA Root-CTRL (LiftInitData [V0 0, V1 0])*)
 [++] ("*CarAudioSystem*",*CDPlayer-CTRL*)
 [++] ("*CarAudioSystem*",*AudioPlayer-CTRL*)
 [++] ("*On*", *TunerMode-CTRL*)
 [++] ("*On*", *CDMode-CTRL*))

12.2 Lemmas

12.2.1 Sequential Automaton *CDMode-CTRL*

lemma *check-Root-CTRL*:

(*Root-CTRL-States*,*Root-CTRL-Init*,*Root-CTRL-Labels*,*Root-CTRL-Delta*) : seqauto

apply (unfold seqauto-def SeqAuto-def *Root-CTRL-States-def Root-CTRL-Init-def Root-CTRL-Labels-def Root-CTRL-Delta-def*)

apply *simp*

done

lemma *States-Root-CTRL*:

States Root-CTRL = *Root-CTRL-States*

apply (*simp add: Root-CTRL-def*)

apply (*unfold States-def*)

apply (*simp add: Abs-seqauto-inverse check-Root-CTRL*)

done

lemma *Init-State-Root-CTRL*:

InitState Root-CTRL = *Root-CTRL-Init*

apply (*simp add: Root-CTRL-def*)

apply (*unfold InitState-def*)

apply (*simp add: Abs-seqauto-inverse check-Root-CTRL*)

done

lemma *Labels-Root-CTRL*:

Labels Root-CTRL = *Root-CTRL-Labels*

apply (*simp add: Root-CTRL-def*)

apply (*unfold Labels-def*)

apply (*simp add: Abs-seqauto-inverse check-Root-CTRL*)

done

lemma *Delta-Root-CTRL*:

Delta Root-CTRL = *Root-CTRL-Delta*

apply (*simp add: Root-CTRL-def*)

```

apply (unfold Delta-def)
apply (simp add: Abs-seqauto-inverse check-Root-CTRL)
done

```

```

schematic-goal Events-Root-CTRL:
  SAEvents Root-CTRL = ?X
apply (unfold SAEvents-def expr-def)
apply (rule trans)
apply (simp add: expr-def Delta-Root-CTRL Root-CTRL-Delta-def)
apply (rule refl)
done

```

12.2.2 Sequential Automaton *CDPlayer-CTRL*

```

lemma check-CDPlayer-CTRL:
  (CDPlayer-CTRL-States, CDPlayer-CTRL-Init, CDPlayer-CTRL-Labels, CDPlayer-CTRL-Delta)
  : seqauto
apply (unfold seqauto-def SeqAuto-def CDPlayer-CTRL-States-def CDPlayer-CTRL-Init-def
CDPlayer-CTRL-Labels-def CDPlayer-CTRL-Delta-def)
apply simp
done

```

```

lemma States-CDPlayer-CTRL:
  States CDPlayer-CTRL = CDPlayer-CTRL-States
apply (simp add: CDPlayer-CTRL-def)
apply (unfold States-def)
apply (simp add: Abs-seqauto-inverse check-CDPlayer-CTRL)
done

```

```

lemma Init-State-CDPlayer-CTRL:
  InitState CDPlayer-CTRL = CDPlayer-CTRL-Init
apply (simp add: CDPlayer-CTRL-def)
apply (unfold InitState-def)
apply (simp add: Abs-seqauto-inverse check-CDPlayer-CTRL)
done

```

```

lemma Labels-CDPlayer-CTRL:
  Labels CDPlayer-CTRL = CDPlayer-CTRL-Labels
apply (simp add: CDPlayer-CTRL-def)
apply (unfold Labels-def)
apply (simp add: Abs-seqauto-inverse check-CDPlayer-CTRL)
done

```

```

lemma Delta-CDPlayer-CTRL:
  Delta CDPlayer-CTRL = CDPlayer-CTRL-Delta
apply (simp add: CDPlayer-CTRL-def)
apply (unfold Delta-def)
apply (simp add: Abs-seqauto-inverse check-CDPlayer-CTRL)
done

```

```

schematic-goal Events-CDPlayer-CTRL:
  SAEvents CDPlayer-CTRL = ?X
apply (unfold SAEvents-def)
apply (rule trans)
apply (simp add: expr-def Delta-CDPlayer-CTRL CDPlayer-CTRL-Delta-def CD-
Player-CTRL-Action1-def CDPlayer-CTRL-Action2-def CDPlayer-CTRL-Action3-def
Label-def)
apply (rule refl)
done

```

12.2.3 Sequential Automaton *AudioPlayer-CTRL*

```

lemma check-AudioPlayer-CTRL:
  (AudioPlayer-CTRL-States, AudioPlayer-CTRL-Init, AudioPlayer-CTRL-Labels, AudioPlayer-CTRL-Delta)
  : seqauto
apply (unfold seqauto-def SeqAuto-def AudioPlayer-CTRL-States-def AudioPlayer-CTRL-Init-def
AudioPlayer-CTRL-Labels-def AudioPlayer-CTRL-Delta-def)
apply simp
done

```

```

lemma States-AudioPlayer-CTRL:
  States AudioPlayer-CTRL = AudioPlayer-CTRL-States
apply (simp add: AudioPlayer-CTRL-def)
apply (unfold States-def)
apply (simp add: Abs-seqauto-inverse check-AudioPlayer-CTRL)
done

```

```

lemma Init-State-AudioPlayer-CTRL:
  InitState AudioPlayer-CTRL = AudioPlayer-CTRL-Init
apply (simp add: AudioPlayer-CTRL-def)
apply (unfold InitState-def)
apply (simp add: Abs-seqauto-inverse check-AudioPlayer-CTRL)
done

```

```

lemma Labels-AudioPlayer-CTRL:
  Labels AudioPlayer-CTRL = AudioPlayer-CTRL-Labels
apply (simp add: AudioPlayer-CTRL-def)
apply (unfold Labels-def)
apply (simp add: Abs-seqauto-inverse check-AudioPlayer-CTRL)
done

```

```

lemma Delta-AudioPlayer-CTRL:
  Delta AudioPlayer-CTRL = AudioPlayer-CTRL-Delta
apply (simp add: AudioPlayer-CTRL-def)
apply (unfold Delta-def)
apply (simp add: Abs-seqauto-inverse check-AudioPlayer-CTRL)
done

```

```

schematic-goal Events-AudioPlayer-CTRL:
  SAEvents AudioPlayer-CTRL = ?X
apply (unfold SAEvents-def)
apply (rule trans)
apply (simp add: expr-def Delta-AudioPlayer-CTRL AudioPlayer-CTRL-Delta-def
Label-def)
apply (rule refl)
done

```

12.2.4 Sequential Automaton *On-CTRL*

```

lemma check-On-CTRL:
  (On-CTRL-States, On-CTRL-Init, On-CTRL-Labels, On-CTRL-Delta) : seqauto
apply (unfold seqauto-def SeqAuto-def On-CTRL-States-def On-CTRL-Init-def On-CTRL-Labels-def
On-CTRL-Delta-def)
apply simp
done

```

```

lemma States-On-CTRL:
  States On-CTRL = On-CTRL-States
apply (simp add: On-CTRL-def)
apply (unfold States-def)
apply (simp add: Abs-seqauto-inverse check-On-CTRL)
done

```

```

lemma Init-State-On-CTRL:
  InitState On-CTRL = On-CTRL-Init
apply (simp add: On-CTRL-def)
apply (unfold InitState-def)
apply (simp add: Abs-seqauto-inverse check-On-CTRL)
done

```

```

lemma Labels-On-CTRL:
  Labels On-CTRL = On-CTRL-Labels
apply (simp add: On-CTRL-def)
apply (unfold Labels-def)
apply (simp add: Abs-seqauto-inverse check-On-CTRL)
done

```

```

lemma Delta-On-CTRL:
  Delta On-CTRL = On-CTRL-Delta
apply (simp add: On-CTRL-def)
apply (unfold Delta-def)
apply (simp add: Abs-seqauto-inverse check-On-CTRL)
done

```

```

schematic-goal Events-On-CTRL:
  SAEvents On-CTRL = ?X
apply (unfold SAEvents-def)

```



```

apply (rule trans)
apply (simp add: expr-def Delta-On-CTRL On-CTRL-Delta-def Label-def)
apply (rule refl)
done

```

12.2.5 Sequential Automaton *TunerMode-CTRL*

```

lemma check-TunerMode-CTRL:
  (TunerMode-CTRL-States, TunerMode-CTRL-Init, TunerMode-CTRL-Labels, TunerMode-CTRL-Delta)
  : seqauto
apply (unfold seqauto-def SeqAuto-def TunerMode-CTRL-States-def TunerMode-CTRL-Init-def
TunerMode-CTRL-Labels-def TunerMode-CTRL-Delta-def)
apply simp
done

```

```

lemma States-TunerMode-CTRL:
  States TunerMode-CTRL = TunerMode-CTRL-States
apply (simp add: TunerMode-CTRL-def)
apply (unfold States-def)
apply (simp add: Abs-seqauto-inverse check-TunerMode-CTRL)
done

```

```

lemma Init-State-TunerMode-CTRL:
  InitState TunerMode-CTRL = TunerMode-CTRL-Init
apply (simp add: TunerMode-CTRL-def)
apply (unfold InitState-def)
apply (simp add: Abs-seqauto-inverse check-TunerMode-CTRL)
done

```

```

lemma Labels-TunerMode-CTRL:
  Labels TunerMode-CTRL = TunerMode-CTRL-Labels
apply (simp add: TunerMode-CTRL-def)
apply (unfold Labels-def)
apply (simp add: Abs-seqauto-inverse check-TunerMode-CTRL)
done

```

```

lemma Delta-TunerMode-CTRL:
  Delta TunerMode-CTRL = TunerMode-CTRL-Delta
apply (simp add: TunerMode-CTRL-def)
apply (unfold Delta-def)
apply (simp add: Abs-seqauto-inverse check-TunerMode-CTRL)
done

```

```

schematic-goal Events-TunerMode-CTRL:
  SAEvents TunerMode-CTRL = ?X
apply (unfold SAEvents-def)
apply (rule trans)
apply (simp add: expr-def Delta-TunerMode-CTRL TunerMode-CTRL-Delta-def
Label-def)

```

```

apply (rule refl)
done

```

12.2.6 Sequential Automaton *CDMode-CTRL*

```

lemma check-CDMode-CTRL:
  (CDMode-CTRL-States, CDMode-CTRL-Init, CDMode-CTRL-Labels, CDMode-CTRL-Delta)
  : seqauto
apply (unfold seqauto-def SeqAuto-def CDMode-CTRL-States-def CDMode-CTRL-Init-def
  CDMode-CTRL-Labels-def CDMode-CTRL-Delta-def)
apply simp
done

```

```

lemma States-CDMode-CTRL:
  States CDMode-CTRL = CDMode-CTRL-States
apply (simp add: CDMode-CTRL-def)
apply (unfold States-def)
apply (simp add: Abs-seqauto-inverse check-CDMode-CTRL)
done

```

```

lemma Init-State-CDMode-CTRL:
  InitState CDMode-CTRL = CDMode-CTRL-Init
apply (simp add: CDMode-CTRL-def)
apply (unfold InitState-def)
apply (simp add: Abs-seqauto-inverse check-CDMode-CTRL)
done

```

```

lemma Labels-CDMode-CTRL:
  Labels CDMode-CTRL = CDMode-CTRL-Labels
apply (simp add: CDMode-CTRL-def)
apply (unfold Labels-def)
apply (simp add: Abs-seqauto-inverse check-CDMode-CTRL)
done

```

```

lemma Delta-CDMode-CTRL:
  Delta CDMode-CTRL = CDMode-CTRL-Delta
apply (simp add: CDMode-CTRL-def)
apply (unfold Delta-def)
apply (simp add: Abs-seqauto-inverse check-CDMode-CTRL)
done

```

```

schematic-goal Events-CDMode-CTRL:
  SAEvents CDMode-CTRL = ?X
apply (unfold SAEvents-def)
apply (rule trans)
apply (simp add: expr-def Label-def Delta-CDMode-CTRL CDMode-CTRL-Delta-def
  CDMode-CTRL-Action1-def CDMode-CTRL-Action2-def)
apply (rule refl)
done

```

12.2.7 Hierarchical Automaton *CarAudioSystem*

lemmas *CarAudioSystemStates* = *States-Root-CTRL States-CDPlayer-CTRL States-AudioPlayer-CTRL States-On-CTRL*

States-TunerMode-CTRL States-CDMode-CTRL
Root-CTRL-States-def CDPlayer-CTRL-States-def
AudioPlayer-CTRL-States-def
On-CTRL-States-def TunerMode-CTRL-States-def
CDMode-CTRL-States-def

lemmas *CarAudioSystemInitState* = *Init-State-Root-CTRL Init-State-CDPlayer-CTRL Init-State-AudioPlayer-CTRL*

Init-State-On-CTRL Init-State-TunerMode-CTRL
Init-State-CDMode-CTRL
Root-CTRL-Init-def CDPlayer-CTRL-Init-def
AudioPlayer-CTRL-Init-def
On-CTRL-Init-def TunerMode-CTRL-Init-def
CDMode-CTRL-Init-def

lemmas *CarAudioSystemEvents* = *Events-Root-CTRL Events-CDPlayer-CTRL Events-AudioPlayer-CTRL Events-On-CTRL*

Events-TunerMode-CTRL Events-CDMode-CTRL

lemmas *CarAudioSystemthms* = *CarAudioSystemStates CarAudioSystemEvents CarAudioSystemInitState*

schematic-goal *CarAudioSystem-StatesRoot:*

HStates (PseudoHA Root-CTRL (LiftInitData [V0 0, V1 0])) = ?X

apply (*wellformed CarAudioSystemthms*)+

done

lemmas *CarAudioSystemthms-1* = *CarAudioSystemthms CarAudioSystem-StatesRoot*

schematic-goal *CarAudioSystem-StatesCDPlayer:*

HStates (PseudoHA Root-CTRL (LiftInitData [V0 0, V1 0])) [++]

("CarAudioSystem", CDPlayer-CTRL)) = ?X

apply (*wellformed CarAudioSystemthms-1*)+

done

lemmas *CarAudioSystemthms-2* = *CarAudioSystemthms-1 CarAudioSystem-StatesCDPlayer*

schematic-goal *CarAudioSystem-StatesAudioPlayer:*

HStates (PseudoHA Root-CTRL (LiftInitData [V0 0, V1 0]))

[++] ("CarAudioSystem", CDPlayer-CTRL)

[++] ("CarAudioSystem", AudioPlayer-CTRL)) = ?X

apply (*wellformed CarAudioSystemthms-2*) +
done

lemmas *CarAudioSystemthms-3* = *CarAudioSystemthms-2 CarAudioSystem-StatesAudioPlayer*

schematic-goal *CarAudioSystem-StatesTunerMode*:
HASates (PseudoHA Root-CTRL (LiftInitData [V0 0, V1 0])
 [++] ("*CarAudioSystem*", *CDPlayer-CTRL*)
 [++] ("*CarAudioSystem*", *AudioPlayer-CTRL*)
 [++] ("*On*", *TunerMode-CTRL*)) = ?X

apply (*wellformed CarAudioSystemthms-3*) +
done

lemmas *CarAudioSystemthms-4* = *CarAudioSystemthms-3 CarAudioSystem-StatesTunerMode*

schematic-goal *CarAudioSystem-StatesCDMode*:
HASates (PseudoHA Root-CTRL (LiftInitData [V0 0, V1 0])
 [++] ("*CarAudioSystem*", *CDPlayer-CTRL*)
 [++] ("*CarAudioSystem*", *AudioPlayer-CTRL*)
 [++] ("*On*", *TunerMode-CTRL*)
 [++] ("*On*", *CDMode-CTRL*)) = ?X

apply (*wellformed CarAudioSystemthms-4*) +
done

lemmas *CarAudioSystemthms-5* = *CarAudioSystemthms-4 CarAudioSystem-StatesCDMode*

schematic-goal *SAsCarAudioSystem*:
SAs CarAudioSystem = ?X
apply (*unfold CarAudioSystem-def*)
apply (*wellformed CarAudioSystemthms-5*) +
done

schematic-goal *EventsCarAudioSystem*:
HAEvents CarAudioSystem = ?X
apply (*unfold CarAudioSystem-def*)
apply (*wellformed CarAudioSystemthms-5*) +
done

schematic-goal *CompFunCarAudioSystem*:
CompFun CarAudioSystem = ?X
apply (*unfold CarAudioSystem-def*)
apply (*wellformed CarAudioSystemthms-5*) +
done

schematic-goal *StatesCarAudioSystem*:
HASates CarAudioSystem = ?X
apply (*unfold CarAudioSystem-def*)
apply (*wellformed CarAudioSystemthms-5*) +
done

```

schematic-goal ValueCarAudioSystem:
  HASInitValue CarAudioSystem = ?X
apply (unfold CarAudioSystem-def)
apply (wellformed CarAudioSystemthms-5)+
done

```

```

schematic-goal HASInitStatesCarAudioSystem:
  HASInitStates CarAudioSystem = ?X
by (simp add: HASInitStates-def SAsCarAudioSystem CarAudioSystemInitState)

```

```

schematic-goal HARootCarAudioSystem:
  HARoot CarAudioSystem = ?X
apply (unfold CarAudioSystem-def)
apply (wellformed CarAudioSystemthms-5)+
done

```

```

schematic-goal HASInitStateCarAudioSystem:
  HASInitState CarAudioSystem = ?X
by (simp add: HARootCarAudioSystem HASInitState-def CarAudioSystemInitState)

```

```

lemma check-DataSpace [simp]:
  [range V0, range V1] ∈ dataspace
apply (unfold dataspace-def DataSpace.DataSpace-def)
apply auto
apply (rename-tac D)
apply (case-tac D)
apply auto
done

```

```

lemma PartNum-DataSpace [simp]:
  PartNum (DSpace) = 2
apply (unfold PartNum-def DSpace-def)
apply (simp add: Abs-dataspace-inverse)
done

```

```

lemma PartDom-DataSpace-V0 [simp]:
  (PartDom DSpace 0) = range V0
apply (unfold PartDom-def DSpace-def)
apply (simp add: Abs-dataspace-inverse)
done

```

```

lemma PartDom-DataSpace-V1 [simp]:
  (PartDom DSpace (Suc 0)) = range V1

```

```

apply (unfold PartDom-def DSpace-def)
apply (simp add: Abs-dataspace-inverse)
done

```

```

lemma check-InitData [simp]:
  ([V0 0, V1 0], DSpace)  $\in$  data
apply (unfold data-def Data.Data-def)
apply auto
apply (rename-tac d)
apply (case-tac d=0  $\vee$  d = 1)
apply auto
done

```

```

lemma Select0-InitData [simp]:
  Select0 (LiftInitData [V0 0, V1 0]) = 0
apply (unfold LiftInitData-def Select0-def DataPart-def DataValue-def)
apply (simp add: Abs-data-inverse)
done

```

```

lemma Select1-InitData [simp]:
  Select1 (LiftInitData [V0 0, V1 0]) = 0
apply (unfold LiftInitData-def Select1-def DataPart-def DataValue-def)
apply (simp add: Abs-data-inverse)
done

```

```

lemma HAINitValue1-CarAudioSystem:
  CarAudioSystem  $\models$  H = Atom (VAL ( $\lambda$  d. (Select0 d) = 0))
apply (simp add: ValueCarAudioSystem)
done

```

```

lemma HAINitValue2-CarAudioSystem:
  CarAudioSystem  $\models$  H = Atom (VAL ( $\lambda$  d. (Select1 d) = 0))
apply (simp add: ValueCarAudioSystem)
done

```

```

lemma HAINitValue-DSpace-CarAudioSystem [simp]:
  Data.DataSpace (LiftInitData [V0 0, V1 0]) = DSpace
apply (unfold LiftInitData-def Data.DataSpace-def)
apply (simp add: Abs-data-inverse)
done

```

```

lemma check-InitStatus [simp]:
  (CarAudioSystem, InitConf CarAudioSystem, {}, LiftInitData [V0 0, V1 0])  $\in$ 
status
apply (unfold status-def Status-def)
apply (simp add: ValueCarAudioSystem)
done

```

```

lemma InitData-InitStatus [simp]:

```

```

  Value (InitStatus CarAudioSystem) = LiftInitData [V0 0, V1 0]
apply (simp add: ValueCarAudioSystem)
done

```

```

lemma Events-InitStatus [simp]:
  Events (InitStatus CarAudioSystem) = {}
apply (unfold InitStatus-def Events-def)
apply (simp add: Abs-status-inverse)
done

```

```

lemma Conf-InitStatus [simp]:
  Conf (InitStatus CarAudioSystem) = InitConf CarAudioSystem
apply (unfold InitStatus-def Conf-def)
apply (simp add: Abs-status-inverse)
done

```

```

lemma CompFunCarAudioSystem-the:
  the (CompFun CarAudioSystem "On") = {CDMode-CTRL, TunerMode-CTRL}
apply (unfold CarAudioSystem-def)
apply (subst AddSA-CompFun-the)
prefer 3
apply (subst AddSA-CompFun-the)
prefer 3
apply (subst AddSA-CompFun-the2)
apply (wellformed CarAudioSystemthms-5)+
done

```

```

lemma CompFunCarAudioSystem-the2:
  the (CompFun CarAudioSystem "CarAudioSystem") = {AudioPlayer-CTRL,
  CDPlayer-CTRL}
apply (unfold CarAudioSystem-def)
apply (subst AddSA-CompFun-the3)
prefer 5
apply (subst AddSA-CompFun-the3)
prefer 5
apply (subst AddSA-CompFun-the)
prefer 3
apply (subst AddSA-CompFun-the)
prefer 3
apply (subst PseudoHA-CompFun-the)
apply (wellformed CarAudioSystemthms-5)+
done

```

```

lemma CompFunCarAudioSystem-the3:
  the (CompFun CarAudioSystem "Off") = {}
apply (unfold CarAudioSystem-def)
apply (subst AddSA-CompFun-the3)
prefer 5
apply (subst AddSA-CompFun-the3)

```

```

prefer 5
apply (subst AddSA-CompFun-the2)
apply (wellformed CarAudioSystemthms-5)+
done

```

```

schematic-goal CompFunCarAudioSystem-ran:
  ran (CompFun CarAudioSystem) = ?X
apply (unfold CarAudioSystem-def)
apply (rule AddSA-CompFun-ran3-IFF)
prefer 6
apply (subst AddSA-CompFun-PseudoHA-ran2)
prefer 4
apply (simp add: insert-commute)
apply (wellformed CarAudioSystemthms-5)+
done

```

```

lemma Root-CTRL-CDPlayer-CTRL-noteq [simp]:
  Root-CTRL ≠ CDPlayer-CTRL
apply (rule SA-States-disjunct)
apply (wellformed CarAudioSystemthms-5)+
done

```

```

lemma Root-CTRL-AudioPlayer-CTRL-noteq [simp]:
  Root-CTRL ≠ AudioPlayer-CTRL
apply (rule SA-States-disjunct)
apply (wellformed CarAudioSystemthms-5)+
done

```

```

lemma Root-CTRL-TunerMode-CTRL-noteq [simp]:
  Root-CTRL ≠ TunerMode-CTRL
apply (rule SA-States-disjunct)
apply (wellformed CarAudioSystemthms-5)+
done

```

```

lemma Root-CTRL-CDMode-CTRL-noteq [simp]:
  Root-CTRL ≠ CDMode-CTRL
apply (rule SA-States-disjunct)
apply (wellformed CarAudioSystemthms-5)+
done

```

```

lemma CDPlayer-CTRL-AudioPlayer-CTRL-noteq [simp]:
  CDPlayer-CTRL ≠ AudioPlayer-CTRL
apply (rule SA-States-disjunct)
apply (wellformed CarAudioSystemthms-5)+
done

```

```

lemma CDPlayer-CTRL-TunerMode-CTRL-noteq [simp]:
  CDPlayer-CTRL ≠ TunerMode-CTRL
apply (rule SA-States-disjunct)

```


apply (*wellformed CarAudioSystemthms-5*)+
done

lemma *CDPlayer-CTRL-CDMode-CTRL-noteq [simp]:*
CDPlayer-CTRL $\neq$ *CDMode-CTRL*
apply (*rule SA-States-disjunct*)
apply (*wellformed CarAudioSystemthms-5*)+
done

lemma *AudioPlayer-CTRL-TunerMode-CTRL-noteq [simp]:*
AudioPlayer-CTRL $\neq$ *TunerMode-CTRL*
apply (*rule SA-States-disjunct*)
apply (*wellformed CarAudioSystemthms-5*)+
done

lemma *AudioPlayer-CTRL-CDMode-CTRL-noteq [simp]:*
AudioPlayer-CTRL $\neq$ *CDMode-CTRL*
apply (*rule SA-States-disjunct*)
apply (*wellformed CarAudioSystemthms-5*)+
done

lemma *TunerMode-CTRL-CDMode-CTRL-noteq [simp]:*
TunerMode-CTRL $\neq$ *CDMode-CTRL*
apply (*rule SA-States-disjunct*)
apply (*wellformed CarAudioSystemthms-5*)+
done

schematic-goal *Chi-CarAudioSystem:*
Chi CarAudioSystem "CarAudioSystem" = ?X
apply (*unfold Chi-def*)
apply (*rule trans*)
apply (*simp add: SAsCarAudioSystem StatesCarAudioSystem restrict-def Comp-FunCarAudioSystem-the2*)
apply (*rule trans*)
apply (*simp add: not-sym*)
apply (*simp add: CarAudioSystemStates insert-or*)
done

schematic-goal *Chi-CarAudioSystem-On:*
Chi CarAudioSystem "On" = ?X
apply (*unfold Chi-def*)
apply (*rule trans*)
apply (*simp add: SAsCarAudioSystem StatesCarAudioSystem restrict-def Comp-FunCarAudioSystem-the*)
apply (*rule trans*)
apply (*simp add: not-sym*)
apply (*simp add: CarAudioSystemStates insert-or*)
done

```

schematic-goal Chi-CarAudioSystem-Off:
  Chi CarAudioSystem "Off" = ?X
apply (unfold Chi-def)
apply (simp add: SAsCarAudioSystem StatesCarAudioSystem restrict-def Comp-
FunCarAudioSystem-the3)
done

```

```

schematic-goal InitConf-CarAudioSystem:
  InitConf CarAudioSystem = ?X
apply (unfold CarAudioSystem-def)
apply (rule AddSA-InitConf-IFF) +
apply simp
apply (wellformed CarAudioSystemthms-5)
apply fast
apply (wellformed CarAudioSystemthms-5)
apply simp
apply (simp add: CarAudioSystemthms-5)
apply (wellformed CarAudioSystemthms-5)
apply fast
apply (wellformed CarAudioSystemthms-5)
apply fast
apply simp
apply (wellformed CarAudioSystemthms-5)
apply fast
apply (wellformed CarAudioSystemthms-5)
apply fast
apply simp
apply (wellformed CarAudioSystemthms-5)
apply force
apply (wellformed CarAudioSystemthms-5)
apply fast
apply (simp add: CarAudioSystemthms-5)
done

```

```

lemma Initial-State-CarAudioSystem:
  CarAudioSystem |= H = Atom (IN "Off")
apply (simp add: InitConf-CarAudioSystem )
done

```

end

References

- [HN96] D. Harel and D. Naamad. A STATEMATE semantics for state-charts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, Oct 1996.

- [MLS97] E. Mikk, Y. Lakhnech, and M. Siegel. Hierarchical automata as model for statecharts. In *Asian Computing Science Conference (ASIAN'97)*, Springer LNCS, **1345**, 1997.
- [HK01] S. Helke and F. Kammüller. Representing Hierarchical Automata in Interactive Theorem Provers. In R. J. Boulton, P. B. Jackson, editors, *Theorem Proving in Higher Order Logics, TPHOLs 2001*, Springer LNCS, **2152**, 2001.
- [HK05] S. Helke and F. Kammüller. Structure Preserving Data Abstractions for Statecharts. In F. Wang, editors, *Formal Techniques for Networked and Distributed Systems, FORTE 2005*, Springer LNCS, **3731**, 2005.
- [Hel07] S. Helke. *Verification of Statecharts using Structure- and Property-Preserving Data Abstraction [german]* . PhD thesis, Fakultät IV, Technische Universität Berlin, Germany, 2007.